PLDI: U: Translation Validation of Thread-Level Parallelizing Transformations using Color Petri Nets

Abstract

Software applications often require the transformation of an input source program into a translated one for optimization. In this process, preserving the semantics across the transformation also called equivalence checking is essential. In this paper, we present ongoing work on a novel translation validation technique for handling loop transformations such as loop swapping and distribution, which cannot be handled by state-of-the-art equivalence checkers. The method makes use of a reduced size Petri net model integrating SMT solvers for validating arithmetic transformations. The approach is illustrated with two simple programs and further validated with a programs benchmark.

Keywords: Translation Validation, Equivalence Checking, Color Petri Net, Z3 Theorem Prover

1 INTRODUCTION

Software applications often require the transformation of an input source program into a translated version while preserving the semantics across the transformation. These translations are performed to efficiently utilize the intrinsic computer architecture, such as multiple cores and vector registers. For safety-critical systems, these translations need to be formally validated before they are used, to certify system reliability and accuracy. Checking the equivalence of the functional behaviors of source and translated programs is thus an important step. This process is called translation validation.

Instruction-level parallelism is one such translation that is widely used in high level synthesis during the scheduling phase. Petri nets are a popular modeling paradigm that can capture and express instruction-level parallelism.

Path-Based Equivalence Checking (PBEC) is a popular method for translation validation, which is based on graphical models/representations of code. Petri net PBEC methods have been proposed in [14, 16] but they are not able to validate code with complex arithmetic expressions. CDFG PBEC [9] methods are not able to validate parallelizing transformations either.

Satisfiability Modulo Theories (SMT) solvers are tools used to solve constraint satisfaction problems. They are used in verification as a means of analyzing the symbolic execution and semantics of programs. Z3 Theorem Prover [2] is an industry-standard SMT Solver developed by Microsoft Research to solve such problems.

In this paper, we propose an approach for translation validation of several loop-involving and parallelizing code transformations. The approach, which is a work-in-progress, has three major parts: a Petri net model constructor, a Petri net path constructor, and an equivalence checker which consists of a path analyzer and the Z3 Theorem Prover.

The major contributions of this paper are as follows:

- Approach to validate several transformations such as loop swapping and distribution, and parallelization which cannot be handled by state-of-the-art CDFG-based equivalence checkers.
- Refinement and reduction in size of Petri net model from that employed in [14], which enhances the efficiency of the equivalence checking mechanism and helps with scalability issues.
- Integration of SMT solvers in the approach to check equivalence between two programs.

This paper is organized as follows: Section 2 presents an overview of the entire workflow of the approach. Through a motivating example, the workflow is explained in Section 3. Through a small set of experimentation, we compare our approach with [14, 15] and other CDFG-based PBEC. Section 4 compares the experimental results with these equivalence checkers. Section 5 describes the state of the art. We conclude the paper in Section 6.

2 WORKFLOW

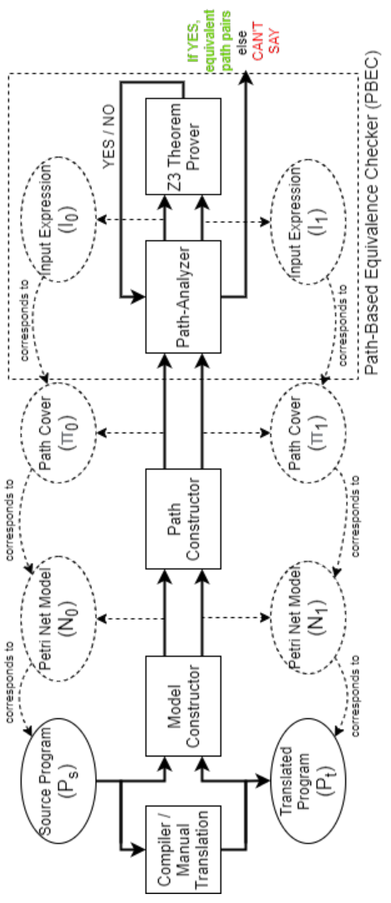


Figure 1. Workflow of proposed approach

The workflow of the proposed approach is illustrated in Fig. 1. Initially, a source program P_s, is subjected to a series of transformations, that result in a translated program P_t. In our approach, we have used Color Petri Net (CPN) as an intermediary modeling paradigm. This task is performed by the Model Constructor module which outputs two CPNs: N_0 and N_1 corresponding to the source and translated programs respectively.

