Automated Software Engineering (2019) 26:653–704
https://doi.org/10.1007/s10515-019-00264-4

Automatic B-model repair using model checking and machine learning

Cheng-Hao Cai[1] · Jing Sun[1] · Gillian Dobbie[1]

Received: 12 November 2018 / Accepted: 2 August 2019 / Published online: 14 August 2019

Abstract
The B-method, which provides automated verification for the design of software systems, still requires users to manually repair faulty models. This paper proposes B-repair, an approach that supports automated repair of faulty models written in the B formal specification language. After discovering a fault in a model using the B-method, B-repair is able to suggest possible repairs for the fault, estimate the quality of suggested repairs and use a suitable repair to revise the model. The suggestion of repairs is produced using the $Isolation$ method, which suggests changing the pre-conditions of operations, and the $Revision$ method, which suggests changing the post-conditions of operations. The estimation of repair quality makes use of machine learning techniques that can learn the features of state transitions. After estimating the quality of suggested repairs, the repairs are ranked, and a best repair is selected according to the result of ranking and is used to revise the model. This approach has been evaluated using a set of finite state machines seeded with faults and a case study. The evaluation has revealed that B-repair is able to repair a large number of faults, including invariant violations, assertion violations and deadlock states, and gain high accuracies of repair. Using the combination of model checking and machine learning-guided techniques, B-repair saves development time by finding and repairing faults automatically during design.

Keywords Model repair · B-method · Model checking · Formal verification · Machine learning

This work is supported by the State Scholarship Fund sponsored by the China Scholarship Council [Grant Number: 201708060334].

✉ Cheng-Hao Cai
chenghao.cai@auckland.ac.nz

Jing Sun
jing.sun@auckland.ac.nz

Gillian Dobbie
g.dobbie@auckland.ac.nz

1 School of Computer Science, University of Auckland, 38 Princes Street, Auckland 1142, New Zealand

1 Introduction

Formal verification focuses on using rigorous mathematical reasoning to verify the correctness of systems. For example, the railway control system in Paris has been verified using the B-method, which is a formal verification technique based on Zermelo-Fraenkel (ZF) set theory (Abrial 2005; Behm et al. 1999). As ZF set theory is built upon natural deduction and nine axioms of sets (Bagaria 2017), all reasoning processes in B are traceable, which means that the verification of B is highly reliable and trustworthy. Formal verification treats a system as premises and system requirements as goals. Checking the system requirements can thus be treated as proving the goals using the premises.

Although formal verification techniques can automatically discover faults in a system, in most cases, users need to fix the faulty parts of the system. As a result, in order to remove the faults, a considerable amount of time is usually spent changing the design of the system, making system development very inefficient. In order to solve this problem, this research focuses on the automation of design repair. That is, after discovering faults in the system using formal verification techniques, computers can automatically remove the faults by changing the design of the system. Formally, the research question of this work is as follows. Suppose that a system is represented as a model $M = \{R_1, R_2, \ldots, R_n\}$, where $R_1, R_2, \ldots, R_n$ are logical rules, and the expected properties of the system are represented as a set of specifications $S = \{\phi_1, \phi_2, \ldots, \phi_m\}$. It is expected that $M \models S$ is proved to be true. If there exists a specification ϕ_i that is proved to be false, are there any methods that can automatically change $R_1, R_2, \ldots, R_n$ to make ϕ_i true?

Recently, a number of approaches to automated system repair have targeted the above problem. Alrajeh and Craven (2014) have proposed an automated error-detection and repair method for finite state machines. It can find error transitions and revise model descriptions by adding new operations and deleting faulty operations. Schmidt et al. (2016) have proposed an interactive repair technique for Event-B. It discovers faults of synthesised events, such as invariant violations and missing transitions, and then changes the state transitions until there is no faulty event. Yang et al. (2012) have proposed an automated program repair approach based on a logical method and a test repair method. When discovering faults in a program, it makes use of test data and a SAT (satisfiability) solver to generate possible repairs and then applies the generated repairs to the program. Gopinath et al. (2011) have proposed a repair technique based on relational logics and SAT solving. The relational logics are used to encode programs. If an error is detected from the encoding of a program, an existing variable will be replaced by a new variable, and the SAT solver is used to find possible valuations of the new variable. It is often the case that many possible repairs are available for a given model, but evaluating the quality of repairs remains a challenge (Le Goues et al. 2013). For model repair methods based on testing, the evaluation of repairs is often based on the number of passed test cases in test suites (Yang et al. 2012). For those based on formal verification, however, evaluating repairs is difficult, because formal verification is usually done without test suites. Thus, suggesting high-quality repairs within the framework of formal verification remains a challenging task.

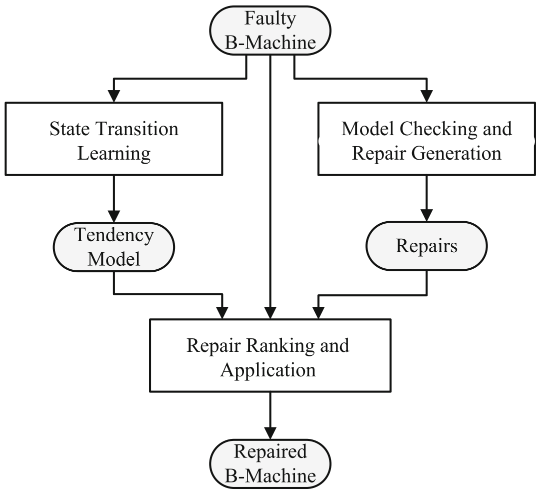
Faulty B-Machine
State Transition Learning
Model Checking and Repair Generation
Tendency Model
Repairs
Repair Ranking and Application
Repaired B-Machine

Fig. 1 A diagram of B-Repair

In order to solve the problem of suggesting high-quality repairs within the framework of formal verification, we propose B-repair. Figure 1 shows the basic working of B-repair, which includes three components. The first component is the state transition learning part. Given a faulty abstract machine, a tendency model reflecting the behaviours of the machine is built by learning from available state transitions in the machine. This step requires model checking and machine learning. Model checking can perform symbolic analysis and compute available state transitions from the given abstract machine, and machine learning can use these transitions as training data to train the tendency model. The second component is the model checking and repair generation part. In order to diagnose the abstract machine, the model checker detects errors including invariant violations, assertion violations and deadlocks. Then the repair generator suggests repairs that can remove the detected errors. Usually, a single error can be removed using different repairs. In order to find the best repair, the suggested repairs need to be ranked. The third component is the repair ranking and application part. Suggested repairs are ranked using the tendency model. According to their ranks, a suitable repair can be selected and applied to the abstract machine, leading to a repaired machine.

The innovation of the proposed B-repair method is that it combines model checking and machine learning in order to repair faulty abstract machines. Model checking techniques derive state graphs of abstract machines and then discover faulty states. Thus, available repairs can be suggested in order to eliminate the faulty states. Moreover, machine learning techniques can learn features of transitions in the state graphs and then evaluate the suggested repairs. If a repair can maintain the learnt features, it will be considered a high-quality repair, and vice versa.

This study extends our previous study on B model repair (Cai et al. 2018). The differences between this study and the previous one include the following aspects. Firstly, this study provides both theoretical and empirical explanations of B-repair, while the previous study focuses on empirical explanations. Secondly, this study provides a large scale evaluation that shows the overall performance of B-repair on large models, while the previous study only provides preliminary results on small models. Finally, this study demonstrates that B-repair can process the post-condition of an operation and the relation between the pre- and post-conditions of an operation, while the previous study does not. Contributions of our study are listed below.

- The B-repair method for repairing faulty abstract machines.
- The isolation and revision method for producing local repairs.
- The tendency model approach as a general criterion of repair ranking.

The rest of this paper is organised as follows. Section 2 provides the background of this study. Section 3 presents a brief introduction to the important preliminary knowledge, such as model checking and supervised learning. Section 4 introduces the proposed B-repair method, especially the isolation and revision methods. Section 5 presents the repair ranking approach based on tendency models. Section 6 describes the design and implementation of the B-repair tool. Section 7 presents a running example on how B-repair works for a read world problem. Section 8 evaluates the B-repair method using various faulty abstract machines. Section 9 discusses the findings and compares the results with related work. Section 10 concludes the paper and outlines the future directions.

2 Background

Automated reasoning plays a key role in computer-aided verification, because verifying the correctness of a system is often considered a problem of proving the consistency of a logical representation of the system, and thus the problem can be solved using reasoning engines. Fundamental components of automated reasoning include propositional logic, first-order (predicate) logic and higher-order (predicate) logic (Huth and Ryan 2004; Nipkow et al. 2002). Propositional logic represents problems using atomic propositions, which are assigned $True$ (T) and $False$ (F), and logical connectives such as negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). Natural deduction, which makes use of introduction rules and elimination rules of the above logical connectives, can be used to solve problems of propositional logic. Based on propositional logic, first-order logic uses universal quantifiers ($\forall$), existential quantifiers ($\exists$) and predicates, which consist of not only atomic propositions, but also constants and variables, to represent more complex descriptions. A predicate is of the form $f(x_1, x_2, \ldots, x_n)$, where f is a functor and each x_i ($i = 1, 2, \ldots, n$) is an argument. In first-order logic, only arguments can be quantified. Higher-order logic extends first-order logic by allowing the quantification of functors. Moreover, term manipulation algorithms, such as unification (Siekmann 1989) and rewriting (Baader and Nipkow 1998), are used to process predicates. Unification is an algorithm that finds possible substitutions of variables by matching terms in two predicates, and rewriting is an algorithm that simplifies predicates by replacing their terms and sub-terms with equivalent ones. In order to solve a problem of predicate logic, natural deduction, unification and rewriting are iteratively applied to the problem, until it is proved to be $True$ or $False$. Further, temporal logics are often used to verify time-dependent properties of systems. Two major branches of temporal logics are linear temporal logic (LTL)

(Pnueli 1977) and computational tree logic (CTL) (Clarke and Emerson 1981). LTL can describe properties using temporal connectives such as "X" (next), "G" (globally), "F" (future) and "U" (until). CTL can describe properties using path quantifiers that include "A" (for all paths) and "E" (there exists a path), and the temporal connectives, and thus CTL operators include AX, EX, AG, EG, AF, EF, AU and EU.

Model checking techniques, which makes use of the theory of automated reasoning, are used to verify whether or not a design model of a system satisfies properties such as safety, liveness, reachability and termination (D'Silva et al. 2008). There are a number of well-developed model checking techniques. For instance, B is a formal notation based on first-order predicate logic, which is built on ZF set theory (Abrial 2005; Behm et al. 1999). Currently, B has enabled the functional representations of higher-order logic (Leuschel et al. 2009). Moreover, NuSMV is a model checker supporting the verification of LTL and CTL assertions (Cimatti et al. 1999). It uses binary decision diagram (BDD)-based symbolic model checking that can maximally reduce the scale of state graphs by merging redundant states into sets. Thus, it is able to check models with over 10^{20} states (Burch et al. 1992). However, it does not support set notations and predicate representations as complex as those of the B-method.

In the B-method (Abrial 2005), a system is described by an abstract machine that consists of variables, an invariant, initialisation, operations and other optional components. The variables are used to record states of the system. The initialisation assigns initial values to the variables and produces an initial state(s) of the system. Starting from the initial state(s), the operations are iteratively applied to existing states and derive new states. The invariant describes properties that the system should satisfy, i.e., true for all states of the system. Invariant checking and refinement checking are main functions of the B-method. The goal of invariant checking is to ensure that all states satisfy the invariant (and satisfy assertions that are extensions of the invariant), and the goal of refinement checking is to ensure that a refined model satisfies all properties of an original model. Currently, the B-method is supported by the ProB tool (Leuschel and Butler 2008). Additionally, Event-B, which is a variant of the classical B, is supported by the Rodin tool (Abrial et al. 2010). The constraint solver of satisfiability modulo theories (SMT) is one of the core components of both ProB and Rodin (Krings and Leuschel 2016). Currently, the SMT solver of ProB can solve constraints of Boolean values, integers, sets, relations and records. The B-method has been applied to many fields. A typical application of B was the verification of the automatic train operating system in Paris (Behm et al. 1999). The system had safety critical properties that were described by a software requirement document, and the B-method was used to ensure that the source code of the software fulfilled the document. Moreover, refinement checking of the B-method was used to verify circuits designed in VHDL (Boulanger et al. 2002). Ports of the circuits were considered to be variables, connections between ports were modelled by invariants, and signal propagation was modelled by operations. The goal of verification was to ensure that each refined circuit satisfied conditions of the original circuit. Additionally, the B-method was used to model and verify a real interaction protocol named the Contract Net Protocol in a multi-agent system (Fadil and Koning 2005).

Supervised machine learning models, such as classification and regression trees (CART), logistic models and residual networks (ResNet), are able to separate labelled

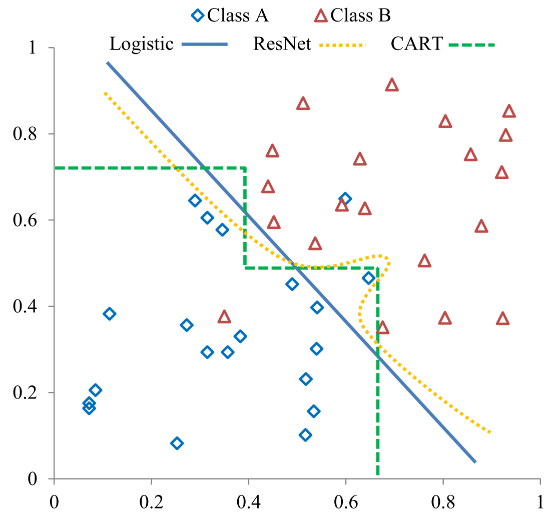

Fig. 2 Decision boundaries of the machine learning models

data using decision boundaries. For example, a CART can recursively split data into subtrees according to attributes of the data, and many CARTs with random attributes can construct a random forest (Ho 1995; Breiman et al. 1984). As a result, these CARTs can specify vertical decision boundaries that are able to separate the data into different zones. A logistic model is able to specify plain decision boundaries. Given multi-dimensional data points, the logistic model can learn to classify the data points using hyperplanes in a multi-dimensional space (Cox 1958). Instead of hyperplanes, a ResNet, which consists of many layers of neural networks and residual building blocks, can use hypersurfaces to classify the data points (He et al. 2016). Figure 2 shows how the above machine learning models form decision boundaries in a two-dimensional space.

3 Preliminaries

B-repair is based on model checking of B and supervised machine learning. In this section, both aspects are briefly revisited. Readers could skip this section if they have been familiar with the B notation, the ProB model checker, classification and regression trees, logistic models and residual networks.

3.1 The B notation and the ProB model checker

This section provides a brief introduction to the B notation and the ProB model checker. For full and rigorous descriptions of the B notation, please refer to the work by Abrial (2005). An abstract machine of B can be constructed by the clauses listed below.

– The MACHINE clause indicates the name of the machine.
– The SETS clause indicates sets used by the machine, and each set includes a number of distinct elements.

– The CONSTANTS clause indicates constants used by the machine, and each constant is a set, a Boolean, an integer, a distinct element or a function.
– The PROPERTIES clause indicates properties that the constants should preserve. The properties are formed as a conjunction of predicates.
– The VARIABLES clause indicates variables used by the machine, and each variable is a set, a Boolean, an integer or a distinct element. The variables and their values form a state of the machine.
– The INVARIANT clause indicates invariants that the variables should preserve. The invariants are formed as a conjunction of predicates.
– The ASSERTIONS clause indicates assertions that are supposed to be preserved by the machine. The assertions are formed as a conjunction of predicates.
– The INITIALISATION clause indicates a substitution that assigns initial values to the variables. All states of the machine originate from the initialisation.
– The OPERATIONS clause indicates operations that are used to derive new states from existing states. Each operation should have a pre-condition and a post-condition. The pre-condition is a predicate, and the post-condition includes substitutions. If the pre-condition is true for the current state, the operation will be triggered and lead to a new state satisfying the post-condition.
– The END clause indicates the end of the machine.

The ProB model checker (Leuschel and Butler 2008) is able to check the correctness of abstract machines described by the B notation. Firstly, ProB initialises constants via the CONSTANTS clause and the PROPERTIES clause. Then ProB initialises variables via the VARIABLES clause, the INVARIANT clause and the INITIALISATION clause, deriving an initial state or a set of initial states. Next, ProB derives new states via the OPERATIONS clause and checks (a) if each state satisfies the INVARIANT clause, (b) if each state satisfies the ASSERTIONS clause and (c) if each state has at least one outgoing transition. If a state does not meet one of the three requirements, ProB will report the faulty state with (a) an invariant violation, (b) an assertion violation and (c) a deadlock respectively. Finally, a state graph will be outputted, recording all derived states, all derived operations connecting the states and an annotation that highlights the faulty state. Moreover, ProB has provided the SMT solver (Krings and Leuschel 2016) that can solve constraints written in the B notation. Practically, the SMT solver can be triggered using the "ANY s WHERE p THEN q END" syntax, where s is a set of variables, p is a set of constraints that the variables should satisfy, and q is a "container" that records any solutions satisfying the constraints. Additionally, below are B's terminologies frequently used in this work.

– *Predicate* A predicate is a statement that can be proved true or false, and it is of the form $f(x_1, x_2, \ldots, x_n)$, where f is a functor and each x_i ($i = 1, 2, \ldots, n$) is an argument. In particular, the functor can be logical connectives such as negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). Moreover, universal quantifiers ($\forall$) and existential quantifiers ($\exists$) can be used to quantify arguments in predicates.
– *Substitution* A substitution is used to change the value of a variable. A primitive substitution is denoted by $v_1 := x_1, v_2 := x_2, \ldots, v_n := x_n$, where each v_i ($i = 1, 2, \ldots, n$) is a variable, and x_i is a value or an expression that has the same

type as v_i. A conditional substitution is denoted by $p \implies (v_1 := x_1, v_2 := x_2, \ldots, v_n := x_n)$, which means that the substitution happens only if the predicate p is proved true.

