VP_TT: A value propagation based equivalence checker for testability transformations

Ramanuj Chouksey | Sachin Kumar Maddheshiya | Chandan Karfa

Abstract
Testability transformation (TT) is a source-to-source programme transformation that aims to improve the ability of a given test generation method to generate test data for the original programme. Herein, the correctness of testability transformations is shown. Translation validation is the process of proving that the transformed programme is a correct translation of the source programme being compiled. It is widely used to verify the correctness of various compiler optimizations and transformations during scheduling. The value propagation based equivalence checking (VP) method is an efficient translation validation approach proposed to verify the correctness of various compiler optimization applied during scheduling in high-level synthesis. VP-based translation validation of testability transformations is proposed. In particular, it is identified that the existing VP method fails to show the equivalence for some of the TTs. A dynamic cutpoint selection scheme and an enhancement to the VP method to overcome these limitations are shown. The enhanced VP method, called VP_TT, successfully shows the equivalence for the TTs where the VP method fails. Experimental results confirm the usefulness of VP_TT in the verification of testability transformations.

1 | INTRODUCTION

Testability transformations (TTs) [1–7] are applied in software testing to improve the test data generation. TTs are a set of programme transformation techniques applied to improve the original programme’s ‘testability’. Testability transformation has been demonstrated to be viable at improving coverage for search-based test data generation. It has been used to tackle problems such as flag variables, unstructured control flow, data flow testing, state-based testing, and nesting to enhance the original programme so that it becomes easier for the generation of adequate test data. Though the test data is to be applied to the original programme, it is generated from the transformed version. Therefore, it is important that the transformed programme must preserve the semantic equivalence against the original programme. Semantic equivalence ensures that the atomic tests are free of side effects and prevent the target programme to perform redundant or useless evaluations of tests. The correctness by construction cannot be expected from a testability transformation tool with several hundred thousand code lines. Therefore, it is important to verify whether the transformed programme is semantically equivalent to the original programme.

Herein, we plan to develop a translation validation framework for checking the correctness of testability transformations. Translation validation [8] is a popular approach for the verification of compiler optimizations. Translation validation is a formal verification approach for compilers whereby each translation is followed by a validation phase which verifies that the transformed behaviour generated by compiler correctly implements the source behaviour. Path-based equivalence checking (PBEC) [9–14], a popular translation validation approach, has made significant progress in the verification of the compiler optimizations applied during scheduling in High-level Synthesis (HLS). The PBEC approaches model the input and the transformed behaviours as a finite state machine with datapaths (FSMDs). In general, PBEC approaches decompose each FSMD into a set of finite paths, and the equivalence of FSMDs is established by showing path level equivalence between two FSMDs using a symbolic bi-simulation approach. The value propagation based equivalence checking (VP) method proposed in Ref. [9] is a powerful PBEC method that can handle code...
motion across loops along with all the compiler optimizations supported by [10–14]. We explored the applicability of the VP method in the verification of TTs. We found that the VP can handle most of the TTs. The equivalence problem of two behaviours is, however, an undecidable problem in general. Therefore, it is possible that the VP method may produce a false-negative result. In fact, the VP method fails to show the equivalence for TTs which involves removal of flag variables [2] and exit statements [15] from the original programme. Thus, the VP method produces false-negative results in these cases.

We investigated the limitations of the VP method in handling flag variables and exit statements removal during TTs. We have identified two limitations in the VP methods. Specifically, in flag removal, multiple conditional blocks are merged into one. The VP method fails to handle this type of scenario due to its static cutpoint selection scheme (Limitation 1). The exit statements create multiple exit paths from the loop body which cannot handle by the VP method (Limitation 2).

We then proposed two enhancements in the VP method to overcome the above limitations to handle two most common TTs: that is removal of flag variables and exit statements. Specifically, we proposed (i) a dynamic cutpoint selection scheme to handle flag variable removal and (ii) a loop invariant checking at all loop exit points to handle exit statement removal. With these two enhancements, we showed that the VP method can handle both flag variables and exit statements removals. To the best of our knowledge, this is the first method proposed for verification of testability transformations. The contributions of this work can be summarised as follows:

- We model the equivalence checking of the testability transformations problem as the equivalence checking of two FSMDs.
- We propose an equivalence checker, called value propagation based equivalence of testability transformation (VP_TT), for the verification of TTs. The method overcomes the limitations of the existing VP method to handle the removal of flag variables and exit statements in TTs.
- Experimental results show that the proposed VP_TT method successfully establishes the equivalence for TTs.

The rest is organised as follows. The related works are briefly discussed in Section 2. The FSMD model and the VP-based method are briefly explained in Section 3. The limitations of the VP method are highlighted in Section 4. A solution to overcome the limitations of the VP method is presented in Section 5. The VP_TT method is presented in Section 6. The correctness and complexity of the proposed method are discussed in Section 7. Experimental results are given in Section 8. Section 9 gives the conclusion.

## 2 RELATED WORK

Translation validation is a well-known technique for the verification of programme transformations. Translation validations aimed to check that each translation performed by the compiler preserves the semantics of the input programme. Translation validation for variety of real-world compilers on large and complex programs has been reported in literature. Translation validation infrastructure (TVI) [16] for GCC demonstrated the validation of the gcc-2.91 compiler and the Linux-2.2 kernel, across five IR passes in GCC, namely branch optimization, common sub-expression elimination, loop unrolling and inversion, register allocation, and instruction scheduling. A value graph translation validator [17,18] to validate the various optimizations of the LLVM compiler is available. The work in Ref. [18] can verify constant propagation, dead-code elimination, global value numbering, loop-invariant code motion, loop deletion, loop un-switching, and dead-store elimination. In a recent work, Dahiya et al. [19] offers a novel black-box equivalence checking algorithm which can perform black-box checking across almost all the composed transformations produced by modern compilers. Some other notable translation validation works for compiler transformations are Refs. [20–22].

Translation validation is also commonly used to verify the correctness of scheduling transformation during HLS. A formal verification of the scheduling phase of HLS using the FSMD model is reported in Ref. [23]. The method presented herein fails if the scheduler applies the non-uniform code motion transformations. Karfa et al. proposed an equivalence checking method for verification of scheduling in Ref. [11]. This method works only for uniform code motion techniques. The study by Karfa et al. [13] improved the equivalence checking method presented in Ref. [11] to handle both uniform and non-uniform code motions applied during the scheduling phase of HLS. The study by Banerjee et al. [9] introduces a notion of value propagation and widen the scope of the path-based mechanism to handle code motion across loops. This VP method is also capable of handling control structure modification of input behaviour and uniform and non-uniform code motion. The VP method is enhanced in Ref. [24] to handle code motion involving loops. An counter example generation process using SMT solver in case of non-equivalence shown by VP method is shown in Ref. [25]. Recently, the scheduling of conditional behaviours is verified using an translation validation approach in Ref. [26].

A number of testability theories have been published in the literature. Voas and Miller [27] defined the testability as the probability that a piece of software will fail on its next execution during testing, provided it contains a fault. McGregor and Srinivas [28] attempted to determine the testability of an object-oriented system. Harman et al. [1] remove flag variables from branch predicates, replacing them with the expression that led to their determination. In the transformed version of the programme is flag-free. Barales and Shwimer [29] introduced a new approach for solving evolutionary testing problems with flag conditions. The study by Gong and Yao [5] uses genetic algorithm to tackle the flag problem in testability transformation. Testability transformations have also been applied to unstructured programs for branch coverage [15]. The exit statements removal for test data generation is also discussed herein. Harman et al. [2] focuses on ameliorating problems for automated test data generation techniques. The studies by Ramshaw and Ward [30,31] focus on the goto removal problem.
As discussed above, translation validation is commonly used for verification of compiler transformations. On the other hand, testability transformations are popular programme transformation approaches in software testing applied to improve the test data generation. To the best of the authors’ knowledge none of the existing works targets verification of testability transformations. We target the translation validation of TTs. Specifically, we consider a robust VP-based translation validation approach [9] proposed originally for scheduling transformation and enhance it to verify testability transformations.