To formally check behavioral equivalence between programs, there is a necessity to characterise the computations. However, in the case of loop(s), the number of loop traversals is indeterminate. To overcome this computational barrier, we represent the CPN model computations as a finite set of paths. This task of extracting the set of paths is performed by Path Constructor module, which gives the set π_0 from N_0 and π_1 from N_1.

Using the path-cover data, the process of equivalence checking is carried out by the Path-Based Equivalence Checking (PBEC) module that is composed of the Path Analyzer and Z3 Theorem Prover. The equivalence checking process is dynamically performed by the Path Analyzer module.

This establishment of equivalence (or non-equivalence) of the characteristics of the two programs (rather, their corresponding path covers) is facilitated by the Z3 Theorem Prover. To utilize Z3, the Path Analyzer module generates a set of Z3-compatible input expressions from the path cover data ($\mathcal{I}_0$ from π_0 and $\mathcal{I}_1$ from π_1), to check for equivalence between paths.

After all candidate paths have been checked, a 'Yes' answer from the Path Analyzer implies equivalence while a 'No' answer is interpreted as 'Can't Say', since the proposed equivalence checking method is sound but not complete. In the case of 'Yes', the Path Analyzer also outputs the equivalent pairs of paths from N_0 and N_1.

3 MOTIVATING EXAMPLE

```
int i=0,a,b,c,d,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);
while ( i < l ) {
    m = m * 10;
    n = n / 10;
    i++;}
c = (a*a*a) - (b*b*b);
d = (a*a) + (b*b) + (a*b)
e = c / d;
k = m + n + e;
```

Listing 1. The source program P_s

```
int i=j=0,a,b,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);
#parbegin scop
while ( i < l ) {
    m = m * 10;
    i++;}
||
while ( j < l ) {
    n = n / 10;
    j++;}
#parend scop
e = a - b;
k = m + n + e;
```

Listing 2. The transformed program P_t

In this section, we detail the major steps of the equivalence checking workflow using a simple example source program P_s and its transformed version P_t as given in Listings 1 and 2 respectively.

The program P_s takes five inputs a, b, l, m, and n, and computes the function:

$$k = (m \times 10^l) + (n \div 10^l) + (a - b) \tag{1}$$

The corresponding transformed program P_t is obtained by loop distribution followed by thread level parallelizing transformation of P_s; the independent sub-expressions $m \times 10^l$ and $n \div 10^l$ are computed separately in two parallelized loops.

3.1 Model Formalism

A Petri net model N, is a bipartite directed graph; one subset P, say, of vertices comprises *places* and the other subset T, say, comprises *transitions.* If there is an arc (p, t) from a place p to a transition t, then p is called a *pre-place* of t and the arc is called *in-coming arc* of t. The set of all pre-places of t is denoted as $^\circ t$. If there is an arc (t, p') from a transition t to a place p', then p' is called a *post-place* of t; the set of all post-places of t is denoted as t°. The arc is called an *out-going arc* of t.

