Fast Automated Abstract Machine Repair Using Simultaneous Modifications and Refactoring

CHENG-HAO CAI, JING SUN, and GILLIAN DOBBIE, School of Computer Science, University of Auckland, New Zealand
ZHÉ HÓU and HADRIEN BRIDE, Institute for Integrated and Intelligent Systems, Griffith University, Australia
JIN SONG DONG, School of Computing, National University of Singapore, Singapore and Institute for Integrated and Intelligent Systems, Griffith University, Australia
SCOTT UK-JIN LEE, College of Computing, Hanyang University ERICA, Korea

Automated model repair techniques enable machines to synthesise patches that ensure models meet given requirements. B-repair, which is an existing model repair approach, assists users in repairing erroneous models in the B formal method, but repairing large models is inefficient due to successive applications of repair. In this work, we improve the performance of B-repair using simultaneous modifications, repair refactoring and better classifiers. The simultaneous modifications can eliminate multiple invariant violations at a time so that the average time to repair each fault can be reduced. Further, the modifications can be refactored to reduce the length of repair. The purpose of using better classifiers is to perform more accurate and general repairs and avoid inefficient brute-force searches. We conducted an empirical study to demonstrate that the improved implementation leads to the entire model process achieving higher accuracy, generality and efficiency.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; *Software development techniques*; • **Computing methodologies** → Machine learning.

Additional Key Words and Phrases: B-method, model checking, automated model repair, repair evaluator training

ACM Reference Format:
Cheng-Hao Cai, Jing Sun, Gillian Dobbie, Zhé Hóu, Hadrien Bride, Jin Song Dong, and Scott Uk-Jin Lee. 2022. Fast Automated Abstract Machine Repair Using Simultaneous Modifications and Refactoring. *Form. Asp. Comput.* X, X, Article 111 (December 2022), 34 pages. https://doi.org/xx/xxxx.xxxxx

Authors' addresses: Cheng-Hao Cai, chenghao.cai@auckland.ac.nz; Jing Sun, jing.sun@auckland.ac.nz; Gillian Dobbie, g.dobbie@auckland.ac.nz, School of Computer Science, University of Auckland, 38 Princes Street, Auckland, New Zealand, 1142; Zhé Hóu, z.hou@griffith.edu.au; Hadrien Bride, h.bride@griffith.edu.au, Institute for Integrated and Intelligent Systems, Griffith University, 170 Kessels Road, Queensland, Australia, 4111; Jin Song Dong, dongjs@comp.nus.edu.sg, School of Computing, National University of Singapore, 13 Computing Drive, Singapore, 117417 and Institute for Integrated and Intelligent Systems, Griffith University, 170 Kessels Road, Queensland, Australia, 4111; Scott Uk-Jin Lee, scottlee@hanyang.ac.kr, College of Computing, Hanyang University ERICA, 55 Hanyangdeahak-ro, Ansan, Korea, 15588.

0934-5043/2022/12-ART111 $15.00
https://doi.org/xx/xxxx.xxxxx

DECLARATIONS

This page includes required declarations for our submission.

Funding

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

Conflicts of Interest / Competing Interests

Not applicable.

Availability of Data and Material

We conducted experiments using a number of third-party B machines, which were downloaded from:

- https://www3.hhu.de/stups/downloads/prob/source/
- https://github.com/hhu-stups/abz2020-models
- https://github.com/hhu-stups/specifications/tree/update/prob-examples/B/MobileComm

Code Availability

Our code is available via https://github.com/cchrewrite/ambm.

Authors' Contributions

- **Conceptualisation:** Cheng-Hao Cai, Jing Sun and Gillian Dobbie
- **Data curation:** Cheng-Hao Cai
- **Formal Analysis:** Cheng-Hao Cai and Hadrien Bride
- **Funding acquisition:** Jing Sun and Jin Song Dong
- **Investigation:** Cheng-Hao Cai, Gillian Dobbie, Zhé Hóu and Hadrien Bride
- **Methodology:** Cheng-Hao Cai, Jing Sun, Gillian Dobbie, Zhé Hóu, Hadrien Bride and Scott Uk-Jin Lee
- **Project administration:** Jing Sun and Jin Song Dong
- **Resources:** Jing Sun and Jin Song Dong
- **Software:** Cheng-Hao Cai, Zhé Hóu, Hadrien Bride and Jin Song Dong
- **Supervision:** Jing Sun, Gillian Dobbie and Jin Song Dong
- **Validation:** Cheng-Hao Cai
- **Visualisation:** Not applicable
- **Writing - original draft:** Cheng-Hao Cai
- **Writing - review & editing:** Jing Sun, Gillian Dobbie, Zhé Hóu, Hadrien Bride and Scott Uk-Jin Lee

1 INTRODUCTION

Automatic Software Repair (ASR) [4, 13] aims to use verification, testing and program synthesis techniques to assist humans to repair erroneous programs. In general, repairing software requires machines to locate faults before generating the repairs. Spectrum-based fault localisation has been heavily used to locate faults in imperative programs [1, 20]. It uses a set of I/O pairs to test a program and records traces of successful cases and failed cases. According to the occurrences of operations in the traces, suspicious code that causes the failed cases can be found, and candidate repairs can be generated using various techniques. For example, GenProg [21] and SCRepair [16] can generate mutation repairs; PASAN [34], AutoFix-E [26] and SPR [24] can use pre-defined template repairs to generate repairs; CASC [37], pyEDB [3] and GenProg [21] can use genetic programming to generate repairs. However, these ASR tools only focus on traditional test-based software development at the concrete code level. Little work has been done on ASR for correct-by-construction software development at the abstract design level, as much work in this field has focused on the computer-assisted diagnosis of faulty models. For example, a theory for identifying consistent behavioural modes in abstract models has been proposed by [11]. Moreover, Linear Temporal Logic (LTL) specifications, which are used to specify sequential properties of programs at the abstract design level, can be diagnosed using SAT encodings and reasoning [27]. However, model repair after diagnosis requires more investigation. In this work, we study automatic model repair techniques based on the B method.

The B method [2] is a formal software development method at the abstract design level, where design specifications are represented as abstract machines (called "models"). B has been used to develop a number of automatic railway control systems in France, Sweden, and USA [6], formalise the security properties of the L4 microkernel [15] and verify industrial PLC controllers [5]. The idea of B model repair is proposed by Schmidt et al. [32], and the goal of B model repair is to automatically (or semi-automatically) eliminate invariant violations and deadlocks in abstract machines. Moreover, Schmidt et al. [33] have developed a model repair approach that eliminates invariant violations by strengthening pre-conditions and relaxing invariants and eliminates deadlocks by weakening pre-conditions and generating new operations. This approach is semi-automatic because users are required to manually give I/O examples to synthesise new operations, and the code of new operations is constructed using a pre-defined program component library. Another automated B model repair approach is called B-repair [10], which uses machine learning techniques to learn the state spaces of abstract machines and select well-behaved repairs that preserve the original state spaces as much as possible. However, B-repair eliminates only one fault during each loop of repair, which means that repairing a large number of faults is time-consuming.

In this paper, we improve B-repair by implementing Abstract Machine Batch Modification (AMBM), which is more automatic and efficient than the previous B-repair. AMBM can repair multiple invariant violations at a time using simultaneous modifications, repair refactoring and better classifiers. AMBM inherits the concept of B-repair, which aims to repair design models at a high level of abstraction. The design models consist of operations describing changes in model states and invariants describing model properties, and the operations are expected to satisfy the properties with respect to the system requirements [2]. If the design models are logically verified to be correct, they can be further developed into executable software via different techniques, e.g., they can be rewritten as concrete models by refinement and finally converted to concrete programs. If faults exist in a design model, succeeding concrete models and final concrete programs can be faulty. Thus, repairing faulty design models is of great importance to the secure software development process.

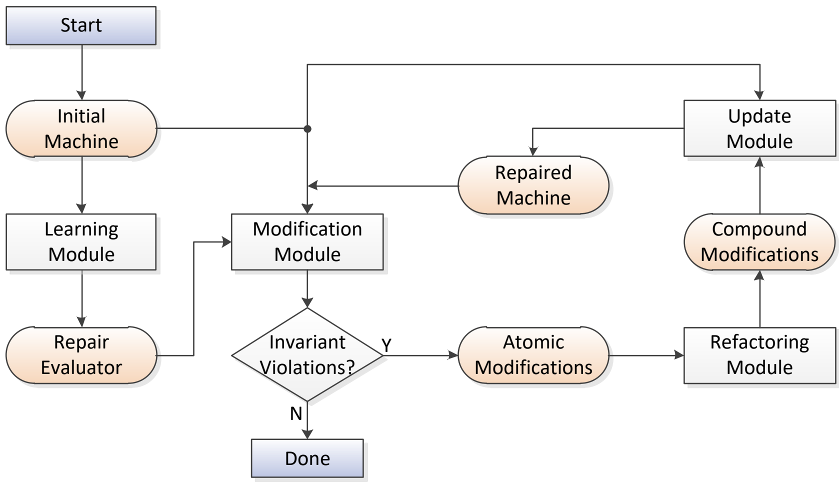


Fig. 1. The Overall Flow Chart of AMBM.

The workflow of AMBM is revealed in Fig. 1. Firstly, an initial abstract machine that specifies the design model, which possibly has invariant violations, is constructed by the user. The machine consists of initialisation and a number of operations. In the *learning module*, the state space of the machine is analysed using a model checker and learnt using a classifier, leading to a repair evaluator. We have the following two assumptions. Firstly, high-quality repairs are expected to retain the state space of the original model as much as possible. Secondly, instead of manual estimation, the quality of repair can be automatically estimated using the repair evaluator. Based on the above assumptions, the repair evaluator will be used to maximise the similarities of state space before and after repair. Next, in the *modification module*, the model checker is used to detect invariant violations in the machine. If invariant violations exist, a set of atomic modifications that can eliminate the invariant violations will be synthesised. These modifications are found via a constraint solver and selected using the repair evaluator. A modification will be selected if the repair evaluator predicts a high likelihood that the modification can lead to a minimal change in the state space. Each selected atomic modification can eliminate exactly one invariant violation. After that, in the *refactoring module*, the atomic modifications are simplified as compound modifications. Each compound modification can eliminate one or more invariant violations. Finally, in the *update module*, the compound modifications are applied to the initial abstract machine, leading to an updated machine. The updated machine will be forwarded to the modification module. If any invariant violations are detected, further modifications will be required; otherwise, the workflow terminates.

The contributions of this work include:

- B-repair [10] is extended by implementing batch modifications and integrating better machine learning models, which leads to higher speed and accuracy on B model repair tasks.
- A repair refactoring algorithm is introduced to generalise the code of modifications.

- An empirical study is conducted to demonstrate the accuracy, generality and efficiency of the extended model repair tool[1].

The rest of this paper is organised as follows. Section 2 revisits the B method, supervised machine learning and B-repair. Section 3 presents technical details of AMBM. Section 4 presents a case study on the use of AMBM to repair a faulty B model. Section 5 presents an empirical study of our approach. Section 6 discusses modifications on non-determinism and compares our study with related work. Section 7 concludes this work and outlines future directions.

2 PRELIMINARIES

This section revisits the B method, supervised machine learning and B-repair. *Emphasised words* are terminology that will be used in later discussions.

2.1 The B Method

The B method [2] is a correct-by-construction formal design modelling technique, where models are represented as abstract machines consisting of constants, variables, initialisations, operations, invariants and properties. *Constants* define unchangeable values in models, and *variables* define changeable values that record states of models. *Initialisations* can assign initial values to variables, and *operations* can generate new states by assigning new values to variables. *Invariants* describe conditions that all states must satisfy, and *properties* describe conditions that must be satisfied by the constants. The derivation and checking of model states can be achieved using model checkers such as the ProB tool [22]. Given an abstract machine that has N variables, the ProB model checker can approximate a *state space* that consists of a set of *state transitions* of the form $[v_1, v_2, \ldots, v_N] \xrightarrow{\alpha} [v'_1, v'_2, \ldots, v'_N]$, where α is an operation, $v_1, v_2, \ldots, v_N$ are the values of the variables before applying the operation, and $v'_1, v'_2, \ldots, v'_N$ are the values of the variables after applying the operation. In other words, the new state $[v'_1, v'_2, \ldots, v'_N]$ results from the application of the operation α to the existing state $[v_1, v_2, \ldots, v_N]$. When deriving such transitions, the model checker verifies whether all $v_1, v_2, \ldots, v_N$ and $v'_1, v'_2, \ldots, v'_N$ satisfy the given invariants. If not, an *invariant violation* will be triggered and reported. Initialisations can trigger invariant violations as well, but in this study we do not repair faulty initialisations.

Operations are the core components of abstract machines as they determine the derivation of transitions. They are described using different forms of substitutions, such as pre-conditioned substitutions and conditional substitutions. A *pre-conditioned substitution* is of the form PRE P THEN Q END, where P is a predicate, and Q is a substitution. P is a pre-condition that must be true for a state s when the operation is applied. If P is false for s, the operation will not be activated. A *conditional substitution* is of the form IF P THEN Q ELSE R END, where P is a predicate, and Q and R are substitutions. If P is true for a state s, Q will be applied. If P is false for s, R will be applied to s. In this study, the above two types of substitution are used to construct repair operators that can generally handle other types of substitution. Additionally, a pre-state and an operation can either deterministically lead to only one post-state, or non-deterministically lead to more than one post-state. In this work, we focus on such determinism and leave non-determinism as future work.