- *State* A state s is a conjunction that indicates a pattern of all variables. It is denoted by $v_1 = s[v_1] \wedge v_2 = s[v_2] \wedge \cdots \wedge v_n = s[v_n]$. Each v_i $(i = 1, 2, \ldots, n)$ is a variable, and each $s[v_i]$ is the value of v_i.
- *Pre-condition* The pre-condition of an operation is a predicate, which has to be proved true for the current state before the operation is triggered.
- *Post-condition* The post-condition of an operation is a substitution, which indicates the next state after the operation is triggered.
- *State transition* A state transition is the derivation from an existing state s_x to a new state s_{new} via an operation α. It is denoted by $s_x \xrightarrow{\alpha} s_{new}$. If s_x satisfies the pre-conditions of α, the substitutions of α will be applied to s_x, and s_{new} will be created.
- *Path* A path p is a sequence of states connected by available operations. It is formed as $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_l} s_l$, where each s_i $(i = 0, 1, 2, \ldots, l)$ is a state and each α_j $(j = 1, 2, \ldots, l)$ is an operation.
- *Invariant violation* An invariant violation is triggered when a state s does not satisfy one or more invariants.
- *Assertion violation* An assertion violation is triggered when a state s does not satisfy one or more assertions.
- *Deadlock state* A deadlock state s_{dead} is a state that no operation can be applied to. There does not exist any pre-condition that s_{dead} can satisfy.

3.2 Supervised learning

Supervised learning focuses on learning a function using input signals and corresponding label signals. Given N training data that are of the form (x_i, t_i) $(i = 1, 2, \ldots, N)$, where x is an input signal, and t is a label signal, the goal of learning is to train a function $t = F(x)$ that can map each x_i to t_i as far as possible. In Sects. 3.2.1 and 3.2.2, two types of supervised learning models are introduced.

3.2.1 Classification and regression trees

A CART can perform a regression function by the variance-based node impurity (Breiman et al. 1984; Loh 2011). In the case of binary attributes (i.e., their values are either 0 or 1), the variance-based node impurity (NI) can be defined as

$$NI(X, \alpha) = \frac{num(X_{\alpha=0}) \cdot var(X_{\alpha=0}) + num(X_{\alpha=1}) \cdot var(X_{\alpha=1})}{num(X_{\alpha=0}) + num(X_{\alpha=1})} \quad (1)$$

where X is a set of training samples, each sample has a label and a number of attributes, α is an attribute, $X_{\alpha=p}$ (p is 0 or 1) is the set containing all samples in X that satisfy $\alpha = p$, $num(S)$ is the number of samples in a set S, and $var(S)$ is the variance of labels in S. The training of the CART starts from a root node with a set of training samples X_0. Firstly, N attributes $\{\alpha_1, \alpha_2, \ldots, \alpha_N\}$ are randomly selected. Then each $NI(X_0, \alpha_i)$ $(i = 1, 2, \ldots, N)$ is computed. Suppose that α_I leads to the minimum NI. Splitting conditions are defined as $\alpha_I = 0$ and $\alpha_I = 1$. Next, a left node and a right node are created using the splitting conditions. The left node contains all samples satisfying $\alpha_I = 0$, and the right node contains all samples satisfying $\alpha_I = 1$. Finally, the above steps are recursively applied to both left and right nodes, until the minimal NI is smaller than a predefined limit NI_0. After finishing the training, the CART can be used to predict labels of samples. Starting from the root node, a sample x can find a path from the current node to the next node by the splitting conditions, and x can finally reach a leaf node V. Then the average of labels of the training samples in V is considered a predicted label of x. Many CARTs can be combined together to form a random forest, where each tree is trained using randomly selected X_0 and NI_0 (Ho 1995).

3.2.2 Logistic models and residual networks

The logistic function is

$$logistic(u) = \frac{1}{1 + e^{-u}} \tag{2}$$

and the logistic model can be defined as

$$y = logistic(\boldsymbol{A} \cdot \boldsymbol{x} + b) \tag{3}$$

where A is an N-dimensional weight vector, b is a bias, x is an N-dimensional input vector, and y is an output number (Cox 1958). The logistic model is one of the simplest neural network architectures. In order to model more complex data, it can be combined with multi-layer architectures such as residual networks (He et al. 2016). A residual network is constructed by stacking a number of residual building blocks together. Each block is defined as

$$\boldsymbol{y} = F(\boldsymbol{x}) + \boldsymbol{x} \tag{4}$$

where $\boldsymbol{x}$ is an N-dimensional input vector, F is a feed-forward neural network, and $\boldsymbol{y}$ is an N-dimensional output vector. When many blocks are stacked together, the output vector of each block is the input vector of the next block. In particular, the output vector of the last block can be the input vector of another neural network such as the logistic model. The training of the logistic model and the residual network can be realised using stochastic gradient descent (Bottou 2012).

4 The B-repair method

In the following two sections, the B-repair method (Cai et al. 2018), including fault localisation, isolation, revision and repair ranking, is introduced. Section 4 focuses on fault localisation, isolation and revision, and Sect. 5 focuses on repair ranking.

4.1 Fault localisation

The correctness of models can be checked automatically via formal verification techniques. However, the correction (repair) of a faulty model is often a tedious and manual process. Users may spend a great amount of time understanding the cause of faults and working out the possible repairs. To make the repair process more efficient, we propose an approach that can automatically eliminate the faults in a formal design model.

Our approach to model repair is based on the B-method (Abrial 2005) and its associated tool named ProB (Leuschel and Butler 2008). Before repairing a faulty model, the state space of the model is explored and used to localise faulty states. The derivation of the state graph usually starts from an initialisation s_0, and new states are derived by applying available operations to the existing states. Once a state s_l is derived, its correctness can be checked. If no available operation can be applied to s_l, then s_l is a deadlock state. If an invariant or assertion is proved to be false for s_l, then s_l triggers an invariant or assertion violation. In the above cases, s_l is considered faulty.

4.2 Isolation

The basic concept of $Isolation$ is to remove a faulty state by changing the pre-condition of a previous operation. It is a weak adaptation of the b-thread patching algorithm (Harel et al. 2014). The main difference between the two methods is that $Isolation$ performs a local repair by removing a single faulty state, while the b-thread patching algorithm performs a global repair by removing all faulty states. Suppose that s_l is verified to be a faulty state and can be reached via $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{l-1}} s_{l-1} \xrightarrow{\alpha_l} s_l$. s_l can then be isolated by adding $\neg s_{l-1}$ to the pre-condition of α_l. After making this change, the operation α_l is no longer activated by s_{l-1}, because the pre-condition is false with respect to s_{l-1}. Thus, the path is changed to $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{l-1}} s_{l-1}$, and s_l has been removed. Formally, the $Isolation$ method is defined as follows.

Definition 1 (*The isolation method*) If $s_p \xrightarrow{\alpha} s_q$ is a state transition, and s_q is a faulty state, then α is changed to α_{Iso} via:

$$\begin{array}{rcl} \alpha & \widehat{=} & \text{PRE } P \text{ THEN } Q \text{ END} \\ & & \downarrow^{Isolation} \\ \alpha_{Iso} & \widehat{=} & \text{PRE } S_{Iso} \wedge P \text{ THEN } Q \text{ END} \end{array} \tag{5}$$

In the above expression, S_{Iso} is a condition:

$$\neg(v_1 = s_p[v_1] \wedge v_2 = s_p[v_2] \wedge \cdots \wedge v_n = s_p[v_n]) \tag{6}$$

where v_i ($i = 1, 2, \ldots, n$) is a variable identifier, and $s_p[v_i]$ is the value of v_i in s_p.

4.3 Revision

The basic concept of *Revision* is to correct a faulty state by changing the post-conditions of a previous operation. Suppose that s_l is verified to be a faulty state and can be reached via $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{l-1}} s_{l-1} \xrightarrow{\alpha_l} s_l$. s_l can be revised via a conditional substitution that rewrites s_l to s_l', where s_l' is a correct state found by the SMT solver of ProB (Krings and Leuschel 2016). The conditional substitution should be added to the end of α_l. Let α_l' denote the resulting operation, thus, the path is changed to $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{l-1}} s_{l-1} \xrightarrow{\alpha_l'} s_l'$, and the faulty state s_l is no longer reached. Formally, the *Revision* method is defined as follows.

Definition 2 (*The revision method*) If $s_p \xrightarrow{\alpha} s_q$ is a state transition, and s_q is a faulty state, then α is changed to α_{Rev} via:

$$\begin{array}{rcl} \alpha & \widehat{=} & \text{PRE } P \text{ THEN } Q \text{ END} \\ & & \downarrow^{Revision} \\ \alpha_{Rev} & \widehat{=} & \text{PRE } P \text{ THEN } Q\,;\ T_{Rev} \text{ END} \end{array} \tag{7}$$

In the above expression, T_{Rev} is a conditional substitution:

$$\begin{array}{l} \text{IF} \\ \quad v_1 = s_q[v_1] \wedge v_2 = s_q[v_2] \wedge \ldots \wedge v_n = s_q[v_n] \\ \text{THEN} \\ \quad v_1 := s_{Rev}[v_1]\,;\ v_2 := s_{Rev}[v_2]\,;\ \ldots\,;\ v_n := s_{Rev}[v_n] \\ \text{END} \end{array} \tag{8}$$

where s_{Rev} is a revision state without any faults, v_i $(i = 1, 2, \ldots, n)$ is a variable identifier, $s_q[v_i]$ is the value of v_i in s_q, and $s_{Rev}[v_i]$ is the value of v_i in s_{Rev}.

The value of s_{Rev} should satisfy all constraints established by invariants and assertions, and it should not be a deadlock state. This problem can be converted to a constraint solving problem and can be solved using the SMT solver. Its solutions should satisfy the following four constraints.

Definition 3 (*Invariant constraint, IC*) The invariant constraint of an abstract machine is formed as a conjunction of all invariants. It is denoted by $inv_1 \wedge inv_2 \wedge \cdots \wedge inv_r$, where $inv_i (i = 1, 2, \ldots, r)$ is an invariant.

Definition 4 (*Assertion constraint, AC*) The assertion constraint of an abstract machine is formed as a conjunction of all assertions. It is denoted by $ast_1 \wedge ast_2 \wedge \cdots \wedge ast_s$, where $ast_i (i = 1, 2, \ldots, s)$ is an assertion.

Definition 5 (*Liveness constraint, LC*) The liveness constraint can guarantee that the solutions are not deadlock states. It is formed as a disjunction of the pre-conditions of all operations. It is denoted by $cond_1 \vee cond_2 \vee \cdots \vee cond_t$, where $cond_i (i = 1, 2, \ldots, t)$ is the pre-condition of an operation.

Definition 6 (*Distance constraint, DC*) The distance constraint is used to restrict the search space of the SMT solver. Note that the distance constraint is optional and only used when the search space is huge. It is formed as $dist(s_q, s_{Rev}) \leq \Delta_{max}$, where $dist(s_x, s_y)$ is a distance function, and Δ_{max} is the upper bound of the distance.

The distance function can be defined as the following absolute distance function.

Definition 7 (*Absolute distance function*) The absolute distance function of revising a state s_x to another state s_y is defined as:

$$dist_{abs}(s_x, s_y) = \sum_{i=1}^{n} Dist(s_x[v_i], s_y[v_i]) \tag{9}$$

where $Dist(u, v)$ is a polymorphic distance function:

- If u and v are sets, then $Dist(u, v) = Card(u \cup v - u \cap v)$, where $Card(S)$ is the cardinality of a set S.
- If u and v are Boolean values, then $Dist(u, v) = \begin{cases} 0\,, & u = v \\ 1\,, & u \neq v \end{cases}$.
- If u and v are integers, then $Dist(u, v) = Abs(u - v)$, where $Abs(x)$ is the absolute value of x.
- If u and v are distinct elements, then $Dist(u, v) = \begin{cases} 0\,, & u \equiv v \\ 1\,, & u \not\equiv v \end{cases}$.

Based on these constraints, the SMT solver can find all solutions satisfying $IC \wedge AC \wedge LC \wedge DC$ if they exist. These solutions satisfy the requirement of liveness and do not trigger any invariant violations or assertion violations. Thus, they can be used as the value of s_{Rev}.

4.4 An example of *Isolation* and *Revision*

Figure 3 provides an example explaining how $Isolation$ and $Revision$ repair a faulty system. In this example, the precedence of operators (from the highest one to the lowest one) is [(), =, ∈, ¬, ∧, ∨, :=, ;]. Suppose that the system has three boolean variables p, q and r. Its invariant is $p \in BOOL \wedge q \in BOOL \wedge r \in BOOL$, its assertion is $(p \vee q \vee r) = True$, and its initial state is $p = True \wedge q = True \wedge r = True$. λ and δ are operations, and they are defined below.

$$\begin{aligned} \lambda \ \widehat{=}\ & \text{PRE } p = True \\ & \text{THEN } p := q\,;\ q := False\,;\ r := False \\ & \text{END} \end{aligned} \tag{10}$$

$$\begin{aligned} \delta \ \widehat{=}\ & \text{PRE } (p \wedge r) = False \wedge (p \vee r) = True \\ & \text{THEN } p := True\,;\ q := True\,;\ r := True \\ & \text{END} \end{aligned} \tag{11}$$

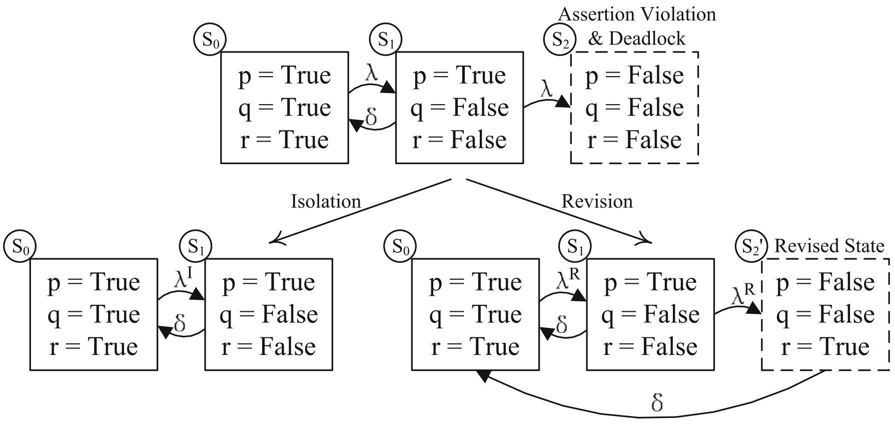

Fig. 3 An example of the *Isolation* and *Revision* method

As S_2 is a faulty state that violates the assertion and is a deadlock state, $S_1 \overset{\lambda}{\rightarrow} S_2$ is considered a faulty state transition. If *Isolation* is applied to this state transition, λ will be changed to

$$\begin{aligned} \lambda^I \ \widehat{=}\ & \text{PRE } \neg(p = True \wedge q = False \wedge r = False) \\ & \quad \wedge p = True \\ & \text{THEN } p := q;\ q := False;\ r := False \\ & \text{END} \end{aligned} \tag{12}$$

and S_2 will be removed. If *Revision* is applied to this state transition, and Δ_{max} is set to 4, the four constraints will be

- $IC : p \in BOOL \wedge q \in BOOL \wedge r \in BOOL$
- $AC : (p \vee q \vee r) = True$
- $LC : p = True \vee (p \wedge r) = False \wedge (p \vee r) = True$
- $DC : dist_{abs}([False, False, False], [p, q, r]) \leq 4$

As $p = False \wedge q = False \wedge r = True$ is a solution under these constraints, λ can be changed to

$$\begin{aligned} \lambda^R \ \widehat{=}\ & \text{PRE } p = True \\ & \text{THEN } p := q;\ q := False;\ r := False; \\ & \quad \text{IF } p = False \wedge q = False \wedge r = False \\ & \quad \text{THEN } p := False;\ q := False;\ r := True \\ & \quad \text{END} \\ & \text{END} \end{aligned} \tag{13}$$

and S_2 can be replaced with the revision state S_2', which satisfies both the invariant and the assertion and is not a deadlock state. S_2' is not a deadlock state because it has a new outgoing transition $S_2' \xrightarrow{\delta} S_0$. According to Eq. (11), the transition is derived because S_2' satisfies the pre-condition of δ, and S_0 satisfies the post-condition of δ. Note that S_2' is just a special solution under the constraint $IC \wedge AC \wedge LC \wedge DC$. Thus, Eq. (13) is a special case of revision, and there are alternative revision repairs for λ. For example, $p = True \wedge q = False \wedge r = False$ and $p = True \wedge q = True \wedge r = True$ are solutions different from S_2' and can be used to construct revision repairs for λ as well.

5 Repair ranking using machine learning

The isolation and revision method usually suggests many repairs, and only one repair should be selected according to the ranking result. Although the distance function can be used to rank revision repairs, different repairs may have the same distance value. Moreover, this function is not able to estimate the distance value of an isolation repair. Thus, a general-purpose function for estimating the quality of repair is needed. This section introduces the quality estimation function with the machine learning-based tendency models (Cai et al. 2018). It provides a general criterion for the ranking of both isolation and revision repairs.

5.1 Quality estimation

Definition 8 (*Tendency model*) Suppose that $\mathcal{M}_0$ is the original abstract machine. A tendency model of $\mathcal{M}_0$ is a function $\mathcal{W}(p, \alpha, q)$ that reflects the likelihood that a transition $p \xrightarrow{\alpha} q$ is valid in $\mathcal{M}_0$. The input of this function includes two states p and q and an operation α, and the output of this function is a real number. This function must satisfy:

$$0 \leq \mathcal{W}(p, \alpha, q) \leq 1 \tag{14}$$

The tendency model should reflect the likelihood that the given transition $p \xrightarrow{\alpha} q$ is a valid one in $\mathcal{M}_0$. In order to train the tendency model, training data are required. In most cases, available training data can be extracted from $\mathcal{M}_0$ via the sampling methods described in Sect. 5.2, and the learning algorithms described in Sects. 3.2 and 5.4 can be used to train the tendency models. Based on the tendency model, the quality estimation of a repaired abstract machine with respect to the original one can be computed.

