Noname manuscript No.
(will be inserted by the editor)

Translation Validation of Coloured Petri Net Models of Programs on Integers

Soumyadip Bandyopadhyay · Dipankar Sarkar · Chittaranjan Mandal · Holger Giese

Received: date / Accepted: date

Abstract Programs are often subjected to significant optimizing and parallelizing transformations. Such transformations are carried out using extensive dependence analysis. To formally validate such transformations, it is necessary to use modelling paradigms which can capture both control and data dependences in the program vividly. Being value based with an inherent scope of capturing parallelism, the untimed coloured Petri net (CPN) models, reported in the literature, fit the bill well; accordingly, they are likely to be more convenient as the intermediate representations (IRs) of both the source and the transformed codes for translation validation than strictly sequential variable-based IRs like sequential control flow graphs (CFGs). In this work, an *efficient* path based equivalence checking method for CPN models of programs on integers is presented. Extensive experimentation has been carried out on several sequential and parallel examples. Complexity and correctness issues have been treated rigorously for the method.

Keywords: Equivalence checking, CPN model, FSMD model, Translation validation, Path based analysis.

Soumyadip Bandyopadhyay
BITS Pilani K K Birla Goa Campus, Goa, India
E-mail: soumyadipby@goa.bits-pilani.ac.in

Dipankar Sarkar
B. P. Poddar Institute of Management and Technology, Kolkata, India
E-mail: ds@cse.iitkgp.ac.in

Chittaranjan Mandal
Indian Institute of Technology, Kharagpur, India
E-mail: chitta@iitkgp.ac.in

Holger Giese
Hasso Plattner Institute, Potsdam, Germany
E-mail: holger.giese@hpi.de

1 Introduction

Recent advancement of multi-core and multi-processor systems has enabled incorporation of concurrent applications in embedded systems through extensive optimizing transformations for better time performance and resource utilization [24,48,23,33,36]. Several code transformation techniques such as, code motions, common sub-expression elimination, dead code elimination, etc., loop based transformation techniques such as, un-switching, reordering, skewing, tiling, unrolling, etc., and thread level parallelizing transformations such as, loop distribution, loop parallelizing [2,51,43,33,6,30,21,50,40,5], etc., may be applied at the pre-processing stage of embedded system synthesis. These transformations are carried out by some compilers or design experts. Even for the former case, if the compiler used is an untrusted one, then it can result in software bugs. Validation of a compiler ensuring the correct-by-construction property is a very difficult task. Instead, using behavioural equivalence checking techniques, it is possible to verify whether the optimized output of each run of the compiler faithfully represents the behaviour of the input source code.

For checking equivalence of behaviours of the two programs, it is necessary to represent them using a formal model. A comprehensive list of models proposed to represent programs for various application areas and analysis mechanisms around these models can be found in [20,3,37,4]. Petri nets have long been popular for modeling concurrent behaviours [49,13,54,38,53]. The untimed coloured Petri net (CPN) models reported in [27,53,26], enhances the classical Petri net models to capture computations over integers, reals and general data structures by permitting the places to hold tokens with data values and the transitions to have associated data transformations and conditions of executions. Analysis of dependencies among the operations in a program lie at the core of many optimizing transformations. The operational semantics of (value based) Petri nets permit depiction of operations having no mutual dependences in a sequential program more vividly as parallel subnets in the model structure; thus the Petri net model of a sequential program becomes structurally more akin to the model of the corresponding parallelized version. This makes Petri nets as potentially ideal models for validating parallelizing transformations making them more convenient intermediate representations (IRs) of both source and transformed codes for translation validation than strictly sequential (variable-based) IRs like all types of control data-flow graphs (CDFGs), communicating sequential processes [25], etc. Accordingly, in the present work the CPN has been chosen as the modelling paradigm; only programs on integers are addressed. (As an aside, it may be noted that several literature [18,9,7] have used the term PRES+ models (Petri net based Representation for Embedded Systems) to capture the parallel model of computation which is same as the restricted CPN model [8] used in this work.)

Equivalence checking of a source program P_s, say, with its transformed version P_t, involves demonstrating the output equivalence of each computa-

tion of P_s with a computation of P_t, and vice-versa. In general, as the input domains of the programs are infinite, the set of all computations, however, is infinite; So to cover this set, it is needed to have some concise representation for the set, such as closed forms of symbolic expressions over the input variables as values of the output variables. If such expressions can be *synthesized mechanically* from the programs P_s and P_t, then equivalence of these expressions can be checked for establishing the equivalence of P_s and P_t. In the presence of program loops, however, synthesis of these expressions is an undecidable problem [35]. A common technique followed by path based analysis mechanism consists in introducing cut-points to cut the loops so that any computation can be represented as a concatenation of finite paths where a path extends from an input point to a cut-point, from a cut-point to a cut-point or from a cut-point to an output point. The term "cut-points" encompasses the input points and the output points also in addition to the loop cut-points. Paths being finite having no loops in their structures, it is possible to synthesize closed forms of functional transformations of the variable values they represent by their symbolic executions. Equivalence checking of programs can thus be reduced to the problem of checking equivalence of the closed form of the functional transformation due to each path in a program with that of a corresponding path in the other program. We refer to such a mechanism as *path based equivalence checking*.

An equivalence checking method based on the path structure in a CPN model is far more complex than those in control and data flow graph (CDFG) based models due to the presence of parallel threads of computation. In CDFG models, it is possible to capture any computation *syntactically* as a concatenation of paths. For CPN models, however, difficulties arise in projecting computations as syntactic concatenations of paths; more specifically, presence of parallel threads, each involving loops which execute unequal number of times, prevent such simple views of model computations in terms paths induced by loop cut-points. This observation led to the need for additional dynamic cut-points over and above the static cut-points in [8]. While the static cut-points can be inserted in a model by identifying loops statically, introduction of dynamic cut-points necessitates execution of the models tracking the progress of the tokens abstracting out the token vales. A dynamic cut-point (DCP) based analysis mechanism (`DCPEQX`, in short) has been devised and reported in [8]. The validity of `DCPEQX` hinges upon the fact that any computation can be viewed as a *syntactic concatenation* of DCP induced paths. There are two major disadvantages of `DCPEQX` namely,

1. Introduction of DCPs involves an extra overhead in the form of a token tracking execution of the model;
2. DCP induced paths are shorter; in fact, for parallel threads with loops, the degeneration can be acute needing DCPs at almost all the places till the threads merge. In the presence of code motion transformations, the short lengths of paths force a special stage called *path extension* during equivalence checking adding to the "cost".

In the present work it has been formally established that static cut-point (SCP) induced paths can also capture any computation *semantically (although not syntactically)*. This permits us to devise valid equivalence checking mechanisms based on equivalence of SCP induced paths, the presently described method being one such approach. The above listed disadvantages of `DCPEQX` methods are all eliminated in `SCPEQX` methods. The fact that SCP induced paths can be concatenated in a certain way to capture model computations can be fruitfully leveraged in other forms of program analysis too, such as program verification.

The major *contributions* of the present paper are summarized as follows:

1. It has been formally established that CPN model computations can be captured by paths induced by cut-points at loops. This fact can be fruitfully used for devising path based equivalence checking methods including the one described presently.
2. An algorithm for path based equivalence checking of two CPN models is described and applied on several examples using the tool *SamaTulyata*. [1]
3. Formal proofs of the modules used at various stages of the mechanism are provided.

The rest of the paper is *organized* as follows. Section 2 describes the work flow of the method. Section 3 illustrates the entire method through a motivating example; in particular, the section describes the computational semantics of CPN models used in this work and also the notion of computational equivalence between two CPN models, the concept of loop cut-points and model paths induced by such cut-points, the mechanism of capturing computations in terms of paths obtained just from the static cut-points and finally the equivalence checking method. Experiments on some examples can be found in section 4. Section 5 states the related work on translation validation. The paper is concluded in section 6. There are several appendices which complement the basic method, described in Section 3 around an example, with the general method and its formal treatment. Appendix A formally defines a restricted CPN model used in this work. Appendix B formalizes computations in a CPN model. Appendix C defines the notion of finite computation paths of a model. Appendix D formally establishes the mechanism by which model computations can be captured semantically as concatenations of paths. Appendix E demonstrates the validity of path based equivalence checking mechanism(s). Appendix F presents the formal description of the equivalence checking algorithm, its complexity analysis and its correctness proof.

2 Work flow

Figure 1 displays the workflow of the current work. A high level program P_s is compiled using some compiler transformation techniques which generate an optimized intermediate (translated) code P_t.

[1] http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db/samatulyata.html

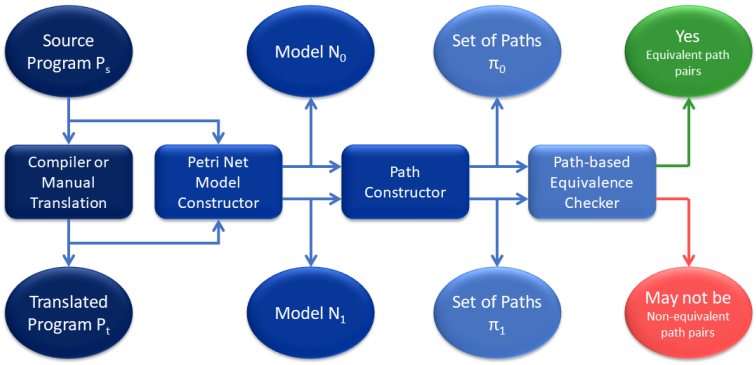


Fig. 1 Basic workflow

It is important to validate the translation. For analysis of this translation, it is necessary to convert these programs into equivalent formal models. In this work, we have chosen CPN (Coloured Petri Net) as our modelling paradigm. This is because a CPN can capture instruction-level parallelism in a vivid manner. The Petri Net Model Constructor module in our work generates the CPN models N_0 and N_1 corresponding to the programs P_s and P_t, respectively.

In a general program with data dependent loop(s), we do not know how many times the loop(s) will be executed. Moreover, the functional transformations produced by loop(s) (and hence the entire program) cannot be obtained in closed form. Such functional transformations, however, are needed to establish equivalence between two programs. To analyze translations involving loops, it is, therefore, necessary to express the CPN model computations (with a possibly infinite number of loop traversals) into a finite number of paths. This is facilitated by the Path Constructor module that gives us the set of paths, Π_0 for N_0, and Π_1 for N_1. A path is characterized by its corresponding symbolic data transformation functions and related condition of execution both of which are needed to identify an equivalent path in the other model.

The notion of equivalence checking between two models used in this work is as follows: "for any path of the model N_0, there exists an equivalent path in the model N_1, and vice-versa". The Path Based Equivalence Checker module accomplishes this task. It yields the answer "Yes" if all paths of N_0 are found to have equivalent paths in N_1, and vice-versa; in such a case, the pairs of equivalent paths are made available. Otherwise, it yields the answer "May not be equivalent". The process is sound in the sense that if the checker module yields an answer "Yes", then the models are indeed equivalent. In other words, there is no false positive outputs. However, if for a path in a model, the checker fails to find an equivalent path in the other model, it simply

hints at a possibility of non-equivalence of the models and reports the path; equivalent path(s) may still exist in the other model but the task of establishing such an equivalence lies beyond the power of the module. (In short, there can be false negatives.) Thus, the checker module is incomplete which should be the case as the problem of program equivalence is undecidable[35]. It is to be noted that the automated model constructor [47] constructs the CPN model from the program.

3 A Motivating Example

```
int i=0,k,m,n,l;
scanf ("%d, %d, %d",
       &m, &n, &l);
while (i<l){
       m=m*10;
       n=n/10;
       i++;
}
k=m+n;
printf ("%d \n", k);
      (a)
```

```
int i=j=0,k,m,n,l;
scanf ("%d, %d, %d",
       &m, &n, &l);
#parbegin scop
while (i<l){
       m=m*10;
       i++;
}
while (j<l){
       n=n/10;
       j++;
}
#parend scop
k=m+n;
printf ("%d \n", k);

 (b)
```