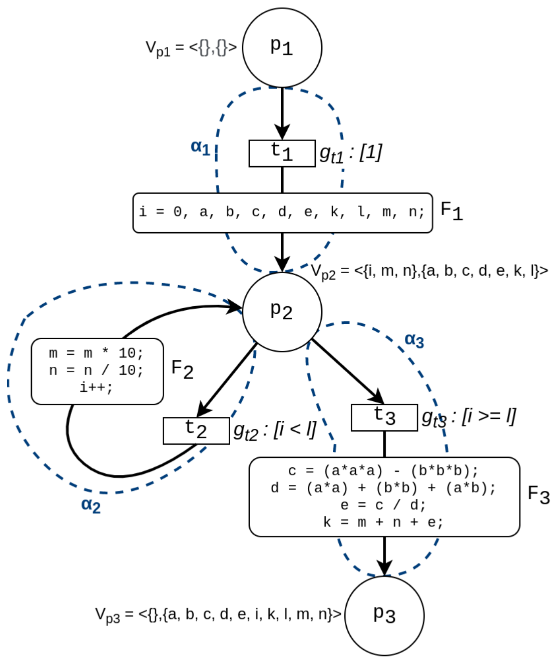
The set $P_{in} \subset P$ is designated as the set of *in-ports* of the model. It comprises all places that are not post-places of any transition. Similarly, another set $P_{out} \subset P$ is called the set of *out-ports*, which comprises the places that are not pre-places of any transition. A place can hold an entity called *token.* A token is a set of variable-value pairs that can hold values for program variables. The *marking* of a net is a particular distribution of tokens over the net.

Each out-going arc is associated with a set of functions. This *function-set* F, say, is a set of arithmetic expressions over (a subset of) the program variables. Each transition t is associated with a *guard condition* g_t, which is a Boolean function over (a subset of) the program variables. A transition t is said to be *enabled* when all its pre-places have tokens and they hold values which satisfy g_t. Consequent to the *firing* of an enabled transition t, tokens are removed from all $p \in {}^\circ t$ and tokens are placed in all $p \in t^\circ$. The value vector of the token(s) in the post-place(s) depends respectively, on the associated function-set F.

Each place $p \in P$ is associated with a vector of program variables V_p, say. For places that are in-ports, the vector consists of no variables. For places that are neither in-ports nor out-ports, there are two kinds of such variables: *changed variables* and *unchanged variables.* Changed variables are those variables whose values are changed from when the place was last marked. Similarly, unchanged variables are those whose values don't change. The partition between changed and unchanged variables for each place, is defined dynamically during the computations of the Petri net and the same will be illustrated in the next subsection. Out-ports have no changed variables in the associated variable vector.

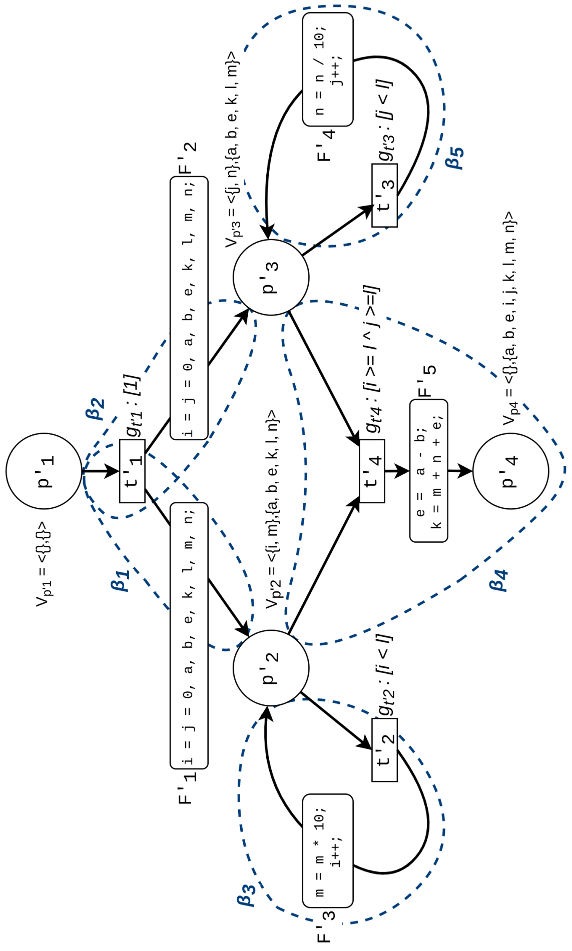
3.2 Model Construction

Using compiler internal infrastructure, the program is transformed into an intermediate representation. This representation is transformed to a Control Flow Graph (CFG) using the *fdump* process of the GCC compiler. The elements of the CFG are mapped to the Petri net, for which, a rudimentary set of rules is described in Table 1.