Definition 9 (*Quality estimation*) Suppose that $\mathcal{M}_R$ is the repaired abstract machine. The quality estimation (QE) of $\mathcal{M}_R$ is computed via

$$\mathcal{Q} = \mathcal{Q}_{val} \cdot \mathcal{Q}_{inv} \tag{15}$$

where $\mathcal{Q}_{val}$ is the quality estimation of valid state transitions in $\mathcal{M}_R$, and $\mathcal{Q}_{inv}$ is the quality estimation of invalid state transitions in $\mathcal{M}_R$. They are computed via

$$\mathcal{Q}_{val} = \prod_{p,\alpha,q}^{p \xrightarrow{\alpha} q} \mathcal{W}(p, \alpha, q) \tag{16}$$

and

$$\mathcal{Q}_{inv} = \prod_{p,\alpha,q}^{\neg(p \xrightarrow{\alpha} q)} (1 - \mathcal{W}(p, \alpha, q)) \tag{17}$$

where p and q are any states including all reachable states and all unreachable states, α is an operation, and $\prod_{x}^{P(x)} F(x)$ is the product of all $F(x)$ such that $P(x)$ is $True$.

The quality estimation consists of two parts, $\mathcal{Q}_{val}$ and $\mathcal{Q}_{inv}$. As a transition $p \xrightarrow{\alpha} q$ is either valid or invalid in $\mathcal{M}_R$, the transition influences either $\mathcal{Q}_{val}$ or $\mathcal{Q}_{inv}$. If the transition is valid, then it influences $\mathcal{Q}_{val}$ by the value of $\mathcal{W}(p, \alpha, q)$. Otherwise, it influences $\mathcal{Q}_{inv}$ by the value of $1 - \mathcal{W}(p, \alpha, q)$. The meaning of the influence is that if the tendency model indicates that $\mathcal{W}(p, \alpha, q)$ is high, then $p \xrightarrow{\alpha} q$ is likely to be a valid transition in $\mathcal{M}_0$. If $p \xrightarrow{\alpha} q$ is a valid transition in $\mathcal{M}_R$, then $\mathcal{M}_R$ is similar to $\mathcal{M}_0$ with respect to $p \xrightarrow{\alpha} q$. Consequently, $p \xrightarrow{\alpha} q$ contributes a high value to $\mathcal{Q}_{val}$. Moreover, if $p \xrightarrow{\alpha} q$ is invalid in $\mathcal{M}_R$, then $\mathcal{M}_R$ is dissimilar to $\mathcal{M}_0$ with respect to $p \xrightarrow{\alpha} q$. As $\mathcal{W}(p, \alpha, q)$ is high, $1 - \mathcal{W}(p, \alpha, q)$ is low. Consequently, $p \xrightarrow{\alpha} q$ contributes a low value to $\mathcal{Q}_{inv}$. On the contrary, if the tendency model indicates that $\mathcal{W}(p, \alpha, q)$ is low, then $p \xrightarrow{\alpha} q$ is unlikely to be a valid transition in $\mathcal{M}_0$. If $p \xrightarrow{\alpha} q$ is a valid transition in $\mathcal{M}_R$, then $\mathcal{M}_R$ is dissimilar to $\mathcal{M}_0$ with respect to $p \xrightarrow{\alpha} q$. Consequently, $p \xrightarrow{\alpha} q$ contributes a low value to $\mathcal{Q}_{val}$. Moreover, if $p \xrightarrow{\alpha} q$ is invalid in $\mathcal{M}_R$, then $\mathcal{M}_R$ is similar to $\mathcal{M}_0$ with respect to $p \xrightarrow{\alpha} q$. As $\mathcal{W}(p, \alpha, q)$ is low, $1 - \mathcal{W}(p, \alpha, q)$ is high. Consequently, $p \xrightarrow{\alpha} q$ contributes a high value to $\mathcal{Q}_{inv}$. The above discussion indicates that if $\mathcal{M}_R$ is similar to $\mathcal{M}_0$ with respect to $p \xrightarrow{\alpha} q$, then $p \xrightarrow{\alpha} q$ contributes a high value to $\mathcal{Q}$, and if $\mathcal{M}_R$ is dissimilar to $\mathcal{M}_0$ with respect to $p \xrightarrow{\alpha} q$, then $p \xrightarrow{\alpha} q$ contributes a low value to $\mathcal{Q}$. Thus, $\mathcal{Q}$ is able to indicate the similarity between $\mathcal{M}_R$ and $\mathcal{M}_0$. Based on the above quality estimation function, a general criterion of repair ranking can be defined as follows.

Definition 10 (*General repair ranking*) Suggested repairs can be ranked by quality estimation. Suppose that the original abstract machine is $\mathcal{M}_0$, and its quality estimation is $\mathcal{Q}_0$. The ranking result is of the form

$$(\mathcal{R}_1, \mathcal{M}_1, \mathcal{Q}_1),\ (\mathcal{R}_2, \mathcal{M}_2, \mathcal{Q}_2),\ \ldots,\ (\mathcal{R}_N, \mathcal{M}_N, \mathcal{Q}_N) \tag{18}$$

where each $(\mathcal{R}_i, \mathcal{M}_i, \mathcal{Q}_i)$ $(i = 1, 2, \ldots, N)$ is a tuple including a repair $\mathcal{R}_i$, a repaired machine $\mathcal{M}_i$ and its quality estimation $\mathcal{Q}_i$. The tuples are sorted in descending order of $\mathcal{Q}_i$.

The general ranking method requires that the quality estimation of each repaired machine is computed after applying the repairs. The model checker needs to compute state graphs for all repaired machines. If there are a huge number of possible repairs, it will be unrealistic to finish the computation. Fortunately, a simplified ranking method can be used to ease the computation.

Definition 11 (*Simplified repair ranking with SQE*) Isolation and revision repairs can be ranked by the tendency model. Suppose that the faulty state transition is $s_0 \xrightarrow{\psi} s_f$, and $\mathcal{W}(p, \alpha, q)$ is a tendency model. The ranking result is of the form

$$(\mathcal{R}_1, \widehat{\mathcal{Q}}_1),\ (\mathcal{R}_2, \widehat{\mathcal{Q}}_2),\ \ldots,\ (\mathcal{R}_N, \widehat{\mathcal{Q}}_N) \tag{19}$$

where each $(\mathcal{R}_i, \widehat{\mathcal{Q}}_i)$ $(i = 1, 2, \ldots, N)$ is a tuple including a repair $\mathcal{R}_i$ and its selective quality estimation (SQE) $\widehat{\mathcal{Q}}_i$. The tuples are sorted in descending order of SQE. The SQE of an isolation repair is:

$$\widehat{\mathcal{Q}}_{ISO} = 0.5 \tag{20}$$

The SQE of a revision repair is:

$$\widehat{\mathcal{Q}}_{REV} = \mathcal{W}(s_0, \psi, s_r) \tag{21}$$

where s_r is a revision of s_f.

The results of simplified repair ranking are equivalent to those of general repair ranking. This statement is supported by the following two theorems.

Theorem 1 (The Order of Two Revision Repairs) *Suppose that $\mathcal{M}_0$ is an abstract machine, its quality estimation is $\mathcal{Q}_0$, and $s_0 \xrightarrow{\psi} s_f$ is a faulty state transition. $\mathcal{R}_A$ and $\mathcal{R}_B$ are two revision repairs that change the faulty state transition to $s_0 \xrightarrow{\psi} s_A$ and $s_0 \xrightarrow{\psi} s_B$ respectively. The quality estimations of the two repairs are $\mathcal{Q}_A$ and $\mathcal{Q}_B$ respectively, and their SQEs are $\widehat{\mathcal{Q}}_A$ and $\widehat{\mathcal{Q}}_B$ respectively. In this case, the following statement holds.*

$$\mathcal{Q}_A \geq \mathcal{Q}_B \longleftrightarrow \widehat{\mathcal{Q}}_A \geq \widehat{\mathcal{Q}}_B \tag{22}$$

Proof Equation (22) is a proof goal. By Eq. (15), we have

$$\mathcal{Q}_A = \mathcal{Q}_0 \cdot \frac{1 - \mathcal{W}(s_0, \psi, s_f)}{\mathcal{W}(s_0, \psi, s_f)} \cdot \frac{\mathcal{W}(s_0, \psi, s_A)}{1 - \mathcal{W}(s_0, \psi, s_A)} \tag{23}$$

and

$$\mathcal{Q}_B = \mathcal{Q}_0 \cdot \frac{1 - \mathcal{W}(s_0, \psi, s_f)}{\mathcal{W}(s_0, \psi, s_f)} \cdot \frac{\mathcal{W}(s_0, \psi, s_B)}{1 - \mathcal{W}(s_0, \psi, s_B)} \tag{24}$$

By Eq. (21), we have

$$\widehat{\mathcal{Q}}_A = \mathcal{W}(s_0, \psi, s_A) \tag{25}$$

and

$$\widehat{\mathcal{Q}}_B = \mathcal{W}(s_0, \psi, s_B) \tag{26}$$

Therefore, the proof goal becomes

$$\begin{aligned} &\mathcal{Q}_0 \cdot \frac{1-\mathcal{W}(s_0,\psi,s_f)}{\mathcal{W}(s_0,\psi,s_f)} \cdot \frac{\mathcal{W}(s_0,\psi,s_A)}{1-\mathcal{W}(s_0,\psi,s_A)} \\ &\quad \geq \mathcal{Q}_0 \cdot \frac{1-\mathcal{W}(s_0,\psi,s_f)}{\mathcal{W}(s_0,\psi,s_f)} \cdot \frac{\mathcal{W}(s_0,\psi,s_B)}{1-\mathcal{W}(s_0,\psi,s_B)} \\ &\quad \longleftrightarrow \mathcal{W}(s_0,\psi,s_A) \geq \mathcal{W}(s_0,\psi,s_B) \end{aligned} \tag{27}$$

The above goal can be directly proved by Eq. (14). □

Theorem 2 (The Order of an Isolation Repair and a Revision Repair) *Suppose that $\mathcal{M}_0$ is an abstract machine, its quality estimation is $\mathcal{Q}_0$, and $s_0 \xrightarrow{\psi} s_f$ is a faulty state transition. $\mathcal{R}_{Iso}$ is an isolation repair that removes the faulty state transition, and $\mathcal{R}_{Rev}$ is a revision repair that changes faulty state transition to $s_0 \xrightarrow{\psi} s_{Rev}$. The quality estimations of the two repairs are $\mathcal{Q}_{Iso}$ and $\mathcal{Q}_{Rev}$ respectively, and their SQEs are $\widehat{\mathcal{Q}}_{Iso}$ and $\widehat{\mathcal{Q}}_{Rev}$ respectively. In this case, the following two statements hold.*

$$\mathcal{Q}_{Iso} \geq \mathcal{Q}_{Rev} \longleftrightarrow \widehat{\mathcal{Q}}_{Iso} \geq \widehat{\mathcal{Q}}_{Rev} \tag{28}$$

$$\mathcal{Q}_{Iso} \leq \mathcal{Q}_{Rev} \longleftrightarrow \widehat{\mathcal{Q}}_{Iso} \leq \widehat{\mathcal{Q}}_{Rev} \tag{29}$$

Proof Equations (28) and (29) are proof goals. By Eq. (15), we have

$$\mathcal{Q}_{Iso} = \mathcal{Q}_0 \cdot \frac{1-\mathcal{W}(s_0,\psi,s_f)}{\mathcal{W}(s_0,\psi,s_f)} \tag{30}$$

and

$$\mathcal{Q}_{Rev} = \mathcal{Q}_0 \cdot \frac{1-\mathcal{W}(s_0,\psi,s_f)}{\mathcal{W}(s_0,\psi,s_f)} \cdot \frac{\mathcal{W}(s_0,\psi,s_{Rev})}{1-\mathcal{W}(s_0,\psi,s_{Rev})} \tag{31}$$

By Eq. (20), we have

$$\widehat{\mathcal{Q}}_{Iso} = 0.5 \tag{32}$$

By Eq. (21), we have

$$\widehat{\mathcal{Q}}_{Rev} = \mathcal{W}(s_0,\psi,s_{Rev}) \tag{33}$$

Therefore, the proof goal in Eq. (28) becomes

$$\begin{aligned} &\mathcal{Q}_0 \cdot \frac{1-\mathcal{W}(s_0,\psi,s_f)}{\mathcal{W}(s_0,\psi,s_f)} \geq \mathcal{Q}_0 \cdot \frac{1-\mathcal{W}(s_0,\psi,s_f)}{\mathcal{W}(s_0,\psi,s_f)} \cdot \frac{\mathcal{W}(s_0,\psi,s_{Rev})}{1-\mathcal{W}(s_0,\psi,s_{Rev})} \\ &\longleftrightarrow 0.5 \geq \mathcal{W}(s_0,\psi,s_{Rev}) \end{aligned} \tag{34}$$

This goal can be directly proved by Eq. (14). Similarly, the goal in Eq. (29) can be proved. □

As the statements in Eqs. (22), (28) and (29) are true, results of simplified repair ranking are equivalent to those of general repair ranking. Thus, the SQE can be used to rank repairs instead of the original QE. Another reason why the QE is not used is that computing the QE of a large model is practically impossible. To compute the QE, pairs of pre- and post-states need to be enumerated. As the states can be all reachable states and all unreachable states, combinatorial explosions will occur if the model has a large state space. Nevertheless, we are able to calculate the change of QE by calculating $\mathcal{Q}_R/\mathcal{Q}_0$, where $\mathcal{Q}_0$ is the QE of the original model $\mathcal{M}_0$, and $\mathcal{Q}_R$ is the QE of the repaired model $\mathcal{M}_R$. We can minimise $\mathcal{Q}_R/\mathcal{Q}_0$ by selecting a repair with a highest SQE. This is the reason why the SQE is used instead of the QE.

5.2 Transition sampling

Major sampling is the process of extracting examples of state transitions from the state graph of an abstract machine. Given an abstract machine $\mathcal{M}$, major sampling follows two steps. Firstly, the model checker is asked to generate a state graph $\mathcal{G}$ for $\mathcal{M}$. When developing the state graph, the model checker is asked to skip faulty states. If a state violates invariants, assertions or the requirement of liveness, it will be added into the state graph, but no successive operations will be applied to the state. Secondly, positive samples and negative samples are extracted from $\mathcal{G}$. The positive samples and the negative samples are defined as follows.

Definition 12 (*Positive sample*) Given a state graph $\mathcal{G}$, a positive sample is of the form $p \overset{\alpha}{\to} q$ such that p and q are states in $\mathcal{G}$, and α is an operation transitioning p to q.

Definition 13 (*Negative sample*) Given a state graph $\mathcal{G}$, a negative sample is of the form $\neg(p \overset{\alpha}{\to} q)$ such that p and q are states in $\mathcal{G}$, and α is an operation that cannot transition p to q.

Additional sampling is a process of extracting hidden state transitions that do not occur in the state graph of the abstract machine. It can be used to increase the number of samples. Given an abstract machine $\mathcal{M}$, additional sampling follows three steps. Firstly, states satisfying invariants and assertions of $\mathcal{M}$ are randomly generated. These states are considered to be initial states. Then the model checker is asked to derive state transitions from these states. When deriving state transitions, the model checker is asked to skip states that violate invariants, assertions or the requirement of liveness, and results are collected into a state graph $\mathcal{G}_{\mathcal{A}}$. Finally, positive examples and negative samples are extracted from $\mathcal{G}_{\mathcal{A}}$. This process is used to sample a number of unreachable transitions in $\mathcal{M}$. Although the unreachable transitions are currently infeasible, they may be feasible in the future, because a revision repair may turn an unreachable state to a reachable state. Thus, we use a number of the unreachable transitions to train the tendency models. In fact, if reachable transitions in $\mathcal{M}$ are sufficient for the training of the tendency models, the unreachable transitions are not necessary. However, if the reachable transitions are insufficient for the training, the resulting tendency models will have poor performance, because the machine learning models such as residual networks and random forests usually require a significant number of training data to achieve a high accuracy. In this case, the unreachable transitions can provide more training data for the tendency models.

5.3 Encodings

In order to use machine learning models to process samples, states of the samples need to be vectorised. Suppose that $p \overset{\alpha}{\rightarrow} q$ is a state transition. Both p and q can be vectorised by encoding their variables using the following methods (Turian et al. 2010).

- *Integer* Before encoding integers, all integers occurring in the samples are collected into base vector $(b_1, \ldots, b_N)$, where each b_i $(i = 1, 2, \ldots, N)$ is an integer. Then an integer x can be encoded as a vector $(x_1, \ldots, x_N)$, where each x_i $(i = 1, 2, \ldots, N)$ is either 1 or 0. If $x \leq b_i$, then x_i is 1. Otherwise, x_i is 0.
- *Distinct element* A distinct element $d \in \tau$ (d is its name, and τ is its type) is encoded as a vector $(e_1, \ldots, e_N)$, where τ is an enumerated set $\{t_1, t_2, \ldots, t_N\}$, and e_i $(i = 1, 2, \ldots, N)$ is 1 when d is the ith element of τ, or it is 0 in other cases.
- *First-order set* A first-order set $s \subseteq \tau$ (s is its name, and the power set of τ is its type) is encoded as a vector $(e_1, \ldots, e_N)$, where τ is an enumerated set $\{t_1, t_2, \ldots, t_N\}$, and each e_i $(i = 1, 2, \ldots, N)$ is either 1 or 0. If t_i $(i = 1, 2, \ldots, N)$ occurs in s, then e_i will be 1. Otherwise, it will be 0.
- *Boolean value* A Boolean value is encoded as $(1, 0)$ when it is true, or $(0, 1)$ when it is false.