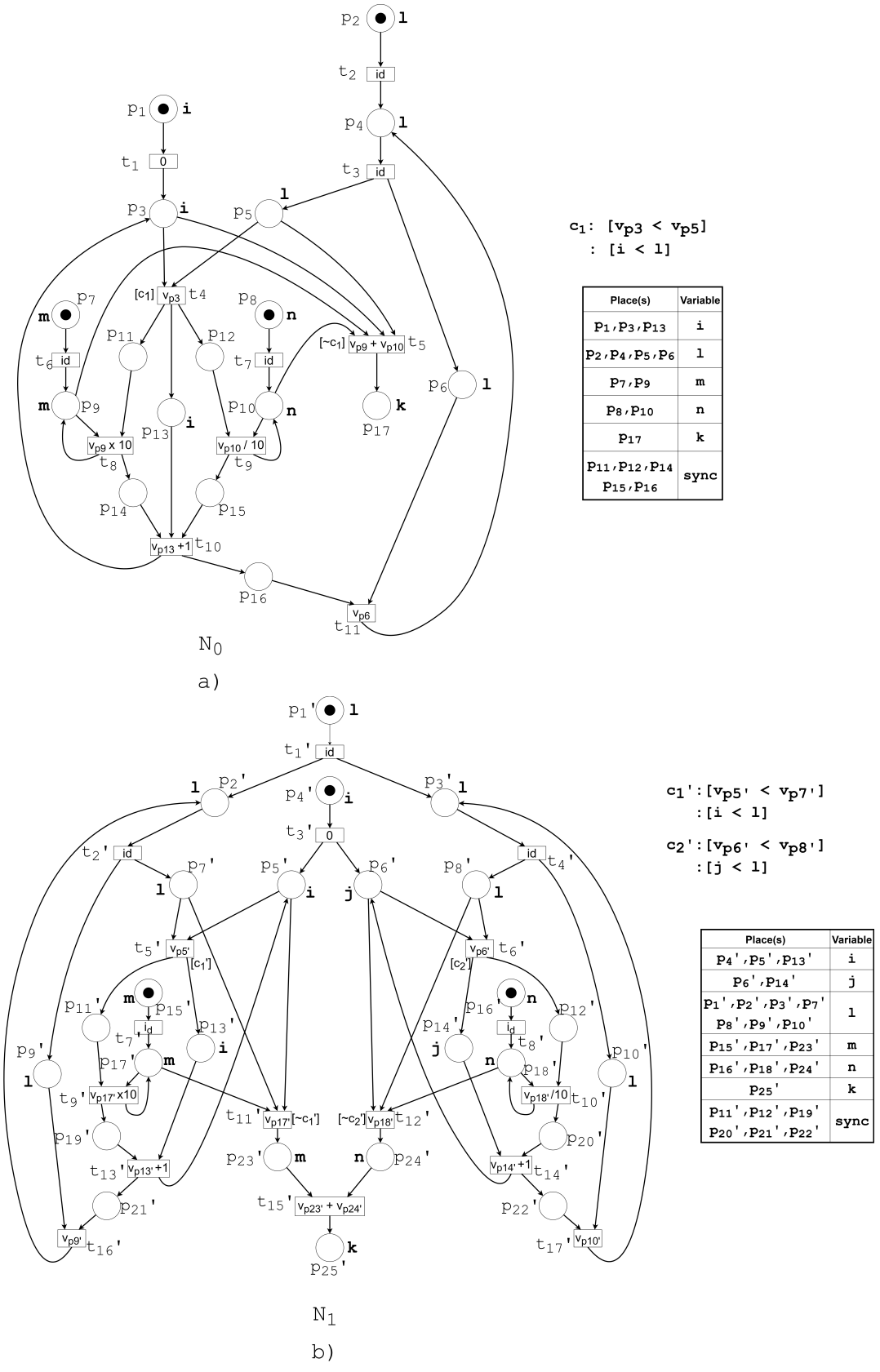
Fig. 2 A thread level parallelizing transformation–(a) P_s: source program and (b) P_t: transformed program.

The present section navigates through the major steps of the equivalence checking framework using a simple source program P_s and its transformed version P_t as given in Figure 2.

Figure 2(a) depicts the program P_s which takes three input integers m, n, l; it computes the function

$$k = m.10^l + n/10^l, l \geq 0 \quad (1)$$
$$= m + n, l < 0 \quad (2)$$

The corresponding transformed program P_t given in Figure 2(b) is obtained by loop distribution followed by thread level parallelizing transformation of P_s; in Figure 2(b), the independent subexpressions $m * 10^l$ and $n/10^l$ are computed separately in two parallelized loops.



Place(s)	Variable
p1,p3,p13	i
p2,p4,p5,p6	l
p7,p9	m
p8,p10	n
p17	k
p11,p12,p14 p15,p16	sync

Place(s)	Variable
p4',p5',p13'	i
p6',p14'	j
p1',p2',p3',p7' p8',p9',p10'	l
p15',p17',p23'	m
p16',p18',p24'	n
p25'	k
p11',p12',p19' p20',p21',p22'	sync

Fig. 3 Illustrative example for validation of a parallelizing transformation – (a) N_0: corresponds to P_s and (b) N_1: corresponds to P_t.

3.1 Petri net models of programs

The present work uses coloured Petri net (CPN) models for representing programs. CPN models have been formally defined in Appendix A. Here we introduce some basic terminologies in course of describing the example and its progress through the equivalence checking stages; this introduction is by no means exhaustive; many intricacies are left to be covered in Appendix A.

A Petri net model is essentially 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. 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°. If two places $p_1, p_2 \in t^\circ$ for some transition t, then they are said to be co-places of each other.

A subset P_{in} of P is designated as the set of *in-ports* of the model; similarly, another subset P_{out} of P is called the set of *out-ports*.

Each place p is associated with some program variable v_p, say. Each transition t is associated with a *guard condition* g_t, and a transformation function f_t; the condition g_t is essentially an arithmetic predicate over the program variables associated with its pre-places; the function f_t is an arithmetic expression over these variables.

A transition t is said to be enabled when all its input places have tokens and they are *associated with values* which satisfy g_t; consequent to firing of t, the tokens are removed from its pre-places ${}^\circ t$, tokens are placed at its post places in t°, all with a token value equal to the value of f_t for the token values at the pre-places in ${}^\circ t$; in other words, this value is assigned in parallel to the variables associated with the post-places in t°.

Figure 3(a) depicts a CPN model N_0 which can be obtained from the program P_s. Figure 3(b) depicts the CPN model N_1 corresponding to the program P_t.

Let us examine the CPN model N_0 of Figure 3(a) corresponding to the program P_s. The place p_3 holds the value of the variable i initialized to 0 through the transition t_1 associated with the constant function 0; since every transition should have some pre-place, an in-port p_1 is kept in ${}^\circ t_1$. The other three in-ports p_2, p_7 and p_8 respectively hold the input values of the variables l, m and n. Note that variable type declaration statements and read statements create only places without creating transitions; the present implementation takes only integer variables. The while-loop entry point is captured by the places p_3 and p_4. The transition t_3 has two post-places p_5 and p_6; the place p_5 is used for checking the condition of the loop and the place p_6 is used to return the token to the place p_4 for the next iteration. The function f_{t_3} associated with transition t_3 is the identity function.

The transition t_8 captures the right-hand side expression of the assignment statement "$m = m * 10$". Similarly, the transition t_9 captures the assignment statement "$n = n/10$". The post-place of t_8 (t_9), therefore, should hold the modified value of the variable m (n); although the in-port p_7 (p_8) is already

designated to hold the input value of the variable m (n), it cannot be in t_8° because in-ports have no incoming transitions. Hence, a different place p_9 (p_{10}) is used as the place holding the values of the variable m (n) updated once corresponding to each iteration of the while-loop. Hence, this place occurs in both ${}^\circ t_8({}^\circ t_9)$ and $t_8^\circ(t_9^\circ)$; it is made to hold the initial input value of m (n) through a separate transition t_6 (t_7) with ${}^\circ t_6 = \{p_7\}$ (${}^\circ t_7 = \{p_8\}$) and $t_6^\circ = \{p_9\}(t_7^\circ = \{p_{10}\})$. Note that since the statements "$m = m * 10$" and "$n = n/10$" have no data dependency between themselves, the associated transitions t_8 and t_9 are kept as transitions that can be fired in parallel. However, their firing have to be after entry to the loop takes place (i.e., after p_3 and p_5 acquire tokens either from some segment external to the loop or after execution of each iteration of the loop.) Hence, two synchronizing places p_{11} and p_{12} are required as pre-places of the transitions t_8 and t_9 respectively; these synchronizing places should acquire tokens through firing of a transition t_4 having p_3 and p_5 as its pre-places. At this stage, since the transition t_4 only serves the purpose of synchronization, its associated function f_{t_4} is of no concern; however, subsequently we can see why f_{t_4} is the identity function. The transition t_4 therefore can be called the entry transition to the loop initiating the execution of all the parallel threads of the loop body. Since it is the entry transition, it is associated with the loop condition $i < l = c_1$, say; obviously, there has to be a loop exit transition associated with the condition $\neg c_1$; the transition t_5 serves this purpose. The transition t_{10} captures the update operation "$i{+}{+}$" of the loop control variable i. Although this update operation has no data dependency with the other two statements in the loop body, it is made to have control dependence with the transitions t_8 and t_9 for synchronization; hence, synchronizing places p_{14}, p_{15} are used as the pre-places of t_{10} and post-places of t_8 and t_9; the place p_{13} holds the value of the loop control variable i which is the only pre-place of t_{10} that gets updated through its execution. It is to be noted that in this program the variable l does not change at all. So for next iteration of the loop the token has to be returned back to the place p_4. To do this the transition t_{11} is used with p_{16} and p_6 as its pre-places and p_4 as its post-place. After update of i, the token is returned back to the place p_4. For this purpose the place p_{16} serves as a synchronizing place and the identify function is associated with the transition t_{11}.

Now we can see that t_4 needs to provide to the pre-place p_{12} a copy of the loop control variable i held in one of the loop entry places p_3; therefore, the transition t_4 is made to have the associated function f_{t_4} as the identity function (id). The segment reached after exit from the loop comprising the statement "$k = m + n$" would require a transition having two pre-places holding values of the variables m and n and a post place corresponding to the variable k. The already conceived exit transition t_5 can serve this purpose with its pre-places p_9 (holding the latest value of m) and p_{10} (holding the latest value of n) in addition to its already conceived pre-places p_3 and p_5. This immediately underlines the need of associating the transitions t_8, t_9 in the loop body with the loop entry condition. The place p_{17} associated with variable k is placed as the post-place of t_5 and it is designated as an out-port because k

is an output variable. We, therefore, find that the place to variable mapping has a co-domain $\{i, k, m, n, l\} \cup \{\delta\}$, where δ is designated as a synchronization variable needed at various points in the model. Specifically, the mapping is $f_{pv} : \{p_1, p_2, \ldots, p_{17}\} \rightarrow \{\{i, k, m, n, l\} \cup \{\delta\}\}$, $\{\{p_1, p_{11}, p_{12}, p_{14}, p_{15}, p_{16}\} \mapsto \delta, \{p_3, p_{13}\} \mapsto i, \{p_7, p_9\} \mapsto m, \{p_8, p_{10}\} \mapsto n, \{p_2, p_4, p_5, p_6\} \mapsto l, p_{12} \mapsto k\}$.

3.2 Computation on a Petri net model

In this subsection, we introduce informally and in brief the notion of computations of Petri net models of programs. A more formal definition of the terms used in this subsection and other associated terms is available in Appendix B.

A model computation is a sequence of model states which are more conventionally called *markings*. (For brevity, we use the term "net" for a Petri net model as described in the previous subsection.) A marking M of a net is an ordered pair $\langle P_M, val_M \rangle$, where P_M, the place marking, is the subset of places where tokens are present and val_M is the corresponding values of these tokens. Diagrammatically, presence of a token in a place is indicated by a dark circle in the vertex corresponding to the place.

A computation trace μ is a sequence of markings of the form $\langle M_0, M_1, \ldots, M_k \rangle$ where each marking $M_i, 1 \leq i \leq k$, results from the preceding marking M_{i-1} by *simultaneous firing of all the enabled transitions* corresponding to the marking M_{i-1}. The marking M_i is said to be *successor marking* of $M_{i-1}, 1 \leq i \leq k$. Note that conventionally, any one of the enabled transitions fires; however, in the context of the present work, the model structure ensures that transitions having some dependence relationship among themselves never become enabled simultaneously. The first member M_0 of μ corresponds to a place marking of only in-ports; it is called the *initial marking*; The final marking M_k corresponds to a place marking having a token in at least one out-port p_t, say; it is called the final marking of μ; often we refer to the computation trace μ as a computation trace of the out-port p_t and denote it as μ_{p_t}.

Note that a computation trace μ starts with some specific input values. The sequence $\langle P_{M_0}, P_{M_1}, \ldots, P_{M_k} \rangle$ of place markings corresponding to the computation trace $\mu = \langle M_0, M_1, \ldots, M_k \rangle$ can also result for other computation traces where they differ from μ and from each other in the starting input values. We refer to the common sequence of place markings of several computation traces as a *computation*. Specifically, therefore, a computation is just a sequence of place markings which when associated with some specific input values results in a computation trace.

A computation $\langle P_{M_0}, P_{M_1}, \ldots, P_{M_k} \rangle$ can also be alternatively designated as a sequence of sets of transitions $\langle T_1, T_2, \ldots, T_{k-1} \rangle$ such that $T_i, 1 \leq i \leq k-1$, is the set of enabled transitions corresponding to P_{M_i} and $P_{M_{i+1}}$ is the set of post places corresponding to the set of transitions T_i.

