Improving Performance of a Path-Based Equivalence Checker using Counter-Examples

Ramanuj Chouksey*, Chandan Karfa† and Purandar Bhaduri‡
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati, 781039, India
Email: *r.chouksey@iitg.ac.in, †ckarfa@iitg.ac.in, ‡pbhaduri@iitg.ac.in

Abstract—Path-based equivalence checkers (PBECs) have been successfully applied for verification of programs from diverse domains and at various stages of high-level synthesis. These verifiers can be sound but not complete. Therefore, non-equivalence cases require further investigation of the two programs being compared by some human expert. In this work, we show how a counter-trace (cTrace) can be generated in the case of non-equivalence reported by the PBEC. We show how a Bounded Model Checker (CBMC) can be used to find suitable initialization values for input variables (i.e., a counter-example) for a given cTrace. With our counter-example generation framework, we show how a strong non-equivalence decision can be taken in a PBEC. We also show that some false negative cases of the PBEC can also be revealed using this framework. Experimental results demonstrate the usefulness of our method.

Index Terms—Equivalence Checking, Finite State Machine with Datapath (FSMD), CBMC, Counter-example Generation.

I. INTRODUCTION

High-level synthesis (HLS) is the process of translating a behavioral description into a Register Transfer Level (RTL) description [1]. HLS tools are large and complex software systems and are very often written without formally proving their correctness. Many path-based approaches [2]–[8] have been proposed for verification of the scheduling phase of HLS where each behavior is represented by a finite state machine with datapaths (FSMD) [1]. In general, path-based approaches decompose each FSMD into a finite set of finite paths and the equivalence of FSMDs is established by showing path level equivalence between two FSMDs. In the case of non-equivalence, these approaches do not provide information sufficient for debugging the issue. A counter-example which will demonstrate the non-equivalence between the input behavior to HLS (i.e., source behavior) and the scheduled behavior generated by HLS (i.e., transformed behavior) will add significant value to the adoption of such PBECs. In this case, PBEC can report “Not equivalent” instead of “May Not be equivalent” Equivalence checking of programs is an undecidable problem in general. Therefore it is possible that a PBEC may produce a false negative result, i.e., a PBEC may report that two behaviors “May Not be equivalent” but these two behaviors are actually equivalent. The process of generating a counter-example helps to identify some false negative cases of a PBEC. Thus, a counter-example generation procedure helps to improve the performance of a PBEC.

Specifically, the contributions of the paper are as follows: 1) We show how the equivalence information of the enhanced value propagation (EVP) based PBEC [8] can be used to find a cTrace in the case of non-equivalence reported by the PBEC. 2) We show how the CBMC [9] tool can be used to find a suitable counter-example for a given cTrace. 3) We show how to improve the performance of the PBEC using this counter-example generation framework. 4) An enhanced version of PBEC [8] after incorporating our counter-example generation scheme is also presented.

To the best of our knowledge, this is the first work which reports a cTrace in the case of non-equivalence and uses it to produce a counter-example and improve the performance of PBECs during verification of the scheduling phase of HLS.

The rest of this paper is organized as follows. Section II describes an FSMD model and the EVP method. Section III focuses on cTrace generation. Section IV presents how that cTrace can be used to produce a counter-examples using CBMC. Section V and VI finally delve into how current PBECs can be enhanced by incorporating our counter-example generation technique. Experimental results are given in Sec. VII. Section VIII contains a summary of the related work. Section IX concludes the paper.

II. THE FSMD MODEL AND PATH-BASED EQUIVALENCE CHECKING

This section briefly explains the FSMD model and the EVP method [8]. The details can be found in [5], [8].

An FSMD is defined as a 7-tuple \( \langle Q,q_0,I,V,O,f:Q \times 2^S \rightarrow Q,h:Q \times 2^S \rightarrow U \rangle \), where \( Q \) is the finite set of states, \( q_0 \) is the reset state, \( I \) is the set of input variables, \( V \) is the set of storage variables, \( O \) is the set of output variables, \( f \) is the state transition function, \( h \) is the update function of the output and the storage variables. Here \( U \) represents a set of storage and output assignments and \( S \) represents a set of relations over arithmetic expressions and Boolean literals.