3 | FSMD MODEL AND THE VP METHOD

In this section, the FSMD model and the VP method are briefly explained. The details can be found in Ref. [9].

3.1 | FSMD model

For verification of testability transformations, the input and the transformed programs need to be represented formally. We model the programs using a finite state machine with data paths (FSMDs) [32]. Although FSMD was initially proposed by Gajski et al. [32] to model hardware designs, it has been used to model sequential programs in the verification of scheduling transformations [9,11]. In fact, the FSMD models the sequential programme efficiently by capturing both control and data flow of the behaviours. The control flow is captured by the sequence of states in the FSMD. The control and data flow is implicitly maintained by the state transitions and the data flow of the behaviours. The control flow is captured by the state transitions and the data flow is implicitly maintained by the state transitions of the form

\[ q_i \Rightarrow q_{i+1} \Rightarrow \cdots \Rightarrow q_{i+n} = q_j \]

where \( q_k \in Q \) for all \( k, i \leq k \leq i + n \), and \( \exists c_k \in 2^S \) such that \( f(q_k, c_k) = q_{k+1} \) for all \( k, i \leq k \leq i + n - 1 \). A (finite) path is a walk where all the states are different, except the end state \( q_j \) may be the same as the start state \( q_i \). The condition of execution \( R_{\beta} \) of a path \( \beta \) is a logical expression over \( I \cup V \), which must be satisfied by initial data state to traverse the path \( a \). The data transformation \( s_\beta \) represents an updated variables vector \( s_\beta = (e_1, e_2, \ldots, e_k) \), where \( k = |V_0 \cup V_i| \) and \( e_i \) is the symbolic value obtained by the variable \( v_i \in V_0 \cup V_1 \) at the end state of \( \beta \). For a path \( \beta, R_{\beta}, \) and \( s_\beta \) are computed by forward or backward substitution based on symbolic execution.

Definition 2 (Path Equivalence). Two paths \( \beta \) and \( \alpha \) are equivalent denoted by \( \beta \simeq \alpha \) if \( R_{\beta} \equiv R_{\alpha} \) and \( s_{\beta} = s_{\alpha} \).

Example 1. Let us consider the input behaviour in Figure 2(a) and its corresponding FSMDs in 2(b). The behaviour checks whether a number \( n \) is perfect number or not. If \( n \) is a perfect number then it sets the value of the variable \( out \) to one otherwise \( 0 \). Let consider the path \( \beta : q_{00} \Rightarrow q_{01} \Rightarrow q_{05} \Rightarrow q_{06} \Rightarrow q_{00} \) in the FSMD \( M_0 \) in 2(b). The computation of \( [R_{\beta}, s_{\beta}] \) for this path \( \beta \) by forward substitution method is as follows:

\[ At \ q_{00} : [True, \{sum = n, temp = out\}] \]
\[ At \ q_{01} : [True, \{1, 2, n, temp, out\}] \]
\[ At \ q_{05} : [\{2 < n\}, \{1, 2, n, temp, out\}] \]
\[ At \ q_{06} : [\{2 < n\} \land (2 = n), \{1, 2, n, temp, 1\}] \]
\[ At \ q_{00} : [\{2 < n\} \land (2 = n), \{1, 2, n, temp, 1\}] \]

3.2 | Equivalence of FSMDs

Let consider the source behaviour is formally represented by the FSMD \( M_0 = \langle Q_0, q_{00}, I, Q, V, f_0, h_0 \rangle \) and the behaviour statements like conditional and loop statements. The FSMD consists of a set of states and transitions among states. Each transition of an FSMD is guarded by condition over variables and constants. A set of assignment operations are also associated with each transition of an FSMD. Therefore, the transitions of an FSMD capture the control flow and also associated assignment operations. It is important to ensure that the inherent data dependencies among the assignment operations within a basic block of a programme are not violated during FSMD-based modelling of programme. In FSMD, operations associated in a transition are assumed to execute in parallel. Therefore, a sequence of assignment operations of a basic block is placed sequentially one in each state to maintain the data dependencies. A sample sequential statement, if-else block and while loop is shown in Figure 1(a), 1(c) and 1(e), respectively and its corresponding FSMD is shown in Figure 1(b), 1(d), and 1(f), respectively. The study by Karfa et al. [13] discusses in detail how the FSMD models can be constructed from the high-level representations of the input and the transformed behaviours.

\[ Q \] is the finite set of states, \( q_0 \in Q \) is the reset (initial) state, \( I \) is the finite set of input variables, \( O \) is the finite set of output variables, \( V \) is the finite set of storage variables, \( f : Q \times 2^S \rightarrow Q \) is the state transition function, \( h : Q \times 2^S \rightarrow U \) is the update function. Here, \( S \) represents the set of relations over arithmetic expression and Boolean literals and \( U \) represents a set of variable and output assignments. An FSMD is an inherently deterministic model.

Any sequential programme can be model as an FSMD. A programme has a set of inputs, a set of outputs, a set of variables, and a set of assignment operations of variables and outputs. The programme also consists of control flow
after testability transformation is formally represented by the FSMD $M_1 = (Q_1, q_{10}, I, O, V_1, f_1, h_1)$. Note that these two FSMDs have the same input(s)/output(s) represented by $I$ and $V$. However, these two FSMDs differ in the number of states, internal variables, state transition function, and the update function. In the following, we discuss how to define the equivalence of these two FSMDs $M_0$ and $M_1$ formally.

A computation of an FSMD is a finite walk from the reset state to itself, and the reset state should not occur in between. The equivalence of computations of two FSMDs are defined as follows.

**Definition 3** (Computation Equivalence). Two computations $\mu_0$ and $\mu_1$ are equivalent, denoted as $\mu_0 \simeq \mu_1$, iff $R_{\mu_0} \equiv R_{\mu_1}$ and $s_{\mu_0} = s_{\mu_1}$, where $R_{\mu_0}$ and $R_{\mu_1}$ are the condition of execution of $\mu_0$ and $\mu_1$, respectively, and $s_{\mu_0}$ and $s_{\mu_1}$ are the data transformation of $\mu_0$ and $\mu_1$, respectively.

The $M_1$ is transformed programme derived from $M_0$ through testability transformations. Our goal is to verify whether $M_0$ behaves exactly as $M_1$. This means that for all possible input sequences, $M_0$ and $M_1$ produce the same sequences of output values and eventually, when the respective reset states are revisited, they are visited with the same storage element values. In other words, for every computation from the reset state back to itself of one FSMD, there exists an equivalent computation from the reset state back to itself in the other FSMD and vice versa. The equivalence of two FSMDs $M_0$ and $M_1$ is captured in the following two definitions.

**Definition 4** (FSMD Containment). An FSMD $M_0$ is contained in another FSMD $M_1$ ($M_0 \subseteq M_1$) if for any computation $\mu_0$ of $M_0$ on some inputs, there exists a computation $\mu_1$ of $M_1$ on the same inputs such that $\mu_0 \simeq \mu_1$.

**Definition 5** (Equivalence of FSMDs). Two FSMDs $M_0$ and $M_1$ are computationally equivalent, if $M_0 \subseteq M_1$ and $M_1 \subseteq M_0$.