5.4 Tendency models

The samples are encoded and used to train a tendency model. When training the tendency model, the encodings of the samples are considered to be input signals. Target signals of the positive samples are 1.00, and those of the negative samples are 0.00. The purpose of training is to make the tendency model be able to reflect the likelihood that $p \overset{\alpha}{\rightarrow} q$ is a valid transition in the original abstract machine $\mathcal{M}_0$. The tendency model is a supervised learning model.

The tendency model can be a random forest that has a number of CARTs. The CARTs are trained using variance-based node impurity (see Sect. 3.2.1) (Breiman et al. 1984; Loh 2011). When training each CART, a certain number of training data are randomly selected from all samples. The encodings of these data are considered to be attributes for producing branches on the CART. Moreover, the CART is pruned using a randomly selected limit of node impurity. When producing a branch, the node impurity is decreasing. If the decreased node impurity is smaller than the limit, the training on this branch will stop. After training the whole CART, it can be used to predict SQE values. Given a transition $p \overset{\alpha}{\rightarrow} q$, the CART can find one and only one leaf node that reflects the attributes of the transition. The leaf node will output a predicted SQE value. As there are a number of CARTs in the random forest, the average of their outputs is the SQE value of the transition.

Moreover, the tendency model can be a feed-forward neural network, such as a logistic model and a residual network (see Sect. 3.2.2) (Cox 1958; He et al. 2016). Figure 4 shows the architecture of the feed-forward neural network as a tendency model. The model is able to take a transition as the input signal and output the SQE

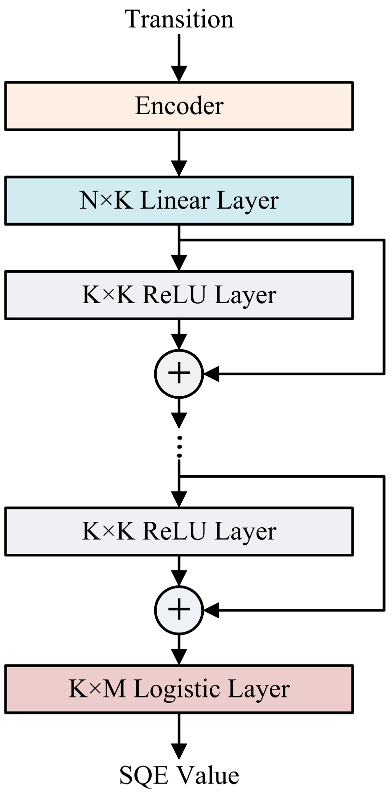

Fig. 4 The feed-forward neural network as a tendency model

value of the transition. Firstly, the transition is encoded as an N-dimensional vector via the encoder. Then the N-dimensional vector is expanded to a K-dimensional vector via the K $\times$ K linear layer, where K is a predefined dimensionality. The equation[1] of the linear layer is simply $\boldsymbol{y} = \boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b}$. Next, the K-dimensional vector is propagated through a number of residual building blocks that each one consists of a K $\times$ ReLU Layer and a shortcut connection (denoted by "+"). The equations of the residual building blocks are $\boldsymbol{y} = relu(\boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b}) + \boldsymbol{x}$ and $relu(u) = max(0, u)$ (Glorot et al. 2011). Finally, the propagated K-dimensional vector is converted to an M-dimensional vector via the K $\times$ M logistic layer, where M is the number of operations in the original abstract machine. The equations of the logistic layer are $\boldsymbol{y} = logistic(\boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b})$ and $logistic(u) = 1/(1 + e^{-u})$. There are M elements in the output vector. Each element is the SQE value of the corresponding operation. The above model is a *residual network* for computing SQE values. Particularly, if the linear layer and the residual building blocks are removed, and the last layer is an N $\times$ M logistic layer, then the model becomes a *logistic model*. Stochastic gradient descent is used to train the residual network and the logistic model (Bottou 2012).

[1] In the following equations, $\boldsymbol{x}$ is an input vector, $\boldsymbol{y}$ is an output vector, $\boldsymbol{W}$ is a weight matrix, and $\boldsymbol{b}$ is a bias vector.

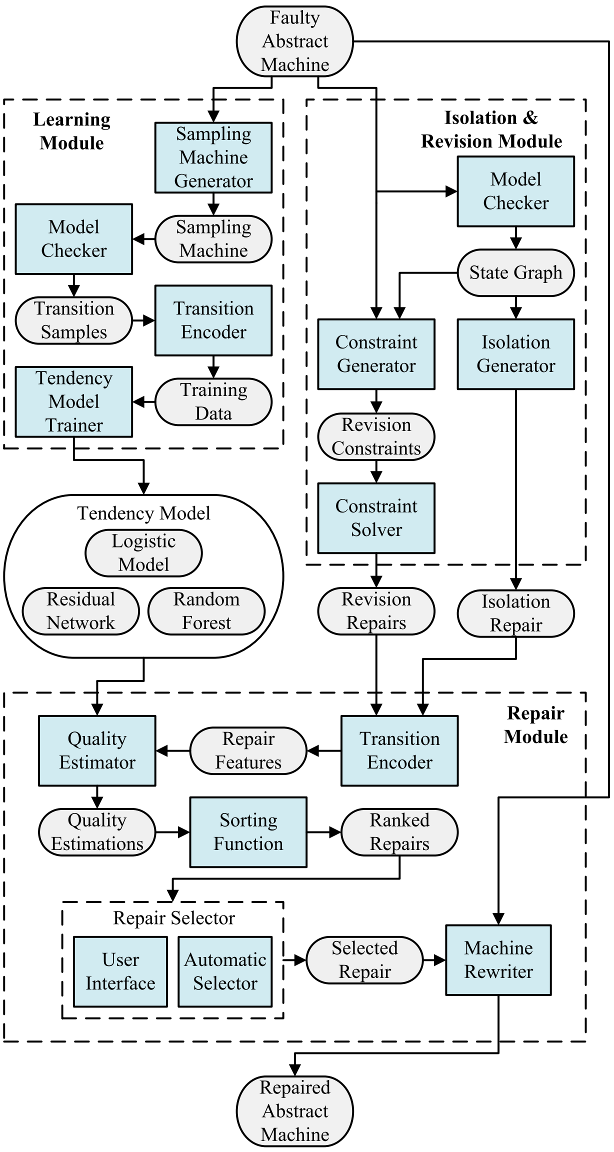

Fig. 5 The architecture of the B-repair system

6 The implementation of B-repair

Sections 4 and 5 has provided the theory of B model repair. Based on the theory, B-repair has been implemented as a tool that consists of three modules, i.e., the learning module, the isolation & revision module and the repair module, as shown in Fig. 5.

6.1 The learning module

The input of the learning module is a faulty abstract machine, and the output is a trained tendency model, such as a logistic model, a residual network and a random forest. The algorithms of logistic models and residual networks are described in Sects. 3.2.2 and 5.4, and the algorithms of random forests are described in Sects. 3.2.1 and 5.4. Note that only one of the three tendency models is outputted according to users' choice. This module consists of the four submodules listed below.

- *Sampling machine generator* It is used to generate sampling machines that allow the model checker to derive state transitions within the scope of major sampling or additional sampling.
- *Model checker* The ProB model checker (Leuschel and Butler 2008) can compute state graphs for the sampling machines and output available state transitions in the state graph. These state transitions are considered to be transition samples.
- *Transition encoder* It is the implementation of the encoding functions described in Sect. 5.3. It is used to vectorise the transitions samples.
- *Tendency model trainer* It includes a random forest trainer and a neural network optimiser. They are the implementation of the training algorithms described in Sects. 3.2.1, 3.2.2 and 5.4. The trainer is able to train three kinds of tendency models, i.e., random forests of CARTs, logistic models and residual networks.

6.2 The isolation & revision module

The input of the isolation & revision module is the faulty abstract machine, and the output includes an isolation repair and revision repairs. This module consists of the four submodules listed below.

- *Model checker* The ProB model checker (Leuschel and Butler 2008) can compute a state graph for an abstract machine and check invariants, assertions and liveness of states. If a faulty state is detected, it will be annotated.
- *Isolation generator* It generates an isolation repair for the annotated faulty state using Eqs. (5) and (6).
- *Constraint generator* It generates revision constraints $IC \wedge AC \wedge LC \wedge DC$ (see Sect. 4.3) for the annotated faulty state. These constraints are of the form of B specifications that can be directly analysed using the constraint solver.
- *Constraint solver* The SMT solver of ProB (Krings and Leuschel 2016) can compute solutions under the revision constraints. Then the solutions are converted to revision repairs using Eqs. (7) and (8).

6.3 The repair module

The input of the repair module includes the trained tendency model, the faulty abstract machine, the isolation repair and the revision repairs. The trained tendency model is a logistic model, a residual network or a random forest. Note that only one of the three tendency models is used during each repair process. Users need to choose an appropriate tendency model before starting the repair process. The output is a repaired abstract machine. This module consists of the five submodules listed below.

– *Repair encoder* It can vectorise the repairs using the encoding functions described in Sect. 5.3. Its input is the isolation repair and the revision repairs, and its output is a list of repair features.
– *Quality estimator* It computes the SQEs of repairs using the tendency model and Eqs. (20) and (21). Its input includes the list of repair features and the trained tendency model, and its output is a list of SQEs.
– *Sorting function* It sorts the repairs according to their SQEs. The input of this function is the list of SQEs, and the output is a list of repair rankings.
– *Repair selector* It includes a manual selector and an automatic selector. The manual selector enables users to view the detected faulty state, the repairs and their SQEs, so that the users can select a suitable repair. The automatic selector always selects the repair with the highest ranking.
– *Machine rewriter* It can automatically apply a selected repair to an appropriate position of the faulty abstract machine, and the result is a repaired abstract machine.

The three modules, including the learning module, the isolation & revision module and the repair module, construct the whole B-repair system. In the following two sections, the implemented B-repair system is evaluated via a case study and a number of experiments.

7 Case study

In this section, an “Accommodation Management System” example illustrates how B-repair repairs a faulty abstract machine. According to Sect. 6, the tendency model can be a logistic model, a residual network or a random forest of CARTs. As the interpretability of CARTs is higher than that of logistic models and residual networks, we use CARTs to explain how the tendency model works for B-repair. The example demonstrates the following.

– The tendency model can learn the post-condition of an operation.
– The tendency model can learn the relation between the pre- and post-conditions of an operation.
– B-repair can repair the faulty abstract machine using $Isolation$, $Revision$ and the tendency model.

7.1 The original abstract machine and the state transitions

The accommodation management system is modelled as the following abstract machine.

```
MACHINE
  Accommodation_Management_System
SETS
  S = {TV, Fridge, SONY_TV, SONY_PS3}
VARIABLES
  Attr
INVARIANT
  Attr : POW(S) &
  not(TV : Attr & SONY_TV : Attr)
INITIALISATION
  Attr := { }
OPERATIONS
  Add_Fridge =
    PRE not(Fridge : Attr)
    THEN Attr := Attr \/ {Fridge}
    END ;
  Add_SONY =
    PRE not(SONY_TV : Attr) & not(SONY_PS3 : Attr)
    THEN Attr := Attr \/ {SONY_TV,SONY_PS3}
    END ;
  Add_TV =
    PRE not(TV : Attr)
    THEN Attr := Attr \/ {TV}
    END ;
  Remove_Fridge =
    PRE Fridge : Attr
    THEN Attr := Attr − {Fridge}
    END ;
  Remove_SONY =
    PRE SONY_TV : Attr & SONY_PS3 : Attr
    THEN Attr := Attr − {SONY_TV,SONY_PS3}
    END ;
  Remove_TV =
    PRE TV : Attr
    THEN Attr := Attr − {TV}
    END
END
```

The abstract machine has a variable $Attr$ that is a set of attributes owned by a room, and the attributes are of appliances such as a Fridge, a normal TV and a suite of a SONY Play Station 3 (PS3) and a SONY TV. Moreover, it has three operations, including Add_Fridge, Add_SONY and Add_TV that can add attributes and three operations, including $Remove\_Fridge$, $Remove\_SONY$ and $Remove\_TV$ that can remove attributes. For instance, if the current value of $Attr$ is $\{TV\}$, then Add_Fridge can change it to $\{TV, Fridge\}$, and $Remove\_TV$ can further change it to $\{Fridge\}$. Additionally, the abstract machine has an invariant requiring that the room does not have both the normal TV and the SONY TV at the same time.

Table 1 shows state transitions that can be derived using the operations of the abstract machine. Each operation is of the form $s_{pre} \xrightarrow{\alpha} s_{post}$, where α is an operation, s_{pre} is a state satisfying the pre-condition of α, and s_{post} is a state satisfying the post-condition of α.

Table 1 State Transitions of the “accommodation management system” abstract machine

State Transition
$\{\} \xrightarrow{Add\_Fridge} \{Fridge\}$
$\{SONY\_PS3\} \xrightarrow{Add\_Fridge} \{Fridge, SONY\_PS3\}$
$\{\} \xrightarrow{Add\_SONY} \{SONY\_TV, SONY\_PS3\}$
$\{Fridge\} \xrightarrow{Add\_SONY} \{Fridge, SONY\_TV, SONY\_PS3\}$
$\{\} \xrightarrow{Add\_TV} \{TV\}$
$\{Fridge\} \xrightarrow{Add\_TV} \{TV, Fridge\}$
$\{Fridge\} \xrightarrow{Remove\_Fridge} \{\}$
$\{Fridge, SONY\_PS3\} \xrightarrow{Remove\_Fridge} \{SONY\_PS3\}$
$\{Fridge, SONY\_TV, SONY\_PS3\} \xrightarrow{Remove\_SONY} \{Fridge\}$
$\{SONY\_TV, SONY\_PS3\} \xrightarrow{Remove\_SONY} \{\}$
$\{TV\} \xrightarrow{Remove\_TV} \{\}$
$\{TV, Fridge\} \xrightarrow{Remove\_TV} \{Fridge\}$
...
(totally 30 state transitions)

Table 2 State transitions of Add_SONY

s_{pre}	s_{post}	Feature
$\{\}$	$\{SONY\_PS3, SONY\_TV\}$	$[-1, -1, -1, -1, -1, 1, 1, -1]$
$\{Fridge\}$	$\{Fridge, SONY\_PS3, SONY\_TV\}$	$[1, -1, -1, -1, 1, 1, 1, -1]$
$\{TV\}$	$\{SONY\_PS3, SONY\_TV, TV\}$	$[-1, -1, -1, 1, -1, 1, 1, 1]$
$\{TV, Fridge\}$	$\{Fridge, SONY\_PS3, SONY\_TV, TV\}$	$[1, -1, -1, 1, 1, 1, 1, 1]$

7.2 Training a decision tree

This section uses the “Add_SONY” operation and a decision tree as an example to explain how to train a tendency model and what can be learnt by the tendency model.

Table 2 shows state transitions available for Add_SONY. Each operation is of the form $s_{pre} \xrightarrow{Add\_SONY} s_{post}$, where s_{pre} is a state satisfying the pre-condition of Add_SONY, and s_{post} is a state satisfying the post-condition of Add_SONY. Each state consists of only one variable $Attr$. The elements of the set $Attr$ and their position annotations are $Fridge/0$, $SONY\_PS3/1$, $SONY\_TV/2$ and $TV/3$, where x/p means that x is an element, and p is its position annotation. Using these position annotations, each state can be encoded as a four-dimensional vector (v_0, v_1, v_2, v_3). If $x \in Attr$, then $v_p = 1$. Otherwise, $v_p = -1$. Both s_{pre} and s_{post} can be vectorised, and they can be concatenated as a state feature. For instance, $\{Fridge\}$

Table 3 Training examples

Feature (V)	Label
Positive training examples	
$[-1, -1, -1, -1, -1, 1, 1, -1]$	Yes
$[1, -1, -1, -1, 1, 1, 1, -1]$	Yes
$[-1, -1, -1, 1, -1, 1, 1, 1]$	Yes
$[1, -1, -1, 1, 1, 1, 1, 1]$	Yes
Negative training examples	
$[1, -1, -1, -1, -1, 1, 1, -1]$	No
$[1, -1, -1, 1, -1, 1, 1, 1]$	No
$[1, -1, -1, -1, 1, 1, 1, 1]$	No
$[1, -1, -1, 1, 1, 1, 1, -1]$	No
$[-1, -1, -1, -1, -1, 1, -1, -1]$	No
$[-1, -1, -1, -1, -1, -1, 1, -1]$	No
$[1, -1, -1, -1, 1, 1, -1, -1]$	No
$[1, -1, -1, -1, 1, -1, 1, -1]$	No

can be encoded as $(1, -1, -1, -1)$, and $\{Fridge, SONY\_TV, SONY\_PS3\}$ can be encoded as $(1, 1, 1, -1)$, so that the feature of $\{Fridge\} \xrightarrow{Add\_SONY} \{Fridge, SONY\_TV, SONY\_PS3\}$ is $[1, -1, -1, -1, 1, 1, 1, -1]$.

Table 3 shows the training examples that are used to train the decision tree. The state transitions in Table 2 are used to produce positive training examples. Each example is of the form (U, Yes), where U is the feature and is of the form $[u_0, u_1, \ldots, u_7]$, and Yes means that it is a positive training example. Moreover, negative training examples are produced by generating invalid state transition. Invalid state transitions are those that cannot be derived by Add_SONY. For instance, $\{SONY\_PS3\} \nrightarrow \{Fridge\}$ is an invalid state transition. Each negative training example is of the form (U, No), where U is the feature and is of the form $[u_0, u_1, \ldots, u_7]$, and No means that it is a negative training example.