V_{p1} = <{},{}>
p1
t1
g_{t1} : [1]
α1
i = 0, a, b, c, d, e, k, l, m, n;
F1
V_{p2} = <{i, m, n},{a, b, c, d, e, k, l}>
p2
m = m * 10; n = n / 10; i++;
F2
t2
g_{t2} : [i < l]
t3
g_{t3} : [i >= l]
α3
c = (a*a*a) - (b*b*b); d = (a*a) + (b*b) + (a*b); e = c / d; k = m + n + e;
F3
α2
V_{p3} = <{},{a, b, c, d, e, i, k, l, m, n}>
p3

Figure 2. CPN model N_0 corresponding to the source program in Listing 1



p'1
V_{p1} = <{},{}>
t'1
g_{t1} : [1]
β1
β2
i = j = 0, a, b, e, k, l, m, n;
F'1
F'2
p'2
V_{p2} = <{i, m},{a, b, e, k, l, n}>
p'3
$V_{p'3}$ = <{j, n},{a, b, e, k, l, m}>
n = n / 10; j++;
F'4
t'3
g_{t3} : [j < l]
β5
t'2
g_{t2} : [i < l]
m = m * 10; i++;
F'3
β3
t'4
g_{t4} : [i >= l ^ j >=l]
e = a - b; k = m + n + e;
F'5
p'4
V_{p4} = <{},{a, b, e, i, j, k, l, m, n}>
β4

Figure 3. CPN model N_1 ≡ program in Listing 2

Table 1. Crude CFG-CPN transformation mapping

Control Flow Graph	Color Petri net
state	place
transition	in-coming arc, transition, out-going arc
transition condition	guard condition associated with transition
transition function	function-set associated with out-going arc

3.3 Notion of Path on CPN Model

In a general program, the number of loop traversals is unbounded. Therefore, we cannot characterize the set of computations. From the classical program verification techniques, we introduce the concept of paths such that any computation can be represented in terms of a finite set of paths. To construct the path, we introduce the notion of *cut-points*. Using cut-points we 'cut' each loop. The notion of cut-points in our CPN model is as follows:

1. All in-ports, $\forall p \in P_{in}$, are cut-points.
2. All out-ports, $\forall p \in P_{out}$, are cut-points.
3. All places that have back-edges are cut-points.

A *path* is a sequence of out-going arcs from a set of cut-points to a cut-point, while having no cut-point in between. Through the backward cone of foci method, we construct the paths in the Petri net model. The detailed discussion of the path construction algorithm is given in [1]. It is to be noted that if an out-going arc is covered in one path, it need not be considered in another path.

3.4 Validity of PBEC

To prove the validity of the path-based equivalence checker, we show that any computation can be represented as a concatenation of parallel paths.

As an example, taking the translated model N_1 in Fig. 3, we can express the computation as follows:

$$\mu_{p'_4} = \langle \{p'_1\}, \{p'_2, p'_3\}^{l+1}, \{p'_4\} \rangle$$

We can express the same computation in terms of the sequence of transitions that are fired. :

$$\mu_{p'_4} = \langle \{t'_1\}, \{t'_2, t'_3\}^{l}, \{t'_4\} \rangle$$

We can now express the computation in terms of the out-going arcs. :

$$\mu_{p'_4} = \langle \{(t'_1, p'_2), (t'_1, p'_3)\}, \{(t'_2, p'_2), (t'_3, p'_3)\}^{l}, \{(t'_4, p'_4)\} \rangle$$

The set of paths of N_1, $\pi_1 = \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5\}$. Initially, $\mu^r_{p'_4} = \phi$. The last member of $\mu_{p'_4}$ is (t'_4, p'_4). The path β_4 has (t'_4, p'_4) as its last member. So β_4 is prepended to $\mu^r_{p'_4}$, and all the out-going arcs in β_4 (only (t'_4, p'_4)) are removed once from $\mu_{p'_4}$.