An FSMD may consist of an infinite number of computations because of loops. Therefore, identifying all computations in an FSMD and then finding their equivalence in the other FSMD is not possible in practice. Alternatively, in path-based equivalence checking (PBEC), any computation $\mu$ of an FSMD $M$ is represented as the concatenation $[\beta_1, \beta_2, \ldots, \beta_n]$ of paths of $M$ where for all $k, 1 \leq k < n, \beta_k$ terminates in the start state of the path $\beta_{k+1}$; the reset state is the start state of $\beta_1$ and
the end state of \( \beta_i \); \( \beta_i \)'s may not all be distinct. The objective is to find a finite set of paths in an FSMD \( M \) so that any computation in \( M \) can be represented by the concatenation of paths from that set. With this observation, the equivalence checking of two FSMDs will reduce to checking equivalence between finite set of paths to two FSMDs. Hence, we have the following definition.

**Definition 6** (Path cover of an FSMD). A finite set of paths \( P = \{ \beta_0, \beta_1, \ldots, \beta_k \} \) is said to be a path cover of an FSMD \( M \) if any computation \( \mu \) of \( M \) can be looked upon as a concatenation of paths from \( P \).

To obtain a path cover for an FSMD model each loop is cut at least one cutpoint. The set of all paths from a cutpoint to another cutpoint without any intermediary occurrence of a cutpoint is a path cover of the FSMD. This is based on the Floyd–Hoare method of programme verification [33].

**Example 2.** Let us consider the FSMDs \( M_0 \) in 2(b). The cutpoints are double circled in the FSMDs \( M_0 \) in 2(b). The states \( q_00, q_01, q_02, \) and \( q_03 \) are cutpoints in \( M_0 \). Note that in the FSMD \( M_0 \), the reset state \( q_00 \) and all the branching state are cutpoints. After setting, the cutpoints in \( M_0 \), the path cover of \( M_0 \) is \( P_0 = \{ \beta_1, \beta_2, \beta_3, \beta_4, \beta_7, \beta_8 \} \). Here, \( \beta_1, \beta_2, \beta_3 \) represents the concatenation of paths \( \beta_1, \beta_2, \beta_3 \).

A PBEC approach inserts cutpoints into FSMDs \( M_0 \) and \( M_1 \), identifies the path covers \( P_0 \) and \( P_1 \) in \( M_0 \) and \( M_1 \), respectively, and then tries to find the equivalence of the paths between \( P_0 \) and \( P_1 \) using a symbolic bi-simulation-based approach [34]. The reset states \( q_00 \) and \( q_{10} \) of \( M_0 \) and \( M_1 \), respectively, are corresponding states. The states \( \beta_{i0} \in Q_0 \) and \( \beta_{i1} \in Q_1 \) are corresponding states if the state \( \beta_{i0} \in Q_0 \) and \( \beta_{i1} \in Q_1 \) are corresponding states and there exists paths \( \beta \) from \( q_{00} \) to \( q_{0k} \) and \( \alpha \) from \( q_{01} \) to \( q_{1l} \) such that \( \beta \simeq \alpha \). Since, the data transformations of the subsequent paths from \( \langle \beta_{i0}, \beta_{i1} \rangle \) are computed using initial symbolic values of the variables, it does not matter what symbolic values it considers as long as it is the same in both FSMDs. Therefore, at corresponding states \( \langle \beta_{i0}, \beta_{i1} \rangle \), all the values of variables are reset. The data dependencies are not lost due to this. The idea is inherited from the Floyd–Hoare method of programme verification. The equivalence between paths of \( P_0 \) and \( P_1 \) may not always exist due to the application of code optimizations. Therefore, the PBEC approaches come up with ideas like path extension [11] or value propagation [9] to show the equivalence in such case.

The correspondence of states between \( M_0 \) and \( M_1 \) are defined as follow.

**Definition 7** (Corresponding States).

1. The reset states \( q_{00} \) and \( q_{10} \) are corresponding states.
2. The states \( q_{0k} \in Q_0 \) and \( q_{1l} \in Q_1 \) are corresponding states if there exists \( \beta \) from \( q_{00} \) to \( q_{0k} \) and \( \alpha \) from \( q_{01} \) to \( q_{1l} \) such that \( \beta \simeq \alpha \).

The following theorem can be concluded from the above discussion.

**Theorem 1** An FSMD \( M_0 \) is contained in another FSMD \( M_1 \) (\( M_0 \subseteq M_1 \)), if there exists a path cover \( P_0 = \{ \beta_{i0}, \beta_{i1}, \ldots, \beta_{ik} \} \) of \( M_0 \) and \( P_1 = \{ \alpha_{i0}, \alpha_{i1}, \ldots, \alpha_{ik} \} \) of \( M_1 \) such that \( \beta_{ij} \simeq \alpha_{ij} \) for all \( 0 \leq i \leq k \).

Proof. \( M_0 \subseteq M_1 \) if, for any computation \( \mu_0 \) of \( M_0 \), there exists a computation \( \mu_1 \) of \( M_1 \) such that \( \mu_0 \alpha \equiv \mu_1 \). Theorem 3 follows from Ref. [3].

Now, let there exists a path cover \( P_0 = \{ \beta_{i0}, \beta_{i1}, \ldots, \beta_{ik} \} \) of \( M_0 \). Corresponding to \( P_0 \), let a set \( P_1 = \{ \alpha_{i0}, \alpha_{i1}, \ldots, \alpha_{ik} \} \) of \( M_1 \) exists such that \( \beta_{ij} \simeq \alpha_{ij} \) for all \( i \), \( 0 \leq i \leq k \). Since \( P_0 \) covers \( M_0 \), any computation \( \mu_0 \) of \( M_0 \) can be looked upon as a concatenated path \( [\beta_{i0}, \beta_{i1}, \ldots, \beta_{ik}] \) from \( P_0 \) starting from the reset state \( q_{00} \) and ending again at the reset state of \( M_0 \). From above, it follows that there exists a sequence \( \Pi \) of states \( \langle \alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{ik} \rangle \) of \( P_1 \), where \( \beta_{ij} \simeq \alpha_{ij} \) for all \( l \), \( 0 \leq l \leq n \). Therefore, in order that \( \Pi \) represents a computation of \( M_1 \), it is required to prove that \( \Pi \) is a concatenated path of \( M_1 \) from its reset state \( q_{10} \) back to itself.

Now, let \( \beta_{i0} : q_{00} \Rightarrow q_{0f} \). Since \( \beta_{i0} \simeq \alpha_{i0} \), from the definition of corresponding states, \( \alpha_{i0} \) must be of the form \( q_{0f} \Rightarrow q_{1f} \). Hence \( q_{0f} \Rightarrow q_{1f} \) is a corresponding state. Thus, by repetitive application of the above argument, it follows that \( \beta_{i0} : q_{00} \Rightarrow q_{1f} \), \( \beta_{i1} : q_{0f} \Rightarrow q_{1f} \), \( \beta_{i2} : q_{0f} \Rightarrow q_{1f} \), \( \beta_{i3} : q_{0f} \Rightarrow q_{1f} \). Hence, \( q_{1f} \) is a corresponding state. Hence, \( \Pi \) is a concatenated path representing a computation \( \mu_1 \) of \( M_1 \), where \( \mu_0 \equiv \mu_1 \).

Two FSMDs \( M_0 \) and \( M_1 \) are equivalent, denoted as \( M_0 \equiv M_1 \), if \( M_0 \subseteq M_1 \) and \( M_1 \subseteq M_0 \). Since FSMDs are deterministic, it can be shown that \( M_0 \subseteq M_1 \) implies \( M_1 \subseteq M_0 \).