Figure 6 shows how the decision tree learns the post-condition of the Add_SONY operation. We use $Attr_{Pre}$ and $Attr_{Post}$ to denote the attribute set $Attr$ before and after the execution of Add_SONY respectively. The post-condition of Add_SONY is $Attr_{Post} = Attr_{Pre} \cup \{SONY\_TV, SONY\_PS3\}$. It implies that $SONY\_TV$ and $SONY\_PS3$ should be members of $Attr_{Post}$. The first branch of the decision tree learns to split the examples by $SONY\_PS3$. If $SONY\_PS3 \in Attr_{Post}$ is true, the examples will be put into the left node. Otherwise, they will be put into the right node. As a result, two negative examples are in the right node, and the remaining examples are in the left node. Similarly, the second branch of the decision tree learns to split the examples by $SONY\_TV$. If $SONY\_TV \in Attr_{Post}$ is true, the examples will be put into the left node. Otherwise, they will be put into the right node. As a result, two negative examples are in the right node, and the remaining examples are in the left node. Thus, the resulting tree is able to find examples that satisfy $SONY\_PS3 \in Attr_{Post} \wedge SONY\_TV \in Attr_{Post}$. As this condition can be implied

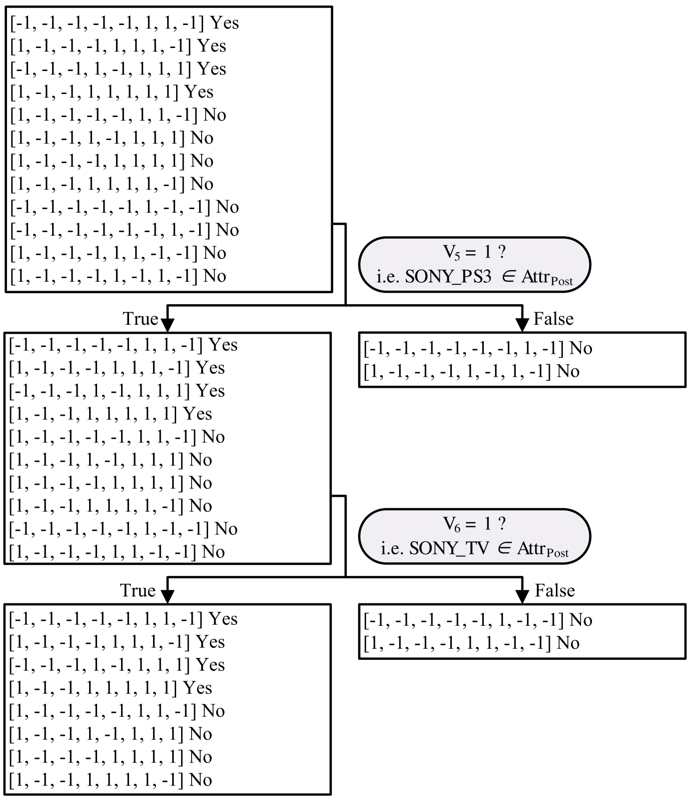
[-1, -1, -1, -1, -1, 1, 1, -1] Yes
[1, -1, -1, -1, 1, 1, 1, -1] Yes
[-1, -1, -1, 1, -1, 1, 1, 1] Yes
[1, -1, -1, 1, 1, 1, 1, 1] Yes
[1, -1, -1, -1, -1, 1, 1, -1] No
[1, -1, -1, 1, -1, 1, 1, 1] No
[1, -1, -1, -1, 1, 1, 1, 1] No
[1, -1, -1, 1, 1, 1, 1, -1] No
[-1, -1, -1, -1, -1, 1, -1, -1] No
[-1, -1, -1, -1, -1, -1, 1, -1] No
[1, -1, -1, -1, 1, 1, -1, -1] No
[1, -1, -1, -1, 1, -1, 1, -1] No
V5 = 1 ? i.e. SONY_PS3 ∈ Attr_Post
True
False
[-1, -1, -1, -1, -1, 1, 1, -1] Yes
[1, -1, -1, -1, 1, 1, 1, -1] Yes
[-1, -1, -1, 1, -1, 1, 1, 1] Yes
[1, -1, -1, 1, 1, 1, 1, 1] Yes
[1, -1, -1, -1, -1, 1, 1, -1] No
[1, -1, -1, 1, -1, 1, 1, 1] No
[1, -1, -1, -1, 1, 1, 1, 1] No
[1, -1, -1, 1, 1, 1, 1, -1] No
[-1, -1, -1, -1, -1, 1, -1, -1] No
[1, -1, -1, -1, 1, 1, -1, -1] No
[-1, -1, -1, -1, -1, -1, 1, -1] No
[1, -1, -1, -1, 1, -1, 1, -1] No
V6 = 1 ? i.e. SONY_TV ∈ Attr_Post
True
False
[-1, -1, -1, -1, -1, 1, 1, -1] Yes
[1, -1, -1, -1, 1, 1, 1, -1] Yes
[-1, -1, -1, 1, -1, 1, 1, 1] Yes
[1, -1, -1, 1, 1, 1, 1, 1] Yes
[1, -1, -1, -1, -1, 1, 1, -1] No
[1, -1, -1, 1, -1, 1, 1, 1] No
[1, -1, -1, -1, 1, 1, 1, 1] No
[1, -1, -1, 1, 1, 1, 1, -1] No
[-1, -1, -1, -1, -1, 1, -1, -1] No
[1, -1, -1, -1, 1, 1, -1, -1] No

Fig. 6 Learning the post-condition of Add_SONY

by the post-condition of Add_SONY, it means that the decision tree has partially learnt the post-condition.

Figure 7, which is the subsequence of Fig. 6, shows how the decision tree continues to learn the relation between the pre- and post-conditions of the Add_SONY operation. The pre-condition is $\neg(SONY\_TV \in Attr_{Pre}) \wedge \neg(SONY\_PS3 \in Attr_{Post})$, and the post-condition is $Attr_{Post} = Attr_{Pre} \cup \{SONY\_TV, SONY\_PS3\}$. They imply that the operation does not change the status of $Fridge$. State transitions produced by the operation should satisfy either $Fridge \in Attr_{Pre} \wedge Fridge \in Attr_{Post}$ or $Fridge \notin Attr_{Pre} \wedge Fridge \notin Attr_{Post}$. The first branch of the decision tree learns to split the examples by $Attr_{Pre}$. If $Fridge \in Attr_{Pre}$ is true, the examples will be put into the left node. Otherwise, they will be put into the right node. The second branch of the decision tree learns to split the examples by $Attr_{Post}$. If $Fridge \in Attr_{Post}$ is true, the examples will be put into the left node. Otherwise, they will be put into the right node. Thus, the resulting decision tree is able to find examples that satisfy $Fridge \in Attr_{Pre} \wedge Fridge \in Attr_{Post}$. As mentioned before, this condition can be implied by the pre- and post-conditions of the operation, which means that the decision tree has partially learnt the relation between the pre- and post-conditions.

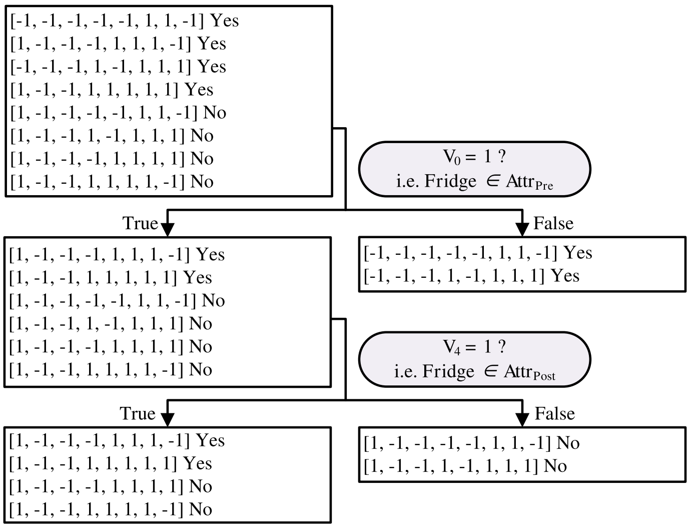
[-1, -1, -1, -1, -1, 1, 1, -1] Yes
[1, -1, -1, -1, 1, 1, 1, -1] Yes
[-1, -1, -1, 1, -1, 1, 1, 1] Yes
[1, -1, -1, 1, 1, 1, 1, 1] Yes
[1, -1, -1, -1, -1, 1, 1, -1] No
[1, -1, -1, 1, -1, 1, 1, 1] No
[1, -1, -1, -1, 1, 1, 1, 1] No
[1, -1, -1, 1, 1, 1, 1, -1] No
V0 = 1 ? i.e. Fridge ∈ AttrPre
True
False
[1, -1, -1, -1, 1, 1, 1, -1] Yes
[1, -1, -1, 1, 1, 1, 1, 1] Yes
[1, -1, -1, -1, -1, 1, 1, -1] No
[1, -1, -1, 1, -1, 1, 1, 1] No
[1, -1, -1, -1, 1, 1, 1, 1] No
[1, -1, -1, 1, 1, 1, 1, -1] No
[-1, -1, -1, -1, -1, 1, 1, -1] Yes
[-1, -1, -1, 1, -1, 1, 1, 1] Yes
V4 = 1 ? i.e. Fridge ∈ AttrPost
True
False
[1, -1, -1, -1, 1, 1, 1, -1] Yes
[1, -1, -1, 1, 1, 1, 1, 1] Yes
[1, -1, -1, -1, 1, 1, 1, 1] No
[1, -1, -1, 1, 1, 1, 1, -1] No
[1, -1, -1, -1, -1, 1, 1, -1] No
[1, -1, -1, 1, -1, 1, 1, 1] No

Fig. 7 Learning the relation between the pre- and post-conditions of Add_SONY

7.3 Ranking suggested repairs using a random forest

A random forest of CARTs can be trained using the training algorithms described in Sect. 3.2.1. Different from the single CART in Sect. 7.2, the random forest is a set of trees with randomly selected node impurities and training data. The advantage of the random forest is that it is more robust than the single CART. When training a single CART, a few features of data may be omitted if the node impurity is too low, or a few features may dominate decision processes if the node impurity is too high. To overcome this phenomenon, many CARTs with randomly selected node impurities are used instead of a single one. Moreover, if all training data are used to train a single CART, it may suffer from overfitting. To ease the overfitting, the CARTs are trained using randomly selected subsets of the data instead of all the data. After training, the random forest can provide SQE values for repair ranking, and each SQE value is computed by averaging outputs of the CARTs.

Table 4 shows suggested repairs for the operation Add_SONY and their SQE values provided by a trained random forest with 256 CARTs. In order to repair Add_SONY, two epochs of repairs were required. During the first epoch, the faulty state transition was $\{TV, Fridge\} \xrightarrow{Add\_SONY} \{TV, Fridge, SONY\_TV, SONY\_PS3\}$. The best repair was a revision repair changing $\{TV, Fridge, SONY\_TV, SONY\_PS3\}$ to $\{Fridge, SONY\_TV, SONY\_PS3\}$. The second best repair was a revision repair removing both TV and $Fridge$. The first one was better than the second one, because Add_SONY should not change the status of $Fridge$. During the second epoch, the faulty state transition was $\{TV\} \xrightarrow{Add\_SONY} \{TV, SONY\_TV, SONY\_PS3\}$. In this case, the best repair was a revision repair removing TV from $\{TV, SONY\_TV, SONY\_PS3\}$.

Table 4 Suggested repairs for Add_SONY

Revision	SQE
Epoch 1—faulty state transition	
$\{TV, Fridge\} \xrightarrow{Add\_SONY} \{TV, Fridge, SONY\_TV, SONY\_PS3\}$	
$\{Fridge, SONY\_TV, SONY\_PS3\}$	0.555
$\{SONY\_TV, SONY\_PS3\}$	0.505
—Borderline of isolation—	0.500
$\{Fridge, SONY\_TV\}$	0.262
$\{SONY\_TV\}$	0.231
$\{TV, Fridge, SONY\_PS3\}$	0.087
$\{TV, SONY\_PS3\}$	0.087
$\{SONY\_PS3\}$	0.075
$\{Fridge, SONY\_PS3\}$	0.067
$\{TV, Fridge\}$	0.012
$\{TV\}$	0.012
$\{Fridge\}$	0.012
$\{\}$	0.012
Epoch 2—faulty state transition	
$\{TV\} \xrightarrow{Add\_SONY} \{TV, SONY\_TV, SONY\_PS3\}$	
$\{SONY\_TV, SONY\_PS3\}$	0.598
—Borderline of isolation—	0.500
$\{Fridge, SONY\_TV, SONY\_PS3\}$	0.497
$\{SONY\_TV\}$	0.274
$\{Fridge, SONY\_TV\}$	0.239
$\{TV, SONY\_PS3\}$	0.106
$\{SONY\_PS3\}$	0.102
$\{TV, Fridge, SONY\_PS3\}$	0.059
$\{Fridge, SONY\_PS3\}$	0.055
$\{\}$	0.028
$\{TV\}$	0.024
$\{TV, Fridge\}$	0.016
$\{Fridge\}$	0.016

Table 5 shows suggested repairs for the operation Add_TV and their SQE values provided by the random forest. In order to repair Add_TV, two more epochs of repairs were required. The results of the third and fourth epochs were similar to those of the first and second epochs. During the third epoch, the best repair was a revision repair removing $SONY\_TV$ from $\{TV, Fridge, SONY\_TV, SONY\_PS3\}$. During the fourth epoch, the best repair was a revision repair removing $SONY\_TV$ from $\{TV, SONY\_TV, SONY\_PS3\}$. Both repairs only changed the status of $SONY\_TV$, but did not influence other attributes.

Table 5 Suggested repairs for Add_TV

Revision	SQE
Epoch 3—faulty state transition	
$\{Fridge, SONY\_TV, SONY\_PS3\} \xrightarrow{Add\_TV} \{TV, Fridge, SONY\_TV, SONY\_PS3\}$	
$\{TV, Fridge, SONY\_PS3\}$	0.519
—Borderline of isolation—	0.500
$\{TV, SONY\_PS3\}$	0.410
$\{TV, Fridge\}$	0.398
$\{TV\}$	0.324
$\{Fridge, SONY\_TV\}$	0.019
$\{SONY\_TV, SONY\_PS3\}$	0.015
$\{SONY\_TV\}$	0.015
$\{Fridge, SONY\_TV, SONY\_PS3\}$	0.011
$\{Fridge\}$	0.011
$\{Fridge, SONY\_PS3\}$	0.008
$\{\}$	0.008
$\{SONY\_PS3\}$	0.004
Epoch 4—faulty state transition	
$\{SONY\_TV, SONY\_PS3\} \xrightarrow{Add\_TV} \{TV, SONY\_TV, SONY\_PS3\}$	
$\{TV, SONY\_PS3\}$	0.515
—Borderline of isolation—	0.500
$\{TV\}$	0.422
$\{TV, Fridge, SONY\_PS3\}$	0.371
$\{TV, Fridge\}$	0.285
$\{SONY\_TV, SONY\_PS3\}$	0.019
$\{SONY\_TV\}$	0.019
$\{\}$	0.011
$\{Fridge, SONY\_TV\}$	0.011
$\{SONY\_PS3\}$	0.008
$\{Fridge, SONY\_TV, SONY\_PS3\}$	0.008
$\{Fridge\}$	0.008
$\{Fridge, SONY\_PS3\}$	0.004

7.4 Results of repair

The faulty operations, including Add_SONY and Add_TV, could be repaired by the best revision repairs. The repaired operations are shown below.

```
Add_SONY =
  PRE not(SONY_TV : Attr) & not(SONY_PS3 : Attr)
  THEN Attr := Attr \/ {SONY_TV,SONY_PS3} ;
    IF Attr = {TV,Fridge,SONY_TV,SONY_PS3}
    THEN Attr := {Fridge,SONY_TV,SONY_PS3}
    END ;
    IF Attr = {TV,SONY_TV,SONY_PS3}
    THEN Attr := {SONY_TV,SONY_PS3}
    END
  END ;
Add_TV =
  PRE not(TV : Attr)
  THEN Attr := Attr \/ {TV} ;
    IF Attr = {TV,Fridge,SONY_TV,SONY_PS3}
    THEN Attr := {TV,Fridge,SONY_PS3}
    END ;
    IF Attr = {TV,SONY_TV,SONY_PS3}
    THEN Attr := {TV,SONY_PS3}
    END
  END ;
```

The best revision repairs were chosen according to the best SQE values in Tables 4 and 5. Two revision repairs were applied to Add_SONY, changing $\{TV, Fridge, SONY\_TV, SONY\_PS3\}$ to $\{Fridge, SONY\_TV, SONY\_PS3\}$ and changing $\{TV, SONY\_TV, SONY\_PS3\}$ to $\{SONY\_TV, SONY\_PS3\}$. The meaning of the repaired operation was that if Add_SONY was triggered, a SONY TV and a SONY PS3 would be added into the room, and if a TV was in the room, the TV would be removed. Moreover, two revision repairs were applied to Add_TV, changing $\{TV, Fridge, SONY\_TV, SONY\_PS3\}$ to $\{TV, Fridge, SONY\_PS3\}$ and changing $\{TV, SONY\_TV, SONY\_PS3\}$ to $\{TV, SONY\_PS3\}$. The meaning of the repaired operation was that if Add_TV was triggered, a TV would be added into the room, and if a SONY TV was in the room, the SONY TV would be removed. After repairing Add_SONY and Add_TV using the revision repairs, the system was free of faults.