2.2 Supervised Machine Learning

Supervised machine learning aims at constructing a function that maps given input-output pairs [29]. Although supervised machine learning and the B method are rooted from two separate domains, the *vectorisation* techniques can bridge the gap between the two domains. According to [35] and [10], the states of B model can be converted into binary vectors by applying a set

[1] The source code is available via https://github.com/cchrewrite/ambm.

of pre-defined transformations to variables in the states. If a variable is an integer, a Boolean value, a distinct element or a first-order set, then it can be vectorised by one-hot encoding. In the vectorisation process, infinite types such as INTEGER and NATURAL are converted to finite sets by collecting all the values that occur in a state space and ignoring all unseen values so that such infinite types can be partially vectorised. The unseen values are ignored because supervised machine learning models usually cannot learn unseen values as they do not occur in training sets; therefore, unseen values should be excluded in order to avoid noise. Besides, in order to vectorise higher-order sets, sets in sets can be considered as string elements. Section 6.2 will discuss limitations and alternatives of the vectorisation method. Regardless of the number of variables, a state s can be vectorised by converting all variables in s to vectors and concatenating the vectors.

Using vectorisation, states of a B formal model can be represented as sequences of 0 and 1, so that they can be learnt using supervised learning models such as Bernoulli Naive Bayes (BNB) classifiers, Logistic Regression (LR) classifiers, Support Vector Machines (SVM), Random Forests (RF) and various neural network architectures [7, 14]. In particular, Silas is an explainable and verifiable classifier that learns patterns using random forests and applies automated reasoning techniques to explain and verify the learning results [8, 9]. Although the theories of these learning models differ from each other, their usages are similar. Each model must have a training algorithm and a prediction algorithm. The *training* algorithm takes as input a training set containing a list of vectors with their labels. Each label is an identifier representing exactly one class, and each vector has exactly one label. The training algorithm updates parameters of the model in order to map the vectors in the training set to their corresponding labels. The *prediction* algorithm takes as input a vector x and returns a vector $y = (y_1, \ldots, y_K)$ such that y_k $(k = 1, \ldots, K)$ is the likelihood that x is mapped to the kth label. In the next section, we will explain how to use the supervised machine learning models to learn the states of a B design model.

2.3 B-repair

B-repair [10] aims to use model checking, constraint solving and machine learning to search for repairs that solve invariant violations in B abstract machines. After the use of ProB [22] to detect a state transition that violates invariants, the constraint solver in ProB is used to suggest candidate repairs that change the state transition to satisfy the invariants. One of the core steps of B-repair is to use a *repair evaluator* based on binary classification to select repairs from the candidate repairs.

Model checkers can approximate an abstract machine using a set of state transitions, where each transition is of the form $S_{pre} \xrightarrow{\alpha} S_{post}$ and can be rewritten as a triple $[S_{pre}, S_{post}, \alpha]$ consisting of a pre-state S_{pre}, a post-state S_{post} and an operation α. The operation α consists of a pre-condition P and a post-condition Q (which is usually represented as a generalised substitution). The triple $[S_{pre}, S_{post}, \alpha]$ means that S_{pre} is a state satisfying P, and S_{post} is a state satisfying Q. The analysis of the state space can be converted into a classification problem, i.e., the triple $[S_{pre}, S_{post}, \alpha]$ can be classified into either a set of "possible" transitions S_P or a set of "impossible" transitions S_I. $[S_{pre}, S_{post}, \alpha]$ is in S_P if and only if $S_{pre} \xrightarrow{\alpha} S_{post}$ is a possible transition with respect to the machine. $[S_{pre}, S_{post}, \alpha]$ is in S_I if and only if $S_{pre} \xrightarrow{\alpha} S_{post}$ is impossible with respect to the machine. B-repair can use binary classifiers to learn the mapping from state transitions to S_P and S_I. The trained classifiers are considered as repair evaluators, i.e., given a repair, the classifiers use their prediction functions to output *repair scores* indicating the likelihood that the repair results in a state transition in S_P.

3 ABSTRACT MACHINE BATCH MODIFICATION

This section gives details on how B-repair is improved by implementing AMBM. AMBM reuses the learning and update modules of B-repair and adapts the modification module to support batch modifications. Additionally, a new refactoring module is used to simplify the code of batch modifications. Algorithm 1 describes the main function of AMBM. It takes as input a source B machine that contains invariant violations and outputs a repaired machine without any invariant violations. The algorithm consists of the learning phase (Line 1-2), the modification phase (Line 3-18), the refactoring phase (Line 19-23) and the update phase (Line 24-25), which are indicated using the "▷" symbols. The motivation and intuition of the four phases are as follows:

- The learning phase is used to train a classifier that learns the state space of the B machine. The trained classifier can be used to rank repairs. Without the ranking process, it will be difficult to select appropriate repairs.
- The modification phase is used to detect invariant violations and suggest repairs. As different candidate repairs are available, the classifier is used to rank the repairs.
- The refactoring phase is an optional process that can simplify the code of repair when multiple repairs are applied to an operation. Without simplification, the repair still works, but the resulting B machine may have tedious code.
- The update phase is used to update the B machine based on the suggested repairs.

Referring to Fig. 1 in Section 1, the four phases correspond to the learning module, the modification module, the refactoring module and the update module, which describe a single loop of modification. During the *learning phase*, the state space of the source machine is approximated using the ProB model checker [22] and used to train a repair evaluator. During the *modification phase*, invariant violations are detected and removed from the source machine. Firstly, the model checker is used to find all faulty transitions that trigger invariant violations. Secondly, the constraint solver embedded in ProB is used to randomly compute a set of candidate states that satisfy all invariants in the source machine. Thirdly, a set of candidate atomic modifications, which can repair single faulty transitions, are produced using the candidate states. (To understand how the candidate atomic modifications are generated, refer to the `Atomic-Modifications` function and the `Update` function in Section 3.1.) Their repair scores are estimated using the learnt classifier. For each faulty transition, an atomic modification with the highest repair score is selected. Fourthly, the source machine is updated using all selected modifications. The modification phase is repeated until no faulty transitions can be found. During the *refactoring phase*, atomic modifications applied to each operation are collected and rewritten using Algorithm 2, resulting in a set containing compound modifications and atomic modifications that cannot be refactored. Finally, during the *update phase*, the source machine is changed using the modifications, and the updated machine is returned.

Algorithm 2 is an algorithm that rewrites atomic modifications into compound modifications. It takes as input a set of atomic modifications and outputs a set of compound modifications. Firstly, the atomic modifications are converted to atomic modification predicates (which are generated using the `Modifications-To-Predicates` function in Section 3.1). Secondly, Context-Free Grammars (CFG) are used to generate a set of relation predicates describing possible relations between the pre- and post-states. Thirdly, candidate relation predicates that are satisfied by each atomic modification predicate are collected. According to the candidate relation predicates, the atomic modification predicates are split into two partitions. The first partition P_B includes all atomic modification predicates that satisfy a common relation predicate. The second partition P_A includes all atomic modification predicates that do not satisfy the relation predicate. Finally, the above partition process is applied to P_A iteratively until no further partitions can be produced. If such a partition cannot

Algorithm 1 Abstract Machine Batch Modification

Input: source machine M_S
Parameter: the maximum number of candidate states N_X, classifier type T_W
Output: repaired machine M_R

1: $D_S \leftarrow$ `State-Space`(M_S) ▷ *Learning Phase*
2: $W \leftarrow$ `Repair-Evaluator-Training`(D_S, T_W)
3: $X \leftarrow$ `Invariant-Solutions`(M_S, N_X) ▷ *Modification Phase*
4: $M_T \leftarrow M_S$
5: $R_{All} \leftarrow \emptyset$
6: **while** *True* **do**
7: $\quad D_T \leftarrow$ `State-Space`(M_T)
8: $\quad T_F \leftarrow$ `Faulty-Transitions`(D_T)
9: $\quad$**if** $T_F = \emptyset$ **then**
10: $\quad\quad$**break**
11: $\quad$**end if**
12: $\quad S_M \leftarrow$ `Correct-States`$(D_T) \cup X$
13: $\quad R_M \leftarrow$ `Atomic-Modifications`(T_F, S_M)
14: $\quad P_M \leftarrow$ `Repair-Scores`(R_M, W)
15: $\quad R_C \leftarrow$ `Modification-Selection`(R_M, P_M)
16: $\quad M_T \leftarrow$ `Update`(M_T, R_C)
17: $\quad R_{All} \leftarrow R_{All} \cup R_C$
18: **end while**
19: $R_U \leftarrow \emptyset$ ▷ *Refactoring Phase*
20: **for** α in `Operations`(M_S) **do**
21: $\quad R_\alpha \leftarrow$ `Collect-Modifications`(R_{All}, α)
22: $\quad R_U \leftarrow R_U \cup$ `Refactoring`(R_α) (Algorithm 2)
23: **end for**
24: $M_R \leftarrow$ `Update`(M_S, R_U) ▷ *Update Phase*
25: **return** M_R

be produced, the partitions will be converted to compound modifications using their relation predicates.

In order to help readers understand the algorithms, the following subsection provides details of core functions used in Algorithm 1 and Algorithm 2. The functions are listed in order of line numbers in the algorithms.

3.1 Core Functions

Algorithm 1 includes the following functions.

- $D \leftarrow$ `State-Space`(M) (in Line 1 and Line 7) returns the state space D of a given abstract machine M. It is a function of the ProB model checker. The given abstract machine must be finite with respect to its invariant and have no deadlock states. In order to approximate a finite state space, ProB is run to generate all states that satisfy the invariant and freeze all states that violate the invariant. In particular, if ProB detects any states violating the invariant, ProB will be controlled to check other normal states rather than stopping at the violation points. The resulting finite state space D is converted to a list of triples. Each triple is of the form $[S, S', \alpha]$, where S is a pre-state of the form $[x_1, x_2, \ldots, x_N]$, S'

Algorithm 2 Modification Refactoring

Input: atomic modification set M_A
Parameter: maximum depth of CFG predicates D_{CFG}
Output: compound modification set M_C

$\alpha \leftarrow$ `Get-Operation`(M_A)
$P_A \leftarrow$ `Modifications-To-Predicates`(M_A)
$P_{CFG} \leftarrow$ `CFG-Predicates`(M_A, D_{CFG})
$M_C \leftarrow \varnothing$
while $P_A \neq \varnothing$ **do**
  $W_S \leftarrow$ `Candidate-Predicates`(P_A, P_{CFG})
  $[P_B, P_A] \leftarrow$ `Best-Partition`(P_A, W_S)
  $M_C \leftarrow M_C \cup \{$`Compound-Modification`$(P_B, \alpha)\}$
end while
return M_C

is a post-state of the form $[x'_1, x'_2, \ldots, x'_N]$, and α is an operation. The triple means that $[x_1, x_2, \ldots, x_N] \xrightarrow{\alpha} [x'_1, x'_2, \ldots, x'_N]$ is a possible transition of M, where x_i $(i = 1, \ldots, N)$ and x'_i $(i = 1, \ldots, N)$ are values of variables in M. The above form of triples is consistently used in our functions.

- $W \leftarrow$ `Repair-Evaluator-Training`(D, T) (in Line 2) learns a binary classifier of type T for a state space D and returns the learnt classifier W. The learning of the classifier is performed by the following steps. Firstly, a set of triples Z is randomly produced such that $|Z| = |D|$ and $D \cap Z = \varnothing$. Secondly, the triples in D and Z are vectorised as features with labels. Features of D are labelled as "0" (i.e., possible transitions). Features of Z are labelled as "1" (i.e., impossible transitions). Thirdly, the features with labels are learnt using a classifier of the type T. T can be BNB, LR, SVM, RF or Silas. Regardless of the theories of these classifiers, each classifier has a training algorithm implemented as a method, `fit`(X, Y), that takes as input a list of features X with a list of labels Y and updates parameters of the classifier in order to fit X and Y. The goal of fitting is to map features in X to their corresponding labels in Y as much as possible. For our repair evaluator training function, the method `fit` is used to learn the mappings between the features and the labels of the triples. Finally, the learnt classifier is returned.
- $X \leftarrow$ `Invariant-Solutions`(M, N) (in Line 3) computed at most N states satisfying the invariant of an abstract machine M and returns a set X containing the N computed states. The function consists of the following steps. Firstly, the invariant of M is extracted. Secondly, the invariant is converted into a constraint where all variables in M are considered unknowns. Finally, the constraint solver of ProB searches solutions satisfying the constraint in random order and returns N solutions, where each solution is a state $[x_1, x_2, \ldots, x_N]$ satisfying the invariant of M. The purpose of this function is to find candidate components (i.e., modified states) for producing atomic modifications.
- $T \leftarrow$ `Faulty-Transitions`(D) (in Line 8) returns a set T containing all faulty transitions in the state space D. D must be produced using the function $D \leftarrow$ `State-Space`(M). Note that D is assumed to have no deadlock states. If ProB detects an invariant violation, the computation of the state space will stop and only one faulty state can be reported. In order to report all invariant violations, we use a trick to control ProB to develop a whole state

space and collect all invariant violations, i.e., all states that trigger invariant violations are frozen, and all states that have no outgoing transitions are reported.