Let us now examine how the computation trace of the program, given in Figure 2 (a), for the inputs $m = 7$, $n = 7347$ and $l = 3$ is represented by a computation $\mu_{p_{17}}$ of the out-port p_{17} of the model of Figure 3(a). In the fol-

lowing, for any marking M encountered along the computation, the second component val_M is listed using the same ordering in which the first component P_M is listed. For the inputs $m = 7$, $n = 7347$ and $l = 3$, the initial marking $M_0 = \langle\{p_1, p_2, p_7, p_8\}, \langle\omega, 3, 7, 7347\rangle\rangle$, where ω stands for any integer. Now, the first set of enabled transitions in the computation is $T_1 = T_{M_0} = \{t_1, t_2, t_6, t_7\}$. Note that $^\circ T_1 \subseteq inP = \{p_1, p_2, p_7, p_9\}$. After firing of (all the members of) T_1, the successor marking M_0^+ becomes $\langle\{p_3, p_4, p_9, p_{10}\}, \langle 0, 3, 7, 7347\rangle\rangle = M_1$; now the next set of enable transitions is $\{t_3\}$ so $T_2 = T_{M_1} = \{t_3\}$ and the successor marking of M_1 becomes $\langle\{p_3, p_5, p_6, p_9, p_{10}\}, \langle 0, 3, 3, 7, 7347\rangle\rangle = M_2$; the condition $c_1 = v_{p_3} < v_{p_5}$, i.e., $(i > l)$ associated with t_4 is satisfied and the one $(\neg c_1)$ associated with t_5 is false; the next set of enabled transitions is the unit set $\{t_4\} = T_3$, say. After firing of T_3, the successor marking $M_2^+ = M_3 = \langle\{p_9, p_{11}, p_{13}, p_{12}, p_{10}, p_6\}, \langle 7, 0, 0, 0, 7347, 3\rangle\rangle$. So the next set of transitions $T_4 = T_{M_3} = \{t_8, t_9\}$. After firing of T_4, the successor marking $M_3^+ = M_4 = \langle\{p_9, p_{10}, p_{14}, p_{13}, p_{15}, p_6\}, \langle 70, 734, 70, 0, 734, 3\rangle\rangle$; $T_5 = T_{M_4} = \{t_{10}\}$. After firing of T_5, the successor marking $M_4^+ = M_5 = \langle\{p_3, P_9, p_{10}, p_6, p_{16}\}, \langle 1, 70, 734, 3, 1\rangle\rangle$; $T_6 = T_{M_5} = \{t_{11}\}$. After firing of T_6, the successor marking $M_5^+ = M_6 = \langle\{p_3, p_4, p_9, p_{10}\}, \langle 1, 70, 734, 3\rangle\rangle$; and $T_7 = T_{M_7} = \{t_3\} = T_2$. So the sub-sequence of the sets of transitions $\langle T_3, T_4, T_5, T_6, T_2\rangle$ captures one iteration of the loop. Hence, it repeats another $l - 1$ times and $l = 3$. The prefix of the computation after these many iterations becomes $\langle T_1, T_2, (T_3, T_4, T_5, T_6, T_2)^3\rangle$ and the resulting marking will be $M_{11} = \langle\{p_3, p_9, p_{10}, p_4\}, \langle 4, 7000, 7, 3\rangle\rangle$. At this stage, $T_{M_{11}} = \{t_5\} = T_7$ because the condition associated with t_5 $(= \neg v_{p_3} < v_{p_5})$ holds. So the computation in terms of a sequence of transitions is $\langle T_1, T_2, (T_3, T_4, T_5, T_6, T_2)^3, T_7\rangle$ and in terms of places is: $\langle\langle {}^\circ T_1, T_1^\circ \cup {}^\circ T_2, (T_2^\circ \cup {}^\circ T_3, T_3^\circ \cup {}^\circ T_4, T_4^\circ \cup {}^\circ T_5, T_5^\circ \cup {}^\circ T_6, T_6^\circ \cup {}^\circ T_2)^3, T_2^\circ \cup^\circ T_7, T_7^\circ\rangle, \langle\omega, 3, 7, 7347\rangle\rangle$.

3.3 Equivalence of Two Petri net Models

In the previous subsection we have demonstrated what is meant by a computation of a Petri net model for certain input values. While establishing the equivalence of two models, however, we have to show that for each computation of one model, there is a computation of the other model which yields exactly the same output(s) as the former. Unfortunately, the set of all model computations is infinite as, in general, the input domain is infinite. However, the set of all computations can be partitioned into subsets where each subset of computations has a characteristic predicate over the input variables in the sense that for any member computation in this subset, the predicate holds. Similarly, there is a function over the input variables which represents the outputs produced by the members of a subset of computations.

For instance, for the present example, we may summarize the set of all its computations by identifying a partition comprising two subsets each being characterized in terms of a condition of execution and a data transformation given by equations (1) and (2), respectively. Hence, the set of computations

can be partitioned into two subsets; one of them has the characteristic predicate $l \geq 0$; any computation in this subset yields an output of the form $m * 10^l + n/10^l$; the other subset has the characteristic predicate $l < 0$ and any member in this subset yields an output of the form $m + n$. We designate the characteristic predicate as *condition of execution* of the computation and the expression corresponding to the value produced at the out-port as the *functional transformation* of the inputs produced by the computation.

In general, therefore, any computation μ_p of an out-port p of a net can be given by an ordered pair $\langle R_{\mu_p}(f_{pv}(inP)), r_{\mu_p}(f_{pv}(inP))\rangle$, where $R_{\mu_p}(f_{pv}(inP))$ is the condition of execution of μ_p and $r_{\mu_p}(f_{pv}(inP))$ is the functional transformation produced by the computation μ_p.

At this stage, we may define equivalence of a computation of one model with a computation of another model and equivalence of two models. The notion of these equivalences builds upon a bijection f_{in} from the set of in-ports of a model to that of the other model and another bijection f_{out} from the set of out-ports of a model to that of the other model. The definitions are as follows.

Definition 1 (Equivalent Computations of Two Models) A computation μ_p of a model N_0 having an out-port p is said to be equivalent to a computation $\mu_{p'}$ of another model N_1 having an out-port p' such that $f_{out}(p) = p'$, symbolically denoted as $\mu_p \simeq \mu_{p'}$, if $R_{\mu_p} \equiv R_{\mu_{p'}}$ and $r_{\mu_p} = r_{\mu_{p'}}$.

Definition 2 (Equivalence of Two Models) A model N_0 is said to be an equivalent to a model N_1 if, for any computation μ_p of N_0 having an out-port p, there is a computation $\mu_{p'}$ of N_1 having an out-port $p' = f_{out}(p)$ such that $\mu_p \simeq \mu_{p'}$ and vice-versa.

In the next subsection we argue that the Definitions 1 and 2 still do not lead directly to an effective procedure for equivalence checking and a notion of *finite paths* is to be introduced.

3.4 Path Based Equivalence Checking

Definitions 1 and 2 cannot be applied directly for establishing equivalence between two models as they necessitate synthesizing the conditions of executions and the transformation functions of all the subsets of computations of a model in the partition of the set of all model computations even if the partition is always finite. The task can only be automated when there are no loops in the program. For programs with loops the problem is undecidable [35].

For this reason, the notion of *finite* computation paths, henceforth referred to simply as paths, is used so that any computation of an out-port can be captured in terms of these paths and accordingly, computational equivalence between two models can be reduced to computational equivalence between two corresponding paths of the models. The notion of introducing cut-points in program flowchart models to capture (possibly infinite) set of computations

in terms of a finite set of finite paths dates back over more than fifty decades in program verification. Conventionally, cut-points are put at the input and output statements and at least one point in each loop. For the CPN models used in the present work, we need to cut the loops designating some of the places as cut-points so that each loop contains at least one cut-point. The places which have back edges with respect to a DFS (Depth first search) traversal of the model graph are taken as cut-points in the loops.

Literature [8] describes an equivalence checking mechanism which uses two kinds of cut-points — static and dynamic. The static cut-points are essentially the input, output and the loop cut-points. The present work uses only static cut-points; hence we omit the qualifier "static" and simply use cut-points wherever there is no scope for ambiguity.

A path is a sequence of sets of transitions of the form $\langle T_1, T_2, \ldots, T_n \rangle$ such that the set ${}^\circ T_1$ of pre-places of the transitions in T_1 are all cut-points or co-places of a cut-point, the set T_n° of post-places of T_n contains at least one cut-point and T_i is a set of enabled transitions for the marking ${}^\circ T_i, 1 \leq i \leq n$. A more formal treatment of paths is available in Appendix C. The path construction method is given in [7].

For example, for the model N_0 of Figure 3(a), the set C of static cut-points is $\{p_1, p_2, p_7, p_8$ (as they are in-ports), p_{17} (as it is an out-port), $\{p_3, p_4, p_9, p_{10}\}$ (as there are back edges terminating in them) $\}$. The corresponding path set Π_0 is $\{\alpha_1 = \langle\{t_1\}\rangle, \alpha_2 = \langle\{t_6\}\rangle, \alpha_3 = \langle\{t_7\}\rangle, \alpha_4 = \langle\{t_2\}\rangle, \alpha_5 = \langle\{t_3\}, \{t_4\}, \{t_8\}\rangle, \alpha_6 = \langle\{t_3\}, \{t_4\}, \{t_9\}\rangle, \alpha_7 = \langle\{t_3\}, \{t_4\}, \{t_{10}\}\rangle, \alpha_8 = \langle\{t_3\}, \{t_{11}\}\rangle\}$ and $\alpha_9 = \langle\{t_3\}, \{t_5\}\rangle\}$. It may be noted that for the path α_5, the last transition set $\{t_8\}$ has the post-places $\{p_9, p_{14}\}$ of which p_9 is a cut point and p_{14} is not. Similar observation holds for the paths α_6, α_7. Thus, while all the pre-places of the first set of transitions in a path are cut-points, among the post-places of the last transitions there needs to be at least one cut-point.

3.4.1 Computation in terms of Paths

Let us now examine how a computation of the model N_0 of Figure 3(a) can be captured in terms of the set Π of paths as listed above. Let us consider the computation $\mu = \langle\{t_1, t_2, t_6, t_7\}, (\{t_3\}, \{t_4\}, \{t_8, t_9\}, \{t_{10}\}, \{t_{11}\})^l, \{t_5\}\rangle$ of the out-port p_{17}. It may be noted that it is not possible to concatenate any subset of the paths α_1 through α_9 to obtain a sequence of sets of transitions which is syntactically identical to μ. However, *maintaining semantic equivalence*, we can reorder the transition sets of μ using the fact that any set of enabled transitions can be partitioned arbitrarily and the members of the partition executed in any arbitrary order so that the sequence corresponding to a path occurs as a whole without having member transitions of other paths interspersed within the sequence. This arrangement permits us to view the reordered μ, referred to as μ^r, as a sequence of paths.

The steps for obtaining such a reordered computation μ^r from μ are as follows. The last member in the computation μ is identified as the unit set $\{t_5\}$. From Π, we notice that t_5 occurs as the last transition in the path α_9 only; so α_9

must be the last member in μ^r; the other transition set $\{t_3\}$ of path α_9 occurs in the paths $\alpha_5, \alpha_6, \alpha_7$ and α_8 as well; hence the set $\{t_3\}$ of α_9 is not deleted and the reordered sequence in the first step becomes $\langle\alpha_9\rangle = \mu^{r(1)}$, say. We delete from μ its already included member set $\{t_5\}$; the (remaining) computation becomes $\mu^{(1)} = \langle\{t_1, t_2, t_6, t_7\}, (\{t_3\}, \{t_4\}\ \{t_8, t_9\}, \{t_{10}\}, \{t_{11}\})^l\rangle$.

Now, the last transition set in $\mu^{(1)}$ is the unit set $\{t_{11}\}$; it is found to occur as the last transition in the path $\alpha_8 = \langle\{t_3\}, \{t_{11}\}\rangle$; the transition $\{t_{11}\}$ is deleted from μ; the other transitions t_3 occurs in the paths α_5, α_6 and α_7 as well; hence, t_3 is not deleted from μ. The path α_8 is placed before the path α_9 in μ^r thereby, $\mu^{r(1)}$ becomes $\mu^{r(2)} = \langle\alpha_8.\alpha_9\rangle$ and the computation $\mu^{(1)}$ becomes $\mu^{(2)} = \langle\{t_1, t_2, t_6, t_7\}, (\{t_3\}, \{t_4\}\ \{t_8, t_9\}, \{t_{10}\})^l\rangle$. Note that the superfix l is ignored at this stage. Now the last member of $\mu^{(2)}$ is the unit set $\{t_{10}\}$; it is found to occur as the last transition in the path $\alpha_7 = \langle\{t_3\}, \{t_4\}, \{t_{10}\}\rangle$; the transition $\{t_{10}\}$ is deleted from μ; the other transitions t_3 and t_4 occur in the paths α_5 and α_6 as well; hence, t_3 and t_4 are not deleted from μ. The path α_7 is placed before the path α_8 in μ^r thereby, $\mu^{r(2)}$ becomes $\mu^{r(3)} = \langle\alpha_7.\alpha_8.\alpha_9\rangle$ and the computation $\mu^{(3)}$ becomes $\mu^{(4)} = \langle\{t_1, t_2, t_6, t_7\}, (\{t_3\}, \{t_4\}, \{t_8, t_9\})^l\rangle$.

Now, the last member in $\mu^{(4)}$ is $\{t_8, t_9\}$ which is not a unit set. The transition t_8 is the last member of the path α_5; the transition t_9 is the last member of the path α_6. As the transitions t_8 and t_9 are parallelizable (i.e., have no dependence on each other), the paths α_5 and α_6 are also parallelizable. This fact is formalized in Theorem 4 given subsequently and proved in detail in Appendix D. Hence, they can be chosen to be placed in any order before α_7 in $\mu^{r(3)}$. Let us decide to place α_6 and then α_5; so $\mu^{r(4)} = \langle\alpha_5.\alpha_6.\alpha_7.\alpha_8.\alpha_9\rangle$ and $\mu^{(5)}$ becomes $\langle\{t_1, t_2, t_6, t_7\}, (\{t_3\}, \{t_4\}\ )^l\rangle$. Now, in $\mu^{(5)}$, the last member comprising $\{t_4\}$ is not the last transition of any paths. This implies that all the paths having $\{t_4\}$ must have got included by now in $\mu^{r(4)}$ (which is indeed the case as $\{t_4\}$ occurs in the paths α_5, α_6 and α_7 all of which occur in $\mu^{r(4)}$). Hence, t_4 is deleted from $\mu^{(5)}$. Therefore, $\mu^{(5)}$ becomes $\mu^{(6)} = \langle\{t_1, t_2, t_6, t_7\}, (\{t_3\})^l\rangle$. For the same reason, t_3 will be deleted from μ^6 resulting in μ^7. It may now be noted that the last five members in $\mu^{(1)}$ iterates l times so $\mu^{(1)}$ is same as those of $\mu^{(7)}$; so the process by which $\mu^{(1)}$ got transformed to $\mu^{(7)}$ and $\mu^{r(1)}$ got transformed to $\mu^{r(4)}$, as described in the above paragraphs, will be repeated resulting in $\mu^{r(4+l)} = \langle(\alpha_5.\alpha_6.\alpha_7.\alpha_8)^l.\alpha_9\rangle$ and $\mu^{(7+l)} = \langle\{t_1, t_2, t_6, t_7\}\rangle$.