### 3.3 Overview of the VP method

A **propagated vector** for a path \( \beta \) is an ordered pair \( \langle R_\beta, \gamma_\beta \rangle \), where the \( R_\beta \) is the condition of execution and the \( \gamma_\beta \) is an updated variable vector representing the value obtained by the variables at the end state of \( \beta \), detailed can be found in Ref. [11].

The VP method [9] finds the path cover in both the behaviours by selecting all the branching states as cutpoint (static cutpoint selection strategy). The VP method of FSMDs consists in propagating the mismatched values of live variables (as propagated vectors) through all the subsequent path segments until the values match or the final path segment ending in the reset state is reached. It may be noted that in the VP method the mismatched values are propagated in subsequent paths in depth-first search (DFS) manner.

In the course of equivalence checking of two FSMDs, two paths, \( \beta \) and \( \alpha \) say (one from each FSMD), are compared with respect to their corresponding propagated vectors for finding a path equivalence. If \( R_\beta \equiv R_\alpha \) and \( \gamma_\beta = \gamma_\alpha \), then these paths are
declared as unconditionally equivalent (U-equivalent in short, denoted by $\beta \simeq \alpha$). If some mismatch in data transformation is detected, then they are declared to be conditionally equivalent (C-equivalent in short, denoted by $\beta \simeq \alpha$) provided their final state-pairs eventually lead to some U-equivalent paths; otherwise, these two paths and, therefore, two FSMDs are declared to be not equivalent. The working of VP is explained in Example 3.

4 | MOTIVATING EXAMPLES

Test data generation by hand is a difficult, expensive, and error-prone task, yet testing is a necessary part of the development process. A programme that has unstructured control flow creates problems in generation of test data. Testability transformation transforms such programs to make it easier for generation of test data (by improving the original programme’s ‘testability’) [1,2]. The presence of flag variables in a programme greatly reduces the effectiveness of the search. The removal of flag variables improves the performance of evolutionary test data generation. An exit statement inside a loop for example leads to more than one way in which the flow of control may leave the loop. Removing the exit statement from an unstructured programme creates single-entry and single-exit control flow structure and make the programme easier to understand. In this section, one example with flag removal and one example with exit removal are presented. The flag removal example is formed based the method presented in Refs. [1] and [2] and the exit statement removal example is presented based on the method [15]. The limitations of the VP method are identified with these two examples. We primarily targeted the flag and exit statements removal methods presented in Refs. [1], [2], and [15]. Since, the equivalence of programs is in general undecidable problem, our proposed method may also produce false-negative results for certain cases of testability transformations.

4.1 | Testability transformation for flag variables

The source and transformed C codes snippet of a flag removal example is given in Figure 3(a) and 3(b). Note that the original programme (Figure 3(a)) contains a flag variable and it has been removed in its equivalent transformed code is shown in Figure 3(b). However, the VP method fails to show the equivalence for these behaviours, as shown in Example 3.

Example 3. Let us consider the original behaviour $M_0$ and its transformed behaviour $M_1$ shown in Figure 4. Figures 4(a) and 4(b) shows the FSMDs corresponding to the original and transformed programme of 3, respectively. It may be noted that the transition from the last state to the reset state for these FSMDs are not shown for clarity in presentation. The same is true for the FSMDs in Figure 6 as well. For this example, the removal of the flag variable causes the merging of two adjacent conditional blocks of $M_0$ into one conditional block in $M_1$. In the equivalence checking, the VP method finds that $q_00, q_01, q_02, q_04$ of $M_0$ and $q_{10}, q_{11}, q_{13}$ of $M_1$ are cutpoints. Let the variable ordering is $\langle P_0, num, out, flag \rangle$. The propagated vector at the reset state $q_{00}(\vartheta_{10})$ is $\vartheta_{00}(\vartheta_{10}) = \langle T, \langle P_0, num, out, flag \rangle \rangle$ where $T$ stands for True. Note that flag variable is uncommon variable that is flag variable is defined in $M_0$ but not in $M_1$. Therefore, flag variable retains its symbolic value in the propagated vectors of $M_1$ throughout a computation of $M_0$. For the path $\alpha_1 = q_{00} \Rightarrow q_{01}$ of $M_0$ and the path $\alpha_2 = q_{10} \Rightarrow q_{11}$ of $M_1$, the propagated vector at $q_{01}$ and $q_{11}$ are $\vartheta_{01} = \langle T, \langle num, num, out, flag \rangle \rangle$ and $\vartheta_{11} = \langle T, \langle num, num, out, flag \rangle \rangle$, respectively. The propagated vector $\vartheta_{01}$ and $\vartheta_{11}$ are the same at $q_0$ and $q_1$. Therefore, the path $\beta_1$ is said to be U-equivalent to the path $\alpha_1$. Note that $\vartheta_{01}$ and $\vartheta_{11}$ are the same, so there is no need to propagate the propagated vector for the path emanating from $q_{01}$ and $q_{11}$. Therefore, all variables in the propagated vector associated with the states $q_{01}$ and $q_{11}$ will be reverted back to their symbolic value. Thus, the updated propagated vector at $q_{01}$ and $q_{11}$ will be $\vartheta_{01}(\vartheta_{11}) = \langle T, \langle P_0, num, out, flag \rangle \rangle$. For the path $\beta_2 = q_{02} \Rightarrow q_{03}$ of $M_0$ and the path $\alpha_2 = q_{12} \Rightarrow q_{13}$ of $M_1$, the propagated vector at $q_{02}$ and $q_{12}$ are $\vartheta_{02} = \langle P_0, 0 = 0, \langle P_0, num, out, 1 \rangle \rangle$ and $\vartheta_{12} = \langle P_0, 0 = 0, \langle P_0, num, 1, flag \rangle \rangle$, respectively. The propagated vectors $\vartheta_{02}$ and $\vartheta_{12}$ are not the same since there is a mismatch in the values of the variable out. Therefore, value propagation is required and $\beta_2$ is said to be candidate C-equivalent (i.e. condition of execution matches but data transformations do not match) to the path $\alpha_2$. As a result, mismatched values need to be propagated in the subse-quent path from the state $q_{02}$ of $M_0$ and $q_{12}$ of $M_1$. However, there is no path emanating from the state $q_{12}$ of $M_1$. Thus, the VP method fails to show the equivalence and reports that behaviours ‘may not be equivalent’.
4.2 Testability transformation for exit statements

The source and transformed C code snippet of an exit removal example is given in Figure 5(a) and 5(b). The source code (Figure 5(a)) contains an exit statement at line 7 and it has been removed in the transformed code, as shown in Figure 5(b).

The VP method can handle the scenario which involves code motions across loops. The variables whose values are propagated beyond a loop must be invariant to that loop for valid code motions across loops. To determine the loop invariant of such variables, the VP method invokes the function loopInvariant at the loop header after one traversal of the loop. Loop invariant checking details can be found in Ref. [9]. The VP method checks the loop invariant at the loop header only. For a programme with exit inside a loop has multiple exit points from the loop. The VP method cannot handle this type of scenario and reports that behaviours ‘may not be equivalent’, as shown in Example 4.

Example 4. Consider the original behaviour $M_0$ and its transformed behaviour $M_1$ shown in Figure 6. Figure 6(a) and 6(b) shows the FSMDs corresponding to the original and transformed programme of Figure 5, respectively. The VP method sets $q_{01}, q_{02}, q_{04}$ of $M_0$ and $q_{10}, q_{11}, q_{13}$ of $M_1$ as cutpoints. The path $q_{00} \Rightarrow q_{01} \Rightarrow q_{02}$ of $M_0$ and $q_{10} \Rightarrow q_{11} \Rightarrow q_{12}$ of $M_1$ are U-equivalent since the values of variables $P$ and $i$ match. Note that, in original behaviour (Figure 6(a)) for the loop body $q_{02} \Rightarrow q_{03} \Rightarrow q_{02}$, the states $q_{02}$ and $q_{03}$ are loop exit points. The FSMD $M_0$ has two exit points. During equivalence checking, the VP method identifies that the loop body $q_{02} \Rightarrow q_{03} \Rightarrow q_{02}$ contains more than one loop exit points. Therefore, the VP method terminates its execution and reports that “behaviours may not be equivalent”.