- $S \leftarrow$ `Correct-States`(D) (in Line 12) returns a set S containing all correct states in the state space D. D must be produced using the function $D \leftarrow$ `State-Space`(M). Correct states in D are collected by finding all states with at least one outgoing transition.
- $R \leftarrow$ `Atomic-Modifications`(T, S) (in Line 13) takes as input a set of faulty transitions T and a set of correct states S. Any transition $S_{pre} \xrightarrow{\alpha} S_{post} \in T$ and any state $S_{mod} \in S$ correspond to an atomic modification $[\alpha, S_{pre}, S_{post}, S_{mod}]$, which means that changing $S_{pre} \xrightarrow{\alpha} S_{post}$ to $S_{pre} \xrightarrow{\alpha} S_{mod}$ can eliminate the faulty state S_{post}. All possible atomic modifications are collected into a set R and returned. The purpose of this function is to synthesise atomic modifications that can eliminate invariant violations.
- $P \leftarrow$ `Repair-Scores`(R, W) (in Line 14) predicts repair scores of a set of atomic modifications R via a classifier W. The classifier must be produced using the function $W \leftarrow$ `Repair-Evaluator-Training`(D, T). The repair scores are computed via the following steps. Firstly, each atomic modification $[\alpha, S_{pre}, S_{post}, S_{mod}]$ is reduced to a triple $[S_{pre}, S_{mod}, \alpha]$. Secondly, $[S_{pre}, S_{mod}, \alpha]$ is vectorised as a binary feature x_0. Thirdly, the classifier W is used to predict the repair score of x_0. Regardless of the type of W, it must have a prediction algorithm implemented as a method `predict`(x, y) that takes as input a feature x and a label y and returns the likelihood that x is mapped to y. For our repair evaluator training algorithm, the repair score of the triple is the value of `predict`$(x_0, \text{“0”})$. Finally, a list P containing the repair scores of all the modifications is returned.
- $S \leftarrow$ `Modification-Selection`(R, P) (in Line 15) selects modifications in a list of atomic modifications R with a list of repair scores P and returns a list of selected modifications S. The ith number in P is the repair score of the ith modification in R. The selecting process has the following steps. Firstly, modifications in R are sorted by their repair scores in descending order. Secondly, for any modification $[\alpha, S_{pre}, S_{post}, S_{mod}]$ in the sorted R, if α, S_{pre} and S_{post} is the first occurrence, $[\alpha, S_{pre}, S_{post}, S_{mod}]$ will be considered as the best modification for the transition $S_{pre} \xrightarrow{\alpha} S_{post}$. Finally, all the best modifications are collected into a list S and returned. The purpose of this function is to find the best atomic modifications, i.e., for each invariant violation, the trained classifier is used to estimate the repair scores of all applicable candidate modifications, and only the modification with the highest repair score will be selected.
- $U \leftarrow$ `Update`(M, R) (in Line 16 and 24) updates an abstract machine M using a list of modifications R and returns an updated machine U. A modification is applied to an operation α via a pair (P', Y'), where P' is a condition, and Y' is a substitution. For an atomic modification $[\alpha, S_{pre}, S_{post}, S_{mod}]$, P' is the predicate form of S_{pre}, and Y' is the substitution form of S_{mod}. For a compound modification $[\alpha, P, Y]$, P' is the predicate form of P, and Y' is the substitution form of Y. If α is a substitution T without any pre-conditions, applying the modification to α will lead to a conditional substitution as follows:

$$\text{IF } not(P') \text{ THEN } T \text{ ELSE } Y' \text{ END} \tag{1}$$

If α is a pre-conditioned substitution “PRE S THEN T END”, applying the modification to α leads to a pre-conditioned substitution as follows.

$$\begin{array}{l} \text{PRE } S \text{ THEN} \\ \quad \text{IF } not(P') \text{ THEN } T \text{ ELSE } Y' \text{ END} \\ \text{END} \end{array} \tag{2}$$

Moreover, when N ($N \geq 1$) modifications $(P'_1, Y'_1), \ldots, (P'_N, Y'_N)$, where P'_i $(i = 1, \ldots, N)$ is a condition and Y'_i $(i = 1, \ldots, N)$ is a substitution, are applied to the same operation with a substitution T and without any pre-conditions, the following template can be used:

$$\begin{array}{l} \text{IF } P'_1 \text{ THEN } Y'_1 \\ \ldots \\ \text{ELSIF } P'_N \text{ THEN } Y'_N \\ \text{ELSE } T \\ \text{END} \end{array} \tag{3}$$

If the operation has a pre-condition S and a substitution T, the following template can be used:

$$\begin{array}{l} \text{PRE } S \text{ THEN} \\ \quad \text{IF } P'_1 \text{ THEN } Y'_1 \\ \quad \ldots \\ \quad \text{ELSIF } P'_N \text{ THEN } Y'_N \\ \quad \text{ELSE } T \\ \quad \text{END} \\ \text{END} \end{array} \tag{4}$$

After applying all modifications to M, the resulting machine U is returned.

- $S \leftarrow$ `Collect-Modifications`(R, α) (in Line 21) takes as input a set of modifications R and an operation α and returns a set S containing all modifications that are in R and can be applied to α.
- $S \leftarrow$ `Refactoring`(R) (in Line 22) takes as input a set of atomic modifications R and returns a set of compound modifications S via Algorithm 2. Each compound modification is of the form $[\alpha, P, Y]$, where α is an operation, P is a list of pre-states, and Y is a list of substitutions reflecting the relations between the pre-states and the post-states. The purpose of this function is to simplify atomic modifications.

Algorithm 2 includes the following functions.

- $\alpha \leftarrow$ `Get-Operation`(M) (in Line 1) takes as input a set of atomic modifications M. If all modifications correspond to a certain operation α, then α is returned. If the modifications correspond to two or more operations, an error is raised.
- $U \leftarrow$ `Modifications-To-Predicates`(M) (in Line 2) converts a set of atomic modifications M to a set of predicates U. Each atomic modification is of the form $[\alpha, S_{pre}, S_{post}, S_{mod}]$, suggesting that a transition $S_{pre} \xrightarrow{\alpha} S_{post}$ should be changed to $S_{pre} \xrightarrow{\alpha} S_{mod}$. Only S_{pre} and S_{mod} need to be converted to predicate forms. Suppose that S_{pre} and S_{mod} are $[x_1, \ldots, x_N]$ and $[y_1, \ldots, y_N]$ respectively, and v_i^{pre} and v_i^{mod} $(i = 1, \ldots, N)$ are identifiers of variables in the pre-state P and the modified state Y respectively. The predicate forms of P and Y are $v_1^{pre} = x_1 \wedge \ldots \wedge v_N^{pre} = x_N$ and $v_1^{mod} = y_1 \wedge \ldots \wedge v_N^{mod} = y_N$ respectively. Thus, the predicate form of $[\alpha, S_{pre}, S_{post}, S_{mod}]$ is the conjunction of the above two predicates, which is $v_1^{pre} = p_1 \wedge \ldots \wedge v_N^{pre} = p_N \wedge v_1^{mod} = y_1 \wedge \ldots \wedge v_N^{mod} = y_N$. This predicate is called a *modification predicate*. The purpose of producing modification predicates is to enable the constraint solving function in ProB to find relationships between pre-states and modified states.
- $P \leftarrow$ `CFG-Predicates`(M, D) (in Line 3) takes as input a set of atomic modifications M and a search depth D and synthesises a set of predicates P using Context-Free Grammars

(CFG). The CFGs are constructed using: (1) the identifiers of variables in the pre-states and the modified states, including v_i^{pre} and v_i^{mod} $(i = 1, \ldots, N)$, (2) values of variables that occur in M, and (3) the B operators such as arithmetic, Boolean and set operators. The maximum depth of synthesised predicates is D. Synthesised predicates are of the form $v_i^{mod} = F(v_1^{pre}, \ldots, v_N^{pre})$, where F is a function. The synthesised predicates are called *CFG predicates*. The purpose of synthesising CFG predicates is to find candidate predicates that represent relations between pre-states and modified states. By default, the maximum depth of the CFG predicate is set to 3, and the number of candidate predicates is set to 1,000. Each variable can match at least one candidate predicate because a variable v_i with a value x_i can be directly converted to a predicate $v_i^{mod} = x_i$.

- $W \leftarrow$ `Candidate-Predicates`(X, P) (in Line 6) takes as input a set of modification predicates X and a set of CFG predicates P and returns a set W containing candidate pairs that are of the form (R, S) such that $R \in X$, $S \in P$, and S is a candidate predicate of R. To obtain such pairs, for any $R \in X$ and any $S \in P$, the constraint solving function in the ProB model checker is used to resolve $R \wedge S$. If $R \wedge S$ is true, S will be considered as a candidate predicate of R. The pair (R, S) will be a member of W and is called a *candidate pair*. After obtaining all candidate pairs, W is returned. The purpose of finding candidate pairs is to discover hidden relations between pre-states and modified states.
- $[P_B, P_A] \leftarrow$ `Best-Partition`(X, W) (in Line 7) takes as input a set of modification predicates X and a set of candidate pairs W. Recall the explanations of `Modifications-To-Predicates` and `CFG-Predicates`. We continue to use the notation of modification predicates $v_1^{pre} = p_1 \wedge \ldots \wedge v_N^{pre} = p_N \wedge v_1^{mod} = y_1 \wedge \ldots \wedge v_N^{mod} = y_N$ and the notation of candidate predicates $v_i^{mod} = F(v_1^{pre}, \ldots, v_N^{pre})$. For each v_i^{mod} $(i = 1, \ldots, N)$, a candidate predicate U_i is found in W such that:
  - $U_1 \wedge \ldots \wedge U_N$ is true for all predicates in a subset $X_1 \subseteq X$,
  - $U_1 \wedge \ldots \wedge U_N$ is false for all predicates in a subset $X_2 \subseteq X$,
  - $X_1 \cup X_2 = X \wedge X_1 \cap X_2 = \emptyset$, and
  - the cardinality of X_1 is maximised.

  In the above process, $U_1 \wedge \ldots \wedge U_N$ is called a *compound modification predicate*, which uses a set of CFG predicates to describe atomic modifications. After finding such X_1 and X_2, the best partition $P_B = [X_1, [U_1, \ldots, U_N]]$ and a partition $P_A = X_2$ that contains all the remaining atomic modifications will be returned. The purpose of this function is to find common relations between pre-states and modified states.
- $Z \leftarrow$ `Compound-Modification`(P_B, α) (in Line 8) converts the partition $P_B = [X_1, [U_1, \ldots, U_N]]$, which is produced using `Best-Partition`, to a compound modification Z for an operation α. Z is of the form $[\alpha, P, [U_1, \ldots, U_N]]$, where P is a set containing all pre-states covered by X_1. The purpose of this function is to synthesis modifications using common relations between pre-states and modified states.

3.2 Implementation

AMBM is implemented by extending B-repair [10]. Its main dependencies include ProB 1.7.1 [22], scikit-learn 0.19.2 [25] and Silas Edu 0.8.5 [8, 9]. The model checker in ProB is used to approximate state spaces of abstract machines and detect invariant violations. The constraint solver in ProB is used to find candidate modifications. The constraint solving function in ProB is used to find relations between pre-states and modified states. Regarding scikit-learn, it provides well-behaved classifiers, including BNB, LR, SVM and RF, as well as training and prediction functions that can be directly used for the purpose of repair evaluator training. Besides, Silas Edu provides an improved

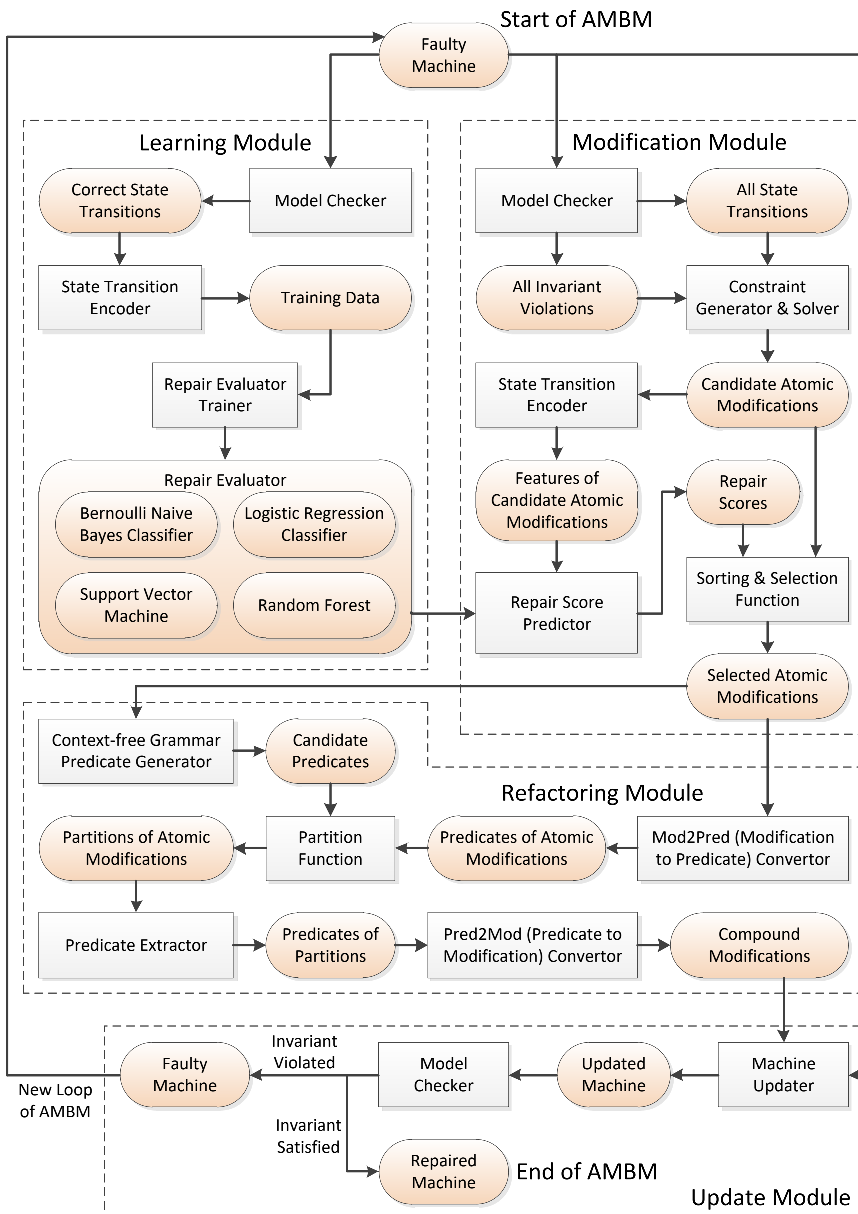


Fig. 2. The Implementation of AMBM.

implementation of random forest and corresponding training and prediction functions. In our tool, these classifiers are used as black boxes. The source code of the tool can be downloaded[2].