Now the last (and the only member) of $\mu^{(7+l)}$ is $\{t_1, t_2, t_6, t_7\}$. They are respectively the last (and only) transitions of the parallelizable paths $\alpha_1, \alpha_2, \alpha_3$ and α_4. Hence these paths can be placed in arbitrary order in $\mu^{r(8+l)}$. Repeating the steps, described above, thrice for the three paths, we get the final reordered sequence $\mu^r = \mu^{r(8+l)} = \langle\alpha_1.\alpha_2.\alpha_3.\alpha_4.(\alpha_5.\alpha_6.\alpha_7.\alpha_8)^l.\alpha_9\rangle$ and $\mu^{(11+l)}$ becomes empty whereupon the process terminates.

We formalize the discussion using the following two theorems to show the validity of the path based equivalence checking mechanism. Proofs of these two theorems are given in Appendix D.

Theorem 1 *Let* $\alpha_1 = \langle T_{1,1}, T_{2,1}, \ldots, T_{m,1} \rangle$ *and* $\alpha_2 = \langle T_{1,2}, T_{2,2}, \ldots, T_{n,2} \rangle$ *be two paths such that their last sets of transitions* $T_{m,1}$ *and* $T_{n,2}$ *are parallelizable. Then,* α_1 *and* α_2 *are parallelizable.*

Theorem 2 *Let* Π *be the set of all paths of a CPN model obtained from a set of static cut-points. For any computation* μ_p *of an out-port* p *of the model, there exists a reorganized sequence* μ_p^r *of paths of* Π *such that* $\mu_p \simeq \mu_p^r$.

Theorem 2 establishes the semantic equivalence of a computation with a corresponding concatenation of some SCP induced paths. In Appendix D, we present the formal algorithm of obtaining the reordered sequence μ^r of paths corresponding to any computation μ; a formal treatment of correctness of the algorithm is also included. Theorem 2, therefore, permits us to designate the set Π of paths obtained using static cut-points as *a path cover* of the model. The validity of any path based equivalence checking methods is captured by the following theorem. The proof of the theorem is given in Appendix E.

Theorem 3 *For any two models* N_0 *and* N_1*, if there exists a finite path cover* $\Pi_0 = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ *of* N_0 *for which there exists a set* $\Psi_1 = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_m\}$ *of sets of paths of* N_1 *such that for all* $i, 1 \leq i \leq m$, $\alpha_i \simeq \beta$*, for all* $\beta \in \Gamma_i$*, then* N_0 *is contained in* N_1 *(*$N_0 \sqsubseteq N_1$*).*

3.5 Equivalence checking mechanism

In this subsection, we describe how the equivalence checking method works for the current example comprising the source program P_s and the transformed program P_t of Figure 2(a). We have explained the CPN model N_0, say, for P_s in subsection 3.1. So here we first describe the CPN model N_1 of P_t before describing the equivalence checking steps.

In the transformed program P_t of Figure 2(b), the two independent statements "$m = m * 10$ and $n = n/10$" in the single `while` loop of P_s are distributed over two different parallel loops. The loop control variables corresponding to these two loops are i and j which start with an identical initial value 0. The input variable l serves as the limits of both the loops over the two different parallel loops.

In the corresponding CPN model N_1 of Figure 3(b), the places p_5' and p_6' represent the variables i and j which serve as the loop control variables for the respective the parallel loops; the corresponding loop conditions become $i < l \simeq c_1'$ and $j < l \simeq c_2'$; the places p_2' and p_3' hold two copies of the variable l specifying the limits of the two parallel loops in P_t; the transitions t_1' and t_3' not only initialize the places p_5', p_6', p_2' and p_3' but also create two parallel threads corresponding to the `parbegin` statement. The subnet corresponding to the while-loops are obtained by the same reasoning used to obtain the subnet of the while-loop of P_s in N_0. In the present case, however, the loop exit transitions t_{11}' and t_{12}' associated with $\neg c_1$ and $\neg c_2$ respectively, only achieve exits from the loops; more specifically, unlike the exit transition t_5 of N_0 in Figure 3(a), it

cannot accomplish the task of the assignment statement "$k = m + n$" because that happens only after merging of the two parallel threads. The transition t'_{15} serves two purposes — it accomplishes the merging of the two parallel threads corresponding to the `parend` statement and accordingly have p'_{23} and p'_{24} as its pre-places; secondly, it captures the computation corresponding to the assignment statement "$k = m + n$" producing the output token corresponding to the output variable k at the out-port p'_{25}.

For N_0 with the set P_0 of places, the set of variables $V = \{i, l, m, n, k\}$ and the place to variable mapping is f^0_{pv}: $P_0 \rightarrow V \cup \{\delta\}$, where $\{p_1, p_{11}, p_{12}, p_{14}, p_{15}, p_{16}\} \mapsto \delta$, for dummy in-ports and synchronizing places, $\{p_3, p_{13}\} \mapsto i, \{p_7, p_9\} \mapsto m, \{p_8, p_{10}\} \mapsto n, \{p_2, p_4, p_5, p_6\} \mapsto l, p_{12} \mapsto k$. For N_1 having the set P_1 of places, the set of variables is $V_1 = V \cup \{j\}$ and the place to variable mapping is f^1_{pv} : $P_1 \rightarrow V_1 \cup \{\delta\}$, where $\{p'_4, p'_{11}, p'_{14}, p'_{21}, p'_{22}\} \mapsto \delta, \{p'_{15}, p'_{17}, p'_{23}\} \mapsto m, \{p'_5, p'_{13}\} \mapsto i, \{p'_6, p'_{14}\} \mapsto j, \{p'_{16}, p'_{17}, p'_{24}\} \mapsto n, \{p'_1, p'_2, p'_3, p'_7, p'_8, p'_9, p'_{10}\} \mapsto l, p'_{23} \mapsto k$. The program P_s works on the variable set V while the program P_t works on the variable set $V_1 = V \cup \{j\}$. However, the sets of input variables and output variables are the same namely, $\{i, l, m, n, k\}$; in other words, $f^0_{pv}(inP_0) = f^1_{pv}(inP_1)$ and $f^0_{pv}(outP_0) = f^1_{pv}(outP_1)$. In general, both $V - V_1$ and $V_1 - V$ may be nonempty. In course of checking equivalence, a relation between these two subsets is revealed which consistently holds for all the paths of the models.

For the model N_1, the set C' of cut-points is $\{p'_1, p'_4, p'_{15}, p'_{16}$ (in-ports), p'_{25} (out-port), $p'_2, p'_3, p'_5, p'_6, p'_{17}, p'_{18}$ (loop cut-points)$\}$. Using the path construction algorithm, the sets of paths obtained from Figures 3(a) and 3(b) are $\Pi_0 = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8, \alpha_9\}$, where, $\alpha_1 = \langle\{t_1\}\rangle, \alpha_2 = \langle\{t_6\}\rangle, \alpha_3 = \langle\{t_7\}\rangle, \alpha_4 = \langle\{t_2\}\rangle, \alpha_5 = \langle\{t_3\}, \{t_4\}, \{t_8\}\rangle, \alpha_6 = \langle\{t_3\}, \{t_4\}, \{t_9\}\rangle, \alpha_7 = \langle\{t_3\}, \{t_4\}, \{t_{10}\}\rangle, \alpha_8 = \langle\{t_3\}, \{t_{11}\}\rangle\}, \alpha_9 = \langle\{t_3\}, \{t_5\}\rangle$ and $\Pi_1 = \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7, \beta_8, \beta_9, \beta_{10}, \beta_{11}\}$, where, $\beta_1 = \langle\{t'_1\}\rangle, \beta_2 = \langle\{t'_3\}\rangle, \beta_3 = \langle\{t'_7\}\rangle, \beta_4 = \langle\{t'_9\}\rangle, \beta_5 = \langle\{t'_2\}, \{t'_{16}\}\rangle, \beta_6 = \langle\{t'_4\}, \{t'_{17}\}\rangle, \beta_7 = \langle\{t'_2\}, \{t'_5\}, \{t'_9\}\rangle, \beta_8 = \langle\{t'_4\}, \{t'_6\}, \{t'_{10}\}\rangle, \beta_9 = \langle\{t'_2\}, \{t'_5\}, \{t'_9\}, \{t'_{13}\}\rangle, \beta_{10} = \langle\{t'_4\}, \{t'_6\}, \{t'_{10}\}, \{t'_{14}\}\rangle, \beta_{11} = \langle\{t'_2, t'_4\}, \{t'_{11}, t'_{12}\}, \{t'_{15}\}\rangle$. Interestingly, the number of paths in the two models may be different.

The equivalence checker needs the following two *bijective* mappings associating the in-ports and the out-ports of the two models. Let inP_0, inP_1 represent the in-ports of N_0 and N_1, respectively; similarly, let $outP_0, outP_1$ be their respective out-ports. Let $f_{in} : inP_0 \leftrightarrow inP_1$ be $\{p_1 \mapsto p'_4, p_7 \mapsto p'_{15}\rangle, p_8 \mapsto p'_{16}\}$; let $f_{out} : outP_0 \leftrightarrow outP_1$ be $p_{17} \mapsto p'_{25}$. Using f_{in} and f_{out}, a place correspondence relationship η_p, say, is initialized as $\{\langle p_1, p'_4\rangle, \langle p_7, p'_{15}\rangle, \langle p_8, p'_{16}\rangle, \langle p_{17}, p'_{25}\rangle\}$. In addition, there should be an association of the uncommon variables of the models; let this relation be $\eta_v \subseteq ((V - V_1) \times V_1) \cup (V \times (V_1 - V))$; thus, if $\langle v, v'\rangle \in \eta_v$, then either v is a variable of N_0 which does not occur in the variable set V_1 of N_1 or v' is a variable in V_1 which does not occur in the variable set V of N_0. This relation is initialized as an empty set.

Definition 3 (Path equivalence, Transition correspondence and Place correspondence)

Let N_0 and N_1 be two CPN models with their in-port bijection f_{in} and out-port bijection f_{out}. Equivalence of paths of N_0 and N_1, a transition cor-

respondence relation, denoted as $\eta_t \subseteq T_0 \times T_1$, and a place correspondence relation, denoted as $\eta_p \subseteq P_0 \times P_1$, are defined as follows:

1. $f_{in} \subseteq \eta_p$,
2. Two paths α of N_0 and β of N_1 are said to be equivalent, symbolically denoted as $\alpha \simeq \beta$, if $\forall p \in {}^\circ\alpha$, there exists exactly one $p' \in {}^\circ\beta$ such that $f^0_{pv}(p) = f^1_{pv}(p')$, $\langle p, p' \rangle \in \eta_p$, $R_\alpha(f_{pv}({}^\circ\alpha)) \equiv R_\beta(f_{pv}({}^\circ\beta))$ and $r_\alpha(f_{pv}({}^\circ\alpha)) = r_\beta(f_{pv}({}^\circ\beta))$.
3. For any two equivalent paths α, β, $\langle$ last(α), last(β) $\rangle \in \eta_t$.
4. For any two equivalent paths α of N_0 and β of N_1, for any two places $p \in \alpha^0, p' \in \beta^0, \langle p, p' \rangle \in \eta_p$ if
   (a) $f^0_{pv}(p) = f^1_{pv}(p')$ or
   (b) if $f^0_{pv}(p) \in V - V_1$ or $f^1_{pv}(p') \in V_1 - V$ then
      i. either $\langle f^0_{pv}(p), f^1_{pv}(p') \rangle$ have already been associated with each other in η_v, or
      ii. they have not yet been associated with any other variables and can now be associated associated in η_v; also $\langle p, p' \rangle$ is put in η_p;

Let us now examine the equivalence checking steps. Instead of giving the detailed steps of identifying the equivalent path of the model N_1 of Figure 3(b) for each path of the model N_0 of Figure 3(a), only some of the cases are given to highlight the important issues involved.

3.5.1 For the Path α_1

Find candidate path(s) of α_1: The method first identifies some candidate paths from Π_1, whose pre-places are in place-correspondence with the pre-places of the path α_1. The path β_2 is identified as the only candidate path for α_1 because $\langle {}^\circ\alpha_1 = p_1, {}^\circ\beta_2 = p'_4 \rangle \in \eta_p$.

Check equivalence of the condition of execution and equality of transformation: Since the conditions of execution of the paths α_1 and β_2 are equivalent and their data transformations are same, the method returns the set $\Gamma = \{\beta_2\}$ as the set of paths equivalent to α_1.