Now, the last member of $\mu_{p'_4}$ is $\{(t'_2, p'_2), (t'_3, p'_3)\}$. (t'_2, p'_2) is the last member of β_3 and (t'_3, p'_3) is the last member of β_5. So $\{\beta_3 || \beta_5\}$ is prepended to $\mu^r_{p'_4}$ and all the out-going arcs from β_3 and β_5 are removed once from $\mu_{p'_4}$. This step will be repeated $l-1$ times until the only element left in $\mu_{p'_4}$ is $\{(t'_1, p'_2), (t'_1, p'_3)\}$. Since (t'_1, p'_2) is the last element of β_1 and (t'_1, p'_3) is the last element of β_2, $\{\beta_1 || \beta_2\}$ is prepended to $\mu^r_{p'_4}$. The algorithm is now terminated since $\mu_{p'_4}$ is empty. Therefore

$$\mu^r_{p'_4} = \langle \{\beta_1 || \beta_2\}, \{\beta_3 || \beta_5\}^l, \{\beta_4\} \rangle$$

3.5 Equivalence Checking Mechanism

There are two entities associated with every path

1. *Condition of execution*, R_α, which is associated with the guard conditions g_t.
2. *Data transformation*, r_α, which is associated with the function-sets F.

Two paths α and β are considered equivalent when $R_\alpha \simeq R_\beta$ and $r_\alpha = r_\beta$. The equivalence checking mechanism is based on the principle: "$\forall\ \alpha \in \pi_0, \exists\ \beta \in \pi_1$ and $\forall\ \beta \in \pi_1, \exists\ \alpha \in \pi_0 \mid \alpha \simeq \beta \implies \pi_0 \simeq \pi_1 \implies N_0 \simeq N_1$". During checking, the algorithm constructs correspondence relationships between the places, variables, and transitions, respectively. To check two arithmetic or logical expressions, we integrate the Z3 Theorem Prover with the equivalence checker. Following are the informal algorithmic steps for checking equivalence between N_0 and N_1:

In our motivating example, the set of paths in N_0 and N_1 are $\{\alpha_1, \alpha_2, \alpha_3\}$ and $\{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5\}$ respectively. Also, $R_{\alpha_1} = g_{t_1}$, $R_{\alpha_2} = g_{t_2}$, $R_{\alpha_3} = g_{t_3}$ and $r_{\alpha_1} = F_1$, $r_{\alpha_2} = F_2$, $r_{\alpha_3} = F_3$. Similarly, $R_{\beta_1} = R_{\beta_2} = g_{t'_1}$, $R_{\beta_3} = g_{t'_2}$, $R_{\beta_4} = g_{t'_4}$, $R_{\beta_5} = g_{t'_3}$ and $r_{\beta_1} = F'_1$, $r_{\beta_2} = F'_2$, $r_{\beta_3} = F'_3$, $r_{\beta_4} = F'_5$, $r_{\beta_5} = F'_4$.