[2]Silas Edu is developed by us and can be downloaded from https://www.depintel.com/silas_download.html.

Fig. 2 shows the architecture of the implemented AMBM tool. The tool consists of four modules that correspond to the four phases of Algorithm 1. The following descriptions provide details of the four modules.

- The *learning module* trains a repair evaluator for a given abstract machine. To obtain the training data, correct state transitions of the abstract machine are approximated using the ProB model checker. The state transitions are vectorised as training data using the state transition encoder in B-repair. Repair evaluators can be a Bernoulli naive Bayes classifier, a logistic regression classifier, a support vector machine, or a random forest. Training functions for these classifiers are inherited from scikit-learn and Silas.
- The *modification module* generates atomic modifications for the abstract machine. In this module, the abstract machine is checked using the ProB model checker, and all state transitions and all invariant violations are collected. After analysing the state transitions and the invariant violations, the constraint generator and solver work out candidate atomic modifications that can remove all the invariant violations from the state transitions. Features of the candidate atomic modifications are computed using the state transition encoder in B-repair. These features and the trained repair evaluator are used to predict repair scores in the repair score predictor. The predictor makes use of the prediction functions in scikit-learn and Silas to estimate the repair scores. After that, the candidate atomic modifications are sorted by their repair scores, and those with high repair scores are selected.
- The *refactoring module* converts the selected atomic modifications to compound modifications. First, the Mod2Pred convertor rewrites the atomic modifications to their predicate forms, and the context-free grammar predicate generator generates candidate predicates from predefined context-free grammars of B and types of variables in the atomic modifications. Then the atomic modifications are clustered into a number of partitions by the satisfiability between the predicate forms of the atomic modifications and the candidate predicates. Next, the predicate extractor collects predicates of each partition. Finally, these predicates are converted to compound modifications via the Pred2Mod convertor.
- The *update module* uses a machine updater to apply the compound modifications to the original abstract machine and uses the ProB model checker to check the correctness of the updated abstract machine. If the model checker does not report any invariant violation, the AMBM tool will terminate and return the abstract machine. Otherwise, a new loop of AMBM is started to eliminate invariant violations in the abstract machine.

In Sections 4 and 5, the AMBM tool is used to conduct experiments.

4 A CASE STUDY ON AMBM

This section provides a case study to explain Algorithm 1 and Algorithm 2. The two algorithms are used to repair the bus control model in Fig. 3. The model has the following features:

- Buses are numbered as $1, 2, \ldots, N$.
- "Selected_Bus" is a variable that denotes the ID of a selected bus that is being controlled. When Selected_Bus = 0, no bus is selected.
- "Current_Location(i)" is a variable that denotes the current location of the ith bus.
- "Next_Location(i)" is a variable that denotes the next scheduled location of the ith bus.
- "Moving(i)" is a variable that denotes whether or not the ith bus is moving.
- $St1$, $St2$, $St3$ and $St4$ are stations, and $Airport$ is an airport.
- Initially, all buses are at $St1$, not moving and not scheduled.
- "Bus_Selector" is an operation that selects a bus to send a signal.

```
MACHINE Bus_Control
SETS Location = {St1, St2, St3, St4, St5, Airport}
CONSTANTS Map, N
PROPERTIES Map = {(St1,St2), (St2,St3), (St1,St3),
  (St3,St4), (St4,St5), (St5,Airport), (Airport,St1)} & N = 2
VARIABLES Selected_Bus, Moving, Current_Location, Next_Location
INVARIANT Selected_Bus : 0..N & Moving : 1..N --> BOOL &
  Current_Location : 1..N --> Location & Next_Location : 1..N --> Location &
  !(ii,jj).(ii : 1..N & jj : 1..N & not(ii = jj) & Current_Location(ii) = St4 &
  Moving(ii) = FALSE => not(Current_Location(jj) = St4 & Moving(jj) = FALSE))
INITIALISATION Selected_Bus := 0; Moving := (1..N) * {FALSE};
  Current_Location := (1..N) * {St1}; Next_Location := (1..N) * {St1}
OPERATIONS
  Bus_Selector =
    ANY Bus_ID WHERE Bus_ID : 1..N & Selected_Bus = 0
    THEN Selected_Bus := Bus_ID
    END;
  Signal_Sender =
    ANY Destination WHERE not(Selected_Bus = 0) &
      Destination : Location & (Current_Location(Selected_Bus),Destination) : Map
    THEN Next_Location(Selected_Bus) := Destination ; Selected_Bus := 0
    END;
  Bus_Controller =
    PRE not(Selected_Bus = 0) &
      not(Next_Location(Selected_Bus) = Current_Location(Selected_Bus))
    THEN
      IF Moving(Selected_Bus) = FALSE
      THEN Moving(Selected_Bus) := TRUE
      ELSE Current_Location(Selected_Bus) := Next_Location(Selected_Bus) ;
        Moving(Selected_Bus) := FALSE
      END;
      Selected_Bus := 0
    END
END
```

Fig. 3. A Bus Control Model

- "Signal_Sender" is an operation that sends a signal to the selected bus to schedule a destination.
- "Bus_Controller" is an operation that moves the selected bus to the scheduled destination.

In the model, a bus controller is described by the Bus_Controller operation. It has a pre-conditioned substitution of the form PRE ... THEN ... END, where the predicate between PRE and THEN is a pre-condition meaning that if the selected bus does not reach its next scheduled location, the bus controller will start controlling it to reach the location. The substitution between the first THEN and the last END describes how to change the state of the bus. If the bus is not moving, then it starts moving to the next location. If the bus is on the way to the next location, it will stop at the next location. The operation is required to satisfy the invariant, where variable identifiers "ii" and "jj" are used instead of "i" and "j" because single letters are not allowed to be identifiers in some B model checkers. The last two lines of the invariant mean that St4 can hold at most one bus, but the invariant is violated because the operation allows two buses to stop at St4. We use Algorithm 1 to solve such invariant violations.

4.1 The Learning Phase

We used the case of $N = 2$ as an example, i.e., the bus controller controlled two buses with IDs 1 and 2. In the learning phase, the `State-Space` function can approximate state transitions of the model. To aid the readability, we recorded a state as [Selected_Bus, Moving(1), Moving(2), Current_Location(1), Current_Location(2), Next_Location(1), Next_Location(2)], and "TRUE" and "FALSE" are recorded as "T" and "F" respectively. Due to the large state space, we give the following state transitions as examples, where changed values are underlined.

- $[0, F, F, St1, St1, St1, St1] \xrightarrow{Bus\_Selector} [\underline{1}, F, F, St1, St1, St1, St1]$
- $[1, F, F, St1, St1, St1, St1] \xrightarrow{Signal\_Sender} [\underline{0}, F, F, St1, St1, \underline{St2}, St1]$
- $[0, F, F, St1, St1, St2, St1] \xrightarrow{Bus\_Selector} [\underline{1}, F, F, St1, St1, St2, St1]$
- $[1, F, F, St1, St1, St2, St1] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{T}, F, St1, St1, St2, St1]$
- $[0, T, F, St1, St1, St2, St1] \xrightarrow{Bus\_Selector} [\underline{1}, T, F, St1, St1, St2, St1]$
- $[1, T, F, St1, St1, St2, St1] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St2}, St1, St2, St1]$

The above state transitions show how the first bus moves from St0 to St1, and they are considered possible state transitions.

Using the `Repair-Evaluator-Training` function, a set of impossible state transitions can be generated so that a binary classifier can be trained to distinguish the two classes of state transitions. Due to a large number of impossible state transitions, we give the following examples.

- $[0, F, F, St1, St1, St1, St1] \xrightarrow{Bus\_Selector} [0, F, F, St1, St1, \underline{St2}, St1]$
- $[1, F, F, St1, St1, St1, St1] \xrightarrow{Signal\_Sender} [\underline{0}, \underline{T}, F, St1, St1, \underline{St2}, St1]$
- $[0, F, F, St1, St1, St2, St1] \xrightarrow{Bus\_Selector} [\underline{1}, \underline{T}, F, St1, St1, St2, St1]$
- $[1, F, F, St1, St1, St2, St1] \xrightarrow{Bus\_Controller} [1, F, F, St1, St1, \underline{St1}, St1]$
- $[0, T, F, St1, St1, St2, St1] \xrightarrow{Bus\_Selector} [0, \underline{F}, F, St1, St1, \underline{St1}, St1]$
- $[1, T, F, St1, St1, St2, St1] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, St1, St1, St2, St1]$

To avoid unnecessary details, we suggest readers refer to [10] for details of repair evaluator training. The trained classifier is stored for future use.

4.2 The Modification Phase

In the modification phase, in order to produce atomic modifications, the `Invariant-Solutions` function can compute a set of solutions satisfying the invariant. For example:

- $[0, F, F, St3, St4, St4, St4]$
- $[0, F, F, St3, St4, St4, St5]$
- $[0, F, F, St4, St3, St4, St4]$
- $[0, F, F, St4, St3, St5, St4]$

Next, the model is iteratively repaired until no further invariant violations can be detected. To detect invariant violations, the `State-Space` function is used to approximate the state space of the current model, and the `Faulty-Transitions` function is used to collect invariant violations in the state space. The following four faulty state transitions are found.

- $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St4}, St4, St4, St4]$
- $[1, T, F, St3, St4, St4, St5] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St4}, St4, St4, St5]$
- $[2, F, T, St4, St3, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, F, \underline{F}, St4, \underline{St4}, St4, St4]$
- $[2, F, T, St4, St3, St5, St4] \xrightarrow{Bus\_Controller} [\underline{0}, F, \underline{F}, St4, \underline{St4}, St5, St4]$

The above state transitions violate the invariant because they allow the two buses to stop at St4 at the same time. To repair the above state transitions, their post-states will be replaced with other candidate post-states satisfying the invariant. All correct states in the state space, which are collected using the `Correct-States` function, and the states computed using the `Invariant-Solutions` function, are considered as candidate post-states. For each faulty state transition, the `Atomic-Modifications` function can use the candidate post-states to generate a set of candidate modifications, and the `Repair-Score` function can estimate their repair scores. For example, to repair $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St4}, St4, St4, St4]$ using the four solutions, the following candidate modifications with repair scores (σ) are generated.

- $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, St3, St4, St4, St4]$ ($\sigma = 0.458$)
- $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, St3, St4, St4, \underline{St5}]$ ($\sigma = 0.274$)
- $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St4}, \underline{St3}, St4, St4]$ ($\sigma = 0.403$)
- $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St4}, \underline{St3}, \underline{St5}, St4]$ ($\sigma = 0.159$)

As the first modification has the highest σ value, it is selected to repair the model. Similarly, for all the faulty state transitions, the `Modification-Selection` function can select the following modifications, which are of the form "pre-state $\xrightarrow{\text{operation}}$ faulty post-state $\hookrightarrow$ modified post-state".

- $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller}$ $[\underline{0}, \underline{F}, F, \underline{St4}, St4, St4, St4] \hookrightarrow [\underline{0}, \underline{F}, F, St3, St4, St4, St4]$
- $[1, T, F, St3, St4, St4, St5] \xrightarrow{Bus\_Controller}$ $[\underline{0}, \underline{F}, F, \underline{St4}, St4, St4, St5] \hookrightarrow [\underline{0}, \underline{F}, F, St3, St4, St4, St5]$
- $[2, F, T, St4, St3, St4, St4] \xrightarrow{Bus\_Controller}$ $[\underline{0}, F, \underline{F}, St4, \underline{St4}, St4, St4] \hookrightarrow [\underline{0}, F, \underline{F}, St4, St3, St4, St4]$
- $[2, F, T, St4, St3, St5, St4] \xrightarrow{Bus\_Controller}$ $[\underline{0}, F, \underline{F}, St4, \underline{St4}, St5, St4] \hookrightarrow [\underline{0}, F, \underline{F}, St4, St3, St5, St4]$

The `Update` function can use the above atomic modifications to repair the Bus_Controller operation, leading to the repaired operation in Fig. 4. As a result, the repaired operation no longer triggers any invariant violations. A problem is that each atomic modification can only remove one invariant violation, resulting in tedious code. The code can be simplified in the refactoring phase.