A computation of an FSMD is a finite walk from the reset state \( q_0 \) to itself, and \( q_0 \) should not occur in between. The papers [5], [8] breaks down an FSMD into smaller segments by introducing cutpoints so that each loop in an FSMD is cut at at least one cutpoint. A path \( \alpha \) is a finite sequence of states from a cutpoint to another cutpoint without an intermediate occurrence of a cutpoint. The condition of execution \( R_\alpha \) of a path \( \alpha \) is a logical expression over \( I \cup V \) such that \( R_\alpha \) is
satisfied by the (initial) data state of the path iff the path $\alpha$ is traversed. The data transformation $r_\alpha$ of a path $\alpha$ over $V$ is the tuple $(s_\alpha, \theta_\alpha)$; the first member $s_\alpha$ represents the value of the variables $v_i$ after the execution of the path in terms of the initial state $s_0$ of the path; the second member $\theta_\alpha$ represents the output list along the path $\alpha$. Two paths $\alpha$ and $\beta$ are equivalent denoted by $\alpha \simeq \beta$ if $R_\alpha \equiv R_\beta$ and $r_\alpha = r_\beta$. A finite set of paths $P = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ is said to be a path cover of an FSM $M$ if any computation of $M$ can be looked upon as a concatenation of paths from $P$ [10].

An FSM $M_0$ is contained in another FSM $M_1$, symbolically $M_0 \subseteq M_1$, if there exists a finite path cover $P_0 = \{\alpha_1, \alpha_2, \ldots, \alpha_l\}$ of $M_0$ for which there exists a set $P_1 = \{\beta_1, \beta_2, \ldots, \beta_l\}$ of paths of $M_1$ such that $\alpha_i \simeq \beta_i$, $1 \leq i \leq l$. Two FSMs $M_0$ and $M_1$ are said to be computationally equivalent ($M_0 \equiv M_1$), if $M_0 \subseteq M_1$ and $M_1 \subseteq M_0$.

The EVP method [8] is based on propagating the mismatched variable values (as propagated vectors) over a path to the subsequent paths until the values match or the final path segments end in the reset state without a match. During the course of equivalence checking of two behaviors, two paths, $\alpha$ and $\beta$ say, (one from each behavior) are compared with respect to their corresponding propagated vectors for finding path equivalence. If $R_\beta \equiv R_\alpha$ and $r_\beta = r_\alpha$, then these paths are declared as unconditionally equivalent (U-equivalent, represented as $\alpha \simeq \beta$); if some mismatch is detected in data transformation, then they are declared to be conditionally equivalent (C-equivalent, represented as $\alpha \simeq_c \beta$), if their final state-pair always eventually lead to some U-equivalent paths; otherwise they are declared to be not equivalent.