5 | PROPOSED ENHANCEMENTS

In this section, we propose solutions to overcome the limitations discussed in Section 4.

5.1 Dynamic cutpoint selection

The VP method uses a static cutpoint selection strategy (cutpoint) and selects $q_{00}, q_{01}, q_{02}$ and $q_{04}$ as cutpoints in $M_0$ (Figure 4(a)) and $q_{10}, q_{11}$, and $q_{13}$ in $M_1$ (Figure 4(b)) as cutpoints. Note that there are two conditional blocks in $M_0$. The transformed behaviour $M_1$ generated by TT for flag variables has only one conditional block. The difference in the number of the conditional block between these two behaviours hints
that we may avoid the state \( q_{02} \) as a cutpoint in \( M_0 \). If we avoid the \( q_{02} \) as cutpoint then the VP method can show the equivalence between these two behaviours. However, the static cutpoint selection strategy also selects \( q_{02} \) as a cutpoint henceforth, the VP method fails to show the equivalence.

We propose a dynamic cutpoint selection scheme, cutpointSelection (Algorithm 1), to choose the proper cutpoints in both the source (say \( M_0 \)) and transformed (say \( M_1 \)) behaviours. The function cutpointSelection selects all the loop headers in \( M_0 \) and \( M_1 \) as cutpoints so that each loop is cut in at least one cutpoint [33] (step 1). The rest of the function finds other set of cutpoints. The function cutpointSelection uses the dominator tree concept [35] to find the number of conditional blocks between two states. It computes the immediate dominator and PIDom of each state in steps 1 and 1, respectively. It computes the dominator tree for each FSMD in step 1. Note that in cutpointSelection method, we find the number of conditional block exists between an articulation node and its PIDom in each FSMD. If the number of conditional blocks are the same for both the FSMDs then we declare the all conditional nodes as cutpoint exists between the articulation point and its PIDom in each FSMD. Otherwise, we do not select any of the nested condition as cutpoint. For each articulation node (say \( q_{0i} \)) in \( M_0 \), it invokes the function checkCondWithZ3 to find the corresponding state (say \( q_{1j} \)) in \( M_1 \) (step 1). If checkCondWithZ3 succeeds (i.e., condition at both states matches) then cutpointSelection invokes the function findCondBlock to find the number of conditional blocks (say \( n_j \)) exists in between the state \( q_{0i} \) and its post immediate dominator (step 1). Similarly, it finds the number of conditional blocks (say \( n_i \)) in between the state \( q_{1j} \) and its post immediate dominator (step 1). If \( n_i = n_j \) then it invokes the function makeCutNode to make all the conditional node as cutpoints exist in between \( q_{0i} \) and its post immediate dominator (steps 1 and 1). Note that the function cutpointSelection finds the cutpoints at step 1 in bi-simulation manner. The function cutpointSelection repeats the steps 1 to 1 for each articulation node present in \( M_0 \). Example 5 shows if the VP method uses the function cutpointSelection to find cutpoints for the behaviours shown in Figure 4 then it can show the equivalence.

Example 5. Consider the original behaviour \( M_0 \) and its transformed behaviour \( M_1 \) shown in 4. There are two conditional states \( q_{01} \) and \( q_{02} \) in the FSMD \( M_0 \) whereas there is only one conditional state \( q_{11} \) in the FSMD \( M_1 \). For the articulation node \( q_{01} \) in the function cutpointSelection finds the state \( q_{11} \) in \( M_1 \). The function declares the states \( q_{01} \) and \( q_{11} \) as cutpoints. The state \( q_{01} \) is PIDom of \( q_{02} \) and the state \( q_{11} \) is PIDom of \( q_{12} \). Note that there is no internal conditional block between the state \( q_{01}(q_{11}) \) and its PIDom \( q_{02}(q_{12}) \). So, no further cutpoint is selected inside this conditional block. In the next step, the function cutpointSelection finds the state in \( M_1 \) for the articulation point \( q_{02} \). However, it does not find any state in \( M_1 \) for the state \( q_{02} \). So, \( q_{02} \) is not selected as cutpoint. Thus, the function cutpointSelection (Algorithm 1) selects \( q_{00}, q_{01}, q_{04} \) of \( M_0 \) and \( q_{10}, q_{11}, q_{13} \) of \( M_1 \) as cutpoints.

Now consider the equivalence checking between these two behaviours. The path \( q_0 \Rightarrow q_{01} \) of \( M_0 \) is found to be \( U \)-equivalent to the path \( q_0 \Rightarrow q_{11} \) of \( M_1 \) since the value of \( P_0 \) matches. Note that the state \( q_{02} \) of \( M_0 \) is not selected as cutpoint. Therefore, there are two valid paths from the state \( q_{01} \): \( \alpha_1: q_{01} \rightarrow q_{02} \rightarrow q_{03} \rightarrow q_{04} \) and \( \alpha_2: q_{01} \rightarrow q_{02} \rightarrow q_{03} \rightarrow q_{04} \). The path \( \alpha_1 \) is \( U \)-equivalent to the path \( q_{11} \Rightarrow q_{12} \Rightarrow q_{13} \) and the path \( \alpha_2 \) is \( U \)-equivalent to the path \( q_{11} \Rightarrow q_{12} \Rightarrow q_{13} \) since the values of all the variables match. Hence, the VP_TT method reports \( M_0 \) and \( M_1 \) are equivalent.

5.2 Handling multiple exits from loop

As shown in Example 4, the VP method fails if a behaviour has more than one loop exit point. The VP method must also check the loop invariant at all the loop exit points inside the loop. Example 6 shows that if the VP method checks the validity of code motion at all the exit points inside the loop body, then it shows the equivalence for the behaviours shown in Figure 6.

Example 6. Consider the original behaviour \( M_0 \) and its transformed behaviour \( M_1 \) shown in 6. In the course of equivalence checking, the path \( q_{01} \Rightarrow q_{02} \Rightarrow q_{03} \) of \( M_0 \) and \( q_{10} \Rightarrow q_{11} \Rightarrow q_{12} \) of \( M_1 \) found to be \( U \)-equivalent since the values of variables \( P_0 \) and \( i \) match. Next, for the path \( q_{02} \Rightarrow q_{03} \), there does not exist any equivalent path in \( M_1 \), but the condition at the state \( q_{12} \) implies the condition at the state \( q_{03} \), so need to extend the path at the state \( q_{03} \). In this case, value propagation is required and need to store the propagated vector \( \theta_{03} \) at the state \( q_{03} \). Therefore, with respect to \( \theta_{03} \) at the state \( q_{03} \) there are two paths \( \alpha_1: q_{03} \rightarrow q_{02} \) and
6 | OVERALL VERIFICATION METHOD

