AES: Automated Evaluation Systems for Computer Programing Course

Shivam[1], Nilanjana Goswami[2], Veeky Baths[1] and Soumyadip Bandyopadhyay[1,3]
[1]*BITS Pilani K K Birla Goa Campus, India*
[2]*University of Heidelberg, Germany*
[3]*Hasso Plattner Institute, Germany*

Keywords: CPN Model, Program Equivalence, Automatic Evaluation Systems, Program Analysis.

Abstract: Evaluation of descriptive type questions automatically is an important problem. In this paper, we have concentrated on programing language course which is a core subject in CS discipline. In this paper, we have given a tool prototype which evaluates the descriptive type of questions in computer programing course using the notion of program equivalence.

1 INTRODUCTION

Due to a significant increase in the number of students in academics, automated checking has become very essential for many types of examination evaluation system. In largely populated countries such as India, where competitive examinations like IITJEE, BITSAT, GATE, etc., are held, the automated evaluation system is used for evaluating the results. Also for international examination like GRE and TOEFL automatic evaluation systems are used. In present day, the Moodle and Mooc type of systems can be used for automated evaluation. However, the systems only deal with objective type answers. In such cases, classical OMR based technology has been widely accepted. In recent times, the system like HackerRank [1] are being used for the evaluation of program using test cases. However, the generation of test cases is neither sound nor complete, i.e., it may gives both false negative as well as false positive result which is very dangerous for any automated marking system. The tool called Automata tutor [2](D'Antoni et al., 2015) can also evaluate descriptive type question answer symbolically. It is worthwhile to apply symbolic analysis technique in PL domain.

Symbolic program equivalence (Bandyopadhyay et al., 2018; Kundu et al., 2008; Necula, 2000) is a very useful underlying technique for building automated tool. However, the program equivalence is an undecidable problem.

Here we would like to propose an automated evaluation systems for checking of computer programs using symbolic equivalence checking method such that the checking method will always be able to give at least a sound answer, i.e., if the checker says that the two programs are equivalent then they are indeed equivalent. For the "No" answer, the checker will call partial marking module.

The proposed algorithm can be applied on any industry specific training and product development program. Furthermore, the system proposed to be developed will have impact on educational institutions (in terms of automating examination/ evaluation systems related to coding).

The main objective of this paper is to develop a well proven and sound automated program evaluation tool using program equivalence. Previously we have developed several equivalence checking methods for validating several human guided transformations that have been reported in (Bandyopadhyay et al., 2018; Bandyopadhyay et al., 2015); however, these methods can not handle loop shifting (Kundu et al., 2008), loop reversal, loop swapping etc. Therefore, we need to enhance the path based equivalence checker to overcome the limitations of the above referred works. The objective our research problem is as follows:

- Building an *efficient* path based equivalence checking algorithm for program equivalence.

Organization: Rest of the paper is organized as follows. Section 2 states the related works on translation validation. Methodology of the overall method is introduced in section 3. The section 4 describes a case study of the method. The paper is concluded in section 5.

[1]https://www.hackerrank.com/
[2]http://automatatutor.com/

Shivam, ., Goswami, N., Baths, V. and Bandyopadhyay, S.
AES: Automated Evaluation Systems for Computer Programing Course.
DOI: 10.5220/0007951205080513
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 508-513
ISBN: 978-989-758-379-7

2 STATE OF THE ART

There is no automated evaluation tools for descriptive type question in computer science course. However, using program verification as well as checking equivalence between two programs, the automated evaluation for descriptive type questions in computer programming course is possible. To achieve this, extension of program verifier as well as program equivalence checker is needed. To the best of our knowledge, program verifier and the equivalence between two programs can be established only for semantic preserving transformations, for example code optimizing transformation, loop transformations etc., where there is an input program and after some human guided and (or) compiler invoked transformations transformed version can be generated. Then the verifier checks the equivalence between two programs either behaviourally or functionally. Behavioural verification is also called as translation validation.