### III. Counter-Trace Generation

Suppose the source behavior and the transformed behavior are represented as FSMs $M_0$ and $M_1$, respectively. Let us assume that the PBEC fails to find an equivalent for the path $\alpha$ of $M_0$. We now discuss how to generate a unique computation starting from the reset state that leads to the path $\alpha$. It may be noted that the EVP method maintains two lists: EQ_LIST contains equivalent path pairs explored so far and C_LIST contains candidates for conditionally equivalent path pairs. In the EVP method C_LIST is obtained in a depth first search (DFS) manner. So, if we traverse backward from the start state of $\alpha$, we will obtain a sequence of paths from the set C_LIST. This trace would always be a unique trace. Let the sequence be $\langle p_{o_1}, p_{o_1}+1, \ldots, p_{o_k}, \alpha \rangle$ in FSM $M_0$. The segment of the FSM $M_0$ from the reset state $q_{00}$ to the start state of $p_{o_1}$, (say $p_{o_1}'$) is already proved to be equivalent to its corresponding part in FSM $M_1$. However, there may be many paths from $q_{00}$ to $p_{o_1}'$. For our purpose, we can choose one of the paths from this segment. Let us choose the sequence $\langle p_{o_0}, p_{o_1}, \ldots, p_{o_i} \rangle$ where $p_{o_0}$ starts from the state $q_{00}$ and the path $p_{o_i}$ ends at $p_{o_1}'$. Therefore, the sequence $cTrace = \langle p_{o_0}, p_{o_1}, \ldots, p_{o_i}, p_{o_j}, p_{o_j}+1, \ldots, p_{o_k}, \alpha \rangle$ is the cTrace in the FSM $M_0$ that we are interested in. From EQ_LIST, we will obtain the paths corresponding to $p_{00}, p_{01}, \ldots, p_{oi}$ in FSM $M_1$. Let the corresponding paths be $p_{10}, p_{11}, \ldots, p_{1i}$, respectively. Similarly, the corresponding paths of $p_{0j}, p_{0j+1}, \ldots, p_{ok}$ in the FSM $M_1$ can be found using C_LIST. Let the corresponding paths be $p_{1j}, p_{1j+1}, \ldots, p_{1k}$, respectively. The potential corresponding path of $\alpha$ can also be obtained in the FSM $M_1$; let it be $\beta$. The EVP method identifies the potential candidate for equivalence, $\beta$, in $M_1$ in most of the cases (see [8] for details). It fails to find $\beta$ only if there does not exist any path from the corresponding state in $M_1$ whose condition of execution matches even partially with that of $\alpha$. In this case, we can take any path from the corresponding state in $M_1$. Therefore, the corresponding cTrace in FSM $M_1$ is $\langle p_{10}, p_{11}, \ldots, p_{i1}, p_{1j}, p_{1j+1}, \ldots, p_{1k}, \beta \rangle$.

**Example 1.** Consider the input behavior $M_0$ and its transformed behavior $M_1$ shown in Fig. 1. The operation $x \leftarrow 5$, a loop invariant for input behavior $M_0$, is placed after the loop body in the transformed behavior $M_1$. Note that the input behavior $M_0$ and the transformed behavior $M_1$, shown in Fig. 1, are not equivalent since there is mismatch in values of the out variable. The EVP method reports that behaviors “May Not be equivalent”. The EVP method also reports that the path pairs $(p_{00}, p_{10})$ and $(p_{01}, p_{11})$ are U-equivalent, the path pair $(p_{02}, p_{12})$ is a candidate for C-equivalence and the path pair $(p_{03}, p_{13})$ is not equivalent. During the course of equivalence checking the EVP method stores these U-equivalent and candidate for C-equivalent path pairs in the EQ_LIST and C_LIST list, respectively. As explained in Sec. III, using these lists the generated cTrace of $M_0$ and $M_1$ is shown in I(c) and I(d), respectively.

### IV. Counter Example Generation Using Counter-Trace

To obtain the counter-example, i.e., assigning suitable value to the inputs, we rely on CBMC [9]. Specifically, for a given upper bound, CBMC verifies the specified assertions. If any violation of an assertion is detected, a counter-example is generated. Let us consider the cTraces as shown in Fig. 1(c) and Fig. 1(d). The input to the CBMC in C for this case is shown in Fig 2.