4.3 The Refactoring Phase

The atomic modifications are refactored using the `Refactoring` function, i.e., Algorithm 2. We use $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St4}, St4, St4, St4] \hookrightarrow [\underline{0}, \underline{F}, F, St3, St4, St4, St4]$ as an example. The `Modifications-To-Predicates` function converts the pre-state $[1, T, F, St3, St4, St4, St4]$ and the modified state $[0, F, F, St3, St4, St4, St4]$ to the predicate "Pre_Selected_Bus = 0 & Pre_Moving(1) = FALSE & Pre_Moving(2) = FALSE &

```
Bus_Controller =
  PRE not(Selected_Bus = 0) &
    not(Next_Location(Selected_Bus) = Current_Location(Selected_Bus))
  THEN
    IF Selected_Bus = 1 & Moving(1) = TRUE & Moving(2) = FALSE &
      Current_Location(1) = St3 & Current_Location(2) = St4 &
      Next_Location(1) = St4 & Next_Location(2) = St4
    THEN Selected_Bus := 0; Moving(1) := FALSE; Moving(2) := FALSE;
      Current_Location(1) := St3; Current_Location(2) := St4;
      Next_Location(1) := St4; Next_Location(2) := St4
    ELSIF Selected_Bus = 1 & Moving(1) = TRUE & Moving(2) = FALSE &
      Current_Location(1) = St3 & Current_Location(2) = St4 &
      Next_Location(1) = St4 & Next_Location(2) = St5
    THEN Selected_Bus := 0; Moving(1) := FALSE; Moving(2) := FALSE;
      Current_Location(1) := St3; Current_Location(2) := St4;
      Next_Location(1) := St4; Next_Location(2) := St5
    ELSIF Selected_Bus = 2 & Moving(1) = FALSE & Moving(2) = TRUE &
      Current_Location(1) = St4 & Current_Location(2) = St3 &
      Next_Location(1) = St4 & Next_Location(2) = St4
    THEN Selected_Bus := 0; Moving(1) := FALSE; Moving(2) := FALSE;
      Current_Location(1) := St4; Current_Location(2) := St3;
      Next_Location(1) := St4; Next_Location(2) := St4
    ELSIF Selected_Bus = 2 & Moving(1) = FALSE & Moving(2) = TRUE &
      Current_Location(1) = St4 & Current_Location(2) = St3 &
        Next_Location(1) = St5 & Next_Location(2) = St4
    THEN Selected_Bus := 0; Moving(1) := FALSE; Moving(2) := FALSE;
      Current_Location(1) := St4; Current_Location(2) := St3;
      Next_Location(1) := St5; Next_Location(2) := St4
    ELSE
      IF Moving(Selected_Bus) = FALSE
      THEN Moving(Selected_Bus) := TRUE
      ELSE Current_Location(Selected_Bus) := Next_Location(Selected_Bus);
        Moving(Selected_Bus) := FALSE
      END;
      Selected_Bus := 0
    END
  END
```

Fig. 4. The Repaired Bus Controller Operation Before Refactoring

Pre_Current_Location(1) = St3 & Pre_Current_Location(2) = St4 & Pre_Next_Location(1) = St4 & Pre_Next_Location(2) = St4 & Mod_Selected_Bus = 0 & Mod_Moving(1) = FALSE & Mod_Moving(2) = FALSE & Mod_Current_Location(1) = St3 & Mod_Current_Location(2) = St4 & Mod_Next_Location(1) = St4 & Mod_Next_Location(2) = St4".

Next, the `CFG-Predicates` function generates a set of predicates using context-free grammars, e.g., Mod_Selected_Bus = 0, Mod_Selected_Bus = 1, Mod_Selected_Bus = Pre_Selected_Bus, Mod_Selected_Bus = 1 - Pre_Selected_Bus, Mod_Moving(1) = TRUE, Mod_Moving(1) =

FALSE, Mod_Next_Location(1) = St4, Mod_Next_Location(1) = St5 and Mod_Next_Location(1) = Pre_Next_Location(2). Based on the predicates, the `Candidate-Predicates` function will find candidate predicates that describe the modifications. For example, the modification $[1, T, F, St3, St4, St4, St4] \xrightarrow{Bus\_Controller} [\underline{0}, \underline{F}, F, \underline{St4}, St4, St4, St4] \hookrightarrow [\underline{0}, \underline{F}, F, St3, St4, St4, St4]$ can be described using $Mod\_Selected\_Bus = 0$, $Mod\_Selected\_Bus = 1 - Pre\_Selected\_Bus$, $Mod\_Moving(1) = FALSE$, $Mod\_Next\_Location(1) = St4$, $Mod\_Next\_Location(1) = Pre\_Next\_Location(2)$, etc.

The `Best-Partition` function can produce the best partitions of modification predicates using the following predicates.

- Mod_Selected_Bus = 0
- Mod_Moving(1) = FALSE
- Mod_Moving(2) = FALSE
- Mod_Current_Location(1) = PRE_Current_Location(1)
- Mod_Current_Location(2) = PRE_Current_Location(2)
- Mod_Next_Location(1) = PRE_Next_Location(1)
- Mod_Next_Location(2) = PRE_Next_Location(2)

The above predicates can lead to the best partition because they are satisfied by all the modifications, leading to a partition that includes all the modifications and an empty partition. As the second partition is empty, the current partitions are the best partitions, and no further partitions will be produced. As a counterexample, if another candidate predicate “Mod_Next_Location(1) = St4” is used to split the modification predicates, then the resulting two partitions will include three modifications and one modification, respectively, which are not the best partitions.

Finally, the `Compound-Modification` can convert the partition to a compound modification that covers the following pre-states.

- $[1, T, F, St3, St4, St4, St4]$
- $[1, T, F, St3, St4, St4, St5]$
- $[2, F, T, St4, St3, St4, St4]$
- $[2, F, T, St4, St3, St5, St4]$

The compound modification uses the following refactored substitutions to generate post-states.

- Selected_Bus := 0
- Moving(1) := FALSE
- Moving(2) := FALSE

The above compound modification means that if a bus is already at St4, and the other bus is currently on the way to St4, then the latter should stop at the current location. After returning the compound modification, Algorithm 2 terminates.

4.4 The Update Phase

During the update phase, the compound modification is applied to the original Bus_Controller operation, leading to the repaired operation in Fig. 5. Due to the existence of the extra **IF-THEN-ELSE-END** construct, the compound modification can disable the faulty post-states while maintaining other correct behaviours of the original operation. For any pre-states that are covered by the compound modification, their post-states are determined using the refactored substitution. For any other pre-states, their post-states are determined using the original substitution.

```
Bus_Controller =
  PRE not(Selected_Bus = 0) &
    not(Next_Location(Selected_Bus) = Current_Location(Selected_Bus))
  THEN
    IF Selected_Bus = 1 & Moving(1) = TRUE & Moving(2) = FALSE &
      Current_Location(1) = St3 & Current_Location(2) = St4 &
      Next_Location(1) = St4 & Next_Location(2) = St4
    or Selected_Bus = 1 & Moving(1) = TRUE & Moving(2) = FALSE &
      Current_Location(1) = St3 & Current_Location(2) = St4 &
      Next_Location(1) = St4 & Next_Location(2) = St5
    or Selected_Bus = 2 & Moving(1) = FALSE & Moving(2) = TRUE &
      Current_Location(1) = St4 & Current_Location(2) = St3 &
      Next_Location(1) = St4 & Next_Location(2) = St4
    or Selected_Bus = 2 & Moving(1) = FALSE & Moving(2) = TRUE &
      Current_Location(1) = St4 & Current_Location(2) = St3 &
      Next_Location(1) = St5 & Next_Location(2) = St4
    THEN Moving(1) := FALSE ; Moving(2) := FALSE ; Selected_Bus := 0
    ELSE
      IF Moving(Selected_Bus) = FALSE
      THEN Moving(Selected_Bus) := TRUE
      ELSE Current_Location(Selected_Bus) := Next_Location(Selected_Bus);
        Moving(Selected_Bus) := FALSE
      END;
      Selected_Bus := 0
    END
  END
```

Fig. 5. The Refactored Bus Controller Operation

5 EVALUATION

This section presents an empirical study of the AMBM tool. The experiments consist of two parts: Part I includes experiments of repair evaluator training, and Part II includes experiments of the entire abstract machine modification processes.

5.1 Purpose of the Evaluation

The objective of Part I was to demonstrate that the classifiers can distinguish between possible state transitions and impossible state transitions. Part I evaluated five types of classifiers, including Bernoulli Naive Bayes (BNB) classifiers, Logistic Regression (LR) classifiers, Support Vector Machines (SVM) with radial basis function kernels, Random Forests (RF) and Silas, on repair evaluator training tasks. In order to compare this work with our previous work [10], we conducted the same evaluation for the traditional B-repair with Classification And Regression Trees (CART) and ResNet. We used a set of correct abstract machines (in Table 1, which will be explained in Section 5.2) to generate training and test sets. Firstly, for each subject, the model checker was used to approximate a state space of the abstract machine, and possible transitions were extracted from the state space. Secondly, impossible transitions were randomly generated. This process did not randomly generate any new states, but used existing states to randomly generate impossible state transitions. For example, if we have a variable $x = 0$ and an operation $Add2$ = PRE $x <$ 3 THEN $x := x + 2$ END, then possible state transitions will include $x = 0 \xrightarrow{Add2} x = 2$ and $x = 2 \xrightarrow{Add2} x = 4$, and existing states will be $x = 0$, $x = 2$ and $x = 4$. Impossible state transitions will include $x = 0 \xrightarrow{Add2} x = 0$, $x = 0 \xrightarrow{Add2} x = 4$, $x = 2 \xrightarrow{Add2} x = 0$, $x = 2 \xrightarrow{Add2} x = 2$, $x = 4 \xrightarrow{Add2} x = 0$, $x = 4 \xrightarrow{Add2} x = 2$ and $x = 4 \xrightarrow{Add2} x = 4$. In our experiments, in order to avoid combinatorial explosion, existing states were randomly selected to generate impossible state transitions, and the generation process would terminate if the number of generated impossible state transitions reached the number of possible state transitions. As a result, 50% of the transitions were of the "possible" class, and the remaining 50% of the transitions were of the "impossible" class. All transitions were shuffled and split into a training set and a test set that contained 80% and 20% of the transitions respectively. Thirdly, the five classifiers were trained using the training set, and consistent hyper-parameters were used during the training.[3] Finally, the trained classifiers were evaluated on the test set, and evaluation metrics included the *classification accuracy* and the area under the receiver operating characteristic curve (*ROC-AUC*) [12].

Part II evaluated the whole abstract machine modification process with the five classifiers and the modification refactoring function. We used fault seeding and removal to evaluate the algorithms. Firstly, for each subject, faults were randomly seeded into 100 deterministic transitions of the correct abstract machine, leading to a faulty machine that could trigger 100 invariant violations and a set of standard answers indicating correct modifications. For instance, if $S_{pre} \xrightarrow{\alpha} S_\top$ is a correct transition, and $S_\perp$ is a state that triggers an invariant violation, a fault will be seeded by replacing $S_{pre} \xrightarrow{\alpha} S_\top$ with $S_{pre} \xrightarrow{\alpha} S_\perp$, and the corresponding standard answer is $[S_{pre}, \alpha, S_\perp, S_\top]$, which means that $S_{pre} \xrightarrow{\alpha} S_\perp$ is a faulty transition and should be repaired by replacing $S_\perp$ with $S_\top$. Faulty state transitions were randomly injected into a correct abstract machine using the following steps.

- The type constraints of variables are extracted from the model.
- A large number of states satisfying the type constraints are randomly generated. The states are verified against the invariant of the abstract machine, and faulty states that violate the invariant are collected.
- A faulty state $S_\perp$ is randomly selected, and a correct state transition $S_{pre} \xrightarrow{\alpha} S_\top$ that is generated by the abstract machine is randomly selected as a position to inject $S_\perp$. In order to inject $S_\perp$ into $S_{pre} \xrightarrow{\alpha} S_\top$, we produce an atomic modification $[\alpha, S_{pre}, S_\top, S_\perp]$ and use the `Update` function in Section 3.1 to apply the modification.

The above method was used to make 10 faulty machines based on each correct machine in Table 1. Consequently, $10 \times 24 = 240$ faulty machines were made. In total, 240 faulty machines with 24,000 faulty transitions were produced. After using AMBM to repair all faulty machines, suggested modifications were compared with the standard answers and evaluated using the following metrics:

- *modification accuracy MA* $= N_{cor}^{val} \ / \ N_{tot}^{val}$, where N_{cor}^{val} denotes the number of correctly modified values with reference to the standard answers, and N_{tot}^{val} denotes the total number of values;
- *refactoring generality RG* $= 1 - N_{rec} \ / \ N_0$, where N_0 denotes the number of modifications before refactoring, and N_{rec} denotes the number of modifications after refactoring;

[3]For BNB, LR, SVM and RF, default settings in scikit-learn were used. The default settings of Silas is for datasets containing over 1 million entries. We therefore used another set of settings for smaller datasets, where the number of decision trees in Silas is equal to its counterpart in RF.

- *average repair time* $ART = T / N_F$, where T denotes running time and N_F denotes the number of faults.