In this section, we present our VP_TT verification method. Note that we use the functions of the VP method [1] as they are except the function correspondenceChecker and containmentChecker functions. The overall flow of our verification method is given in Figure 7. We start the procedure of equivalence checking of FSMDs $M_0$ and $M_1$ by invoking the function containmentChecker (Algorithm 2). The function containmentChecker first invokes the function cutpointSelection to find the appropriate cutpoints and then computes the path cover for both the FSMDs. The function containmentChecker invokes the function correspondenceChecker (Algorithm 3) for each corresponding state pairs to check the equivalence for all paths starting from the corresponding state pairs in FSMDs $M_0$ and $M_1$. The function correspondenceChecker invokes the function findEquivalentPath to find a U- or C-equivalent path $\alpha : (q_0 \Rightarrow q_1)$ in the transformed FSMD $M_1$ for each path $\beta : (q_0 \Rightarrow q_m)$ of the original FSMD $M_0$. The input vector for the function findEquivalentPath contains a path $\beta$, the propagated vector $\vartheta_0$ associated with start state of $\beta$, the corresponding state $q_i$ of $q_0$, the propagated vector $\vartheta_1$ associated with the state $q_j$, and the path cover $P_0$ and $P_1$ of $M_0$ and $M_1$, respectively. The correspondenceChecker uses function loopInvariant to check the loop invariant at each loop exit points. We have used the same definition of these two functions from Ref. [9] in our work. Therefore, the detail of them are omitted for brevity. At any point of time if the function correspondenceChecker returns ‘Failure’, then containmentChecker reports ‘$M_0$ and $M_1$ may not be equivalent’ and exit the programme. Furthermore, in case of successful completion of for loop (line 2 of Algorithm 2), it returns ‘$M_0 \subseteq M_1$’.

The function findEquivalentPath returns a 4-tuple $\langle \delta(\alpha), \vartheta(\alpha), \vartheta(\beta), \vartheta(\beta) \rangle$ where $\delta(\alpha)$ and $\vartheta(\alpha)$ are corresponding paths as described above, $\vartheta(\beta)$ is the propagated vector at the end state $q_m$ of $\beta$ and $\vartheta(\beta)$ is the propagated vector at the end state $q_m$ of $\alpha$. If $\vartheta(\alpha) \equiv \vartheta(\beta)$, then the path $\alpha$ is U-equivalent to path $\beta$. Consequently, the data structure $W_{\text{exp}}$ gets updated (line 5.2). If $\text{findEquivalentPath}$ does not find any path $\alpha$ in $M_1$ whose condition of execution $R_\alpha$ satisfies either $R_\beta \equiv R_\alpha$ or $R_\alpha \Rightarrow R_\beta$, or $R_\beta \Rightarrow R_\alpha$ then it returns $\alpha = \text{NULL}$ (i.e. $M_0$ and $M_1$ may not be equivalent, handled in line 5.2). If $\alpha \neq \text{NULL}$ and $\vartheta(\alpha) \equiv \vartheta(\beta)$, then the path $\alpha$ is candidate C-equivalent to the path $\beta$ and hence further value propagation is required. However, the

---

**Algorithm 2 containmentChecker($M_0$, $M_1$)**

1. **cutpointSelection($M_0$, $M_1$);**
2. Compute the path cover $P_0$ and $P_1$ of $M_0$, $M_1$, respectively; $W_{\text{exp}}$ is a set of corresponding state pairs and initially contains $(q_0, q_1)$; $E_u$ contains the pair of U(C)-equivalent pair; $E_u$ contains the candidate of C-equivalent path pairs; LIST: a list of paths maintained to keep track of C-equivalent paths; initially these lists are empty; $\vartheta_0$, $\vartheta_1$ are the propagated vectors associated with the states $q_0$, $q_1$ of $M_0$ and $M_1$, respectively;
3. **for each state** $(q_0, q_1) \in W_{\text{exp}}$ **do**
   4. $\text{output} \leftarrow \text{correspondenceChecker}(q_0, q_1, P_0, P_1, W_{\text{exp}}, E_u, E_c, \text{LIST}, \vartheta_0, \vartheta_1)$;
5. **end**
6. **if output is “Failure” then**
   7. Report “Failure detected in correspondence checker, $M_0$ and $M_1$ May Not be Equivalent” and exit;
8. **end**
9. Report “$M_0 \subseteq M_1$”;

**Algorithm 3 correspondenceChecker($q_i$, $q_j$, $P_0$, $P_1$, $W_{\text{exp}}$, $E_u$, $E_c$, LIST, $\vartheta_0$, $\vartheta_1$)**

1. **foreach path** $\beta : (q_0 \Rightarrow q_m)$ in $P_0$ **do**
   2. $(\beta, \alpha, \vartheta_0, \vartheta_1) \leftarrow \text{findEquivalentPath}(\beta, \alpha, q_j, q_j, P_0, P_1)$;
3. **if** path $\alpha : (q_j \Rightarrow q_1)$ **can be found in** $P_1$ **such that** $\beta \equiv U \alpha$ **then**
   4. $E_u = E_u \cup \{((\beta, \alpha))\}$;
   5. $W_{\text{exp}} = W_{\text{exp}} \cup \{q_i, q_j\}$;
   6. **else** if path $\alpha : (q_j \Rightarrow q_1)$ **can be found in** $P_1$ **such that** $\beta < U \alpha$ **then**
   7. $q_i$ or $q_j$ is reset state then
   8. **return failure**;
   9. **end**
   10. **if** $q_i$ or $q_j$ is a loop break point and $\text{loopInvariant}(\beta, \alpha)$ **then**
   11. **return failure**;
   12. **else**
   13. $\vartheta_0, \vartheta_1 \leftarrow \vartheta_0, \vartheta_1$;
   14. $\text{append}(\beta, \alpha)$ to LIST;
   15. $\text{correspondenceChecker}(q_0, q_j, P_0, P_1, W_{\text{exp}}, E_u, E_c, \text{LIST}, \vartheta_0, \vartheta_1)$;
   16. **end**
   17. **else**
   18. **return failure**;
  19. **end**
20. $E_c = E_c \cup \{\text{Last member of LIST}\}$;
21. LIST $\leftarrow$ LIST$\setminus$\{Last member of LIST\};
22. **return success**;
The verification method ensures the validity of code motion across the loop. The function `cutpointSelection` used by the VP method by the VP_TT enhances the VP method by replacing the static cutpoint selection strategy used by the VP method by the dynamic cutpoint and by calling the function `loopInvariant` at all loop exit points. Therefore, we have to ensure that our enhancements do not compromise the soundness of the VP method. The function `cutpointSelection` is used to select cutpoints dynamically. This function selects all loop headers as cutpoints. Therefore, this function selects enough cutpoints to cut all loops in a behaviour. Therefore, the paths between cutpoints are indeed a path cover of an FSMD. Calling the function `loopInvariant` at all loop exits points the VP_TT method ensures the validity of code motion across the loop body. Hence, both enhancements on the VP method do not affect the soundness of the VP method. In the following the proof of soundness is explained intuitively. After introducing the cutpoints, the verification method constructs the path cover and starts with the reset states. For each path $\beta$ on $M_0$, the verification method searches for corresponding U- or C-equivalence path $\alpha$ in other FSMD $M_1$ in a depth-first manner.

Let suppose that the verification method is not sound that is the algorithm `containmentChecker` terminates successfully and the path $\beta$ is C-equivalent path $\alpha$ and the path $\alpha$ leads to the reset state. However, in this case, the function `correspondenceChecker` returns failure to `containmentChecker`, as shown at step 2; consequently, `containmentChecker` terminates at step 2, not at step 2. Hence it is a contradiction and the verification method is sound. The soundness proof of the VP method is given in Ref. [9].