The variables appearing in the cTrace of $M_0$ (Fig. 1(c)) are suffixed with _s, whereas the variables appearing in the cTrace of $M_1$ (Fig. 1(d)) are suffixed with _t. Since program equivalence entails identical output(s) generated by the two programs when fed with the same input(s), the input variable $n$ is not suffixed with either _s or _t. Lines 3 and 4 declare the variables appearing in the cTrace of $M_0$ and the cTrace of $M_1$, respectively, along with their data type which is integer for all the variables. The lines 8–16 and 18–26 capture the data transformations and the conditions of execution of the paths appearing in the cTrace of $M_0$ and $M_1$, respectively. We use __CPROVER_assume statements to allow only those computation that satisfy a given condition. For example CBMC first picks the value for $n$ non-deterministically from the domain of integers. The
```c
#include<assert.h>
void main()
{
    int i_s,x_s,y_s,n,out_s;
    int i_t,x_t,y_t,out_t;
    __CPROVER_assume(n>=0);
    assert(!(n>=0));
    // cTrace for M0
    if(n>=0)
    {
        i_s=0;x_s=0;y_s=0;
        __CPROVER_assume(i_s<=n);
        assert(!(i_s<=n));
        x_s=5;
        y_s=y_s+5;
        i_s=i_s+1;
        out_s=x_s+y_s;
        //cTrace for M1
        if(n>=0)
        {
            i_t=0;x_t=0;y_t=0;
            __CPROVER_assume(i_t<=n);
            assert(!(i_t<=n));
            y_t=y_t+5;
            i_t=i_t+1;
            x_t=x_t+1;
            out_t=x_t+y_t+1;
            assert(x_s = x_t);// Live Variable
            assert(y_s = y_t);// Live Variable
            assert(out_s = out_t);// Output Variable
        }
```

Fig. 2. CBMC input for the cTraces shown in Fig. 1(c) and Fig. 1(d)

statement __CPROVER_assume(n >= 0) at line 5 further restricts the range of n for all program computations to be greater than or equal to 0. Note that if there is no computation satisfying the condition, say P, mentioned in __CPROVER_assume statement, then all the assertions hold vacuously. We check this by adding assert (!P) statement after each __CPROVER_assume statement so that if one of the assert (!P) statement is true then we declare that all the possible computations represented by cTrace are false computations [11] i.e., they never execute. Finally, we check the equivalence of the live variables (x_s, y_s, x_t, y_t) and output variables (out_s, out_t) using the assert statements (lines 27–29).

CBMC is able to automatically determine an upper bound on the number of loop iterations in many cases. It may fail if the number of loop iterations is highly data-dependent. Therefore, to verify the assertions with CBMC we use the following command: `cbmc fileName.c -unwind k --no-unwinding-assertions where fileName.c is the name of the target program, k is the bound on the number of iterations of the loop in the program called as Unwinding Loop Bound (ULB) and --no-unwinding-assertions disables the unwinding assertion check and changes the unwinding assertion to an unwinding assumption. We use the option --no-unwinding-assertions so that a counter-example might be found within the small state space generated with the small ULB. If the target program contains a loop then CBMC unwinds the loop k times and check the properties. Note that if there are multiple loops in the program, the bound k applies to all loops. A violation of the property is reported if it is found within k ULB and CBMC will give a counter-example. Otherwise, we iteratively run CBMC with increasing ULBs for the loops until an assertion violation is found or a given time limit is reached.

V. INCORPORATION OF RESULTS IN EQUIVALENCE CHECKING FRAMEWORK

The PBECs are sound but not complete. Therefore, all the PBECs report that the behaviors “May Not be equivalent” once they fail to prove the equivalence of source and transformed behaviors. Using the output of CBMC, we can actually make the PBEC more powerful. In some scenarios, the PBEC can report that the behaviors are “Not equivalent” (instead of “May Not”) along with a counter-example. Also, in some scenarios, the non-equivalence result reported by the PBEC can be proved to be a false negative and equivalence checking
will proceed further. In the following, we discuss how we can incorporate the CMBC result to improve the equivalence checking framework.

- **Case 1:** One of the conditions mentioned in \_CPROVER\_assume statement is not satisfiable: In this case, we report to PBEC that all the possible computations represented by cTrace are false computations. Consequently, we need to proceed further in the equivalence checking process.

- **Case 2:** The unwinding assertions are valid and CMBC does not find any counter-example: This means the values of all the live variables and output variables are the same for both cTraces. So the non-equivalence reported by the PBEC may be a false negative. In this case, we need to proceed further in equivalence checking by declaring the corresponding path pair \( (\alpha, \beta) \) as an equivalent path. This actually helps the PBEC to avoid false negative results during the course of equivalence checking.