Moreover, the faulty operations could be repaired by the isolation repairs. The repaired operations are shown below.

```
Add_SONY =
  PRE not(SONY_TV : Attr) & not(SONY_PS3 : Attr) &
    not(Attr = {TV,Fridge}) &
    not(Attr = {TV})
  THEN Attr := Attr \/ {SONY_TV,SONY_PS3}
  END ;
Add_TV =
  PRE not(TV : Attr) &
    not(Attr = {Fridge,SONY TV,SONY PS3}) &
    not(Attr = {SONY TV,SONY PS3})
  THEN Attr := Attr \/ {TV}
  END ;
```

According to Tables 4 and 5, the isolation repairs were not the best repairs, because the SQE values of the isolation repairs were slightly lower than those of the best revision repairs. Nevertheless, users could manually select the isolation repairs and ignore the revision repairs. Two isolation repairs could be applied to Add_SONY, disabling the faulty transitions from $\{TV, Fridge\}$ and $\{TV\}$. The meaning of the repaired operation was that Add_SONY could be triggered only if there was no TV, SONY TV and SONY PS3 in the room, and after triggering Add_SONY, a SONY TV and a SONY PS3 would be added into the room. Moreover, two isolation repairs were applied to Add_TV, disabling the faulty transitions from $\{Fridge, SONY\_TV, SONY\_PS3\}$ and $\{SONY\_TV, SONY\_PS3\}$. The meaning of the repaired operation was that Add_TV could be triggered only if there was no TV and SONY TV in the room, and after triggering Add_TV, a TV would be added into the room. After repairing Add_SONY and Add_TV using the isolation repairs, the system was free of faults.

In summary, the case study has shown that B-repair is able to repair the "Accommodation Management System" abstract machine. During the repair process, the tendency model was able to learn the post-condition of the operation and the relation between the pre- and post-conditions of the operation. Using the learnt features, the tendency model was able to provide SQE values of the repairs suggested by $Isolation$ and $Revision$, and the repairs that fitted the learnt features could gain higher SQE values than the others. By applying the repairs with high SQE values to the faulty operations, B-repair was able to change the original machine to a new machine that was free of faults and could reasonably maintain the features of the original machine. Moreover, we explained how the CARTs worked for B-repair, but did not provide an explanation for neural networks such as logistic models and residual networks, because the interpretability of CARTs is the highest one. The interpretability of the neural networks, however, is not as high as that of CARTs, so that a single case study is not sufficient to demonstrate whether or not the neural networks are efficient for B-repair. In order to demonstrate this point, in the next section, large scale experiments are used to evaluate the performance of the neural networks and CARTs.

8 Evaluation

In this section, we evaluate the overall performance of B-repair using a number of experiments. This section consists of two subsections: (1) primary evaluation using our own dataset and (2) further evaluation using public datasets.

8.1 Primary evaluation

In this subsection, we evaluate the performance of B-repair using four categories of models and focus on answering the following questions.

- **RQ1:** How accurately can the B-repair method repair models?
- **RQ2:** How well does the B-repair method repair different kinds of faults?
- **RQ3:** Do the types of tendency models influence results of repair?
- **RQ4:** Do the differences introduced by faults influence results of repair?

Experimental settings and results are presented in the following two subsections.

8.1.1 Settings

The experiments were conducted on a machine with Intel(R) Core(TM) i5-4670 CPU (4 cores, 3.40 GHz) and 8GB memories. The goal of the experiments was to count the accuracies of repair. In order to establish an objective criterion of accuracies, the following methodology was used. Firstly, a correct system A was built, and it was considered a standard answer that had met all requirements. Then a number of faults were injected into A, changing it to a faulty system P. Next, B-repair was used to repair P, resulting in a repaired system P'. Finally, P' was compared with A. Transitions in P' would be considered accurate if they were exactly the same as those in A. Otherwise, they would be considered inaccurate. The dataset[2] contained four subsets that were intentionally designed for the evaluation of B-repair. The motivation of designing our own datasets was that there were no existing datasets that could be directly used to evaluate B-repair. We searched others' studies related to B-model repair, but only found a well-formed model in the work by Schmidt et al. (2018). (This model is tested in Sect. 8.2.) Thus, we designed new models to evaluate B-repair. We tried our best to include essential features of the B-method into these models, and their descriptions were listed below.

- *Accommodation management system (AMS)* It was a reservation system for accommodations. Users could add new appliances into a room or remove existing appliances from the room. Moreover, the price of the room was calculated according to the appliances. This model included 2 variables, 3 invariants, 1 assertion and 22 operations with distinct elements, set computations, integer computations, Boolean computations, pre-conditioned substitutions and conditional substitutions.
- *Lift control system (LCS)* It was a lift controller that enables the lift to move up, move down, stop moving, open its door and close its door according to signals sent by users. This model included 7 variables, 7 invariants, 8 assertions and 8 operations with distinct elements, set computations, integer computations, Boolean computations, existential quantifications, pre-conditioned substitutions, conditional substitutions and non-deterministic substitutions.
- *Tennis player agent (TPA)* It was an agent of tennis players that can use different actions such as serving, receiving, running and jumping. This model included 9 variables, 13 invariants, 3 assertions and 8 operations with distinct elements, set computations, integer computations, Boolean computations, pre-conditioned substitutions, conditional substitutions and non-deterministic substitutions.
- *Course management system (CMS)* It was a course management system guiding students to arrange select courses and check if the students met the requirement of each selected course. This model included 5 variables, 7 invariants, 5 assertions, 12 mappings and 6 operations with distinct elements, set computations, integer computations, Boolean computations, existential and universal quantifications, pre-conditioned substitutions and non-deterministic substitutions.

Each subset contained nine faulty state transition systems and a correct one. The faulty state transition systems were made by injecting faults into the correct one. Injected

[2] The dataset is available at https://github.com/cchrewrite/B-ALTC-36.

faults included invariant violations (IV), assertion violations (AV) and deadlocks (DL), and these faults made the faulty state transition systems partially different from the correct one. The difference between two state transition systems was computed via

$$Diff(s_1, s_2) = Card(s_1 - s_2) + Card(s_2 - s_1) \tag{35}$$

where s_1 and s_2 were sets containing all transitions of the two systems respectively, "–" was used to compute the difference between two sets, and $Card(s)$ was the cardinality of a set s. For each type of faults, three faulty state transition systems were made, and the difference between each faulty system and the correct one was more than 1%, 5% or 10% of the number of available transitions in the correct one. Each faulty system was repaired using B-repair and different tendency models including logistic models (Logistic), residual networks (ResNet) and random forests of classification and regression trees (CART). The correct state transition system was considered a standard answer. When repairing a faulty system, the goal of repair was to remove all faults in the system and make the difference between the repaired system and the standard answer as small as possible. After repairing each system, the difference between the repaired system and the standard answer was computed, and the performance of B-repair was measured via the accuracy of repair

$$Acc(s_a, s_f, s_r) = Max(0,\ 1 - Diff(s_a, s_r)/Diff(s_a, s_f)) \tag{36}$$

where s_a was the standard answer, s_f was the faulty system, s_r was the repaired system, and $Max(x, y)$ was the maximum of x and y. Particularly, the maximum function was used to avoid negative numbers.

8.1.2 Results

Table 6 shows the results of evaluation. The first column of the table shows the name of each correct state transition system and its size, and the size is measured by the number of transitions in the system. The second column shows fault types. The third column shows the difference [i.e., Eq. (35)] between each faulty system and the standard answer, and the remaining columns show the difference between each repaired system and the standard answer. Particularly, the bold results have led to accuracies more than 90%. The results revealed that after the systems were repaired, many repaired transitions could fit those of the standard answer. Particularly, a significant number of repaired systems could perfectly fit the standard answer, and many repaired systems could partially fit the standard answer. Although many transitions did not fit the standard answer, they were still correct, because there was only one standard answer, but there might be many alternatives that can meet the users' requirements. These available answers, however, did not reduce the difference. Moreover, the systems with deadlocks were relatively easy to be repaired. After removing deadlocks from these systems, their differences were reduced to zero. In most of these systems, the pre-conditions of operations were relatively weak, so that there were only a few possible deadlock states in these systems. As a result, the tendency models had not learnt strong information to support revision repairs, so that they tended to suggest

Table 6 Results of repair

System	Fault	Difference	Difference After Repair		
<Size>	Type	Before repair	Logistic	ResNet	CART
AMS <6424>	IV	84	42	**1**	**0**
		396	198	61	**24**
		844	422	221	422
	AV	114	56	**0**	57
		445	141	121	141
		652	261	262	262
	DL	163	**0**	**0**	**0**
		375	**28**	**16**	**0**
		661	**49**	**11**	70
LCS <6768>	IV	70	66	**0**	**0**
		486	456	190	**0**
		1004	980	**4**	**0**
	AV	96	86	23	16
		444	341	**0**	334
		1266	436	**8**	**1**
	DL	80	**0**	**0**	**0**
		448	**0**	**0**	**0**
		926	**0**	**0**	**0**
TPA <11,386>	IV	128	64	48	46
		640	320	446	165
		2240	1120	1172	1120
	AV	248	3130	94	35
		737	3805	384	**0**
		3504	2356	**131**	2356
	DL	194	138	138	138
		610	**0**	**0**	**0**
		1040	448	448	448
CMS <11,402>	IV	265	140	140	140
		676	440	168	**0**
		1320	790	683	480
	AV	430	11,400	233	226
		860	5331	1157	442
		1314	1717	4056	1077
	DL	198	**0**	198	195
		648	**0**	**0**	**0**
		1296	**0**	**0**	**0**

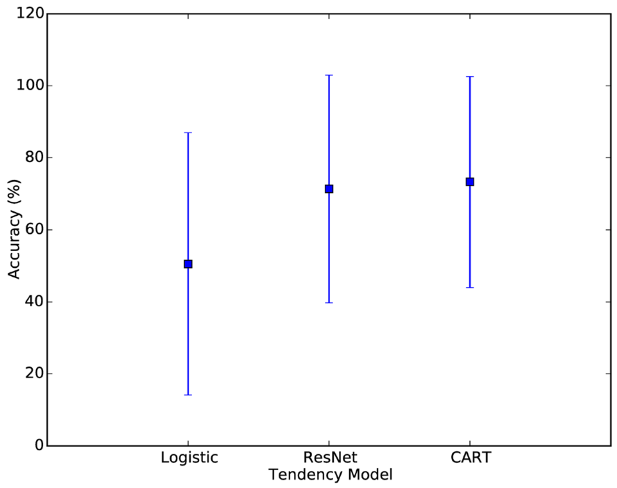

Fig. 8 Error bars of repair accuracies with respect to tendency models

isolation repairs. For the accommodation management system, the lift control system and the course management system, isolation repairs could remove their deadlocks and reduce their difference to zero. For the tennis player agent, however, isolation repairs were not able to reduce the difference to zero in most cases, because it required revision repairs. Additionally, the difficulties of repairing different systems are different from each other. For instance, repairing the tennis player agent is more difficult than repairing the other systems.

Figure 8 shows error bars of repair accuracies with respect to the tendency models. The figures revealed that the CARTs had the best overall performance, and the residual networks had the second best overall performance. In comparison to the residual networks and the CARTs, the logistic models had the worst performance. This was probably because it was difficult for the logistic models to model complex relations. Revisit the equations of logistic models in Sect. 3.2.2. The equations include a linear function [i.e., $\boldsymbol{A} \cdot \boldsymbol{x} + b$ in Eq. (3)] computing the weighted sum of features and a logistic function [i.e., Eq. (2)] mapping the sum to a real number between 0 and 1, meaning that the modelling ability of logistic models is mainly depended on the linear function. Different from the logistic models, the residual building blocks [(i.e., Eq. (4)] is a model that consists of many weighted sum computations, nonlinear activations and short-cut connections, so that they have the ability to model more complex relations. As logical relations in the state transition systems might be complex, this was probably the reason why the residual networks had better performance than the logistic models. Additionally, the CARTs were tree structures with attributes of data, so that they were able to model complex relations as well, as explained in Sect. 7.2. This might be the reason why the CARTs had better performance than the logistic models.

Figure 9 shows error bars of repair accuracies with respect to the differences before repair. When the tendency models were the residual networks, the best accuracies occurred in the "$>$ 10%" cases. When the tendency models were CARTs, the best

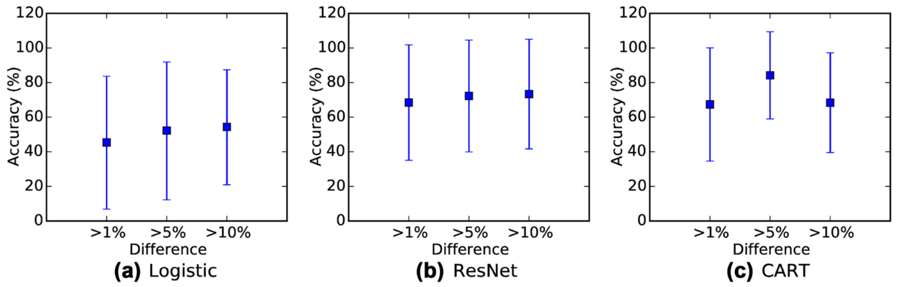

Fig. 9 Error bars of repair accuracies with respect to the differences before repair

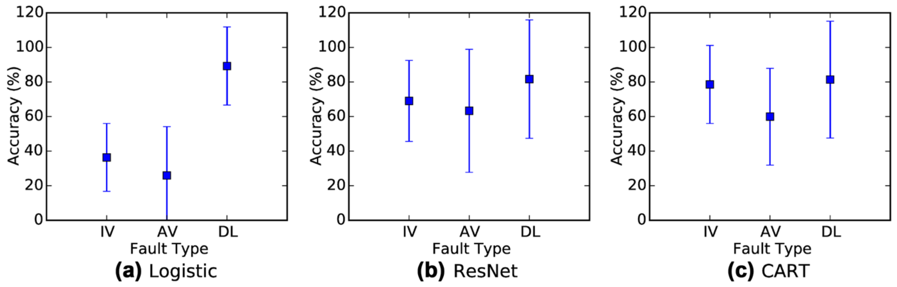

Fig. 10 Error bars of repair accuracies with respect to fault types

accuracies occurred in the "$> 5\%$" case. When the tendency models were the logistic models, the accuracies were lower than their counterparts. This phenomenon meant that the differences were a factor that impacts the accuracies. Moreover, the residual networks were able to deal with more differences than the CARTs and the logistic models. Additionally, although the accuracy of CARTs was decreasing when the difference increased, the accuracy was significantly higher than that of the logistic models.

Figure 10 shows error bars of repair accuracies with respect to fault types. When the tendency models were the logistic models, the accuracies of removing deadlocks were significantly higher than those of removing invariant violations and assertion violations. However, when the tendency models were the residual networks and CARTs, the accuracies of removing the three types of faults were close to each other. The above phenomenon indicated that the residual networks and the CARTs had significantly better performance than the logistic models when dealing with invariant violations and assertion violations. A possible explanation for this phenomenon was that the logistic models had not learnt strong information to suggest revision repairs for these deadlocks, so that they tended to suggest isolation repairs. The effect of isolation repairs and revision repairs were different, as the isolation repairs did not create new transitions, while the revision repairs did. As a result, the new transitions decreased the repair accuracy. If the revision repairs fitted the standard answer, the accuracy would increase, and sometimes the increase would be significant. However, if the revision repairs did not fit the standard answer, new transitions that would decrease the accuracy might be created. By contrast, the isolation repairs would not create new transitions, but they might remove existing and correct transitions. Thus, both isolation and revision had side-effects.

In summary, we have the following findings that can answer the research questions (i.e., **RQ1** to **RQ4**) at the beginning of Sect. 8.

- **RQ1:** The average accuracy of repair could reach 70% if the tendency models were the residual networks and the CARTs, and the average accuracy of repair could reach 50% if the tendency models were the logistic models. For different state transition systems, the accuracies of repair were different. In many cases, the accuracies of repair could reach a level close to 100% if the tendency models were the residual networks and the CARTs.
- **RQ2:** If the tendency models were the residual networks and the CARTs, the accuracies of repair could surpass 60% when dealing with deadlocks, invariant violations and assertion violations. If the tendency models were the logistic models, the accuracy could surpass 80% when repairing deadlocks, but the accuracies were lower than 40% when dealing with invariant violations and assertion violations.
- **RQ3:** The types of tendency models could influence the results of repair. The residual networks and CARTs had better performance than the logistic models. The residual networks had slightly better performance than the CARTs.
- **RQ4:** The differences introduced by faults could influence the results of repair. The difficulty of repair was rising when the number of differences increased. The residual neural networks were able to deal with more differences than the CARTs and the logistic models, and the CARTs were able to deal with more differences than the logistic models.

8.2 Further evaluation on public datasets

In this subsection, we evaluate B-repair on a number of public datasets that have been used in previous studies on the B-method. We focuses on answering more specific research questions as follows.

- **RQ5:** How accurately can the B-repair method repair public models?
- **RQ6:** Does the mechanism of tendency models outperform a baseline mechanism (i.e., random choices without using any tendency models)?
- **RQ7:** Are the top-ranked repairs more accurate than the other repairs?
- **RQ8:** Is one type of repair more effective on a particular system?

8.2.1 Settings

Datasets. To the best of our knowledge, there were no benchmark datasets that could be directly used to evaluate B model repair tools. Nevertheless, a number of public datasets of B models, which had been used in a number of previous studies, were available in the ProB Public Examples Repository.[3] We examined all abstract machines in the "B" directory and selected representative machines via the following steps.

[3] The ProB Public Examples Repository was downloaded from https://www3.hhu.de/stups/downloads/prob/source/.