Translation validation for an optimizing compiler by obtaining simulation relations between programs and their translated versions was first proposed in (Pnueli et al., 1998); such a method is demonstrated by Necula et. al. (Necula, 2000) and Rinder et. al. (Rinard and Diniz, 1999). The procedure mainly consists of two algorithms – an inference algorithm and a checking algorithm. The inference algorithm collects a set of constraints (representing the simulation relation) using a forward scanning of the two programs and then the checking algorithm checks the validity of these constraints. Depending on this procedure, validation of high-level synthesis procedures are reported in (Kundu et al., 2008). Unlike the method of (Necula, 2000), their procedure considers statement-level parallelism since hardware can capture natural concurrency and high-level synthesis tools exploit the parallelization of independent operations. Furthermore, the method of (Kundu et al., 2008) uses a general theorem prover, rather than the specific solvers (as used by (Necula, 2000)). On a comparative basis, a path based method always terminates; however, some sophisticated transformations, like loop shifting, remains beyond the scope of the state of the art path-based methods. The loop shifting can be verified by the method reported in (Kundu et al., 2008). A major limitation the reported bisimulation based method is that the termination is not guaranteed (Necula, 2000; Kundu et al., 2008). Also, it cannot handle non-structure preserving transformations by path based schedulers (Camposano, 1991; Rahmouni and Jerraya, 1995); in other words, the control structures of the source and the target programs must be identical. The authors of (Camposano, 1991) have studied and identified what kind of modifications the control structures undergo on application of some path based schedulers; based on this knowledge, they try to establish which control points in the source and the target programs are to be correlated prior to generating the simulation relations. The ability to handle control structure modifications which are applied by (Rahmouni and Jerraya, 1995), however, still remain beyond the scope of the currently known bisimulation based techniques.

A Petri net based verification strategy is described in (Bandyopadhyay et al., 2012) for sequential high-level synthesis benchmarks for several code motions. In this method, the Petri net representations of an original behaviour and its transformed version are translated into equivalent FSMD models and fed as inputs to the FSMD equivalence checkers of (Karfa et al., 2012; Banerjee et al., 2014); no correctness proof, however, is given for this method; moreover, in the presence of more than one parallel thread, the method fails to construct the equivalent FSMD models.

None of the above mentioned techniques has been demonstrated to handle efficiently combination of several smart human guided code transformations as well as marking scheme for partially equivalent portion of program which is the key important part for automated program evaluation systems. Hence, it would be desirable to have an equivalence checking method that encompasses to verify efficiently several human guided code transformations as well as assign the marking scheme for equivalent code portion. Since automated evaluation system for descriptive type questions has not used till date, so the social analysis of the output derive from the descriptive type questions has not yet been studied.

3 METHODOLOGY

Through the following example, we would like to demonstrate the overall mechanism.

3.1 Methodology for Program Equivalence

Teacher asks to write a program which computes $\lceil \frac{100}{i} \rceil + \lfloor \frac{100}{j} \rfloor$. Teacher also provides a solution of the program. Students write the solution in a different way where the two loops are interchanged. This type of transformation has been referred to as *loop swapping*. The control data flow graph (CDFG) model is well known paradigm for scalar programs. However, the CDFG based equivalence checking methods

```
int i = 1, j = 1;
int k;
 while ( i*7 <=100){
    i++ ;
 }
 while ( (j+1)*11 <=100){
       j++;
 }
k = i+j;

    (a)
```

```
int i = 1, j = 1;
int k;
while ( (j+1)*11 <=100){
       j++;
 }
while (i*7 <=100){
    i++ ;
 }

k = i+j;
```

Figure 1: (a)–Solution Key (b) — Student's solution.