### 7.2 Termination

Figure 7 represents the overall flow of our verification method. The function `loopInvariant` always terminates since it involves a comparison of two propagated vectors. The function `cutpointSelection` finds the cutpoint in a given FSMD. In worst case, it checks all the states of the FSMD. Since the number of states in an FSMD is finite, Algorithm 1 also terminates. The function `findEquivalentPath($\beta$, $q_j$, ...)` tries to find a path $\alpha$ starting from $q_{ij} \in M_1$ such that $\beta \equiv \alpha$ or $\beta \equiv \alpha$. It checks all the transitions from $q_{ij}$ in the worst case. Hence it terminates as well. The function `correspondenceChecker` tries to find a path $\alpha$ starting from $q_{ij} \in M_1$. In worst case, it checks all the paths of $P_1$ starting from $q_{ij}$ which is finite. The recursive call of the function `correspondenceChecker` does not extend beyond the reset state. Therefore, the function `correspondenceChecker` invokes itself recursively only a finite number of times. Hence, Algorithm 3 also terminates.

### 7.3 Complexity

Let $n$ be the number of states in an FSMD and $k$ be the maximum number of parallel edges between any two states and $x$ is time taken to check the equivalence of the two formulas. The maximum possible state transitions from a state is $kn$.

In worst case, for a path $\beta$ in $M_0$ the function `correspondenceChecker` checks all the transitions from $q_{ij}$ to find a path $\alpha$ in $M_1$ such that $\beta \equiv \alpha$ or $\beta \equiv \alpha$. The complexity of finding the path $\alpha$ is $O(x \cdot k \cdot n)$. The number of times `correspondenceChecker` is called from `containmentChecker` is the same as the size of the set of corresponding states pairs which is $O(n)$. In worst case, all the states of $M_0$ can be cutpoints and for each path, the mismatched values are propagated in subsequent paths in depth first search (DFS) manner until the reset state is reached. In this case, `correspondenceChecker` can recursively call itself $k \cdot (n-1) + k^2 \cdot (n-1) \cdot (n-2) + \cdots + k^{n-1} \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \approx k^{n-1} \cdot (n-1)^{n-1}$

Therefore, the complexity of the overall verification method is $O(x \cdot k \cdot n) \cdot n \cdot k^{n-1} \cdot (n-1)^{n-1}$. The worst case complexity of the our verification method is the same as that of the VP method.
We have enhanced the VP method [9] (VP_TT) by incorporating all proposed solutions discussed in Section 5. The VP_TT method is implemented in C. All the experiments have been conducted on a desktop with 1.8 GHz Intel i5 processor with 8 GB of RAM. The benchmarks listed in rows 1–5 of Table 1 have some exit statements and benchmarks listed in rows 6–10 of Table 1 have some flag variables. The benchmarks PRIME checks whether a number is prime; FACT computes the factorial of given number; REPTD checks the repeated number present in a given list; MAX finds the maximum number in a given series; MAXGCD finds the maximum GCD of series; EVEN checks a number is even or odd; PERF checks a number is perfect or not; MIN and MINMAX find the minimum and maximum number in a given series; and ROTATE rotates the series on a given point.

The transformed behaviour ($M_1$) in Table 1 is obtained by applying the TTs for exit and flag on the source behaviour ($M_0$).

In this experiment, we compare the VP_TT with the VP method [9], EVP method [24] and the VP_TT_static method. The EVP method is an enhanced version of the VP method and capable of handling the scenario which involves loop invariant code motion. The VP_TT_static method is another variant of the VP_TT method which uses static cutpoint selection strategy instead dynamic cutpoint selection strategy. The static cutpoint selection strategy is also used by the VP and EVP methods. Note that the VP_TT method uses dynamic cutpoint selection strategy (Algorithm 1). In Table 1, 2nd and 3rd columns represent the number of cutpoints selected by the static cutpoint selection strategy while 10th and 11th columns represent the number of cutpoints selected by the dynamic cutpoint selection strategy (i.e. Algorithm 1) for a given benchmark. This experiment confirms that the VP_TT method shows the equivalence for the benchmarks listed in Table 1. This is reported as ‘Eq’ in Table 1. However, the VP and EVP method reports ‘May Not be Equivalent’ in these benchmarks and denoted as ‘MNEq’ in Table 1. The VP_TT_static method can handle exit statements removal scenario but the fails handle flag variables removal scenario. It is evident from this experiment that the solution proposed reduces the false-negative cases of state-of-the-art PBEC methods. The VP_TT method takes more time in this experiment as compared with the VP and EVP method because these methods do not complete the execution and exits early since it identifies a possible case of non-equivalence. Whereas, our method completes the execution and shows the equivalence. The runtime of both VP_TT_Static and VP_TT are comparable for this experiment.

In the second experiment, we check the effect of our dynamic cutpoint selection on the run time of the equivalence checking procedure. The benchmarks used for this experiment are taken from Ref. [9]. We modify the VP_TT method so that it can use both cutpoint selection strategies in the course of equivalence checking. The result of this experiment is tabulated in Table 2. The number of cutpoints selected by both the

| Benchmarks | Cutpoint | VP_TTSTATIC | VP_TT | VP | EVP |
|------------|----------|-------------|------|---|-----|
| PRIME | 4 | 4 | 4 | MNEq | MNEq |
| FACT | 4 | 4 | 4 | MNEq | MNEq |
| REPTD | 4 | 4 | 4 | MNEq | MNEq |
| MAX | 7 | 7 | 7 | MNEq | MNEq |
| MAXGCD | 7 | 7 | 7 | MNEq | MNEq |
| EVEN | 4 | 4 | 4 | MNEq | MNEq |
| PERF | 6 | 6 | 6 | MNEq | MNEq |
| MIN | 7 | 7 | 7 | MNEq | MNEq |
| ROTATE | 5 | 5 | 5 | MNEq | MNEq |
| MINMAX | 7 | 7 | 7 | MNEq | MNEq |
static and dynamic cutpoint selection methods are tabulated in Table 2. The 4th and 8th columns, ‘CC’ (i.e. #correspondenceChecker) represent the number of times correspondenceChecker function is called while using static and dynamic cutpoint selection strategy, respectively. Note that these benchmarks do not contain any exit statements or flag variables. The VP_TT method can show the equivalence with both the cutpoint selection strategies for these benchmarks. The run time comparison between these two schemes depends on two factors. Dynamic cutpoint selection involves some data flow analysis, whereas static cutpoint selection is done by just checking the states of an FSM. Therefore, dynamic cutpoint selection takes more time as compared to static cutpoint selection. The number of cutpoints selected by dynamic scheme is always less than or equal to that of static scheme. It is evident from both Table 1 and Table 2. Therefore, the number of calls to correspondenceChecker for dynamic scheme is less than or equal to that of static scheme. This improves the runtime of dynamic scheme over static scheme. In case, where CC values are less for dynamic scheme compared with static scheme, the run time of VP_TT improves for dynamic scheme over static scheme. It is evident from the run time shown in Table 2. Therefore, the VP_TT method takes less time for dynamic cutpoint selection strategy than the static cutpoint selection strategy. It may be noted that the dynamic cutpoint selection strategy also helps the VP_TT method to show the equivalence for flag TT as shown in our previous experiment.

In the next experiment, we analyse the performance of our proposed method with respect to its worst case complexity. As discussed in Section 7.3 the VP_TT takes exponential time in the worst case. Note that, the most dominating factor in the time complexity of VP_TT method is how many times correspondenceChecker is called from containmentChecker. In Table 2, we compare the number of times correspondenceChecker function is called by VP_TT method in practice (denoted as Actu. #CC in 8th column) against its theoretical estimation in the worst case (denoted as Theo. #CC in 9th column) for each benchmark listed in Table 1. From the results, it is clearly visible that the static cutpoint selection method calls the correspondenceChecker function less number of times as compared with its theoretical estimation for all the benchmarks. Thus, we can conclude that it is unlikely the our proposed method exposes the worst case behaviour for a wider selection of typical programs in practice.