The intuition of MA is that the suggested atomic modifications are expected to coincide with standard answers. The intuition of RG is that the suggested atomic modifications are expected to be generalised as fewer compound modifications. When computing RG, both correct and incorrect atomic modifications are taken into account. Generalisation is the outcome of refactoring. Before refactoring, all modifications are atomic modifications. After refactoring, a number of atomic modifications have been replaced with fewer compound modifications. As a single compound modification can cover the functions of multiple atomic modifications, it can reduce the total number of modifications and make the modifications more expressive. In the best case, a compound modification covers all atomic modifications, leading to $N_{rec} = 1$ and $RG = 1 - 1 / N_0$ (i.e., the maximum RG). In the worst case, no compound modification is produced, leading to $N_{rec} = N_0$ and $RG = 0$ (i.e., the minimum RG). With regard to ART, the intuition is that the average time for eliminating a fault should be reasonable.

5.2 Experimental Settings

All experiments were run on a machine equipped with Intel(R) Core(TM) i5-4670 CPU (4 cores, 3.40GHz) and 8GB memory. The operating system was Ubuntu Desktop 16.04.[4] A specific dataset to evaluate our solution was constructed using the materials from the ProB Public Examples repository.[5] We used the following filters to select machines.

- Filter #1 selects machines that are syntactically correct.
- Filter #2 selects machines that have variables, invariants and operations.
- Filter #3 selects machines that approximate at most 30K state transitions (in order to avoid memory exhaustion on our equipment).
- Filter #4 selects machines that approximate at least 500 deterministic state transitions. Note that the selected machines may include non-determinism as well. In order to get more machines, the scales of small machines may be expanded by adjusting their set cardinalities and integer scopes.
- Filter #5 selects machines that can pass the model checking.
- Filter #6 selects machines that only have Boolean values, integers, distinct elements and first-order sets as single variables or arrays.

After using the above filters, we manually removed redundant machines and machines without actual meanings. Consequently, we obtained 18 well-formed and error-free machines for evaluation, and the size of their state space (i.e., the number of states plus the number of state transitions) ranged from 1K to 27K. Table 1 provides information on the 18 machines, i.e., M01 - M18, including the source file of each machine, the number of lines of code (# LOC), the number of variables (# Var.), the number of invariants (# Inv.) and the number of operations (# Ope.). A subset of the abstract machines and their essential information (e.g., source files and # LOC) were used by [10], where # LOC was counted after using ProB [22] to convert the source files into the pretty-printed format.

Additionally, we added M19 - M24 into the evaluation dataset. M19 and M20 were relevant to the wireless network protocols studied by [30] and [31]. The original B models of M19 and M20 were found from the repository shared by [19].[6] M21 - M24 were originally a part of automotive

[4]Scripts and datasets of our experiments can be found in https://github.com/cchrewrite/ambm.
[5]https://www3.hhu.de/stups/downloads/prob/source/.
[6]https://github.com/hhu-stups/specifications/tree/update/prob-examples/B/MobileComm

Table 1. Dataset of Abstract Machines

Subject	Source File (.mch)	# LOC	# Var.	# Inv.	# Ope.
	ProB Public Examples				
M01	Paperround_simple	40	2 sets	2	3
M02	Sortarray	41	1 array of 4 integers	1	3
M03	ParallelModelCheckTest	47	1 array of 5 integers; an integer	4	3
M04	POR_TwoThreads_WithSync	48	4 integers	4	3
M05	progress	55	3 sets	6	4
M06	monitor2	56	3 sets	6	4
M07	BridgeTransitions	57	3 integers	4	5
M08	InvolvedSequences2	60	4 integers	4	5
M09	club	75	2 sets	4	7
M10	ADD4	85	5 integers; 1 bool	7	9
M11	TestBZTT3	85	1 array of 10 elements; 1 element; 1 set	3	8
M12	scheduler6	90	3 sets	10	5
M13	BinomialCoefficientConcurrent	109	5 integers	5	8
M14	Lift2	119	2 integers; 2 bools; 1 set	5	8
M15	Mikrowelle	203	an integer; 4 bools; 1 element	8	12
M16	CSM	211	14 integers	15	13
M17	GSM_revue	215	two arrays of 4 bools and 4 elements, etc.	20	6
M18	Cruise_finite1	482	1 integer; 2 sets; 12 bools	39	26
	Wireless Network Protocol Models				
M19	CXCC	124	3 functions	7	6
M20	BarRel	166	4 functions; 1 set	6	9
	ABZ Automotive Models				
M21	GenericTimersMC	79	1 function	1	5
M22	BlinkLamps	151	3 integers; 1 set; 1 bool	11	7
M23	PitmanController	373	1 integer; 1 set; 5 elements; 2 bools	21	10
M24	PitmanController_TIME_MC	489	1 function; 3 integers; 1 set; 3 elements; 2 bools	22	11

models developed by [23].[7] We adapted M19 - M24 to suit the capabilities of AMBM. For example, partial functions were replaced by total functions as partial functions could not be processed by the repair evaluators. Besides, composite models were expanded as whole models as AMBM could not process the INCLUDES mechanism.

[7]https://github.com/hhu-stups/abz2020-models

Table 2. Results of Repair Evaluator Training

Subject	# Examples	ROC-AUC						
		BNB	*LR*	*SVM*	*RF*	*Silas*	*ResNet*	*CART*
M01	24,530	0.615	0.619	1.000	1.000	1.000	0.961	0.973
M02	22,500	0.722	0.736	0.984	0.996	0.990	0.986	0.986
M03	4,608	0.729	0.691	0.965	1.000	1.000	0.976	0.953
M04	10,202	0.660	0.668	0.936	1.000	1.000	0.955	0.955
M05	15,360	0.640	0.645	0.997	0.999	0.999	0.990	0.961
M06	14,160	0.584	0.593	0.999	0.999	0.999	0.979	0.967
M07	4,550	0.663	0.654	0.725	0.965	0.995	0.819	0.840
M08	2,024	0.673	0.709	0.905	0.998	0.988	0.925	0.931
M09	14,580	0.536	0.539	0.994	0.999	0.999	0.992	0.999
M10	6,592	0.625	0.636	0.993	1.000	0.998	0.962	0.998
M11	15,860	0.848	0.868	0.930	0.997	0.995	0.948	0.968
M12	20,976	0.566	0.563	0.996	1.000	1.000	0.996	0.963
M13	10,210	0.536	0.532	0.621	0.999	1.000	0.794	0.813
M14	11,918	0.802	0.814	0.997	1.000	0.997	0.994	0.990
M15	1,944	0.777	0.837	0.844	0.999	0.988	0.886	0.908
M16	2,456	0.642	0.666	0.850	0.999	0.999	0.905	0.912
M17	9,056	0.679	0.735	0.956	0.998	0.996	0.946	0.937
M18	51,384	0.783	0.822	0.995	1.000	0.998	0.975	0.996
M19	11,378	0.736	0.759	1.000	0.999	0.999	0.938	0.966
M20	5,244	0.771	0.786	0.999	1.000	1.000	0.953	0.992
M21	15,662	0.746	0.753	0.982	0.979	0.981	0.924	0.978
M22	5,002	0.884	0.890	0.983	0.996	0.993	0.912	0.968
M23	1,024	0.615	0.718	0.910	0.977	0.937	0.833	0.941
M24	33,944	0.706	0.732	0.999	1.000	0.999	0.952	0.988
All	315,164	0.694	0.711	0.969	**0.998**	**0.997**	0.958	0.970
Classification Accuracy		0.646	0.665	0.917	**0.980**	**0.980**	0.908	0.941

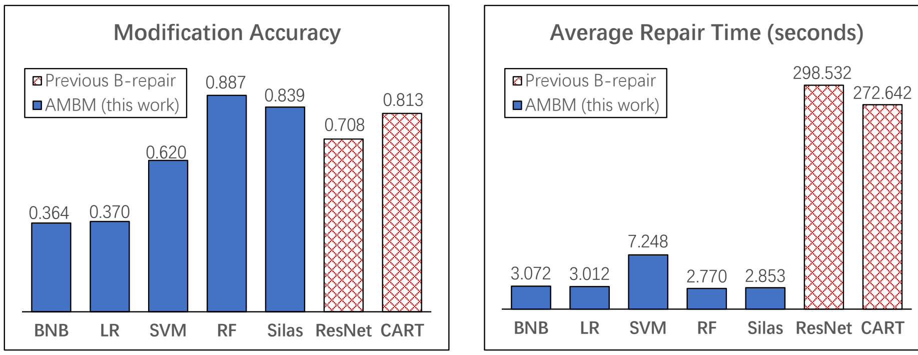
5.3 Results and Discussions

5.3.1 Results of Part I. Table 2 shows the results of repair evaluator training experiments, including the number of training and test examples (i.e., # Examples), the ROC-AUC of classifiers in each subject, the total ROC-AUC and the total classification accuracy (CA). We observed that with regard to ROC-AUC and CA, RF and Silas obtained the best results, i.e., over 99% ROC-AUC and over 98% CA, and the difference between Silas and RF on these metrics was insignificant. Besides, SVM performed well and obtained over 96% ROC-AUC and over 91% CA, whereas LR and BNB fell behind the others. Additionally, both RF and Silas gained better ROC-AUC and CA than the two traditional repair evaluators of B-repair, i.e., ResNet and CART. In summary, RF and Silas showed high predictive performance on the repair evaluator training tasks.

Table 3. Results of Abstract Machine Batch Modification

Subject	Modification Accuracy					Refactoring Generality					Average Repair Time (s)				
	BNB	*LR*	*SVM*	*RF*	*Silas*	*BNB*	*LR*	*SVM*	*RF*	*Silas*	*BNB*	*LR*	*SVM*	*RF*	*Silas*
M01	0.388	0.388	0.611	0.943	0.834	0.959	0.959	0.980	0.965	0.965	2.965	2.965	6.696	3.033	3.157
M02	0.460	0.456	0.398	0.690	0.659	0.990	0.990	0.495	0.744	0.868	2.069	2.066	4.982	2.142	2.138
M03	0.297	0.249	0.844	0.878	0.854	0.952	0.970	0.359	0.215	0.545	3.135	3.111	6.433	3.161	3.619
M04	0.018	0.013	0.187	0.975	0.948	0.980	0.978	0.583	0.137	0.331	10.694	9.161	43.670	3.168	3.094
M05	0.313	0.313	0.457	0.969	0.866	0.960	0.960	0.960	0.895	0.878	2.018	2.023	3.222	1.758	1.828
M06	0.141	0.138	0.333	0.649	0.383	0.960	0.960	0.960	0.841	0.929	1.417	1.398	2.878	1.498	1.530
M07	0.271	0.284	0.518	0.709	0.902	0.960	0.960	0.920	0.361	0.529	0.716	0.715	2.749	0.760	0.748
M08	0.146	0.155	0.456	0.970	0.929	0.961	0.961	0.532	0.342	0.773	2.682	2.584	2.992	3.010	3.350
M09	0.214	0.214	0.399	0.702	0.428	0.876	0.876	0.950	0.861	0.920	1.534	1.531	2.715	1.597	1.684
M10	0.305	0.314	0.943	0.999	0.999	0.920	0.920	0.340	0.359	0.872	0.649	0.648	0.830	0.668	0.688
M11	0.453	0.652	0.656	0.969	0.952	0.953	0.953	0.912	0.695	0.793	4.525	4.538	6.629	4.578	4.646
M12	0.142	0.148	0.402	0.946	0.916	0.950	0.952	0.959	0.845	0.841	2.421	2.390	6.280	2.384	2.451
M13	0.108	0.080	0.148	0.900	0.976	0.930	0.930	0.885	0.159	0.466	1.629	1.616	16.160	1.703	1.660
M14	0.365	0.362	0.799	0.997	0.981	0.920	0.920	0.591	0.499	0.715	3.345	3.504	4.456	3.244	3.444
M15	0.657	0.652	0.671	0.909	0.733	0.890	0.890	0.882	0.496	0.817	3.389	3.353	4.697	3.277	3.558
M16	0.541	0.449	0.714	0.973	0.975	0.871	0.871	0.367	0.229	0.602	2.019	1.995	4.820	2.049	2.267
M17	0.758	0.758	0.917	0.995	0.990	0.980	0.980	0.839	0.792	0.805	2.159	2.152	2.741	2.233	2.283
M18	0.425	0.409	0.811	0.999	0.990	0.802	0.802	0.719	0.795	0.776	10.034	9.970	26.024	9.518	9.803
M19	0.544	0.551	0.857	0.955	0.905	0.958	0.958	0.837	0.746	0.763	4.998	5.093	5.389	5.005	5.004
M20	0.518	0.551	0.927	0.979	0.977	0.913	0.908	0.673	0.700	0.740	3.517	3.502	4.305	3.862	3.694
M21	0.290	0.290	0.510	0.484	0.445	0.980	0.980	0.928	0.773	0.873	0.952	0.955	1.266	0.958	0.957
M22	0.458	0.474	0.712	0.916	0.763	0.951	0.951	0.904	0.859	0.919	1.189	1.201	1.300	1.255	1.251
M23	0.510	0.481	0.711	0.788	0.746	0.891	0.901	0.580	0.531	0.537	1.685	1.607	1.600	1.591	1.591
M24	0.411	0.493	0.893	0.990	0.986	0.890	0.890	0.733	0.559	0.576	3.984	4.204	11.129	4.037	4.028
Average	0.364	0.370	0.620	**0.887**	**0.839**	0.933	0.934	0.745	0.600	**0.743**	3.072	3.012	7.248	2.770	2.853

5.3.2 Results of Part II. Table 3 shows modification accuracies (MA), refactoring generalities (RG) and average repair time (ART) of the abstract machine batch modification experiments. We observed that with regard to MA, RF was the leading classifier with over 88% MA, followed by Silas with over 83% MA. The use of BNB and LR resulted in significantly bad accuracy because the two models were too simple to approximate the encodings of state transitions. SVM obtained better MA because its kernel functions could form more complex transforms than BNB and LR. Besides,



(a) Accuracy Comparisons (b) Performance Comparisons

Fig. 6. Comparing AMBM with the Traditional B-repair [10].