Enhance transition correspondence (η_t), place correspondence (η_p) and uncommon variable correspondence (η_v) The method updates the following sets:
the set of equivalent paths E becomes $\{\langle \alpha_1, \beta_2 \rangle\}$;
The last transition of α_1, i.e., t_1 should be made to have transition correspondence with the last transition of β_2, i.e., t'_3. Therefore, $\langle t_1, t'_3 \rangle$ is put in η_t;
$\alpha^\circ_1 = \{p_3\}$ should be made to have place correspondence with $\beta^\circ_2 = \{p'_5, p'_6\}$; since $f^0_{pv}(p_3) = f^1_{pv}(p'_5) = i$, so, $\langle p_3, p'_5 \rangle$ is put in η_p.
However, $f^1_{pv}(p'_6) = j \neq f^0_{pv}(p_3)$; but j is an uncommon variable and at this point, $j = i$ since $p'_5, p'_6 \in \beta^\circ_2$. So we put $\langle j, i \rangle$ in η_v and $\langle p_3, p'_6 \rangle$ is also put in η_p.
Similarly, it is found that

1. $\alpha_2 \simeq \beta_3 \Rightarrow \langle$ last(α_2), last (β_3)$\rangle = \langle t_6, t'_7 \rangle \in \eta_t \Rightarrow \langle \alpha^\circ_2, \beta^\circ_3 \rangle = \langle p_9, p'_{17} \rangle \in \eta_p$,
2. $\alpha_3 \simeq \beta_4 \Rightarrow \langle$ last(α_3), last (β_4)$\rangle = \langle t_7, t'_9 \rangle \in \eta_t \Rightarrow \langle \alpha^\circ_3, \beta^\circ_4 \rangle = \langle p_{10}, p'_{18} \rangle \in \eta_p$ and
3. $\alpha_4 \simeq \beta_1 \Rightarrow \langle$ last(α_4), last (β_1)$\rangle = \langle t_2, t'_1 \rangle \in \eta_t \Rightarrow \langle \alpha^\circ_4, \beta^\circ_1 \rangle = \{\langle p_4, p'_2 \rangle, \langle p_4, p'_3 \rangle\} \in \eta_p$.

3.5.2 For the Path α_5

Find candidate path(s) of α_5: The candidate path is chosen as β_7 as ${}^\circ\alpha_5 = \{p_4, p_3, p_9\}$, ${}^\circ\beta_7 = \{p'_2, p'_5, p'_{17}\}$ and $\langle p_4, p'_2\rangle, \langle p_3, p'_5\rangle, \langle p_9, p'_{17}\rangle \in \eta_p$.

Check equivalence of the condition of execution and equality of transformation: The conditions of execution $R_{\alpha_5}(v_{p_3} \leq v_{p_5})$ and $R_{\beta_7}(v_{p'_5} \leq v_{p'_7})$ are same because $\langle p_3, p'_5\rangle \in \eta_p$ and hence the values $v_{p_3} = v_{p'_5}$; therefore $v_{p_5} = v_{p'_7}$ always holds. Similarly, from the place correspondence of p_9, p'_{17}, their data transformations $r_{\alpha_5} = v_{p_9} * 10$ and $r_{\beta_7} = v_{p'_{17}} * 10$ are identified to be identical. The fact that $\alpha_5 \simeq \beta_7$ will be inferred identically this time using the correspondence $p_4 \in {}^\circ\alpha_5$ also with $p'_5 \in {}^\circ\beta_7$.

Enhance transition correspondence (η_t), place correspondence (η_p) and uncommon variable correspondence (η_v): In the process, $\langle t_8, t'_9\rangle$ are include in η_t and $\langle p_{14}, p'_{19}\rangle$ are included in η_p. The variable correspondence η_v remains unchanged.

Similarly it is found that $\alpha_6 \simeq \beta_8 \Rightarrow \langle t_9, t'_{10}\rangle \in \eta_t \Rightarrow \langle p_{15}, p'_{20}\rangle \in \eta_p$.

3.5.3 For the Path α_7

Find candidate path(s) of α_7: While choosing the candidate path, the method identifies that ${}^\circ\alpha_7 = \{p_3, p_4\}$ and the pre-places of both paths ${}^\circ\beta_9 = \{p'_2, p'_5\}$, ${}^\circ\beta_{10} = \{p'_3, p'_6\}$ are in η_p with the pre-places of α_7.

Check equivalence of the condition of execution and equality of transformation: The method then identifies $R_{\alpha_7} \equiv R_{\beta_9}$ as well as $R_{\alpha_7} \equiv R_{\beta_{10}}$; similarly, $r_{\alpha_7} = r_{\beta_9}$ and $r_{\alpha_7} = r_{\beta_{10}}$. So it returns $\Gamma = \{\beta_9, \beta_{10}\}$. *The method registers both these paths to be equivalent to α_7.*

Enhance transition correspondence (η_t), place correspondence (η_p) and uncommon variable correspondence (η_v): As $\alpha_7 \simeq \beta_9$, $\langle t_{10}, t'_{13}\rangle$ are include in η_t and $\langle p_{16}, p'_{21}\rangle$ are include in η_p. As $\alpha_7 \simeq \beta_{10}$, $\langle t_{10}, t'_{14}\rangle$ are included in η_t and $\langle p_{16}, p'_{22}\rangle$ are included in η_p. The variable i associated with α_7° and the variable j associated with β_{10}° conform to the already included pair $\langle i, j\rangle$ in η_v. For both the above pairs η_v remains unchanged. Similarly,

1. $\alpha_8 \simeq \beta_5 \Rightarrow \langle t_{11}, t'_{16}\rangle \in \eta_t$, the place correspondence η_p, however, happens to remain unchanged. The variable correspondence η_v remains unchanged.
2. $\alpha_8 \simeq \beta_6 \Rightarrow \langle t_{11}, t'_{17}\rangle \in \eta_t$, the place correspondence η_p is also unchanged. The variable correspondence η_v remains unchanged.

Note that the same path α_7 is equivalent to two paths β_9 and β_{10}; so is the case for the path α_8.

3.5.4 For the Path α_9

Find candidate path(s) of α_9: The pre-places ${}^\circ\alpha_9 = \{p_3, p_4, p_9, p_{10}\}$ are used identically by the method to identify β_{11} as the only candidate path since ${}^\circ\beta_{11} = \{p'_5, p'_6, p'_2, p'_3, p'_{17}, p'_{18}\}$ and $\langle p_3, p'_5\rangle, \langle p_3, p'_6\rangle, \langle p_4, p'_2\rangle, \langle p_4, p'_3\rangle, \langle p_9, p'_{17}\rangle, \langle p_{10}, p'_{18}\rangle \in \eta_p$.

Check equivalence of the condition of execution and equality of transformation: It also identifies the equivalence of their conditions of execution and equality of data transformations; so it returns β_{11} as the equivalent of α_9.

Enhance transition correspondence (η_t), place correspondence (η_p) and uncommon variable correspondence (η_v): The method updates E by adding $\langle \alpha_9, \beta_{11} \rangle$, η_t by adding $\langle t_5, t'_{15} \rangle$ and η_p by adding $\langle p_{17}, p'_{25} \rangle$. The variable correspondence η_v remains unchanged.

Therefore, the set E of equivalent pairs of paths are as follows: $\langle \alpha_1, \beta_2 \rangle, \langle \alpha_2, \beta_3 \rangle, \langle \alpha_3, \beta_4 \rangle, \langle \alpha_4, \beta_1 \rangle, \langle \alpha_5, \beta_7 \rangle, \langle \alpha_6, \beta_8 \rangle, \langle \alpha_7, \beta_9 \rangle, \langle \alpha_7, \beta_{10} \rangle, \langle \alpha_8, \beta_5 \rangle, \langle \alpha_8, \beta_6 \rangle, \langle \alpha_9, \beta_{11} \rangle$.

The method now finds that all the paths of N_0 have equivalent paths in N_1 with proper correspondence of their pre-places; also each path of N_1 are found to have equivalence with some path in N_0; accordingly, it declares the models (and hence the programs P_s, P_t) to be equivalent. The formal treatment of the algorithm is given in the Appendix F.

4 Experimental Results

The static cut-point induced path based equivalence checking method is implemented in *C* and tested on some sequential as well as parallel examples on a 2.0 GHz Intel(R) Core(TM)2 Duo CPU machine (using only a single core). We refer to this implementation as the `SCPEQX` module.

The experiment has been carried out in three parts. In the first part, some sequential programs have been subjected to code optimizing transformations involving instruction level parallelization, code motions, dynamic loop scheduling, loop swapping, etc.; the equivalence checker has been tested for such cases; this part has been described in Subsection 4.1. In the second part, some sequential programs have been subjected to thread level parallelizing transformations; Subsection 4.2 describes this part. Finally, in the third part, described in Subsection 4.3, the performance of the equivalence checker on some manually injected faulty transformations has been examined. In the site[2], the following subdirectories are made available for reproducibility of the current experimental results:

1. *Subdirectory "`SCPEQX`":* It contains the `SCPEQX` module, the automated model constructor module and all the examples of the test suite; each example comprises a pair of source and transformed programs, that of the corresponding models and the corresponding output produced by the `SCPEQX` module.
2. *Subdirectory "`DCPEQX`":* It contains the `DCPEQX` module, the automated model constructor and a copy of the same set of examples as provided under the subdirectory "`SCPEQX`".
3. *Subdirectory "`FSMDEQX`":* This subdirectory includes two versions of the finite state machine models with data paths (FSMD) based equivalence

[2] http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db/samatulyata.html

checking method – one with path extension (PE) capability and the other with value propagation (VP) capability. The examples in the test suite given in the above two subdirectories where parallelizing transformations are applied are not included in the example list given here. Other examples as given under `SCPEQX` subdirectory are included.

4. *CPN tool:* It contains the well known CPN tool [28] as well as all the models – both hand constructed ones and those constructed by the automated model constructor – which are tested using this tool.

4.1 Validation of several code optimizing transformations

The benchmark suite of the prominent high level synthesis compiler SPARK[24] has been used for testing the performance of the equivalence checker on code optimizing transformations. Each of the sequential programs is then transformed using some human guided transformations or by the SPARK compiler. Table 1 depicts the transformations that are applied for each of the sequential examples. It is to be noted that both uniform and non-uniform code optimizing transformations are carried out by the SPARK compiler. The automated model constructor, reported in [47] , has been used to obtain the CPN models of the source and the transformed programs. Each of these models is tested using the CPN simulator[28]. Finally, for each pair of the source and transformed programs, the corresponding CPN models are fed as inputs to all the modules namely, `FSMDEQX` module with path extension and that with value propagation, `DCPEQX` and finally, `SCPEQX` modules, to compare their performances by observing certain parameters as recorded in Table 2. For calculating time we have used `get_cpu_time()`.

For all the examples in the test suite, as listed in Table 2, the source and the transformed programs were successfully declared to be equivalent by the `SCPEQX` module. Column 2 depicts the number of paths obtained from both original and transformed CPN models. From columns 3 and 4, we notice that the costly path extension is needed for almost all the examples for the `FSMDEQX` module using path extension; for the `DCPEQX` module, it is needed for less number of examples; the path extension, however, was not needed for the `SCPEQX` module for any of the examples (and hence not included as a separate column for this module). It may be noted that under the column `FSMDEQX Time`, there are two sub-columns `PE` and `VP`; the former corresponds to the runtimes recorded for the path extension `FSMDEQX` module [29] and the latter to those recorded for the value propagation based `FSMDEQX` module [10]. The path construction times and the equivalence checking times have been separately observed for both the DCPEQX and SCPEQX modules, as unlike the FSMDEQX module(s), the path construction phases involve significant overheads. As the path construction times for the DCPEQX module have been reported in [8], we have not put it as a separate column in Table 2; however, in column 6 (designated as "`DCPEQX Times`") we have put two sub-columns, one for the equivalence checking times and the other for the total

Example	Transformations
MODN	Uniform and non-uniform code motion (SPARK)
SUMOFDIGITS	Dynamic loop scheduling (DLS) (human guided)
PERFECT	DLS, uniform and non-uniform code motion, Code motion across Loop (human guided)
PRIMEFACS	DLS, Uniform and non-uniform code motion (SPARK)
GCD	Uniform and non-uniform code motion (SPARK)
TLC	Uniform and non-uniform code motion (SPARK)
DCT	Uniform and non-uniform code motion (SPARK)
LRU	Uniform and non-uniform code motion (SPARK), Code motion across Loop (human guided)
LCM	Uniform and non-uniform code motion (SPARK), Code motion across Loop (human guided)
MINANDMAX-S	Loop swapping (human guided)
BARCODE	Uniform and non-uniform code motion (SPARK)
DIFFEQ	Uniform and non-uniform code motion (SPARK)
DHRC	Uniform and non-uniform code motion (SPARK)
PRAWN	Uniform and non-uniform code motion (SPARK)
IEEE -754	Uniform and non-uniform code motion (SPARK), Code motion across Loop (human guided)
QRS	Uniform and non-uniform code motion (SPARK), Code motion across Loop (human guided)
EWF	Uniform and non-uniform code motion (SPARK)

Table 1 Transformations applied on sequential examples.

times taken by the DCPEQX module; subtracting the first sub-column entries from the second sub-column entries would yield the path construction times.

The path construction times of DCPEQX modules can be found to be slightly higher than those for SCPEQX because of two reasons — first, the overhead of token tracking execution needed for introducing DCPs in DCPEQX is not there in SCP induced paths; secondly, the number of SCP induced paths being less than that of DCP induced paths, the overhead of finding the corresponding conditions of execution and the data transformations by symbolic execution of paths is also less. The time taken by the equivalence checking phase of the SCPEQX module is slightly better than the corresponding time taken by DCPEQX module (except for PRIMEFACS example). However, no distinct improvement can be identified for the SCPEQX module compared to the DCPEQX module. The Total Time for SCPEQX module is much higher than the FSMDEQX modules. However, the total time for SCPEQX module is slightly better than the total time for DCPEQX module.