- Machines with invariants were selected, because the constraint solving mechanism of the repair search process must be based on invariants.
- ProB was used to verify the machines. Well-formed machines, which were without any invariant (and assertion) violations, deadlock states and syntax errors, were selected.
- ProB was used to count the number of transitions in each machine. Machines that derived finite state spaces within 30,000 ms and had at least 100 transitions were selected.
- Datatypes of the machines were analysed, and machines without any higher-order sets and dynamic variables were selected, because our algorithms focused on first-order sets and static variables.
- If a number of machines were repetitions or approximations of each other, then only one of them was selected.
- In order to balance the number of tests in each subject and avoid memory exhaustion, we changed the scale of machines when necessary. All resulting machines must have at least 500 and at most 10,000 deterministic transitions.
- Machines that included other machines were excluded, because these machines were mostly parts of large systems, but system level repair was out of the scope of our study.
- Machines without actual significance were excluded.

Machines that satisfy all the above conditions were selected, leading to 15 representative models. Table 7 shows these models with subject IDs, lines of code (LOC), scales (i.e., the number of states and transitions), source file names in the public datasets and descriptions. These models were all correct models without any invariant violations, assertion violations or deadlocks. In order to evaluate B-repair, we created 10 faulty models by randomly seeding 200 faulty transitions into each correct model. Among the faulty transitions, half of them were made by randomly replacing the post-states of 100 correct transitions with faulty states, and the remaining half of them were made by randomly inserting 100 faulty transitions. As a result, 150 faulty models were created, and each of them was expected to be repaired via 100 revision repairs and 100 isolation repairs. Standard answers of repair were those that can make each repaired model derive the same state diagram as the correct model.

In order to make faulty models and standard answers, we developed an automatic fault injection program that can randomly inject faulty transitions into a given model. It can inject two types of faulty transitions into a model M. The first type of faulty transitions corresponds to revision repairs, and these transitions are injected into M via the following steps. Firstly, it randomly selects a correct transition $p \xrightarrow{\alpha} q$ in the state graph of M, where p is a pre-state, q is a post-state and α is an operation. Secondly, it analyses M's invariants, assertions and pre-conditions and finds a faulty state r. Thirdly, it replaces $p \xrightarrow{\alpha} q$ with $p \xrightarrow{\alpha} r$. Lastly, a standard answer [“$revision$”, $p \xrightarrow{\alpha} r, q$] is produced. The second type of faulty transitions corresponds to isolation repairs, and these transitions are injected into M via the following steps. Firstly, it analyses M's invariants, assertions and pre-conditions and finds a faulty state t. Secondly, it randomly selects a correct state s and an operation γ in the state graph of M. Thirdly,

Table 7 Public models

Subject	LOC	Scale	Source file name	Description
S-01	47	3587	ParallelModelCheckTest.mch	Two parallel counters
S-02	48	7705	POR_TwoThreads_WithSync.mch	A process with two threads
S-03	55	8705	progress.mch	Managing employees
S-04	56	7797	monitor2.mch	Monitoring users of a room
S-05	60	7220	InvolvedSequences2.mch	Sequences of operations
S-06	75	9480	club.mch	Managing club members
S-07	85	3587	ADD4.mch	Testing a calculator
S-08	85	8311	TestBZTT3.mch	Array assignments
S-09	90	13,533	scheduler6.mch	A process scheduler
S-10	109	7732	BinomialCoefficientConcurrent.mch	Binomial coefficients
S-11	119	7808	Lift2.mch	A lift controller
S-12	203	1479	Mikrowelle.mch	A microwave controller
S-13	211	1569	CSM.mch	A Petri net model
S-14	215	14,403	GSM_revue.mch	A file manager
S-15	482	27,056	Cruise_finite1.mch	Volvo cruise controller

it adds a new transition $s \xrightarrow{\gamma} t$. Lastly, a standard answer ["$isolation$", $s \xrightarrow{\gamma} t$] is produced.

Evaluation metrics As isolation and revision are different repair operators, they need to be evaluated using different metrics. The effectiveness of isolation repairs is evaluated using the Isolation Success Rate (ISR) that is defined as:

$$ISR = Card\left(S_{Iso}^{Sug} \cap S_{Iso}^{Ans}\right) / Card\left(S_{Iso}^{Ans} \cup S_{Iso}^{Ans}\right) \tag{37}$$

where S_{Iso}^{Sug} is a set containing all suggested isolation repairs, S_{Iso}^{Ans} is a set containing all standard answers of isolation repairs, and $Card(S)$ is the cardinality of a set S. The intuition of the above formula is that the higher ISR means that the more correct isolation repairs are suggested. The ISR must satisfy $0 \leq ISR \leq 1$. In the best case, all suggested isolation repairs match the standard answers, so that the ISR is 1. In the worst case, all suggested isolation repairs are different from the standard answers, or no isolation repairs are suggested, so that the ISR is 0.

The effectiveness of revision repairs is evaluated using the Revision Value Accuracy (RVA) that is defined as:

$$RVA = N_{Rev}^{Cor} / N_{Rev}^{Ans} \tag{38}$$

where N_{Rev}^{Cor} denotes the number of correct values in suggested revision repairs with reference to the standard answers, and N_{Rev}^{Ans} denotes the total number of values in the standard answers of revision repairs. The intuition of the above formula is that the higher RVA means that the more values in faulty states are correctly revised. The RVA must satisfy $0 \leq RVA \leq 1$. In the best case, all revised values match the standard answers, so that the RVA is 1. In the worst case, all revised values are different from the standard answers, or no revision repairs are suggested, so that the RVA is 0. The reason why the RVA is chosen to be an evaluation metric is that the RVA can measure the distance between a partially correct repair and its answer. For example, if only one fault is injected into a model such that a correct post-state ($w = 0, x = 0, y = 0, z = 0$) is replaced with a faulty state ($w = 1, x = 2, y = 3, z = 4$), then the answer for revising the faulty state will be ($w = 0, x = 0, y = 0, z = 0$), and $N_{Rev}^{Ans} = 4$. If a revision repair suggests ($w = 0, x = 1, y = 0, z = 0$), then $N_{Rev}^{Cor} = 3$, and $RVA = 0.75$. If a revision repair suggests ($w = 0, x = 1, y = 1, z = 0$), then $N_{Rev}^{Cor} = 2$, and $RVA = 0.50$. Both the revision repairs do not perfectly fit the answer, but their distances from the answer are different. The revision ($w = 0, x = 1, y = 0, z = 0$) is closer to the answer because only the value of x disagrees with its counterpart of the answer, while ($w = 0, x = 1, y = 1, z = 0$) has two values of variables, including x and y, that disagree with their counterparts of the answer.

Based on ISR and RVA, repairs suggested by B-repair with different tendency models, including the logistic model, the ResNets and the CARTs, were compared with the standard answers. In order to investigate whether or not the tendency models had a beneficial effect, they were compared with a baseline mechanism that all repairs were randomly chosen without using any tendency models. Besides, in order to investigate whether or not the top-ranked repairs were more accurate than the other repairs, repairs ranked from 2nd to 10th were compared with the standard answers as well.

8.2.2 Results

Table 8 shows B-repair's ISRs and RVAs on the public B-models. B-repair repaired these B-models with the three tendency models: the logistic model (Logistic), the residual network (ResNet) and the random forest of classification and regression trees (CART). For the purpose of comparison, results of the baseline mechanism, abbreviated as "Random", were obtained as well. These result revealed that both the ISRs and RVAs were impacted by the tendency models. When the tendency models were absent, repairs were randomly ranked and selected. As each faulty transition corresponded to exactly one isolation repair and a large number of revision repairs, most selected repairs were revision repairs, leading to ISRs close to 0. When the tendency models were used to rank the repairs, the ISRs increased. On average, the CART random forest led to the best ISR (i.e., 0.541), which meant that more than half of isolation repairs were successful. On specific subjects such as S-03, S-04, S-06 and S-09, the ISRs of the random forest were lower than their counterparts of the logistic model or the residual network. This was probably because an overfitting phenomenon occurred during the learning process of the random forest, resulting in a bias that could give a number of unexpected revision repairs higher ranks than expected isolation repairs. Regarding RVAs, the CART random forest obtained the best results. On average, its RVA achieved 0.875, which meant that the top-ranked revision repairs suggested by the random forest were mostly correct. On specific subjects such as S-04 and S-06, the residual network led to better results than the random forest.

Table 9 shows the numbers of top-ranked isolation repairs and revision repairs suggested by the different tendency models. It was clear that most tendency models

Table 8 ISRs and RVAs on the public B-models

Subject	Isolation success rate (ISR)				Revision value accuracy (RVA)			
	Random	Logistic	ResNet	CART	Random	Logistic	ResNet	CART
S-01	0.005	0.644	0.670	1.000	0.194	0.268	0.763	0.831
S-02	0.000	0.971	0.970	1.000	0.021	0.016	0.121	0.924
S-03	0.006	0.306	0.060	0.184	0.075	0.297	0.870	0.921
S-04	0.001	0.185	0.222	0.002	0.044	0.237	0.825	0.614
S-05	0.001	0.000	0.186	0.714	0.164	0.150	0.717	0.926
S-06	0.000	0.092	0.133	0.001	0.026	0.206	0.834	0.656
S-07	0.003	0.000	0.449	0.915	0.385	0.302	0.926	0.948
S-08	0.003	0.000	0.654	0.683	0.403	0.598	0.882	0.920
S-09	0.003	0.000	0.125	0.000	0.102	0.151	0.838	0.895
S-10	0.002	0.157	0.441	0.966	0.128	0.068	0.499	0.853
S-11	0.006	0.641	0.094	0.749	0.308	0.358	0.829	0.947
S-12	0.008	0.001	0.011	0.167	0.355	0.617	0.651	0.862
S-13	0.000	0.167	0.069	0.237	0.385	0.374	0.724	0.927
S-14	0.001	0.000	0.465	0.982	0.648	0.722	0.936	0.944
S-15	0.002	0.002	0.044	0.516	0.547	0.394	0.827	0.949
Average	0.003	0.211	0.306	**0.541**	0.252	0.317	0.749	**0.875**

Table 9 Numbers of top-ranked repairs

Subject	# Top-ranked isolation repairs				# Top-ranked revision repairs			
	Random	*Logistic*	ResNet	*CART*	Random	Logistic	ResNet	CART
S-01	5	683	701	1040	1995	1317	1299	960
S-02	0	1012	1033	1041	2000	988	967	959
S-03	6	318	64	193	1994	1682	1936	1807
S-04	1	197	234	2	1999	1803	1766	1998
S-05	1	0	196	748	1999	2000	1804	1252
S-06	0	95	141	1	2000	1905	1859	1999
S-07	3	0	469	962	1997	2000	1531	1038
S-08	3	0	684	721	1997	2000	1316	1279
S-09	3	0	132	0	1997	2000	1868	2000
S-10	2	162	466	1024	1998	1838	1534	976
S-11	6	678	99	775	1994	1322	1901	1225
S-12	9	1	11	177	1991	1999	1989	1823
S-13	0	176	73	251	2000	1824	1927	1749
S-14	1	0	488	1029	1999	2000	1512	971
S-15	2	2	46	543	1998	1998	1954	1457
Total	44	3326	4837	8506	29,956	26,674	25,163	21,494

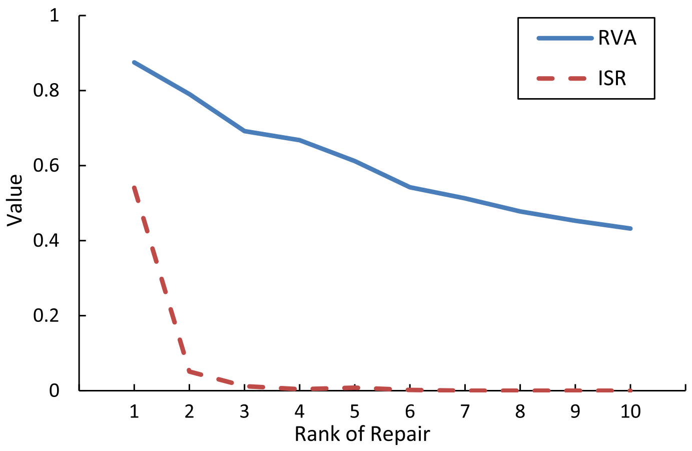

Fig. 11 Curves of ISR and RVA subject to the rank of repair

suggested more revision repairs than isolation repairs. This phenomenon was reasonable, because in each ranking list, there were a large number of candidate revision repairs, but only one candidate isolation repair. Considering the large number of candidate repairs, the effect of the tendency models was to give expected repairs higher ranks. When no tendency models were used (i.e., the "Random" case), ranks were randomly assigned to the repairs, so that isolation repairs only had a small chance to be top-ranked repairs. When the tendency models were used, isolation repairs had a significantly larger chance to be top-ranked repairs. Associating the numbers of top-ranked isolation repairs with the ISRs in Table 8, it was demonstrated that the tendency models had positive effects on the selection of high-quality repairs.

Figure 11 shows the curves of ISR and RVA subject to the rank of repair, i.e., in each ranking list, instead of the 1st repair, the 2nd to 10th repairs are chosen respectively. The data for plotting the curves were the average ISRs and RVAs of the CART random forest on all the 15 subjects. It was clear that the curve of ISR dropped to a level close to 0 when the rank of repair was greater than 1, which meant that the tendency model had crucial effects on the selection of isolation repairs. Regarding RVAs, the curve was gradually dropping when the rank of repair increased, which meant that top-ranked revision repairs were usually more accurate than others.

Based on the above experimental results, we are able to answer the research questions at the beginning of Sect. 8.2.

- **RQ5:** On the public models, B-repair was able to accurately suggest revision repairs at the average accuracy of 87.5%, and it was able to accurately suggest isolation repairs at the average success rate of 54.1%. On a number of specific models, both the accuracy of revision and the success rate of isolation could achieve a level above 90%.
- **RQ6:** The mechanism of tendency models significantly outperformed the baseline mechanism. Without the tendency models, the B-models could not be accurately repaired. Among the three tested tendency models, the CART random forest had

the best predictive performance. On a few specific models, the residual network could outperform the random forest.
- **RQ7:** Probabilistically, for both isolation and revision repairs, the top-ranked repairs were more accurate than the other repairs.
- **RQ8:** On a particular system, one type of repair could be more effective. Consider the repairs ranked using the CART random forest. On systems such as S-01, S-02, S-07, S-10 and S-14, both isolation and revision repairs were effective and could achieve accuracies above 80%. On systems such as S-03, S-05, S-08, S-09, S-11, S-12, S-13 and S-15, revision repairs were more effective and could achieve accuracies above 85%, while isolation repairs seemed not accurate.

8.3 Threats to validity

In this subsection, we discuss threats to validity. Threats to internal validity include the generality of the evaluation datasets and the efficiency of the model checker. Threats to external validity include the access to real defeats of B-models and the size of fault.

8.3.1 Threats to internal validity

The generality of the evaluation datasets is a potential threat to validity. The evaluation datasets include 4 models built by the authors and 15 public models built by other researchers. These models may not cover all features of B-model design. Although we have tried our best to obtain more test data, available test data are still limited. The reason why the test data are limited is that our research topic, i.e., B-model repair, is an emerging topic proposed by Schmidt et al. (2016) and further studied by Schmidt et al. (2018) in recent three years. The above two studies only provided two models for case study purposes, and the two models are not suitable and not sufficient for an empirical evaluation. Consequently, we had to collect more models used in past studies and build our own evaluation datasets. However, most models used in past studies are designed for evaluating various functions of the B-method such as bounded model checking, refinement checking, SMT solving, etc., so that only a limited number of models can be used to evaluate B-repair. Particularly, S-09 in Table 7 has been used in the previous work of interactive B-model repair and synthesis (Schmidt et al. 2018). However, it is difficult to systematically compare our evaluation results with their results, because their premise and goal of evaluation are different from ours. Their premise is that users can provide a number of I/O examples and can interact with their model repair tool, and their goal is to synthesise new operations that work for the examples. Our premise is that users do not interact with B-repair, and our goal is to eliminate faulty transitions by changing existing operations.

Regarding the model checker, it is one of the most important and time consuming components in B-repair. Although it is able to verify the correctness of a model with more than 27K states and transitions (i.e., S-15 in Table 7), verifying larger models using limited computational resources may be a difficult task. As the model checking is a prerequisite of model repair, the efficiency of B-repair is restricted by the efficiency of the model checker. Previous experiences show that B-repair may be inefficient if a large model occupies most memory space during the model checking process. Besides, B-repair can repair models with static variables and the four fundamental datatypes, but cannot deal with complex datatypes and dynamic variables. Although this problem can be solved by decomposing complex datatypes into fundamental datatypes and replacing dynamic variables with static variables, repairing models with a large number of static variables and fundamental datatypes is still difficult. The reason is that the model checker naturally requires more time to complete the tasks of model checking and constraint solving for larger models.

8.3.2 Threats to external validity

A potential threat to validity is the access to real B-models and defeats. Most real B-models are used in industrial projects. For example, a number of automatic railway control systems in North America and Europe have been developed using B (Behm et al. 1999; Benaïssa et al. 2016). Moreover, a number of industrial PLC controllers (Barbosa and Déharbe 2012) and the security properties of a microkernel (Hoffmann et al. 2007) has been verified using B. However, most industrial B-models are not public. Consequently, we are not able to access most of the industrial models except the Vovol cruise controller model (i.e., S-15 in Table 7). Besides, it is difficult to obtain real defeats of B-models and corresponding human-made repairs via version control platforms such as GitHub. When evaluating automated program repair (APR) tools for popular programming languages such as Java, people can download different versions of a program from version control platforms such as GitHub, consider the differences between the two versions as feasible repairs and observe whether or not an APR repair tool is able to produce the same repairs (Le et al. 2016b). However, this methodology seems not suitable for the evaluation of B-repair, because it is difficult to find appropriate B-models on the version control platform due to the fact that the community of B is considerably smaller than the community of the popular programming languages. Consequently, most models and defeats used in our work are artificial.