Silas showed better generalisation capability than RF. The use of RF resulted in significantly bad generality probably because the strong fitting ability caused overfitting. By contrast, BNB and LR had significantly lower accuracy and higher generality because their fitting ability was too weak to cause overfitting.

Regarding the performance of AMBM, all classifiers required similar ART of approximately 3 seconds with the exception of SVM, which required over 8 seconds. These results demonstrate the feasibility and the pertinence of the AMBM approach. More specifically, these results suggest that among the classifiers considered in the experiments, Silas is the most suitable for approximating the repair scores of abstract machines as it has both high accuracy and generalisation capability.

The above experiments have demonstrated that the modification accuracy of B model repair can achieve a high level by means of a well-trained classifier, while the refactoring generality of modifications is dependent upon the classifier. Considering both the modification accuracy and the refactoring generality, Silas has the best performance on model repair tasks. Moreover, the finding implies that model repair processes can benefit from the repair evaluators. With repair evaluator training and constraint solving, machines are able to automatically produce repairs and select well-behaved repairs that eliminate faults in the models and preserve the state spaces of the models as much as possible. As the modification accuracy and the refactoring generality have achieved 0.8 and 0.7, respectively, we infer that a large number of suggested modifications are correct and simple. Furthermore, the study has shown the effectiveness of the AMBM algorithm. AMBM is able to automatically eliminate hundreds of faults within a reasonable time, while manually eliminating these faults may require a considerable amount of human effort. Thus, our study suggests that AMBM can assist programmers in designing abstract machines and possibly realise automated model generation via step-by-step incremental design repairs.

Finally, Fig. 6 compared AMBM and the traditional B-repair with respect to accuracy and performance. Fig. 6 (a) indicated that AMBM with RF and Silas gained better accuracies than the traditional B-repair with CART and ResNet. Moreover, Silas had better accuracy than CART because Silas's internal decision trees could model more data types than CART. Additionally, ResNet failed to suggest accurate modifications probably because such a neural network architecture is not appropriate for learning small data. Fig. 6 (b) indicated that AMBM's performance was significantly better than B-repair. B-repair's performance bottleneck was mainly caused by successive repairs, i.e., after eliminating a single invariant violation, a model checking process was started to detect the next invariant violation. Consequently, B-repair was slowed down when the model repair processes

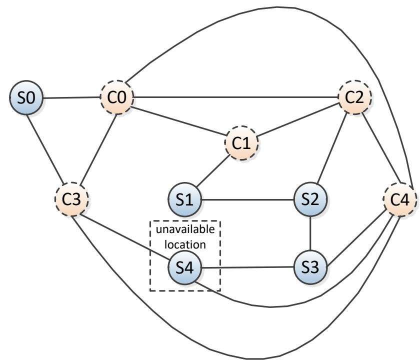


Fig. 7. The Road Map of the AVC Model

were run repeatedly. AMBM solved this problem by detecting multiple invariant violations during only one model checking process so that the number of required model checking processes could be reduced. Overall, the above results showed that AMBM had higher accuracy and performance than the traditional B-repair.

6 DISCUSSIONS

In this section, we discuss a feasible extension to generally repair non-determinism. Additionally, we discuss the limitations of our approach and compare this study with other existing studies.

6.1 Modifications on Non-determinism

As previously mentioned, the modification operator eliminates invariant violations triggered by deterministic state transitions. To repair non-determinism, we suggest a new modification operator. For an operation, given a pre-state p and an erroneous post state q, a modification operator modifies q to a correct post state r. The modification operator can be described using a triple $[u, \text{“}modification\text{”}, r]$, where u is the concatenation of p and q. The triple means that the faulty state pair (p, q) is rewritten as a correct state pair (p, r) using the modification operator. To control the application domain of modification, u is converted to the following condition:

$$v_1^{pre} = p[v_1] \ \wedge \ v_1^{post} = q[v_1] \ \wedge \ \ldots \ \wedge \ v_n^{pre} = p[v_n] \ \wedge \ v_n^{post} = q[v_n] \qquad (5)$$

where v_i^{pre} $(i = 1, 2, \ldots, n)$ is a pre-variable identifier of v_i, v_i^{post} $(i = 1, 2, \ldots, n)$ is a post-variable identifier of v_i, $p[v_i]$ is the value of v_i in p, and $q[v_i]$ is the value of v_i in q. Moreover, r can be converted to the following substitution:

$$v_1^{post} := r[v_1] \ ; \ \ldots \ ; \ v_n^{post} := r[v_n] \qquad (6)$$

where v_i^{post} $(i = 1, 2, \ldots, n)$ is a post-variable identifier of v_i, and $r[v_i]$ is the value of v_i in r. The Non-Determinism Modification (NDM) operator to repair non-determinism is defined as follows. Given M modification operators that are described using M triples $[u_j, \text{“}modification\text{”}, r_j]$ $(j = 1, \ldots, M)$, we use Eq. (5) to convert each u_j to a condition U_j and use Eq. (6) to convert each r_j to a substitution R_j. Given an operation α = **PRE** P **THEN** S **END**, where P is a pre-condition, and S is a substitution, a NDM can be applied to α using the following template.

```
PRE P THEN
  VAR <pre−variables>, <post−variables> IN
    <pre−variables> := <state−variables>;
```

```
      S;
      <post−variables> := <state−variables>;
      IF U_1 THEN R_1 END;
      ...
      IF U_M THEN R_M END;
      <state−variables> := <post−variables>
   END
END
```

In particular, if the original operation does not have a pre-condition, then the repaired operation does not have the outer "**PRE-THEN-END**" statement. The above template uses the "**VAR-IN-END**" construct with two blocks of temporary variables, i.e., "<pre-variables>" and "<post-variables>", to store pre-values and post-values. Before applying the substitution S, the values of the state variables are the pre-values, so that "<pre-variables> := <state-variables>" can assign the pre-values to the pre-variables. After applying S, the values of the state variables become the post-values, so that "<post-variables> := <state-variables>" can assign the post-values to the post-variables. As a result, the pre-values and the post-values are obtained. After using the conditions U_j $(j = 1, \ldots, M)$ and the substitutions R_j $(j = 1, \ldots, M)$ to modify the values of the post-variables, the modified values are assigned to state variables, leading to a new post-state.

The use of NDM requires a slight change on Algorithm 1, i.e., the `Update` function is adapted to use the NDM template to repair faulty operations. The following example of autonomous vehicle control model will demonstrate the use of NDM. The model describes how to control a bus in a city. Fig. 7 visualises a city map, which has a set `Location` containing five bus stations, i.e., S0, S1, S2, S3 and S4, and five other locations, i.e., C0, C1, C2, C3 and C4. The locations are linked by a set `Street` containing bidirectional edges. The model has a variable "loc" recording the location of the bus. The value of loc is changed using the following "*move*" operation.

< Original *move* Operation >
move(x,y) = **PRE** loc = x & (x,y) : Street **THEN** loc := y **END**

Suppose that the bus station S4 is temporarily unavailable. The following invariant is used to specify the constraint.

< Invariant on S4 > not(loc = S4)

The invariant is violated because the *move* operation can change the value of loc from S3, C3 and C4 to S4. Corresponding faulty state transitions and feasible modifications (indicated by "$\hookrightarrow$") are listed below.

< Faulty State Transitions and Modifications >
loc = S3 $\xrightarrow{move}$ loc = S4 $\hookrightarrow$ loc = S2
loc = C3 $\xrightarrow{move}$ loc = S4 $\hookrightarrow$ loc = C4
loc = C4 $\xrightarrow{move}$ loc = S4 $\hookrightarrow$ loc = C3

The meaning of the modifications is that if the bus is scheduled to move from S3, C3 or C4 to S4, the bus will be rescheduled to move from S3 to S2, from C3 to C4 or from C4 to C3, respectively. They are applied to the *move* operation using the NDM operator, leading to the following repaired operation.

< Repaired *move* Operation >
move(x,y) =
PRE loc = x & (x,y) : Street

```
THEN
  VAR loc_pre, loc_post IN
    loc_pre := loc ;
    loc := y ;
    loc_post := loc ;
    IF loc_pre = S3 & loc_post = S4 THEN loc_post := S2 END ;
    IF loc_pre = C3 & loc_post = S4 THEN loc_post := C4 END ;
    IF loc_pre = C4 & loc_post = S4 THEN loc_post := C3 END ;
    loc := loc_post
  END
END
```

The above example demonstrates that even though multiple state transitions can share a common post-state that triggers an invariant violation, the NDM can repair each state transition separately. This is because the conditions of NDM can ensure that each repair acts on one and only one state transition.

6.2 Limitations

As discussed in Section 2.2, the repair evaluators cannot deal with unseen elements in infinite sets. When training the repair evaluators, unseen elements are avoided by reducing infinite sets into finite sets that contain all elements occurred in a state space. For example, if a state space only includes integers 0, 1, 2 and 4, then the finite set {0, 1, 2, 4} is used to generate one-hot encodings, e.g., {1, 4} is encoded as [0, 1, 0, 1]. However, unseen elements such as 3 and 5 are not preserved in one-hot encodings, e.g., {1, 3, 4, 5} is treated as {1, 4} and encoded as [0, 1, 0, 1]. Consequently, the repair evaluators cannot distinguish between {1, 4} and {1, 3, 4, 5}. Besides, to avoid unseen elements, AMBM can control the constraint solver to find modifications that only contain elements occurring in a given state space, but such a restriction may disable a few required modifications. For example, suppose that a faulty operation Inc attempts to increase a set of integers by 1, the operation yields correct state transitions such as $\{0\} \xrightarrow{Inc} \{1\}$, $\{1\} \xrightarrow{Inc} \{2\}$ and $\{0, 1\} \xrightarrow{Inc} \{1, 2\}$, and faulty state transitions such as $\{2\} \xrightarrow{Inc} \{\}$ and $\{1, 2\} \xrightarrow{Inc} \{2\}$. The faulty state transitions should be repaired as $\{2\} \xrightarrow{Inc} \{3\}$ and $\{1, 2\} \xrightarrow{Inc} \{2, 3\}$ respectively. Unfortunately, AMBM cannot suggest the required repairs because 3 does not occur in the given state transitions. A possible approach to solve this problem is model repair based on inductive programming [33]. Given a component library containing essential symbols such as 0, 1, +, -, * and =, inductive programming can synthesise the function $x' = x + 1$, where x and x' are integers, to generalise the correct state transitions of Inc. When $x = 2$, a new element $x' = 3$ can be inferred by the function, so that the required modifications can be constructed. The above discussion indicates that a merger of inductive programming and AMBM can be a way to solve the restrictions of infinite sets.

The repair of higher-order sets, e.g., sets of sets, is another limitation to the applicability of AMBM. The number of candidate modifications with higher-order sets can be huge due to combinatorial explosions. For example, if a set of sets is constructed by integers $1, 2, \ldots, n$, then its value will be a member of $\mathcal{P}^2(\{1, 2, \ldots, n\})$, where $\mathcal{P}$ represents the power set. As the cardinality of $\mathcal{P}^2(\{1, 2, \ldots, n\})$ is 2^{2^n}, it is unrealistic to enumerate the members when n is large. Instead of enumeration, AMBM only uses components that occurred in a given state space to generate modifications. As a result, a required modification will be missing if it includes a component not in the state space. For example, suppose that a faulty operation Inc_Send attempts to increase sets of integers by 1 and transfer the sets from a sender to a receiver, with two sets of sets $[Sender, Receiver]$ to represent states, the operation yields correct state transitions such as:

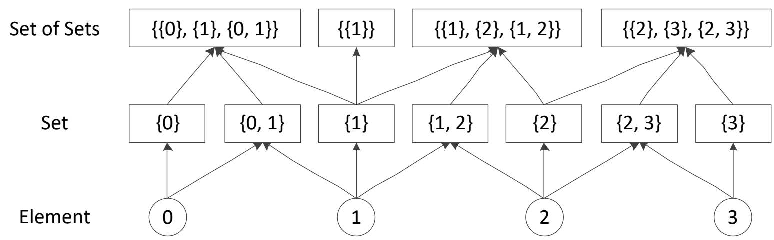


Fig. 8. Graph Representations of Member Relationships in Sets of Sets.