For the MINANDMAX-S example, both FSMDEQX (PE) and FSMDEQX (VP) fail because the loop swapping transformation is involved and this transformation cannot be handled by the both FSMDEQX (PE) and FSMDEQX (VP) modules. The last five rows involve code motion across loops (indicated by the suffix ''CM''). It is to be noted that FSMDEQX (PE) cannot handle the transformation; however, FSMDEQX (VP), DCPEQX and SCPEQX successfully handle the transformation.

.

Example	♯Paths		Path Extension (FSMDEQX)	Path Extension (DCPEQX)	FSMDEQX Time (μs)		DCPEQX Time (μs)		SCPEQX			
					PE	VP	EqChk	Total	Path Construction Time (μs)		EqChk	Total
	Original	Transformed							Original	Transformed	Time (μs)	Time (μs)
MODN	30	30	YES	YES	16001	15892	15581	37789	9451	10231	11079	30761
SUMOFDIGITS	9	9	YES	YES	8000	8000	13302	25477	5231	5187	13033	23451
PERFECT	53	23	YES	YES	8456	8372	9299	53674	20134	9542	7589	37274
LRU	56	56	YES	NO	20001	19872	21462	855785	423142	383451	20123	826716
PRIMEFACS	35	20	YES	YES	6352	6149	5568	27414	9263	9257	5636	24356
DCT	1	1	NO	NO	2102	1902	6717	42354	12345	12876	2902	28123
LCM	45	45	YES	NO	16231	16174	12285	43245	15341	14402	9510	39235
BARCODE	765	765	YES	YES	125189	125189	123175	9316779	5208310	5208190	121969	10538469
MINANDMAX-S	40	38	×	NO	×	×	15989	40763	11534	10243	15256	37033
DIFFEQ	32	21	YES	NO	42500	42389	36195	64189	11123	10524	35123	56770
DHRC	100	85	YES	YES	188300	186729	185674	8772586	4493587	3912432	185321	8591340
PRAWN	610	610	YES	NO	293400	291676	293876	78037279	7106251	7012412	290451	14409114
IEEE 754	210	210	YES	YES	195741	186824	195330	6146482	2776134	2752124	185169	5713427
QRS	56	56	YES	NO	20001	19346	21462	855785	423142	383451	20123	826716
GCD	43	43	YES	NO	12567	12563	12472	41432	14231	13210	10797	38238
TLC	70	70	YES	YES	16121	14230	5795	288671	184352	83456	4321	272129
EWF	351	312	YES	YES	34368	33413	36524	3344664	2034721	1151219	35149	3221089
LCM-CM	45	45	–	NO	×	16035	12285	43245	15341	14402	9510	39235
IEEE 754-CM	210	210	–	YES	×	176572	195330	6146482	2776134	2752124	185169	5713427
LRU-CM	56	56	–	NO	×	18549	21462	855785	423142	383451	20123	826716
QRS-CM	56	56	–	NO	×	19234	21462	855785	423142	383451	20123	826716
PERFECT-CM	53	23	–	YES	×	7278	9299	53674	20134	9542	7589	37274

Table 2 Equivalence checking results for several sequential examples using automated model constructor

Example	Transformations
BCM	Boosting up code motion for parallel threads
MINANDMAX-S	Thread level parallelization
LUP	Thread level parallelization
DEKKER	Thread level parallelization
PETERSON	Thread level parallelization

Table 3 Transformations carried out using parallelizing compilers

4.2 Validation of several thread level parallelizing transformations

In the second part of the experimentation, we have tested the performance of `SCPEQX` module for parallelizing transformations and compared them with the corresponding observations for the `DCPEQX` module; the two `FSMDEQX` modules cannot handle such transformations.

Five hand constructed examples have been taken for this purpose. The example BCM is taken from [31]. For LU Decomposition with Pivoting (LUP), we have only taken the pivoting routine which does not contain any array. The detailed functionality of this source program is given in PLuTo example suite [14]. MINANDMAX-S computes sum of the maximum of four numbers n_1, n_2, n_3, n_4 and the minimum of the four numbers n_1, n_5, n_6 and n_7 (having n_1 as the common element). DEKKER's and PETERSON's examples describe the classical solutions to the mutual exclusion problem of two concurrent processes. Since our mechanism does not handle writable shared variables among parallel threads, we have considered a single process in each of these cases; also we have introduced a series of dummy assignment statements within the critical section which was otherwise left unspecified in the code.

The five sequential programs are transformed by two prominent thread level parallelizing compilers, PLuTo [14] and Par4All; the transformed versions accordingly have parallel structures. Table 3 depicts the type of transformations applied for each of the examples. To be noted that for each of the five source programs, we have two transformed versions, one produced by PLuTo and the other by Par4All [3]. Before submitting the sequential programs to these compilers for parallelization, all the segments in the source program which have scope for thread level parallelization are designated manually using `pragma scop` such that the compiler transforms only that particular portion of the code. To contain the size of the hand constructed models, we have taken only those portions of the codes which are transformed by the compilers excluding the intervening segments. The variables which are only used within the scope are the in-ports of the CPN model. We construct two CPN models by hand – one from the original code snippet(s) present in `pragma scop` of the original program and the other from the corresponding code snippet(s) present in `CLooG scop` of the transformed program. All the above parallel examples do not contain any writable shared variables. To guard against human errors, each of these models is checked for validity using the CPN tool [28].

[3] http://www.par4all.org/

Example	Original CPN				Transformed CPN							
					PLuTo				Par4All			
	♯places	♯transitions	♯SCPs	♯paths	♯places	♯transitions	♯SCPs	♯paths	♯places	♯transitions	♯SCPs	♯paths
BCM	10	6	6	1	11	7	6	1	11	7	6	1
PETERSON	32	30	15	10	30	28	14	10	30	28	14	10
LUP	55	53	30	18	52	50	29	18	52	50	29	18
DEKKER	34	32	20	12	30	29	18	12	30	29	18	12
MINANDMAX-S	28	21	11	14	28	21	11	14	28	21	11	14

Table 4 Characterization of hand constructed models of parallel examples

Finally, we feed these two pairs of CPN models as (two independent) inputs to our SCPEQX module. The one pair of CPN models are constructed from the source program and the program translated by PLuTo compiler; the other pair of CPN models are constructed from the source program and the program translated by Par4All compiler. We observe the set of parameters as indicated in Tables 4, 5 and 6.

Table 4 summarizes the characterization of each of the examples in terms of the the numbers of places and transitions of the corresponding models; the numbers of static cut-points and the SCP induced paths for the models are also recorded.

Columns 2 and 3 of Table 5 depict the number of paths record for DCPEQX module [8] and columns 4 and 5 record the same parameters for SCPEQX module. From Table 5, we observe that the number of paths in SCPEQX method is lesser than the DCPEQX method.

In Table 6, column 2 depicts the path construction times for the SCPEQX module which are again found to be smaller than the corresponding path construction times recorded in [8] because of the same reasons as underlined in subsection 4.1. The last column of Table 6 show the corresponding equivalence checking times of the SCPEQX and the DCPEQX modules. Path extension is needed for the DCPEQX module only for the LUP example; for the SCPEQX module, however, it is not needed for any of the examples. The equivalence checking times for the SCPEQX module are slightly smaller than those for the DCPEQX modules as the numbers of SCP induced paths are less than those of the DCP induced paths.

Example	Orig (DCPEQX)	Transformed (DCPEQX)		Orig (SCPEQX)	Transformed (SCPEQX)	
		PLuTo	Par4All		PLuTo	Par4All
BCM	3	3	3	1	1	1
MINANDMAX-S	21	21	21	14	14	14
LUP	35	34	34	18	18	18
DEKKER	17	17	17	12	12	12
PETERSON	12	12	12	10	10	10

Table 5 Number of model paths for parallelizing transformation examples

4.3 Experimentation with faulty transformations

The third part of the experimentation is aimed at examining the efficacy of the SCPEQX module for faulty transformations. We take the original behaviours

Example	Path Const Time (μs)(SCPEQX)			Equivalence Chk Time (μs)			
				DCPEQX Time (μs)		SCPEQX Time (μs)	
	Org	PLuTo	Par4All	PLuTo	Par4All	PLuTo	Par4All
BCM	455	434	434	4659	4659	3561	3561
MINANDMAX-P	4189	4189	4189	24335	24335	18341	18341
PETERSON	4812	4624	4642	11231	11231	7456	8423
LUP	9873	9113	8978	33633	31235	29845	31012
DEKKER	3902	3123	3123	13428	14352	11231	10234

Table 6 Path construction times and equivalence checking times for parallelizing transformation examples

for some examples taken from the sequential and parallel example suites and manually inject some errors in the transformed program code. We have introduced the following types of (both instruction level and thread level) erroneous code transformations.

1. *Type 1:* non-uniform boosting up code motions from one branch of an if-then-else block to the block preceding it which introduce false data dependencies in the other branch of the if-then-else block; this has been injected in the `GCD` and `MODN` examples.
2. *Type 2:* non-uniform duplicating down code motions from the basic block preceding an if-then-else block to one branch of the if-then-else block which remove data dependency in the other branch; this has been injected in the `TLC` example.
3. *Type 3:* mix of some correct code motions and incorrect code motions in `LCM` and `LRU` examples.
4. *Type 4:* data-locality transformations which introduce false data-locality in the body of the loop in `MINANDMAX-S` and `PETERSON` examples.

For each of the five examples involving code optimizing transformations, the CPN models are constructed using the automated model constructor; for the two examples involving thread level parallelizing transformations, the models are constructed manually. The models are tested using the CPN tool [28]. (The faults injected manifest themselves when the models of the erroneously transformed programs are tested.)

Finally, for each of the five examples, we feed the two CPN models to `SCPEQX` modules and observe the set of parameters as described in Table 7. For all the examples, all the non-equivalent paths are reported correctly. Table 7 depicts the types of the errors introduced in the examples, the number of operations moved, the number of non-equivalent path pairs revealed and the execution times taken by the `FSMDEQX` modules (both PE and VP), the `DCPEQX` module and the `SCPEQX` module; the performances of the `DCPEQX` and `SCPEQX` modules are assessed on the hand constructed as well as automatically constructed models for the five examples namely, GCD, MODN, TLC, LCM and LRU; for the last two examples, namely PETERSON and MINANDMAX-S, only hand constructed models are used as the automated model constructor [47] cannot handle parallelizing transformations.

The last four columns of Table 7 record the total times (including path construction times and equivalence checking times) taken by the four modules. It is to be noted that in all cases, the non-equivalence detection time is comparable with the equivalence checking time. It is worth mentioning that in course of this experiment our equivalence checker has identified a bug of the PLuTo compiler (possibly due to faulty usage of an existing variable namely, $t1$, holding intermediate results in the source program as the loop control variable $t1$ in the transformed program). The error was successfully detected by our equivalence checker.

Errors	Example	Number of operation(s) moved	Number of non equivalent pairs of paths	FSMDEQX (PE) Non-EqChk Time (μs)	FSMDEQX (VP) Non-EqChks Time (μs)	DCPEQX Non-EqChks Time (μs)	SCPEQX Non-EqChks Time (μs)
Type 1	GCD	1	3	10435	10142	42872	35413
	MODN	1	2	15456	13471	34048	27132
Type 2	TLC	2	4	14592	13780	123414	42174
Type 3	LCM	2	3	11412	10619	51231	45987
	LRU	1	1	19278	16143	733452	685412
Type 4	PETERSON	4	3	×	×	10913	6534
	MINANDMAX-S	2	2	×	×	24347	15463

Table 7 Non-Equivalence checking times for faulty translations

5 Related Work

Translation validation was introduced by Pnueli et al. in [42] and were demonstrated by both Necula et al. [39] and Rinard et al. [45]. Then the approach is further enhanced by Kundu et al. [32] where they verified the high-level synthesis tool named, SPARK. It is to be noted that all the above techniques are basically bisimulation based method. The work reported in [46] describes a method of formally verifying a compiler back-end from a simple imperative intermediate language Cminor to PowerPC assembly code using a Coq [1] proof assistant. A loop parallelizing transformation validation method comprising rewrite rules has been reported in [12]. A bisimulation method for parallel programs is also reported in Milner et al. [37]. A major limitation of these above methods [39,32,52,37,22] is that they can verify only structure preserving transformations. If the code moves beyond the basic block boundaries, the methods cannot validate. To alleviate this shortcoming, a path based equivalence checker for the FSMD model is proposed for sophisticated uniform and non-uniform code motions and code motions across loops [29,10]. Presently, however, they cannot handle loop swapping transformations as well as several thread-level parallelizing transformations because being a sequential model of computation, FSMD cannot capture parallel behaviours straightway; modeling concurrent behaviours using CDFGs is significantly more complex due to all possible interleavings of the parallel operations.

The literature records no significant attempts for devising formal equivalence checking methods using Petri net based models which is essentially a parallel model of computation. Literature records several works on property verification using several Petri net based modelling paradigm [17,34,15,44,19,16,28, 53,27]. In [8], the validation of loop swapping and thread level parallelising transformations using Petri net based models of programs was reported.

6 Conclusion

A path based computational equivalence checking mechanism for translation validation of programs using CPN modelling framework has been described. The mechanism hinges upon viewing any computation of the model as a concatenation of finite paths obtained by cutting loops at cut-points. While a similar mechanism, called DCPEQX [8], uses additional cut-points over and above those needed to cut the loops, the present mechanism, called SCPEQX, uses only cut-points needed to cut the loops. The SCPEQX mechanism has been illustrated informally using a non-trivial example. The illustration navigates through the steps of each stage of the mechanism. The discussion has been complemented with appendices where the method has been presented more formally and validated rigorously. Experiments with several examples involving instruction level parallelism, code motions, loop swapping and thread level parallelizations have been described; the performance of the (present) SCPEQX module has been compared with the performances of two other path based equivalence checking modules including the DCPEQX module. Performance of the present module on some erroneously transformed cases has also been examined.