Step 1) Taking the first element of π_0, i.e. α_1, we look at its pre-place p_1. Places p_1 and p'_1 correspond to each other since they are in-ports. Since p'_1 is a pre-place for paths β_1 and β_2, these two paths are candidate paths for α_1. The SMT solver tells us that $R_{\alpha_1} \simeq R_{\beta_1}$ (i.e. $g_{t_1} = g_{t'_1}$) and $R_{\alpha_1} \simeq R_{\beta_2}$ (i.e. $g_{t_1} = g_{t'_2}$). The SMT solver also tells us that $r_{\alpha_1} = r_{\beta_1}$ (i.e. $F_1 = F'_1$) and $r_{\alpha_1} = r_{\beta_2}$ (i.e. $F_1 = F'_2$). Hence, $\alpha_1 \simeq \beta_1$ and $\alpha_1 \simeq \beta_2$. From this information we also infer that the post-places of these paths correspond to each other, i.e. p_2 corresponds to p'_2 and p'_3.

Step 2) Taking the next element of π_0 i.e. α_2. The pre-place of α_2 is p_2, which corresponds to p'_2 and p'_3. Since β_3 and β_5 have the two places respectively as their pre-place, they are candidate paths for α_2. Checking for equivalence between these paths results in a 'No' answer from the SMT solver. So, we go for *path extension.* The paths β_3 and β_5 can be merged parallelly, due to place, variable, and transition correspondence . The SMT solver tells us that $R_{\alpha_2} \simeq R_{\beta_3 || \beta_5}$. Similarly, $r_{\alpha_2} = r_{\beta_3 || \beta_5}$. Hence, $\alpha_2 \simeq (\beta_3 \parallel \beta_5)$.

Step 3) Finally, taking the path α_3, it's pre-place is p_2 which has correspondence to p'_2 and p'_3, which are the pre-places of β_4. Similarly, the post-place of α_3 corresponds to the post-place of β_4 since they are out-ports in their respective nets. Hence, β_4 is a candidate path for α_3. The SMT solver tells us that $R_{\alpha_3} \simeq R_{\beta_4}$ and $r_{\alpha_3} = r_{\beta_4}$. Hence, $\alpha_3 \simeq \beta_4$. So,

$$\alpha_1 \simeq \beta_1,\beta_2\ ;\ \alpha_2 \simeq \{\beta_3 \parallel \beta_5\}\ ;\ \alpha_3 \simeq \beta_4$$

Since "$\forall\ \alpha \in \pi_0\ \exists\ \beta \in \pi_1 \mid \alpha \simeq \beta \implies \pi_0 \simeq \pi_1 \implies N_0 \simeq N_1$". That is, the programs in Listing 1 and Listing 2 are semantically equivalent.

3.5.1 Z3 Theorem Prover.

For two candidate paths α and β , the Z3 Theorem Prover (Z3) receives the conditions of execution, R_α and R_β, and the data transformation, r_α and r_β, from the path analyzer. All the program statements are encoded as *Static Single Assignments* to preserve the order of execution. The sub-scripts '_s' and '_t' are appended for variables of P_s and P_t respectively. The input to Z3 consists of:

1. Variables and corresponding type declarations.
2. Functions in the form of *assert* statements
3. Test statements asserted as negations. Z3 returns a *sat* (true) answer if it finds even one case (from the entire model space) that satisfies equivalence. Using the negation, we can test that equality is satisfied over the entire model space. Mathematically: for ξ (the model space) and c (the cases), by De Morgan's Law, $\neg(\bigcup_{c\in\xi} c) = \bigcap_{c\in\xi} \neg c$.

So, an *unsat* output from Z3 actually corresponds to equivalence and a *sat* output implies non-equivalence. Also, the test statements check for equality only between the common variables of P_s and P_t. In case of multiple assignment of the same variable, only the last executed variable is considered (i.e. the variable with highest numerical suffix).