Another potential threat to validity is the size of fault. In the second part of our evaluations, multiple faults were randomly injected into a model. As each fault created exactly one erroneous transition, the size of fault was 1. However, in the real world, there are two more cases. The first case is that a fault creates two or more erroneous transitions, so that the size of fault is greater than 1. In this case, B-repair is still able to eliminate the fault, and the number of suggested repairs is exactly the same as the number of erroneous transitions. On the other hand, a human expert may eliminate the fault using only one repair, which means that the repair suggested by the human expert is better than those suggested by the algorithm. The second case is that an erroneous transition is created by multiple faults, so that the size of fault is smaller than 1. In order to eliminate these faults, a comprehensive analysis is required in order to localise faults and calculate candidate repairs, which is still a challenging task for both humans and computers.

9 Related work

In Sects. 4–8, we proposed B-repair, which can automatically suggest repairs for faulty abstract machines of B, rank the suggested repairs and apply suitable repairs to the machines. The three key aspects, including the *Isolation* and *Revision* method, repair ranking using tendency models and the evaluation of B-repair using several examples, have been investigated in previous sections.

Our work is relevant to a number of previous studies on the repair of state transition systems. For example, Schmidt et al. (2018) have proposed an interactive method for repairing state transition systems of B. This method can remove deadlock states and invariant violations using the program synthesis technique proposed by Jha et al. (2010). To remove deadlock states, it strengthens the pre-conditions of previous operations or synthesises new operations that resolve the deadlock states. To remove invariant violations, it strengthens the pre-conditions of previous operations or relaxes the violated invariants. This method has been used to repair faulty models of Event-B as well (Schmidt et al. 2016). Moreover, Babin et al. (2016) have proposed a refinement checking-based method for repairing state transition systems of Event-B. If a failure occurs in a system, failed states will be replaced with successful states using substitutions, and the behaviours of the original system will be preserved using refinement checking. Further, Alrajeh and Craven (2014) have suggested the use of inductive logic programming to discover repairs that can remove violation runs and deadlocks from systems and maintain the behaviours of the systems. This method will suggest all possible minimal repairs and require users to select one. Besides, our work is relevant to previous studies on residual networks and CARTs. He et al. (2016) have suggested that the residual networks, which are feed-forward neural networks with shortcut connections, seem to have stronger abilities of modelling complex features than the plain feed-forward neural networks. Kurt et al. (2008) have compared different supervised machine learning models and found that CARTs are able to model complex relations.

In our work on B-repair, the most important parts are *Isolation*, *Revision* and the tendency models. Similar to the fault removal method proposed by Schmidt et al. (2016) and Schmidt et al. (2018), *Isolation* is able to remove faulty transitions by strengthening the pre-condition of the last operation. The difference between the *Isolation* method and the fault removal method is that the former one produces a repair that can remove a single transition, while the later one produces a repair that can remove a set of transitions. Similar to the system substitution mechanism of Event-B proposed by Babin et al. (2016), *Revision* is able to replace faulty transitions with correct ones via conditional substitutions. The difference between the *Revision* method and the system substitution mechanism is that the former one relies on the checking of invariants and assertions, while the later one relies on refinement checking. More importantly, B-repair uses probabilistic machine learning techniques to build tendency models and uses them to rank repairs, so that repair selection processes can be automated. It is different from the repair selection methods proposed by Alrajeh and Craven (2014) and Schmidt et al. (2018), which require users' feedback to select repairs.

The tendency models play an important role of improving the accuracy of repair. They are able to learn features of state transition systems and rank repairs according to the learnt features. Different tendency models lead to different ranking results, and consequently, they lead to different accuracies of repair. The residual networks often lead to better accuracies than the logistic models. A possible explanation for this phenomenon is that the residual networks are able to model more complex features than the logistic models, and this argument can be supported by previous work on the use residual learning to model image features (He et al. 2016). Moreover, the CARTs outperform the logistic models. As explained in Sect. 7.2, the CARTs are able to exclude the impact of irrelevant attributes and model the post-conditions and the relations between the pre- and post-conditions according to a number of certain attributes. On the other hand, it is difficult for the logistic models to exclude irrelevant attributes and model complex relations of these attributes, as they are based on a simple weighted sum function that considers all attributes together. Similar evidences, which show that CARTs usually outperform logistic models on tasks of modelling complex relations, can be found in the work by Kurt et al. (2008).

Our work is related automated program repair (APR) as well. APR is a subfield of program synthesis, aiming to assist programmers in finding faulty components in programs and producing patches (Gazzola et al. 2019). Similar to B-model repair, APR usually has three steps: fault localisation, repair synthesis and repair selection. Spectrum-based fault localisation is one of the most widely used fault localisation methodology (Abreu et al. 2009). It makes use of a set of input-output pairs to test the correctness of a program and produce traces of successful executions and failed executions, and find suspicious faulty components by analysing the occurrences of the components in the traces. Usually, faulty components tend to occur in the traces of failed executions and not to occur in the traces of successful executions. After localising faults, a vast number of candidate repairs are produced using various repair synthesis techniques such as mutation repair, template-based repair and genetic programming. Mutation repair techniques, which apply atomic repairs to the Abstract Syntax Tree (AST) of a program, are supported by various APR tools such as CAPGEN (Wen et al. 2018), GenProg (Le Goues et al. 2012), CASC (Wilkerson and Tauritz 2010), etc. Template-based repair techniques, such as the history driven program repair method proposed by Le et al. (2016b) and the SearchRepair tool developed by Ke et al. (2015), focus on using repairs extracted from version control platforms and those produced by programmers to eliminate common bugs in programs. Genetic programming is a repair synthesis technique that mixes existing repairs together to generate new repairs and has been supported by GenProg (Le Goues et al. 2012), CASC (Wilkerson and Tauritz 2010), etc. The above repair synthesis techniques can synthesise a vast number of candidate repairs, so that repair ranking functions are usually needed in order to obtain high-quality repairs.

Repair ranking functions are crucial for a number of APR methods. Similar to the tendency models of B-repair, the repair ranking functions are able to map candidate repairs to numbers so that the candidate repairs can be ranked. For example, Le et al. (2016b) have proposed a repair ranking function based on historical data of program repair. The historical data are collected from version control platforms such as GitHub, and repair templates are produced by comparing different versions of a program. The repair templates are used to synthesise candidate repairs for a given faulty program, and the candidate repairs are ranked by the frequencies of their patterns occurring in the historical data. The idea of using a frequency function to rank repairs has been used in CAPGEN as well (Wen et al. 2018). In CAPGEN, candidate repairs are ranked by the product of suspicious values, frequencies of mutation operators and context similarities. The suspicious values are calculated using a fault localisation technique that indicates faulty lines of code. The frequencies of mutation operators are calculated based on historical data of successful mutation repairs. The context similarities are calculated by comparing neighbouring nodes of original AST nodes with those of repaired AST nodes. The above repair ranking functions have been demonstrated to be effective for the repair of JAVA programs. A common feature of these repair ranking functions is that they act on programs at the syntactic level. Repair ranking functions can act on programs at the semantic level as well, and Le et al. (2016a) have argued that the combination of the syntactic and semantic levels can lead to better repairs. For instance, in S3 (Le et al. 2017) both syntactic features and semantic features are used to rank repairs. The syntactic features include AST structures, the vectorisation of ASTs and locality of variable and constants, and the semantic features include the satisfiability of particular formulae, the coverage of input-output pairs and counterexamples.

The similarities between B-repair and the APR techniques include the following three aspects. Firstly, they require faulty localisation techniques to find suspicious faulty components in programs. Secondly, they require repair synthesis techniques to produce a large number of candidate repairs. Thirdly, they require repair ranking functions to rank the candidate repairs and obtain high-quality repairs. The differences between B-repair and the APR techniques include the following four aspects. Firstly, B-repair localise faults using the model checker, while the APR tools usually localise faults based on test suites. Secondly, a number of APR tools such as S3 and CAPGEN make use of (symbolic) machine learning techniques to discover candidate repairs from historical data, while these techniques are not used in B-repair. Thirdly, B-repair's tendency models are based on (probabilistic) machine learning and do not rely on historical data, while the repair ranking functions of S3 and CAPGEN are built upon historical data. Lastly, B-repair focuses on eliminating invariant violations, assertion violations and deadlocks, while APR focuses more on making a program consistent with test suites.

In summary, the novelty of B-repair is that it makes use of probabilistic machine learning to learn features of abstract machines and makes use of the learnt features to acquire high-quality repairs. Combining with model checking techniques, B-repair is able to automate model repair processes and gain high accuracies of repair. Thus, this study suggests that B-repair can be used to improve the efficiency of state transition system design. Firstly, a user needs to design an initial abstract machine that describes a state transition system and properties of the system. Then a tendency model learns from the state transition system. Regardless of whether or not the system satisfies the properties, the tendency model can learn relations hidden in operations of the system. Next, the model checker checks whether or not the system satisfy the properties. If not, then $Isolation$ and $Revision$ are used to suggest repairs. After that, the suggested repairs are ranked using the tendency model. Finally, the user can enable B-repair to automatically select the repair with the highest rank or manually select a satisfactory repair, and the selected repair will be automatically applied to a suitable position in the initial abstract machine. As a result, the user only needs to consider whether or not the applied repair meets design requirements, but does not need to consider how to work out possible repairs or how to manually apply the repairs to the abstract machine.

10 Conclusion

B-repair is an automated model repair approach that combines model checking, constraint solving and probabilistic machine learning. It makes use of the model checker to find faults in abstract machines, the SMT solver to suggest repairs and various machine learning models to select repairs. Using B-repair, the repair of abstract machines can be done by changing the pre- and post-conditions of operations. Moreover, the behaviours of abstract machines, which are described by operations with pre- and post-conditions, can be learnt by the machine learning models, so that the machine learning models can help the users improve the quality of repaired machines.

The implementation of B-repair has been evaluated on the tasks of repairing a number of state transition systems. It has been found that B-repair is able to accurately suggest revision repairs that satisfy users' requirements and remove thousands of faulty transitions in the systems. Moreover, it has been found that the machine learning techniques can significantly influence the results of repair, and random forests seem to be the best machine learning technique for B-repair.

To the best of our knowledge, B-repair is the first system that attempts to use probabilistic machine learning to improve the repair of abstract machines, providing a new solution to improve the efficiency of system design. As a result, users can design an initial system that may contain faults. This means that they can focus on developing required functions and do not need to pay much attention to minor faults. After designing the initial system, they can use B-repair to detect faults, suggest possible repairs, evaluate the quality of the repairs and revise the initial system. This means that they only need to consider whether or not the suggested repairs meet their requirements, but do not need to consider how to work out possible repairs and how to apply the repairs to the system.

In the future, our work will target the following limitations of B-repair. Firstly, the current B-repair only suggests repairs that repair single transitions. These repairs may be replaced with an abstract repair that combines complex expressions to cover more than one transition. Secondly, the current B-repair cannot repair abstract machines that do not pass refinement checking, which is another component of B. Thirdly, we will design a benchmark that contains a number of B-models of classical algorithms, i.e., those in textbooks of algorithms and data structures. We will use a version control tool to archive development processes of these models. After that, we will be able to obtain real defeats and use them as a benchmark. Finally, the current B-repair does not deal with temporal logics that are widely used in model checking. To solve these limitations, we will develop repair merging algorithms, parallel repair algorithms and model repair algorithms with refinement checking and temporal logics.

Acknowledgements We would like to thank the anonymous reviewers, whose valuable comments have contributed to the clarification of many ideas presented in the paper.

References

Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C.: A practical evaluation of spectrum-based fault localization. J. Syst. Softw. **82**(11), 1780–1792 (2009)

Abrial, J.: The B-book—Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)

Abrial, J., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transfer **12**(6), 447–466 (2010)

Alrajeh, D., Craven, R.: Automated error-detection and repair for compositional software specifications. In: 12th International Conference Software Engineering and Formal Methods, SEFM 2014, Grenoble, France, September 1–5, 2014. Proceedings, pp. 111–127 (2014)

Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)

Babin, G., Ameur, Y.A., Singh, N.K., Pantel, M.: A system substitution mechanism for hybrid systems in Event-B. In: Proceedings 18th International Conference on Formal Engineering Methods Formal Methods and Software Engineering, ICFEM 2016, Tokyo, Japan, November 14–18, 2016, pp. 106–121 (2016)

Bagaria, J.: Set theory. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy, winter 2017 edn. Stanford University, Stanford (2017)

Barbosa, H., Déharbe, D.: Formal verification of PLC programs using the B method. In: Abstract State Machines, Alloy, B, VDM, and Z—Proceedings Third International Conference, ABZ 2012, Pisa, Italy, June 18–21, 2012, pp. 353–356 (2012)

Behm, P., Benoit, P., Faivre, A., Meynadier, J.: Météor: A successful application of B in a large project. In: FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999, Proceedings, Volume I, pp. 369–387 (1999)

Benaïssa, N., Bonvoisin, D., Feliachi, A., Ordioni, J.: The PERF approach for formal verification. In: Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification—First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings, pp. 203–214 (2016)

Bottou, L.: Stochastic gradient descent tricks. In: Montavon, G. (ed.) Neural Networks: Tricks of the Trade, 2nd edn, pp. 421–436. Springer, Berlin (2012)

Boulanger, J.L., Aljer, A., Mariano, G.: Formalization of digital circuits using the b method. WIT Trans. Built Environ. https://doi.org/10.1002/9781119002727.ch6 (2002)

Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Routledge, Wadsworth (1984)

Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10–20 states and beyond. Inf. Comput. **98**(2), 142–170 (1992)

Cai, C., Sun, J., Dobbie, G.: B-repair: Repairing B-models using machine learning. In: 23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12–14, 2018, pp. 31–40 (2018)

Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model verifier. In: 11th International Conference Computer Aided Verification, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings, pp. 495–499 (1999)

Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981, pp. 52–71 (1981)

Cox, D.R.: The regression analysis of binary sequences. J. R. Stat. Soc. Ser. B (Methodol.) **20**, 215–242 (1958)

D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. CAD Integrated Circuits Syst. **27**(7), 1165–1178 (2008)

Fadil, H., Koning, J.: A formal approach to model multiagent interactions using the B formal method. In: Advanced Distributed Systems: 5th International School and Symposium, ISSADS 2005, Guadalajara, Mexico, January 24–28, 2005, Revised Selected Papers, pp. 516–528 (2005)

Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. IEEE Trans. Softw. Eng. **45**(1), 34–67 (2019)

Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011, pp. 315–323 (2011)

Gopinath, D., Malik, M.Z., Khurshid, S.: Specification-based program repair using SAT. In: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software Tools and Algorithms for the Construction and Analysis of Systems, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings, pp. 173–188 (2011)

Harel, D., Katz, G., Marron, A., Weiss, G.: Non-intrusive repair of safety and liveness violations in reactive programs. Trans. Comput. Collective Intell. **16**, 1–33 (2014)

He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27–30, 2016, pp. 770–778 (2016)

Ho, T.K.: Random decision forests. In: Third International Conference on Document Analysis and Recognition, ICDAR 1995, August 14–15, 1995, Montreal, Canada. Volume I, pp. 278–282 (1995)

Hoffmann, S., Haugou, G., Gabriele, S., Burdy, L.: The b-method for the construction of microkernel-based systems. In: B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17–19, 2007, Proceedings, pp. 257–259 (2007)

Huth, M., Ryan, M.D.: Logic in Computer Science—Modelling and Reasoning About Systems, 2nd edn. Cambridge University Press, Cambridge (2004)

Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, pp. 215–224 (2010)

Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y.: Repairing programs with semantic code search (T). In: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, pp. 295–306 (2015)

Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Integrated Formal Methods—12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings, pp. 361–375 (2016)

Kurt, I., Ture, M., Kurum, A.T.: Comparing performances of logistic regression, classification and regression tree, and neural networks for predicting coronary artery disease. Expert Syst. Appl. **34**(1), 366–374 (2008)

Le, X.D., Chu, D., Lo, D., Le Goues, C., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4–8, 2017, pp. 593–604 (2017)

Le, X.D., Le, Q.L., Lo, D., Le Goues, C.: Enhancing automated program repair with deductive verification. In: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2–7, 2016, pp. 428–432 (2016a)

Le, X.D., Lo, D., Le Goues, C.: History driven program repair. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14–18, 2016—Volume 1, pp. 213–224 (2016b)

Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software Qual. J. **21**(3), 421–443 (2013)

Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)

Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transfer **10**(2), 185–203 (2008)

Leuschel, M., Cansell, D., Butler, M.J.: Validating and animating higher-order recursive functions in B. In: Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday, pp. 78–92 (2009)

Loh, W.: Classification and regression trees. Wiley Interdisc. Rew. Data Min. Knowl. Discov. **1**(1), 14–23 (2011)

Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, vol. 2283. Springer, Berlin (2002)

Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57 (1977)

Schmidt, J., Krings, S., Leuschel, M.: Interactive model repair by synthesis. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z—5th International Conference, ABZ 2016, Linz, Austria, May 23–27, 2016, Proceedings, pp. 303–307 (2016)

Schmidt, J., Krings, S., Leuschel, M.: Repair and generation of formal models using synthesis. In: Integrated Formal Methods—14th International Conference, IFM 2018, Maynooth, Ireland, September 5–7, 2018, Proceedings, pp. 346–366 (2018)

Siekmann, J.H.: Unification theory. J. Symb. Comput. **7**(3/4), 207–274 (1989)

Turian, J.P., Ratinov, L., Bengio, Y.: Word representations: a simple and general method for semi-supervised learning. In: ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11–16, 2010, Uppsala, Sweden, pp. 384–394 (2010)

Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.: Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27–June 03, 2018, pp. 1–11 (2018)

Wilkerson, J.L., Tauritz, D.R.: Coevolutionary automated software correction. In: Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7–11, 2010, pp. 1391–1392 (2010)

Yang, G., Khurshid, S., Kim, M.: Specification-based test repair using a lightweight formal method. In: FM 2012: Formal Methods—18th International Symposium, Paris, France, August 27–31, 2012. Proceedings, pp. 455–470 (2012)