- **Case 3:** CMBC reports counter-example for some variables: This means the data transformation of some variables is not equivalent in the cTraces.
  - **Case 3.1:** A mismatch is found for an output variable: This is surely a non-equivalence case. So the equivalence checker correctly found the non-equivalence of the behaviors. In this case, the PBEC reports that the behaviors are “Not equivalent” along with the counter-example.
  - **Case 3.2:** The outputs are the same: This is not a non-equivalence case. Consequently, we need to proceed further in the equivalence checking process.
  - **Case 3.3:** The outputs of the two programs are not the same: This is surely a non-equivalence scenario; in this case, the equivalence checker will report the behaviors are “Not equivalent” along with the counter-example.

- **Case 4:** CMBC hits the time limit: In this case, CMBC has failed to generate a counter-example because of time out. So no counter-example will be provided to the user. The PBEC reports the behaviors “May Not be equivalent”.

VI. OVERALL EQUIVALENCE CHECKING FRAMEWORK

The abstract version of our counter-example generation represented by the function counterExampleGenerator is presented in Algorithm 1. The control flow of Algorithm 1 is given in Fig. 3. The function counterExampleGenerator takes as input two FSMDs \( M_0 \) and \( M_1 \), a path \( \alpha \) from the path cover of \( M_0 \), a path \( \beta \) from the path cover of \( M_1 \), EQ_LIST contains equivalent path pairs and C_LIST contains candidates for conditionally equivalent path pairs. The function counterExampleGenerator returns \( \langle \bar{v}, \text{Equiv}, \text{falseComp} \rangle \), where \( \bar{v} = (v_1, v_2, \ldots, v_n) \) is the input variable list such that \( v_i \) represents the value of the input variable \( v_i \), \text{Equiv} is True if \( \alpha \simeq \beta \) and False otherwise and \text{falseComp} is True if all
the computations represented by \textit{cTrace} are false computations and \textit{False} otherwise. In lines 1–2 of Algorithm 1, a \textit{cTrace} is constructed from the \texttt{EQ\_LIST} and \texttt{C\_LIST} as discussed in Sec. III. The \textit{cTrace} is encoded as input to CBMC at line 3. The output generated by CBMC may result in various scenarios as discussed in Sec. V. Lines 6–19 of Algorithm 1 handle these cases.

The enhanced version of \texttt{correspondenceChecker} function of the EVP method [8] after incorporating the result of the function \texttt{counterExampleGenerator} is presented in Algorithm 2. In case of failure, Algorithm 2 invokes the function \texttt{counterExampleGenerator} (Algorithm 1) at line 13. It may be noted that the EVP method reports failure under this scenario. If \texttt{counterExampleGenerator} returns a counter-example (i.e., \( \bar{v} \neq \text{NULL} \)) then the function \texttt{correspondenceChecker} returns “Not equivalent” i.e., the two FSMDs are not equivalent (line 17). If CBMC hits the time limit then we cannot decide whether \( M_0 \) is equivalent to \( M_1 \). Hence the function \texttt{correspondenceChecker} returns “May Not be Equivalent” (line 23). If CBMC reports that all the possible computations represented by \textit{cTrace} are false computations (i.e., the variable \textit{falseComp} is True) then the function \texttt{correspondenceChecker} needs to be modified to handle this scenario (line 15). If CBMC finds the mismatch in the values of a live variable but outputs of the two programs are the same then we do not report the counter-example (line 21). To handle this case also \texttt{correspondenceChecker} needs to be modified. If CBMC declares that the path pair \((\alpha, \beta)\) is equivalent (i.e., the variable \textit{Equiv} is True) then it is a false negative result of the \texttt{correspondenceChecker} function (line 19). The \texttt{correspondenceChecker} function must take some decision to avoid the false negative case in the future.