9 | CONCLUSION
We have presented VP_TT method for the verification of testability transformations. We have shown that the VP method fails to show the equivalence for TTs involving flag and exit removals. We have proposed the solutions and presented the VP_TT method to overcome the limitation of the VP method. Specifically, we have presented a dynamic cutpoint selection strategy to handle flag removal and an enhancement to VP method to handle multiple exits from a loop to handle exit removal. We have performed the experiments on several TT test cases. The results show that VP_TT outperforms the VP method in the verification of testability transformations. The VP_TT method fails for the testability transformation presented in Ref. [36] since it checks the equivalence of loops without unrolling the loops. To handle transformation presented in Ref. [36], checking one iteration of the loop is not sufficient. Rather, one has to identify the loop invariant for loops. In future, we intend to enhance the VP_TT method to validate such testability transformations.

ACKNOWLEDGEMENT
C. Karfa was partially supported by the DST, Government of India under Project CRG/2019/001300.

ORCID
Chandan Karfa https://orcid.org/0000-0002-3835-4184

REFERENCES
1. Harman, M., Hu, L., Hierons, R., Baresel, A., Shamer, H.: Improving evolutionary testing by flag removal, in GECCO 2002. In: Proceedings of Genetic and Evolutionary Computation Conference, pp. 1359–1366. San Francisco (2002)
2. Harman, M., Hu, L., Hierons, R., Wegener, J., Shamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Software Eng. 30(no. 1), 3–16 (2004)
3. Harman, M., Baresel, A., Binkley, D., Hierons, R., Hu, L., Korel, B., McMinn, P., Roper, M.: Testability transformation–programme
transformation to improve testability. In: Formal methods and testing, pp. 320–344. Springer, Berlin (2008)
4. Harman, M.: We need a testability transformation semantics. In: International Conference on Software Engineering and Formal Methods, pp. 3–17. Springer, Toulouse (2018)
5. Gong, D., Yao, X.: Testability transformation based on equivalence of target statements. Neural Comput. Appl. 21(no. 8), 1871–1882 (2012)
6. Li, Y., Fraser, G.: Bytecode testability transformation. In: International Symposium on Search Based Software Engineering, pp. 237–251. Springer, Szeged (2011)
7. McMinn, P., Binkley, D., Harman, M.: Empirical evaluation of a nesting testability transformation for evolutionary testing. ACM Trans. Software Eng. Methodol. 18(no. 3), 1–27 (2009)
8. Pnueli, A., Siegel, M., Singerman, E.: Translation Validation. TACAS, ser. LNCS, vol. 1384, pp. 151–166 (1998)
9. Banerjee, K., Karfa, C., Sarkar, D., Mandal, C. A.: Verification of code motion techniques using value propagation. IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst. 33(no. 8), 1180–1193 (2014)
10. Kim, Y., Mansouri, N.: Automated formal verification of scheduling with speculative code motions. In: Proceedings of the 18th ACM Great Lakes Symposium on VLSI 2008, pp. 95–100. ACM, Orlando (2008). May
11. Karfa, C., Sarkar, D., Mandal, C., Kumar, P.: An equivalence-checking method for scheduling verification in high-level synthesis. IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst. 27(no. 3), 556–569 (2008). Mar.
12. Lee, C., Shih, C., Huang, J., Jou, J.: Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In: ASPDAC 2011, pp. 497–502, Yokohama (2011)
13. Karfa, C., Mandal, C. A., Sarkar, D: Formal verification of code motion techniques using data-flow-driven equivalence checking. ACM TODAES. 17(no. 3), 30 (2012)
14. Li, T., Hu, J., Guo, Y., Li, S., Tan, Q.: Equivalence checking of scheduling in high-level synthesis. In: Sixteenth International Symposium on Quality Electronic Design, pp. 257–262. IEEE, Santa Clara (2015)
15. Hierons, R. M., Harman, M., Fox, C. J.: Branch-coverage testability transformation for unstructured programs. Comput. J. 48(no. 4), 421–436 (2005)
16. Necula, G. C.: Translation validation for an optimising compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 83–94. Vancouver, British Columbia (2000)
17. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for llvm. In: International Conference on Computer Aided Verification, pp. 737–742 (2011)
18. Tristan, J.-B., Goverceau, P., Morrisett, G.: Eviating value-graph translation validation for llvm. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 295–305. San Jose (2011)
19. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Asian Symposium on Programming Languages and Systems, pp. 127–147. Springer (2017)
20. Sharma, R., Sulkuzlu, E., Churehill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, ser. OOPSLA ‘13, pp. 391–406. Indianapolis (2013)
21. Gupta, S., Saxena, A., Mahajan, A., Bansal, S.: Effective use of SMT solvers for programme equivalence checking through invariant-sketching and query-decomposition. In: Theory and Applications of Satisfiability Testing SAT 2018, vol. 10929, pp. 365–382. Springer, Oxford (2018)
22. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic programme alignment for equivalence checking. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. ser. PLDI 2019, pp. 1027–1040. Phoenix (2019)
23. Kim, Y., Kopuri, S., Mansouri, N.: Automated formal verification of scheduling process using finite state machines with datapath (FSMD). In: Isqel 2004, pp. 110–115. IEEE Computer Society (2004). Mar
24. Chouksey, R., Karfa, C., Bhaduri, P.: Translation validation of code motion transformations involving loops. IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst. 38(no. 7), 1378–1382 (2019)
25. Chouksey, R., Karfa, C., Banerjee, K., Kalita, P. K., Bhaduri, P.: Counterexample generation procedure for path-based equivalence checkers. IET Softw. 13(no. 4), 280–285 (2019)
26. Chouksey, R., Karfa, C.: Verification of scheduling of conditional behaviours in high-level synthesis. IEEE Trans. Very Large Scale Integr. Syst. 28(no. 07), 1638–1651 (2020)
27. Vaas, J. M., Miller, K. W.: Software testability: the new verification. IEEE Softw. 12(no. 3), 17–28 (1995)
28. McGregor, J. D., Srinivas, S.: A measure of testing effort. In: COOTS. vol. 96, pp. 10–10. Toronto (1996)
29. Baresel, A., Schamer, H.: Evolutionary testing of flag conditions. In: Genetic and Evolutionary Computation Conference, pp. 2442–2454. Springer, Chicago (2003)
30. Ramshaw, L.: Eliminating go to’s while preserving program structure. J. ACM. 35(no. 4), 893–920 (1988)
31. Ward, M. P.: “Reverse engineering through formal transformation: Knuth’s ‘polynomial addition’ algorithm. Comput. J. 37(no. 9), 795–813 (1994)
32. Gajski, D. D., Dutt, N. D., Wu, A. C.-H., Lin, S. Y.-L.: High-level Synthesis: Introduction to Chip and System Design. Kluwer Academic Publishers, Norwell, MA, USA (1992)
33. Floyd, R. W.: Assigning meanings to programs. Math. Aspects Comput. Sci.. 19(no. 1), 19–32 (1967)
34. Banerjee, K., Sarkar, D., Mandal, C.: Deriving bisimulation relations from path extension based equivalence checkers. IEEE Trans. Softw. Eng. 43(no. 10), 946–953 (2017)
35. Lowry, E. S., Medlock, C. W.: Object code optimization. Commun ACM. 12(no. 1), 13–22 (1969)
36. Binkley, D. W., Harman, M., Lakhotia, K., Flagmemo, “A testability transformation for transforming loop-assigned flags. ACM Trans. Softw. Eng. Methodol. 20(no. 3), 1–33 (2011)

How to cite this article: Chouksey R, Maddheshiya SK, Karfa C. VP_TT: A value propagation based equivalence checker for testability transformations. IET Soft. 2021;15:147–159. https://doi.org/10.1049/sfw2.12008