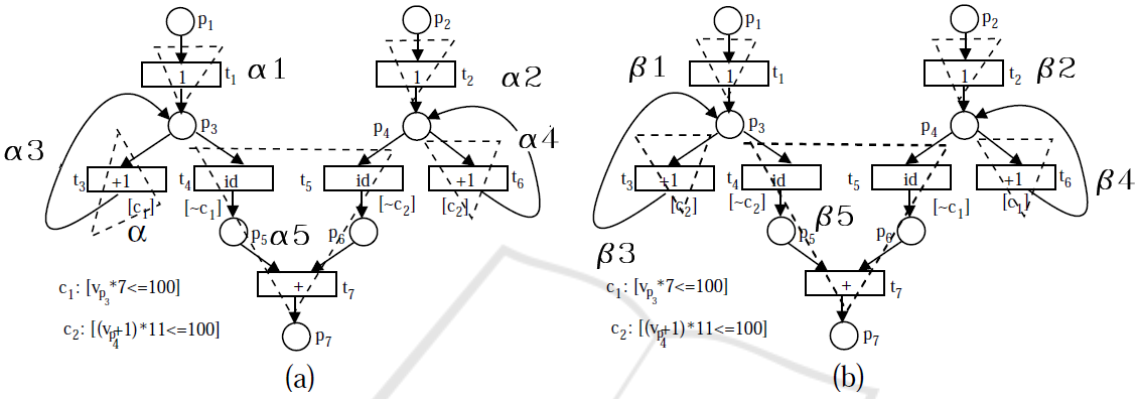

Figure 2: An Illustrative Example for Equivalence Checking.

(Banerjee et al., 2014; Kundu et al., 2008; Necula, 2000) fails to establish the equivalence between two programs because CDFGs are sequential MoCs. To establish the equivalence for loop swapping, we need parallel MoC's like Petri net. Figure 2(a) corresponds Figure 1(a) and Figure 2(b) corresponds Figure 1(b). Note that the semantics preserving model construction from programs to CPN models has been reported in (Bandyopadhyay, 2018).

Example 1. Figure 1(a) represents a simple scalar C program given by teacher. Figure 1(b) represents one version of students program. The PRES+ model generator generates Figure 2(a) (N_0) from the source code depicted in Figure 1(a). Similarly, from the source code Figure 1(b), the generator module generates the CPN model as shown in Figure 2(b) (N_1). Then the analyzer module has been triggered. The equivalence checking algorithm is the main functional part for the analyzer module. Each CPN model has been broken in to paths. In Figure 2 (a) the set of paths are $\alpha_1 = \langle\{t_1\}\rangle, \alpha_2 = \langle\{t_2\}\rangle, \alpha_3 = \langle\{t_3\}\rangle, \alpha_4 = \langle\{t_6\}\rangle$ and $\alpha_5 = \langle\{t_4, t_5\}, \{t_7\}\rangle$. Similarly in Figure 2 the set of paths are $\beta_1 = \langle\{t_1\}\rangle, \beta_2 = \langle\{t_2\}\rangle, \beta_3 = \langle\{t_3\}\rangle$, $\beta_4 = \langle\{t_6\}\rangle$ and $\beta_5 = \langle\{t_4, t_5\}, \{t_7\}\rangle$. Then equivalence checking method gives $\alpha_1 \simeq \beta_1, \alpha_2 \simeq \beta_2, \alpha_3 \simeq \beta_3, \alpha_4 \simeq \beta_4$ and $\alpha_5 \simeq \beta_5$. It is to be noted that equivalence between two expressions will be checked using SMT solver(z3,).

Report Generation. This phase declares that the two models N_0 and N_1 in path level. Therefore student gets full marks. For several semantics preserving transformations like loop shifting, code motion across loop etc., the method fails to detect the equivalence. The current version of the tool is available in (Bandyopadhyay, 2018).

4 CASE STUDY

Example 2. *In this example, we have taken one example Suppose the test input of the program is* $n = 7, a = 5, b = 6$*, then the outputs of both the programs are* 2*. Figure 3 (a) correspondences to the teachers' solution and Figure 3 (b) corresponds to a sample of students' solution. The CPN models constructed automatically for these two programs (Bandyopadhyay et al., 2017). The models are then validated using CPN simulator for the same data set. Then we have fed these two examples one by one as inputs to our equivalence checker module. Figure 4 depicts the output of the equivalence checking module. The output of the equivalence checking module depicts the condition of execution and the data transformation for each path in normalized form. Although in this method has*

```
main() {
    int s, i, n, a, b, sout;
    s = 0;
    for (i = 0; i <= 15; i++) {
        if (b % 2 == 1)
            s = s + a;
        if (s >= n)
            s = s - n;
        a = a * 2;
        b = b / 2;
        if (a >= n)
            a = a - n;
    }
    sout = s;
}
```

(a)

```
main() {
   int s, i, n = 6, b = 6, sout,
       a = 120,k, l, t;
   s = 0;
   i = 0;
   do {
       if (i <= 15) {
          i = (i + 1);
          k = (b % 2);
          l = (a * 2);
          b = (b / 2);
          if (k == 1) {
             s = (s + a);
             t = (l - n);
             a = l;
          } else {
              t = (l - n);
              a = l;
           }
           if (s >= n) {
               s = (s - n);
           }
           if (l >= n) {
              a = t;
           }
       }
      else
          break;
   } while (1);
   sout = s;
}
```

(b)

Figure 3: (a) Solution Key – (b) Students' solution.

no scope for path extension (as cut-points are present only at loop entry points apart from the in-ports and the out-ports) and paths cannot extend beyond the loop entry points, In Figure 4, it may be noted that the path extension is indeed not needed for this example.