VII. EXPERIMENTAL RESULT

We have taken the source code of the EVP method and have implemented our counter-example generation procedure on top of it. Once the EVP method fails to prove the equivalence, a \textit{cTrace} is automatically generated using \texttt{EQ\_LIST} and \texttt{C\_LIST} of the EVP method as discussed in Sec. III. We have then translated the two corresponding \textit{cTraces} as an input to CBMC. For our experiment, we used CBMC version 5.8 [9]. The benchmarks are taken from [5]. The benchmarks are run on a 1.8 GHz Intel i5 processor with 8 GB of RAM with a timeout limit of 60 seconds. The results of our experimentation are tabulated in Table I. We have manually introduced few changes like addition, multiplication or subtraction of a constant to some of the variables in the benchmarks tabulated in rows 3–6 of Table I so that source and transformed behaviors become non-equivalent. For each benchmark, we have reported the number of paths in the path cover, the number of states in the source and transformed behaviors, the equivalence decision taken by the EVP method and our method, the run time in milliseconds (ms) of the EVP method and our method, and the number of lines in the C program given as an input to CBMC.

For the benchmarks DIFFEQ and LRU, both the EVP method and our method report equivalence which is denoted as ‘E’ in Table I. The objective is to make sure that our implementation does not have any side effect on the existing method.
In the benchmarks reported in rows 3–6, the EVP method fails to prove the equivalence of source and transformed behaviors. It reports that the behaviors “May Not be equivalent”. This is reported as ‘MNE’ in Table I. In these cases, CBMC finds the mismatch in the values of output variable and generates a suitable counter-example with \( k = 2 \) loop unwindings. Hence, our method concludes that the behaviors are “Not equivalent” which is denoted as ‘NE’ in Table I. The time required by our method is a little high compared to the EVP method in the case of non-equivalence as we need to run CBMC on the cTrace. This experiment shows that with the help of our counter-example generation scheme a PBEC can take strong decisions about the non-equivalence of behaviors. Moreover, the counter-example provided by the PBEC will help the user to debug the root cause of the non-equivalence.

In our second experiment, we try to explore the false negative scenario of the EVP method. For this purpose, we have taken the example given in [12] and the result is tabulated in row 7 of Table I. This test case involves the inverse operation [12]. For this test case, the EVP method reports that the behaviors “May Not be equivalent”. However CBMC does not generate any counter-example and case 2 as discussed in Sec VI arises here. CBMC reports that cTrace corresponding to these behaviors are equivalent. Our method still reports “May Not be equivalent” since we have not implemented proceed further scenarios in the EVP. This experiment exposes a false negative case of the EVP method. It would be an interesting future work to enhance the EVP to handle the test cases which involves inverse operations.

### VIII. RELATED WORK

The basic path-based equivalence checking of the scheduling step of HLS was first proposed in [2]. The PBEC is further enhanced to handle control structure modification and uniform code motions in [7], non-uniform code motion in [6], speculative code motions in [3], code motions across loops in [5], for better correlation of corresponding paths using machine learning in [4], and to handle false computations and code motions involving loops in [8].

In software model checking [13], if the property of a program fails then model checkers generates a counter-example and helps in understanding the root cause of violation of the property. In this work we utilize the counter-example generation procedure in a PBEC framework to improve its performance.

### IX. CONCLUSION

In this paper, we have presented a counter-example generation mechanism for the PBEC reported in [8]. A similar counter-example generation mechanism can also be developed for other reported equivalence checking methods as well. The idea is to reuse the equivalence information of a PBEC to generate a counter-trace efficiently and then use CBMC to generate a counter-example. We have also shown that a path-based equivalence checking method can be further strengthened with the counter-example generation mechanism. As shown in the experiments, the EVP method can take stronger equivalence decision with help of counter-examples. Our counter-example generation mechanism identifies a false negative result of the EVP method. In the future, we plan to enhance the EVP method to handle the ‘proceed further’ (i.e., false negative cases) scenarios identified by our counter-example generation mechanism.

### ACKNOWLEDGMENT

This work is partially supported by DST, Govt. of India (Project code: ECR/2017/000492). The authors would like to acknowledge Dr. Kunal Banerjee, currently at Intel Labs, Bangalore, India, for his useful insights during the formulation of this work.

### REFERENCES