The mechanism described in this paper is a path based mechanism. Another mechanism that has been effectively used in translation validation is bisimulation equivalence [32]. In [11], it has been shown how bisimulation relation can be derived from the output of the FSMDEQX (PE) module. It has been identified in this work that while bisimulation equivalence methods are not guaranteed to terminate, path based equivalence checkers do not suffer from this limitation. It would be an interesting future work to show how the present path based checker using CPN models yields bisimulation relations between the input models. As such, loop swapping transformations cannot be handled by FSMDEQX (with path extension or value propagation). The present method can handle such transformations and hence this would indirectly yield bisimulation relations for such transformations too.

The mechanism leaves scope for future improvement by alleviating its limitations as follows. One major bottleneck is that the models tend to become large as for each use of a definition of (i.e., an assignment to) a variable, a place is needed to hold a copy of the value computed by the definition. Another aspect of the CPN models that contributes to larger model size is the need for synchronizing places to capture control flow. Large model size affects the scalability of the method. CPN models have many sophisticated features

which, if used properly, can contain the model size significantly. For example, the restriction of one-safeness which permits each place to hold at most one token at a time can be relaxed; in such a case, an output place of a transition corresponding to a definition can hold as many tokens as the number of uses of the definition. Furthermore, instead of associating each transition with a function, we may have several output arcs of the transition, each associated with different functions; this way instruction level parallelism is captured more concisely. However, more concise is the model more is the sophistication needed for the analysis mechanism. Another limitation of the present work is its inability to handle arrays. The parallelizing transformations are applied more predominantly on array handling programs. This inability has been an impediment in examining the scalability of the method for parallelizing transformation validation. The relevant literature, however, use DFDGs (data flow dependence graphs) while Petri net models are essentially CDFGs despite their ability to reflect data dependence more vividly. Efficacy of the CPN modelling framework alleviating above limitations and encompassing arrays can be an interesting future work that merits independent treatment.

Acknowledgments

The authors are grateful for the valuable feedback given by the reviewer which has improved the readability of the paper immensely. We would also like to thanks Mr. Rakshit Mittal for useful discussion.

References

1. Coq. `https://coq.inria.fr/`.
2. A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Trans. Softw. Eng.*, 14(5):584–594, 1988.
3. S. G. Akl. *Parallel Computation: Models and Methods*. Prentice-Hall, Inc., 1997.
4. S. G. Akl. Editorial note. *Parallel Processing Letters*, 25(1), 2015.
5. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, December 1994.
6. H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff. The cetus source-to-source compiler infrastructure: Overview and evaluation. *IJPP*, 41(6):753–767, 2013.
7. S. Bandyopadhyay. *Path Based Equivalence Checking of Petri Net Representation of Programs for Translation Validation*. PhD thesis, I.I.T Kharagpur, India, 2016.
8. S. Bandyopadhyay, D. Sarkar, and C. Mandal. Equivalence checking of petri net models of programs using static and dynamic cut-points. *Acta Informatica*, Apr 2018.
9. S. Bandyopadhyay, D. Sarkar, C. A. Mandal, K. Banerjee, and K. R. Duddu. A path construction algorithm for translation validation using PRES+ models. *Parallel Processing Letters*, 26(2):1–25, 2016.
10. K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. Verification of code motion techniques using value propagation. *IEEE TCAD*, 33(8), 2014.
11. K. Banerjee, D. Sarkar, and C. Mandal. Deriving bisimulation relations from path extension based equivalence checkers. *IEEE Trans. Software Eng.*, 43(10):946–953, 2017.
12. C. J. Bell. Certifiably sound parallelizing transformations. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, pages 227–242, 2013.

13. E. Best and R. R. Devillers. Synthesis of persistent systems. In *Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings*, pages 111–129, 2014.
14. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI 08*, 2008.
15. B. Charron-Bost, S. Merz, A. Rybalchenko, and J. Widder. Formal verification of distributed algorithms (dagstuhl seminar 13141). *Dagstuhl Reports*, 3(4):1–16, 2013.
16. A. Corradini, L. Ribeiro, F. L. Dotti, and O. M. Mendizabal. A formal model for the deferred update replication technique. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, pages 235–253, 2013.
17. L. Cortes, P. Eles, and Z. Peng. Verification of embedded systems using a petri net based representation. In *System Synthesis, 2000. Proceedings. The 13th International Symposium on*, pages 149–155, 2000.
18. L. A. Cortés, P. Eles, and Z. Peng. Modeling and formal verification of embedded systems based on a petri net representation. *JSA*, 49(12-15):571–598, 2003.
19. S. A. da Costa and L. Ribeiro. Verification of graph grammars using a logical approach. *Sci. Comput. Program.*, 77(4):480–504, 2012.
20. S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentellni. Design of embedded systems: Formal models, validation and synthesis. DAC '99, pages 296–299, 1999.
21. Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha. Energy-optimizing source code transformations for operating system-driven embedded software. *Trans. on Embedded Computing Sys.*, 7(1):1–26, 2007.
22. X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 307–316, 2005.
23. M. R. Garey and D. S. Johnson. Complexity results for multi-processor scheduling under resource constraints. *Procs. of 8-th Annual Princeton Conference on Information Sciences and Systems*, 1974.
24. S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *Proc. of Int. Conf. on VLSI Design*, pages 461–466, Washington, DC, USA, Jan 2003. IEEE Computer Society.
25. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
26. K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 3*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1997.
27. K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
28. K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, May 2007.
29. C. Karfa, C. Mandal, and D. Sarkar. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):30, 2012.
30. A. Kejariwal, A. V. Veidenbaum, A. Nicolau, M. Girkar, X. Tian, and H. Saito. On the exploitation of loop-level parallelism in embedded applications. *Trans. on Embedded Computing Sys.*, 8(2):1–34, 2009.
31. J. Knoop and B. Steffen. Code motion for explicitly parallel programs. PPoPP '99, pages 13–24, 1999.
32. S. Kundu, S. Lerner, and R. Gupta. Validating high-level synthesis. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 459–472, Berlin, Heidelberg, 2008. Springer-Verlag.
33. J. Lidman, D. J. Quinlan, C. Liao, and S. A. McKee. Rose:: Fttransform-a source-to-source translation framework for exascale fault-tolerance research. In *DSN workshop*, pages 1–6, 2012.
34. D. Lime, O. H. Roux, C. Seidner, and L. Traonouez. Romeo: A parametric model-checker for petri nets with stopwatches. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 54–57, 2009.

35. Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Kogakusha, Tokyo, 1974.
36. P. Marwedel. *Embedded System Design*. Springer(India) Private Limited, New Delhi, India, 2006.
37. R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.
38. S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 77–88, New York, NY, USA, 2002. ACM.
39. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.
40. P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioural synthesis of asics. *IEEE Transactions on CAD of ICS*, 8(6):661–679, June 1989.
41. C. A. Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.
42. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166, 1998.
43. V. Raghavan. *Principles of Compiler Design*. Tata McGraw Hill Education Private Limited, New Delhi, 2010.
44. L. Ribeiro, O. M. dos Santos, F. L. Dotti, and L. Foss. Correct transformation: From object-based graph grammars to PROMELA. *Sci. Comput. Program.*, 77(3):214–246, 2012.
45. M. Rinard and P. Diniz. Credible compilation. Technical Report MIT-LCS-TR-776, MIT, 1999.
46. J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM (JACM)*, 60(3):art. 22:1–50, 2013.
47. K. Singh. Construction of Petri net based models for C programs, M.Tech. Dissertation, Dept. of Computer Sc. & Engg., I.I.T., Kharagpur, INDIA. `https://cse.iitkgp.ac.in/~chitta/pubs/rep/thesisKulwant.pdf`, `https://github.com/soumyadipcsis/Equivalence-checker/blob/master/thesisKulwant.pdf`, May 2016.
48. M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 248–259, New York, NY, USA, 1992. ACM Press.
49. I. Suzuki and T. Murata. A method for stepwise refinement and abstraction of petri nets. *J. Comput. Syst. Sci.*, 27(1):51–76, 1983.
50. A. Turjan. *Compiling Nested Loop Programs to Process Networks*. PhD thesis, Leiden University, 2007.
51. J. D. Ullman. Polynomial complete scheduling problems. *Operating Systems Review*, 7(4), 1973.
52. V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.
53. M. Westergaard. Verifying parallel algorithms and programs using coloured petri nets. *T. Petri Nets and Other Models of Concurrency*, 6:146–168, 2012.
54. J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.

Appendix

A A restricted CPN model for Programs on Integers

The restricted CPN model [8] used in this work is an eight tuple $N = \langle P, V, f_{pv}, T, I, O, inP, outP \rangle$, where the members are defined as follows. The set $P = \{p_1, p_2, \ldots, p_m\}$ is a finite non-empty set of places which are said to hold tokens. The set V is the set of variables of the program which N seeks to model. The mapping $f_{pv} : P \rightarrow V \cup \{\delta\}$ depicts an association of the places of N to the program variables V; the role of the co-domain element δ for f_{pv} is explained shortly. The variable $f_{pv}(p)$ is denoted as v_p in short; the token at the place p, if present, (synonymously, the variable v_p) assumes values from a domain D_p. Hence, for any two places $p, p', if\ f_{pv}(p) = f_{pv}(p')$ then $D_p = D'_p$. In this paper, we consider only integer type variables. The set $T = \{t_1, t_2, \ldots, t_n\}$ is a finite non-empty set of *transitions*; the relation $I \subseteq P \times T$ is a finite non-empty set of input edges which define the flow relation from places to transitions; a place p is said to be an *input place of a transition* t if $(p, t) \in I$. The relation $O \subseteq T \times P$ is a finite non-empty set of output edges which define the flow relation from transitions to places; a place p is said to be an *output place of a transition* t if $(t, p) \in O$. A place $p \in P$ is said to be an *in-port* if and only if $(t, p) \notin O$, for all $t \in T$. Likewise, a place $p \in P$ is said to be an *out-port* if and only if $(p, t) \notin I$, for all $t \in T$. The set $inP \subseteq P$ is the non-empty set of in-ports and the set $outP \subseteq P$ is the non-empty set of out-ports. The *pre-places* $^\circ t$ *of a transition* $t \in T$ is the non-empty set of *input places* of t. Thus, $^\circ t = \{p \in P \mid (p, t) \in I\}$. Similarly, the *the set of post-places* t° *of a transition* $t \in T$ is the non-empty set of *output places* of t. So, $t^\circ = \{p \in P \mid (t, p) \in O\}$; a place p_1 is said to be a co-place of a place p_2 if $p_1, p_2 \in t^\circ$ for any transition t. For any set T of transitions, $^\circ T \left(= \bigcup_{t \in T} {}^\circ t \right)$ represents all the pre-places of the transitions in T. Similarly, for any set T of transitions, $T^\circ \left(= \bigcup_{t \in T} t^\circ \right)$ represents all the post-places of the transitions in T. The pre-transitions $^\circ p$ and the post-transitions p° of a place $p \in P$ are given by $^\circ p = \{t \in T \mid (t, p) \in O\}$ and $p^\circ = \{t \in T \mid (p, t) \in I\}$, respectively. For any place p, either $^\circ p$ or p° or both are non-empty; for any set P of places, $^\circ P \left(= \bigcup_{p \in P} {}^\circ p \right)$ represents all the pre-transitions of the places in P. Similarly, for any set P of places, $P^\circ \left(= \bigcup_{p \in P} p^\circ \right)$ represents all the post-transitions of the places in P. If the post-places t° of a transition t contains n places, then all these places are associated with identical token type and token value and therefore, the domain of all the post-places of a transition are identical; this property is consistent with the firing rules of Petri net transitions.

A transition t is associated with a *guard condition* $g_t : D_{p_1} \times D_{p_2} \times \ldots \times D_{p_{n_t}} \rightarrow \{\top, \bot\}$ and a *function* $f_t : D_{p_1} \times D_{p_2} \times \ldots \times D_{p_{n_t}} \rightarrow D$, where $^\circ t = \{p_1, p_2, \ldots, p_{n_t}\}$ and $D = D_{p'_1} = D_{p'_2} = \ldots = D_{p'_{m_t}}$ where $t^\circ = \{p'_1, p'_2, \ldots, p'_{m_t}\}$. The guard condition g_t specifies the condition that must hold over the token values in $^\circ t$ for the transition t to be executed. The function f_t captures the functional transformation that takes place on the token values in $^\circ t$ to produce the *same token value* at all the places in t°. For example, for an assignment statement of a high level language of the form $x := y + (c/d) * 4$, the transition t will have $^\circ t = \{p_1, p_2, p_3\}$, $t^\circ = \{p\}$ and f_t will be maintained as $v_{p_1} + (v_{p_2}/v_{p_3}) * 4$, where v_{p_1} is y, v_{p_2} is c, v_{p_3} is d and v_p is x. A transition t corresponding to an initialization operation of some variable(s) with a constant value has the associated function f_t as the corresponding constant function with a pre-place associated with δ; some places are also introduced as synchronizing pre-places of certain transitions to ensure that all the operations of an iteration is completed before the next iteration starts; these synchronizing places are also associated with δ; all places associated with δ are called dummy places. For any transition t, let n be the arity of f_t; then, $X_{f_t}(f_{pv}(^\circ t))$ represents the set of variables associated with the places of $^\circ t$, ordered (as an n-tuple) in conformity with the domain of f_t. Hence when an enabled transition occurs, the expression $f_t(X_{f_t}(f_{pv}(^\circ t)))$ symbolically represents the value produced at the post-place(s) t^o in terms of the symbolic value tuples present at the pre-place(s) $^\circ t$. To avoid notational heaviness, with a little abuse of notation, we use the expression $f_t(f_{pv}(^\circ t))$ in place of $f_t(X_{f_t}(f_{pv}(^\circ t)))$. Similarly, the expression $g_t(f_{pv}(^\circ t))$ is used in place of $g_t(X_{g_t}(f_{pv}(^\circ t)))$, where X_{g_t} is interpreted (as a Cartesian product) in an identical way as X_{f_t}.