5 CONCLUSION

In this paper we have given a program equivalence checking technique for validating several code optimizing transformations like uniform, non-uniform code optimization, variable renaming, code motion across loop and several loop transformations like dynamic loop scheduling and loop swapping. Some of the limitations of the present work are its inability to handle loop-shifting and software pipelining based transformations(Kundu et al., 2008). To solve this we have planned to extend this equivalence checker using the following technique. we reuse the front-end Petri net compiler part of the approach in (Bandyopadhyay et al., 2018) to construct a parallel model for the sequential programs. However, we use a different approach for equivalence checking. We use Static Single Assignment (SSA) logical formalism to model Petri net models and a bi-simulation-based equivalence between them. SSA formalism has been widely and effectively used in several state-of-art symbolic model checkers for C-programs, such as CBMC [3] and 2LS. This allows us to seamlessly integrate our tool to state-of-art symbolic model checker and also leverage several advanced proof techniques, such as K-induction (Brain et al., 2015) and invariant generation to scale our verification method. Our research will include the following main tasks.

1. Develop a method to model behaviour of Petri nets using SSA formalism
2. Develop a symbolic method to prove bi-simulation based equivalence of two Petri net models represented SSA and apply this method to verify a catalogue of parallel compiler transformations targeted for parallel reactive SW.
   - Explore how K-induction and invariant gen-

[3] www.cprover-org

```
#################### PATH  EQUIVALENCE ######################
FOR PATH 1 ...
THE CONDITION IS -- THE TRANSFORMATION IS  s :=  0 + 1 * s
PATH 1 OF MODEL 1 IS MATCHED WITH PATH 1 OF MODEL 2
FOR PATH 2 ...
THE CONDITION IS -- THE TRANSFORMATION IS i :=  0
PATH 2 OF MODEL 1 IS MATCHED WITH PATH 1 OF MODEL 2
FOR PATH 3 ...
THE CONDITION IS -- THE TRANSFORMATION IS a :=  0 + 1 * a
PATH 3 OF MODEL 1 IS MATCHED WITH PATH 1 OF MODEL 2
FOR PATH 4 ...
THE CONDITION IS -- THE TRANSFORMATION IS b :=  0 + 1 * b
PATH 4 OF MODEL 1 IS MATCHED WITH PATH 1 OF MODEL 2
FOR PATH 5 ...
THE CONDITION IS -- THE TRANSFORMATION IS n :=  0 + 1 * n
PATH 5 OF MODEL 1 IS MATCHED WITH PATH 1 OF MODEL 2
FOR PATH 6 ...
THE CONDITION IS (-15 + 1 * i > 0) THE TRANSFORMATION IS s :=  0 + 1 * s
PATH 6 OF MODEL 1 IS MATCHED WITH PATH 6 OF MODEL 2
.                    .                       .                   .
.                    .                       .                   .
FOR PATH 10 ...
PATH 10 OF MODEL 1 IS MATCHED WITH PATH 9 OF MODEL 2
THE CONDITION IS
(-15 + 1 * i <= 0)AND(0 - 1 * n + 1 * s >= 0 )AND(0 + 1 * a - 1 * n >= 0)
THE TRANSFORMATION IS
K :=  0 + 1 * a +  0 + 1 * n +  0 + 1 * n + 0 + 1 * b + 0 + 1 * s
.                .                       .                   .
<<<<<<<<<<<<<<<< THE TWO MODEL ARE EQUIVALENT >>>>>>>>>>>>>>>>
#################### Verification Report ############################
Exec time is 0 sec and 16324 microsecs
#####################################################################
```

Figure 4: Output of AES.