- $[\{\{0\}, \{1\}, \{0, 1\}\}, \{\}] \xrightarrow{Inc\_Send} [\{\{1\}, \{0, 1\}\}, \{\{1\}\}]$
- $[\{\{1\}, \{0, 1\}\}, \{\{1\}\}] \xrightarrow{Inc\_Send} [\{\{0, 1\}\}, \{\{1\}, \{2\}\}]$
- $[\{\{0, 1\}\}, \{\{1\}, \{2\}\}] \xrightarrow{Inc\_Send} [\{\}, \{\{1\}, \{2\}, \{1, 2\}\}]$
- $[\{\{1\}, \{2\}, \{1, 2\}\}, \{\}] \xrightarrow{Inc\_Send} [\{\{2\}, \{1, 2\}\}, \{\{2\}\}]$

and faulty state transitions such as:

- $[\{\{2\}, \{1, 2\}\}, \{\{2\}\}] \xrightarrow{Inc\_Send} [\{\{1, 2\}\}, \{\{2\}, \{\}\}]$
- $[\{\{1, 2\}\}, \{\{2\}, \{\}\}] \xrightarrow{Inc\_Send} [\{\}, \{\{2\}, \{\}\}]$

The faulty state transitions should be repaired as follows:

- $[\{\{2\}, \{1, 2\}\}, \{\{2\}\}] \xrightarrow{Inc\_Send} [\{\{1, 2\}\}, \{\{2\}, \{3\}\}]$
- $[\{\{1, 2\}\}, \{\{2\}, \{3\}\}] \xrightarrow{Inc\_Send} [\{\}, \{\{2\}, \{3\}, \{2, 3\}\}]$

Unfortunately, AMBM cannot suggest the required repairs because {3} and {2, 3} are unseen in the given state space. To suggest repairs with such unseen components, inductive programming is probably feasible because it can construct functions to infer unseen components, as discussed above.

Besides, as the repair evaluators treat sets in sets as string elements, any unseen sets in sets cannot be encoded appropriately. To encode such unseen components, a feasible method is to introduce repair evaluators based on graphs. Fig. 8 shows an example of the conversion from sets of sets (or higher-order sets) to a graph. Compared to the string representation, the advantage of graph representation is that graphs can link two components at multiple levels so that unseen components can be represented by linking with their sub-components. For example, both {{1}, {2}, {1, 2}} and {{2}, {3}, {2, 3}} have a direct link from the set {2} and two indirect links from the element 2. Even though {{2}, {3}, {2, 3}} is unseen in the state space, its encoding can still be partially inferred from the encoding of {{1}, {2}, {1, 2}} using graph learning models [36]. As there are a large number of graph learning models available, their application to B model repair can be considered future work.

6.3 Comparisons to Related Work

Automated B model repair originates from two studies [32, 33]. In order to eliminate faults in abstract machines, they have proposed four methods, including the strengthening of pre-conditions, the relaxation of pre-conditions, the relaxation of invariants and the synthesis of new operations. When relaxing pre-conditions and invariants and synthesising new operations, users are required to manually produce positive and negative I/O examples for program synthesis. The difference between their work and our work is that we focus on repairing substitutions, while they focus on repairing pre-conditions. Moreover, our AMBM algorithm uses repair evaluators and constraint solving to automatically synthesise repairs, while their methods require users to manually produce I/O examples for repair synthesis.

AMBM is an improved implementation of our previous work, B-repair [10]. When revising faulty abstract machines, B-repair uses a constraint solver to generate candidate repairs and uses learnt quality estimation functions to rank repairs. As B-repair eliminates one and only one fault during each loop of repair, it takes considerably longer to repair models with a large number of faults. In this work, as multiple faults are eliminated using fewer compound repairs during each loop of repair, AMBM is significantly more efficient than the previous B-repair, and the resulting corrections are simpler in terms of the predicate structure.

With regard to the previous work on automatic imperative program repair such as RSRepair [28], GenProg [21], CASC [37] and SearchRepair [17], automated B model repair is conceptually different, because B is a design modelling language for constructing formal specifications at the abstract design level, while imperative programs are at the concrete implementation level [2]. Functions in B design models are usually represented as operations with pre-conditions and substitutions (which play the role of post-conditions) that declare the facts between the before and after values (states) of the variables used in the system. Consequently, operation executions are decided by the current states of variables but not decided by the control flows of the program. Thus, repair evaluators can separately analyse each operation, and repairs can be applied to faulty operations asynchronously without considering the execution orders of operations. In automatic imperative program repair, however, execution orders that are decided by control flows are important factors to be considered.

There are a number of similarities between the repair of B models and imperative program repair. Firstly, both of them have fault localisation functions to reduce the search space of the repair. B model repair can rely on a model checker to detect faulty operations, while imperative program repair can rely on Spectrum-based Fault Localisation (SFL) [1] functions that find suspicious code blocks by counting successful and failed paths of program executions. Secondly, the concept of inductive programming can be used to synthesise patches of imperative programs [18] and refactor atomic modifications to compound modifications (this work). Thirdly, conditional statements are often used to avoid the side effects of repair. For example, Staged Program Repair (SPR) [24] can use a number of instances to generate conditions that distinguish between correct and failed executions, so that side effects to the correct executions can be minimised. Similarly, our AMBM uses conditional substitutions to avoid the side effects of modifications. Finally, as repairs are not definitive, evaluation functions seem to be inevitable in order to estimate the appropriateness of candidate repairs and select the best repairs. For example, GenProg [21] evaluates candidate repairs by observing successful and failed executions related to the repairs, while AMBM uses classification models and repair scores to select repairs. The above similarities between imperative program repair and B model repair indicate that the technologies in the two fields may be used by each other in the future.

7 CONCLUSION

We have extended B-repair by implementing abstract machine batch modification, which is an automated method for repairing erroneous B formal models during the correct-by-construction development processes. We have demonstrated that the state spaces of abstract machines can be accurately learnt using classic classifiers such as random forests. The learnt classifiers can be used to select atomic modifications produced by solving invariant constraints. Moreover, atomic modifications can be merged as compound modifications using predicate refactoring. The explainable and verifiable classifier has yielded high modification accuracies and improved the generality of compound modifications. Consequently, we suggest that automated abstract machine modification has the potential to increase the efficiency and productivity of software development.

In the future, AMBM may be improved as described below:

- It is possible to develop repair evaluators based on unsupervised machine learning algorithms, where the generation of negative training examples is no longer needed because the characteristics of state spaces can be directly learnt using the unsupervised methods. We may develop new unsupervised methods similar to random forests and appropriate decision functions to achieve relatively high modification accuracies.
- It is possible to extend AMBM using refinement checking techniques. Firstly, a model checker is used to find faulty state transitions that violate refinement conditions. Next, pre- and post-conditions of a refined abstract machine are rewritten as constraints, so that a set of candidate modifications can be generated by solving the constraints. After that, a repair evaluator is used to select the best modification to repair each faulty state transition. Finally, the original model is updated using the modifications and checked against the refinement conditions.
- It is possible to design an algorithm that can eliminate invariant violations by either weakening invariants or modifying existing state transitions. The problem is how to make a choice between the above two options. A possible solution is to use an evaluator to decide whether a faulty state transition should be kept or removed, i.e., if the state transition gains a relatively high score, then the state transition will be kept, and the corresponding invariant will be weakened. If the state transition does not gain a high score, then the state transition will be modified using atomic modifications, and the corresponding invariant will not be changed.
- In order to use AMBM in industry, we may try to improve the scalability of AMBM by integrating it with more development tools and speeding up the refactoring process using probabilistic techniques.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[2] Jean-Raymond Abrial. 2005. *The B-book - assigning programs to meanings.* Cambridge University Press.

[3] Thomas Ackling, Bradley Alexander, and Ian Grunert. 2011. Evolving patches for software repair. In *Proceedings of 13th Annual Genetic and Evolutionary Computation Conference (GECCO), Dublin, Ireland, July 12-16, 2011.* 1427–1434.

[4] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastián Uchitel. 2015. Automated support for diagnosis and repair. *Commun. ACM* 58, 2 (2015), 65–72.

[5] Haniel Barbosa and David Déharbe. 2012. Formal Verification of PLC Programs Using the B Method. In *Proceedings of ABZ: Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, Pisa, Italy, June 18-21, 2012.* 353–356.

[6] Nazim Benaïssa, David Bonvoisin, Abderrahmane Feliachi, and Julien Ordioni. 2016. The PERF Approach for Formal Verification. In *Proceedings of Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference (RSSRail), Paris, France, June 28-30, 2016.* 203–214.

[7] Christopher M. Bishop. 2007. *Pattern recognition and machine learning, 5th Edition.* Springer.

[8] Hadrien Bride, Cheng-Hao Cai, Jie Dong, Jin Song Dong, Zhé Hóu, Seyedali Mirjalili, and Jing Sun. 2021. Silas: A high-performance machine learning foundation for logical reasoning and verification. *Expert Systems with Applications* 176 (2021), 114806.

[9] Hadrien Bride, Jie Dong, Jin Song Dong, and Zhé Hóu. 2018. Towards Dependable and Explainable Machine Learning Using Automated Reasoning. In *Proceedings of Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods (ICFEM), Gold Coast, Australia, November 12-16, 2018.* 412–416.

[10] Cheng-Hao Cai, Jing Sun, and Gillian Dobbie. 2019. Automatic B-model repair using model checking and machine learning. *Automated Software Engineering* 26, 3 (2019), 653–704.

[11] Johan de Kleer and Brian C. Williams. 1989. Diagnosis with Behavioral Modes. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI), Detroit, MI, USA, August 1989*. 1324–1330.

[12] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern Recognition Letters* 27, 8 (2006), 861–874.

[13] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[14] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of Third International Conference on Document Analysis and Recognition (ICDAR), August 14 - 15, 1995, Montreal, Canada*. 278–282.

[15] Sarah Hoffmann, Germain Haugou, Sophie Gabriele, and Lilian Burdy. 2007. The B-Method for the Construction of Microkernel-Based Systems. In *Proceedings of 7th International Conference of B Users, Besançon, France, January 17-19, 2007*. 257–259.

[16] Tao Ji, Liqian Chen, Xiaoguang Mao, and Xin Yi. 2016. Automated Program Repair by Using Similar Code Containing Fix Ingredients. In *Proceedings of 40th IEEE Annual Computer Software and Applications Conference (COMPSAC), Atlanta, GA, USA, June 10-14, 2016*. 197–202.

[17] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, November 9-13, 2015*. 295–306.

[18] Emanuel Kitzelmann. 2009. Inductive Programming: A Survey of Program Synthesis Techniques. In *Approaches and Applications of Inductive Programming, Third International Workshop (AAIP), Edinburgh, UK, September 4, 2009*. 50–73.

[19] Philipp Körner, Michael Leuschel, and Jannik Dunkelau. 2020. Towards a Shared Specification Repository. In *Proceedings of ABZ: Rigorous State-Based Methods - 7th International Conference, Ulm, Germany, May 27-29, 2020*, Alexander Raschke, Dominique Méry, and Frank Houdek (Eds.). 266–271.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, June 2-9, 2012*. 3–13.

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[22] Michael Leuschel and Michael J. Butler. 2008. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer* 10, 2 (2008), 185–203.

[23] Michael Leuschel, Mareike Mutz, and Michelle Werth. 2020. Modelling and Validating an Automotive System in Classical B and Event-B. In *Proceedings of ABZ: Rigorous State-Based Methods - 7th International Conference, Ulm, Germany, May 27-29, 2020*, Alexander Raschke, Dominique Méry, and Frank Houdek (Eds.). 335–350.

[24] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of ESEC/FSE: 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, August 30 - September 4, 2015*. 166–178.

[25] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[26] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (2014), 427–449.

[27] Ingo Pill and Thomas Quaritsch. 2013. Behavioral Diagnosis of LTL Specifications at Operator Level. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), Beijing, China, August 3-9, 2013*. 1053–1059.

[28] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of 36th International Conference on Software Engineering (ICSE) Hyderabad, India, May 31 - June 07, 2014*. 254–265.

[29] Stuart J. Russell and Peter Norvig. 2010. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

[30] Björn Scheuermann. 2007. *Reading between the packets - implicit feedback in wireless multihop networks*. Ph. D. Dissertation. University of Düsseldorf, Germany.

[31] Björn Scheuermann, Christian Lochert, and Martin Mauve. 2008. Implicit hop-by-hop congestion control in wireless multihop networks. *Ad Hoc Networks* 6, 2 (2008), 260–286.

[32] Joshua Schmidt, Sebastian Krings, and Michael Leuschel. 2016. Interactive Model Repair by Synthesis. In *Proceedings of ABZ: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, Linz, Austria, May 23-27, 2016*. 303–307.

[33] Joshua Schmidt, Sebastian Krings, and Michael Leuschel. 2018. Repair and Generation of Formal Models Using Synthesis. In *Proceedings of Integrated Formal Methods - 14th International Conference (IFM), Maynooth, Ireland, September 5-7, 2018*. 346–366.

[34] Alexey Smirnov and Tzi-cker Chiueh. 2007. Automatic Patch Generation for Buffer Overflow Attacks. In *Proceedings of the Third International Symposium on Information Assurance and Security (IAS), Manchester, UK, August 29-31, 2007*.

165–170.
[35] Joseph P. Turian, Lev-Arie Ratinov, and Yoshua Bengio. 2010. Word Representations: A Simple and General Method for Semi-Supervised Learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL), Uppsala, Sweden, July 11-16, 2010*. 384–394.
[36] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z. Sheng, Mehmet A. Orgun, Longbing Cao, Francesco Ricci, and Philip S. Yu. 2021. Graph Learning based Recommender Systems: A Review. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI), Virtual Event / Montreal, Canada, 19-27 August, 2021*. 4644–4652.
[37] Josh L. Wilkerson and Daniel R. Tauritz. 2010. Coevolutionary automated software correction. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO), Portland, Oregon, USA, July 7-11, 2010*. 1391–1392.