As an example, in ***Step 3)*** for checking equivalence between R_{α_3} and R_{β_4}, the Z3 input is as follows:

```
(declare-const g_t3_s Bool) (declare-const l_t Int)   1
(declare-const g_t4_t Bool) (declare-const l_s Int)   2
(declare-const i_0_s Int) (declare-const j_0_t Int)   3
(declare-const i_0_t Int)                             4
(assert (= g_t3_s(>= i_0_s l_s)))                     5
(assert (= g_t4_t(and(>= i_0_t l_t)(>= j_0_t l_t))))  6
(assert (= l_s l_t)) (assert (= i_0_t j_0_t))         7
(assert (= i_0_s i_0_t))                              8
(assert (not(= g_t3_s g_t4_t)))                       9
(check-sat)                                           10
```

Listing 3. Checking equivalence of R_{α_3} and R_{β_4}

In Listing 3, lines 1-4 define the guard conditions as Boolean functions and define the associated variables. Lines 5-6 define $g_{t3\_s} = i \geq l$ and $g_{t4\_t} = i \geq l\ \&\ j \geq l$. To facilitate equivalence checking, equivalence between variables is asserted in lines 7-8. $i\_0\_t = i\_0\_s$ is infered from F'_1 and F'_2. Line 9 is the assert statement for equivalence checking defined as a negation. In the last line we check equivalence. Z3 returns *unsat* which implies $R_{\alpha_3} = R_{\beta_4}$.

4 EXPERIMENTAL RESULTS

We manually tested our equivalence checking algorithm on five examples, where parallelising transformations are applied using Pluto [18] and Par4All [11] compilers. The programs and their descriptions can be found in [1].

Table 2 presents a comparative study of the model size of our proposed approach with the models of two other Petri net-based equivalence checking tools *ST-1* [15] and *ST-2* [14]. It is to be noted that the model size of the current method is comparable with *ST-2*.

Table 3, presents transformation verification capabilities of the proposed approach, compared with *ST-1*, *ST-2* and two CDFG (Control Data Flow Graph) based PBEC namely, FSMD-VP (FSMD with Value Propagation) [10] and FSMD-EVP (FSMD with Extended Value Propagation) [13]. It is to be noted that

Table 2. Model size for different Petri-net PBEC

Example	ST-1		ST-2		Proposed	
	p	t	p	t	p	t
BCM	34	28	6	6	3	2
MINMAX	31	27	7	7	4	6
PETERSON	11	9	4	2	6	8
DEKKERS	19	14	6	4	6	8
LUP	28	21	6	4	10	16

Table 3. Capabilities of different PBEC

Example	FSMD-VP	FSMD-EVP	ST-1	ST-2	Proposed
BCM	X	X	X	X	✓
MINMAX	X	X	✓	✓	✓
PETERSON	X	X	X	X	✓
DEKKERS	X	X	X	X	✓
LUP	X	X	✓	✓	✓

both FSMD based PBEC cannot handle the parallelizing transformations because FSMD is a sequential model of computation. ST-1 and ST-2 cannot handle arithmetic transformations. They have their own normalizer, which are their limitations. These limitations are overcome by Z3.

5 RELATED WORK

Translation validation was introduced in [4] and was demonstrated in [20] and [12]. The approach was enhanced in [17]. All these techniques are bisimulation based methods. A bisimulation method for parallel programs is reported in [19].

Another equivalence checking method is the inductive-inferencing based technique reported in [7]. The method only works for scalar handing programs.

A major limitation of these methods is that termination is not guaranteed. To alleviate this shortcoming, a path based equivalence checker for the FSMD model was proposed for uniform and non-uniform code motions, code motion across loop and loop invariant code optimizations in [6, 10, 13]. However, they cannot handle loop swapping transformations and many thread-level parallelizing transformations because FSMDs cannot capture parallel behaviors easily.

The literature records no significant attempts for devising formal equivalence checking methods using Petri net based models. Although, there are several works on property verification using Petri net modelling paradigm [3, 5, 8, 21]. In [16], the validation of loop swapping and thread level parallelising transformations using Petri nets was reported. The major limitation of this method is it cannot handle loop invariant code motion as well as polynomial arithmetic transformations. Also, the model size presents a scalability issue. To over come the limitations, a modification in the model construction an equivalence checking was reported in [14]. However, the method cannot handle polynomial arithmetic transformations.