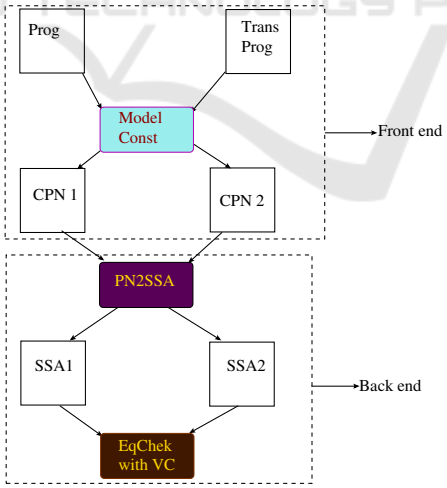

Figure 5: Future Tool Overview and Verification Process for loop shifting.

eration techniques can be used to automatically generate inductive invariants needed in our equivalence checking.

- We also plan to explore compiler transformations for programs that manipulate arrays and pointers.

3. Automate the verification process by implementing our method in a tool

The overall picture of the proposal is given in Figure 5. The Model Const box will re-use the Petri net compiler part of the tool reported in (Bandyopadhyay et al., 2018). PN2SSA is a Petri net to SSA trans-

lator that we need to build. EqCheck with VC box will be implemented by instrumenting the CBMC and 2LS code infrastructure. Other loop transformations for array handling programs is also a future endeavour. Further, the other aspect of the work is to do social analysis. Here we would like to analyse why a group of students in particular examination of particular course, for example computer programming, does better than the other group of students. This analysis technique will be helpful to improve the efficacy of AES.

REFERENCES

Z3 SMT Solver. http:/www.z3.codeplex.com/.

Bandyopadhyay, S. (Jan 2018). SamaTulyata. https://cse.iitkgp.ac.in/~chitta/pubs/rep/thesisBench.zip, https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db/samatulyata.html.

Bandyopadhyay, S., Banerjee, K., Sarkar, D., and Mandal, C. (2012). Translation validation for pres+ models of parallel behaviours via an fsmd equivalence checker. In *Progress in VLSI Design and Test (VDAT)*, volume 7373, pages 69–78. Springer.

Bandyopadhyay, S., Sarkar, D., Banerjee, K., and Mandal, C. (2015). A path-based equivalence checking method for petri net based models of programs. In *ICSOFT-EA 2015 - Proceedings of the 10th International Conference on Software Engineering and Applications, Colmar, Alsace, France, 20-22 July, 2015.*, pages 319–329.

Bandyopadhyay, S., Sarkar, D., and Mandal, C. (2018). Equivalence checking of petri net models of programs using static and dynamic cut-points. *Acta Informatica*.

Bandyopadhyay, S., Sarkar, S., and Banerjee, K. (2017). An end-to-end formal verifier for parallel programs. In *Proceedings of the 12th International Conference on Software Technologies, ICSOFT 2017, Madrid, Spain, July 24-26, 2017.*, pages 388–393.

Banerjee, K., Karfa, C., Sarkar, D., and Mandal, C. (2014). Verification of code motion techniques using value propagation. *IEEE TCAD*, 33(8).

Brain, M., Joshi, S., Kroening, D., and Schrammel, P. (2015). Safety verification and refutation by k-invariants and k-induction. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, pages 145–161.

Camposano, R. (1991). Path-based scheduling for synthesis. *IEEE transactions on computer-Aided Design of Integrated Circuits and Systems*, Vol 10 No 1:85–93.

D'Antoni, L., Weavery, M., Weinert, A., and Alur, R. (2015). Automata tutor and what we learned from building an online teaching tool. *Bulletin of the EATCS*, 117.

Karfa, C., Mandal, C., and Sarkar, D. (2012). Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):30.

Kundu, S., Lerner, S., and Gupta, R. (2008). Validating high-level synthesis. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 459–472, Berlin, Heidelberg. Springer-Verlag.

Necula, G. C. (2000). Translation validation for an optimizing compiler. In *PLDI*, pages 83–94.

Pnueli, A., Strichman, O., and Siegel, M. (1998). Translation validation for synchronous languages. In *ICALP*, pages 235–246.

Rahmouni, M. and Jerraya, A. A. (1995). Formulation and evaluation of scheduling techniques for control flow graphs. In *Proceedings of EuroDAC'95*, pages 386–391, Brighton.

Rinard, M. and Diniz, P. (1999). Credible compilation. Technical Report MIT-LCS-TR-776, MIT.