6 CONCLUSION

In this paper we presented our ongoing work on developing an approach to check the equivalence of software programs using a novel translation validation technique for handling loops. In addition, our approach makes use of SMT solvers to validate arithmetic transformations. Such constructions cannot be handled by state-of-the-art equivalence checkers.

We presented an initial validation of the approach for a standard benchmark. Currently this validation was performed manually. Therefore, our future work is to implement a tool-chain supporting the approach and validate it on a larger benchmark. For this, we will reuse existing compiler front-ends (e.g. GCC) and automatically construct the Petri Net models from the generated intermediate code representation so that the approach can be tested on different programming languages, potentially including existing architecture description languages such as UML, SysML and AADL. This will also allow us to further characterize the domain of applicability of the approach; i.e. which language constructions and translations are handled by our approach and to evaluate scalability for large programs.

Acknowledgments

To Soumyadip and Dominique, for being amazing guides.

References

[1] Soumyadip Bandyopadhyay. 2016. *Path based equivalence checking of Petri net representation of programs for translation validation.* Ph.D. Dissertation. IIT, Kharagpur.

[2] Leonardo de Moura et al. 2008. Z3: An Efficient SMT Solver. In *TACAS.* 337–340.

[3] Andrea Corradini et al. 2013. A Formal Model for the Deferred Update Replication Technique. In *TGC.*

[4] Amir Pnueli et al. 1998. Translation Validation. In *TACAS.*

[5] Bernadette Charron-Bost et al. 2013. Formal Verification of Distributed Algorithms (Dagstuhl Seminar 13141). *Dagstuhl Reports* 3, 4 (2013), 1–16.

[6] Chandan Karfa et al. 2012. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM TODAES* 17, 3 (2012).

[7] Dennis Felsing et al. 2014. Automating regression verification. In *ACM/IEEE International Conference on ASE.*

[8] Didier Lime et al. 2009. Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches. In *TACAS.*

[9] Kunal Banerjee et al. 2014. Extending the FSMD Framework for Validating Code Motions of Array-Handling Programs. *IEEE TCAD* 33, 12 (2014).

[10] Kunal Banerjee et al. 2014. Verification of Code Motion Techniques Using Value Propagation. *IEEE TCAD* 33, 8 (2014).

[11] Mehdi Amini et al. 2012. Par4All: From Convex Array Regions to Heterogeneous Computing. *IMPACT Workshop* (05 2012).

[12] Martin Rinard et al. 1999. *Credible Compilation.* Technical Report MIT-LCS-TR-776. MIT.

[13] Ramanuj Chouksey et al. 2019. Translation Validation of Code Motion Transformations Involving Loops. *IEEE TCADICS* 38, 7 (2019).

[14] Rakshit Mittal et al. 2020. Translation Validation of Loop involving Code Optimizing Transformations using Petri Net based Models of Programs. In *PNSE Workshop.*

[15] Soumyadip Bandyopadhyay et al. 2017. SamaTulyata: An Efficient Path Based Equivalence Checking Tool. In *ATVA.*

[16] Soumyadip Bandyopadhyay et al. 2018. Equivalence checking of Petri net models of programs using static and dynamic cut-points. *Acta Informatica* (2018).

[17] Sudipta Kundu et al. 2008. Validating High-Level Synthesis *(CAV).*

[18] Uday Bondhugula et al. 2008. PLuTo: A practical and fully automatic polyhedral program optimization system. In *PLDI.*

[19] Robin Milner. 1989. *Communication and Concurrency.* Prentice-Hall, Inc.

[20] George Necula. 2000. Translation validation for an optimizing compiler. In *PLDI.*

[21] Michael Westergaard. 2012. Verifying Parallel Algorithms and Programs Using Coloured Petri Nets. *Trans. on Petri Nets and Other Models of Concurrency* (2012).