The model is deterministic denoted symbolically by the following conditions:

1. The CPN model is one-safe, i.e., at any point, a place may hold at most one token [41].

2. For any place p, for any two transitions $t_i, t_j \in p^\circ$, $g_{t_i}(f_{pv}({}^\circ t_i)) \wedge g_{t_j}(f_{pv}({}^\circ t_j)) \equiv \bot(false)$, where $f_{pv}({}^\circ t)$, for any transition t, indicates the image of the set ${}^\circ t$ of places under f_{pv}; likewise for $f_{pv}(t^\circ)$. Thus, two transitions having some common pre-places cannot fire simultaneously.
3. Also the model is completely specified denoted symbolically by the following condition: $\bigvee_{t \in p^\circ} g_t(f_{pv}({}^\circ t)) \equiv \top(true)$.

It is to be noted that the transitions may also have delay and deadline time parameters; models having these features are called timed CPN models. We deal with only untimed restricted CPN models. Henceforth, by a CPN model we only mean a *one-safe, untimed, deterministic, completely specified* restricted CPN model. The relationship between the original CPN model [26] and the restricted CPN model used in this work is reported in [8].

B Computations in a CPN model

A *marking* M is an ordered 2-tuple $\langle P_M, val_M \rangle$, where P_M is a subset of places and val_M a mapping from places to token values; P_M, referred to as place marking of M, designates the set of places where tokens are present for the marking M; the values of these tokens are captured by the second member val_M which is a function defined as follows. Let $D_M = \sqcup_{p \in P_M} D_p$, the disjoint union of the family of sets $D_p, p \in P_M$. The function $val_M : P_M \rightarrow D_M$ maps a place $p \in P_M$ to a value in the domain D_p of that place. The function val_M is consistent with the mapping f_{pv}, that is, $\forall p_1, p_2 \in P_M$, if $f_{pv}(p_1) = f_{pv}(p_2)$, then $val_M(p_1) = val_M(p_2)$. The symbol $val_M(P')$ denotes the values of places in $P' \subseteq P_M$. A marking M_0 is an initial marking with $P_{M_0} = inP$.

A transition $t \in T$ is said to be *bound* for a given marking $M : \langle P_M, val_M \rangle$ if all its input places are marked, i.e., ${}^\circ t \subseteq P_M$. A bound transition $t \in T$ for a given marking M is said to be *enabled* if $g_t(val_M({}^\circ t) \equiv T$, where $val_M({}^\circ t)$ are the values of the tokens present at the pre-places ${}^\circ t$. The set of enabled transitions for a marking M is denoted as T_M. For any $t_1, t_2 \in T_M$, $f_{pv}(t_1^\circ) \cap f_{pv}(t_2^\circ) = \emptyset$ and $f_{pv}({}^\circ t_1) \cap f_{pv}(t_2^\circ) = \emptyset$ ensuring that among the enabled transitions there are no shared variables with write after write (WAW) and read after write (RAW) dependences, respectively. For untimed CPN models, all the enabled transitions are fired simultaneously provided they satisfy the above (freedom of) dependency requirements. It may be noted that conventionally, only one of the enabled transitions fire non-deterministically (thereby possibly disabling some of the erstwhile enabled transitions); however, under the above assumption of absence of shared variables with RAW or WAW dependences over any sequence of enables transitions, the resulting data values of the variables remain independent of any interleaving of the transitions over the entire sequence. Hence, in this work, the computation semantics of the model is defined based on *simultaneous firing of all the enabled transitions*. After firing of all enabled transitions from a given marking M, the successor marking M^+ of M is obtained. The definition of successor marking is as follows:

Definition 4 (Successor marking of a marking [8]) A marking $M^+ = \langle P_{M^+}, val_{M^+} \rangle$ is said to be a successor of the marking $M = \langle P_M, val_M \rangle$, if

1. the first component P_{M^+}, referred to as the successor place marking of P_M, contains all the post-places of the enabled transitions of P_M and also all the places of P_M whose post-transitions are not enabled; symbolically, $P_{M^+} = \{p \mid p \in t^\circ \text{ and } t \in T_M\} \cup \{p \mid p \in P_M \text{ and } p \notin {}^\circ T_M\}$,
   and
2. $\forall p \in P_{M^+}$, if $p = t^\circ$ for some $t \in T_M$, then $val_{M^+}(p) = f_t(val_M({}^\circ t))$ and if $p \notin {}^\circ T_M$, then $val_{M^+}(p) = val_M(p)$.

For formalizing the definition of computation of a CPN model, we need the following definitions.

Definition 5 (Back edge [9]) An edge $\langle t, p \rangle$ from a transition t to a place $p \in t^\circ$ is said to be a back edge with respect to an arbitrary DFS traversal of the CPN model, if p is an ancestor of t in that traversal.

In the sequel, all references to "back edge" involve the same traversal.

Definition 6 (Successor relation between two transitions [9]) A transition t_i succeeds a transition t_j, denoted as $t_i \succ t_j$, if $\exists t_{k_1}, t_{k_2}, \ldots, t_{k_n} \in T$, and $p_1, p_2, \ldots p_{n+1} \in P, n \geq 1$ such that

1. $\langle t_j, p_1\rangle, \langle t_{k_1}, p_2\rangle, \ldots, \langle t_{k_{n-1}}, p_n\rangle, \langle t_{k_n}, p_{n+1}\rangle \in O \subseteq T \times P$,
2. $\langle p_1, t_{k_1}\rangle, \langle p_2, t_{k_2}\rangle, \ldots, \langle p_n, t_{k_n}\rangle, \langle p_{n+1}, t_i\rangle \in I \subseteq P \times T$ and
3. none of $\langle t_j, p_1\rangle$ or $\langle t_{k_m}, p_{m+1}\rangle, 1 \leq m \leq n$, is a back edge.

The expression $t_i \not\succ t_j$ is used as a shorthand for $\neg(t_i \succ t_j)$. The transition t_i is said to "succeed or be same as" the transition t_k, denoted as $t_i \succeq t_k$, if and only if t_i succeeds t_k or t_i is the same as t_k.

The following definition captures transitions which are independent of each other and accordingly can occur (if simultaneously enabled by a marking) in any arbitrary order creating the same final effect when all the transitions have completed their execution; in other words, the effects remain the same if they execute in parallel. In [26] such transitions have been referred to as "concurrently enabled transitions" informally described as transitions which can occur for a given marking without sharing of tokens in their input places (pre-places).

Definition 7 (Set of parallelizable transitions [8]) Two transitions t_i and t_j are said to be parallelizable, denoted as $t_i \asymp t_j$, if (*i*) $t_i \not\succ t_j, t_j \not\succ t_i$ and (*ii*) $\forall t_k, t_l (t_k \neq t_l \wedge t_i \geq t_k \wedge t_j \geq t_l \rightarrow {}^\circ t_k \cap {}^\circ t_l = \emptyset)$. A set $T = \{t_1, t_2, \ldots, t_k\}$ of transitions is said to be parallelizable if $\forall t_i, t_j \in T, t_i \neq t_j \rightarrow t_i \asymp t_j$ holds.

For a (possibly infinite) set of initial markings, sets of parallelizable transitions occur in a sequence resulting in a computation of the model. Two computations may differ from each other by having some sequences of the subsets of parallelizable transitions being repeated different number of times in them. In the sequel, for static analysis leading to equivalence checking we shall find it convenient to visualize a computation as a sequence of parallel paths each of which would contain sequence of non-intersecting subsets of parallelizable transitions occurring in the computation. This would need the notions of parallelizable sets of parallelizable transitions (to capture parallel paths), maximally parallelizable sets of parallelizable transitions (for the models) and their sequences conforming to a successor relation among such sets; these notions are formalized by the following definitions.

Definition 8 (Parallelizable sets of parallelizable transitions [8]) Let $T_1, T_2, \ldots, T_k$ be k sets of parallelizable transitions. They are said to be parallelizable if $\bigcup_{i=1}^{k} T_i$ is a set of parallelizable transitions.

Note that in the above definition, each transition $T_i, 1 \leq i \leq k$, is parallelizable but not maximally parallelizable as it is contained in a larger set $\bigcup_{i=1}^{k} T_i$. Hence we have the following definition.

Definition 9 (Set of maximally parallelizable transitions [8]) A set T is said to be maximally parallelizable if there is no set T' of parallelizable transitions which contains T.

Definition 10 (Succeed Relation over Parallelizable Transitions [8]) Given two sets of parallelizable transitions T_1 and T_2, T_1 is said to succeed T_2, denoted as $T_1 \succ T_2$, if $\exists t_1 \in T_1$ and $\exists t_2 \in T_2$ such that $t_1 \succ t_2$.

Definition 11 (Computation in a CPN model [8]) In a CPN model N with in-port inP, a computation $\mu_{N,p}$ of an out-port p is a sequence $\langle T_1, T_2, \ldots, T_i, \ldots, T_l\rangle$ of sets of maximally parallelizable transitions satisfying the following properties:

1. There exists a sequence of markings of places $\langle P_{M_0}, P_{M_1}, \ldots, P_{M_{l-1}}\rangle$ such that
   (a) $P_{M_0} \subseteq inP$,
   (b) $\forall i, 1 \leq i < l$, P_{M_i} is a successor place marking of $P_{M_{i-1}}$, ${}^\circ T_i \subseteq P_{M_{i-1}}$ and $T_i^\circ \subseteq P_{M_i}$.
2. $p \in T_l^\circ$.

We can represent the computation alternatively as a sequence of place markings $\langle {}^\circ T_1, T_1^\circ \cup {}^\circ T_2, T_2^\circ \cup {}^\circ T_3, \ldots, \{p\} \subseteq T_l^\circ\rangle$; thus, in this alternative representation, a computation is a sequence of markings of places which

1. starts with the pre-places of the first set of transitions,
2. ends with the unit set $\{p\} \subseteq T_l^\circ$, the post-places of the last (unit set of) transition and
3. the remaining members in the sequence are the union of the post-places of a set of transitions with the pre-places of its next set of transitions.

If there are k out-ports, then for each initial marking M_0, there are at most k computations, one for each out-port. (We drop the subscripts of μ when they are clear from the context. Thus, more specifically, when there is no other CPN model we use the symbol μ_p.)

For many valuations of inP (and consequently of ${}^\circ T_1$), the same sequence given by μ_p can result. In general, therefore, a particular sequence μ_p may represent more than one *computation trace* where a trace of the computation μ_p is formally denoted as an ordered pair $\langle \mu_p, val({}^\circ T_1)\rangle$.

Recall that in the context of the example of Section 3 it has been identified that the set of all computation traces of an out-port p can be partitioned into a finite collection of subsets where a subset having a computation μ_p is characterized by a condition of execution, designated as R_{μ_p}, and a data transformation function, designated as r_{μ_p}, both over the set of the input variables (associated with inP).

C Paths of a CPN Model

Definition 12 (Static cut-point (SCP)[9]) A place p of a model is designated as a static cut-point with respect to an arbitrary DFS traversal of the model graph starting from some in-port and covering all the in-ports if (*i*) p is an in-port, or (*ii*) p is an out-port or (*iii*) there is an edge $\langle t, p\rangle$ which is a back edge with respect to that DFS traversal.

Definition 13 (Paths in a CPN model) A finite path α in a CPN model from a set T_1 of transitions to a transition t_j is a finite sequence of distinct sets of parallelizable transitions of the form $\langle T_1 = \{t_1, t_2, \ldots, t_k\}, T_2 = \{t_{k+1}, t_{k+2}, \ldots, t_{k+l}\}, \ldots, T_n = \{t_j\}\rangle$ satisfying the following properties:

1. ${}^\circ T_1$ contains at least one cut-point or one co-place of a cut-point.
2. T_n° contains at least one cut-point.
3. There is no cut-point in T_m°, $1 \leq m < n$.
4. $\forall i, 1 < i \leq n, \forall p \in {}^\circ T_i$, if p is neither a cut-point nor a co-place of a cut-point, then $\exists k, 1 \leq k \leq i-1, p \in T_{i-k}^\circ$; thus, any pre-place of a transition set in the path which is neither a cut-point nor a co-place of a cut-point must be a post-place of some preceding transition set in the path.
5. There do not exist two transitions t_i and t_l in α such that ${}^\circ t_i \cap {}^\circ t_l \neq \emptyset$. Thus, no two transitions in a path can have a common pre-place.
6. $\forall i, 1 \leq i \leq n$, T_i is maximally parallelizable within the path, i.e., $\forall l \neq i, \forall t \in T_l$ in the path, $T_i \cup \{t\}$ is not parallelizable.

The fact that every set T_i succeeds all the transitions T_1 through T_{i-1} follows from clauses 4 and 6 of the above definition; otherwise, if there exists some set T_{i-m}, $m \geq 1$, such that T_i does not succeed T_{i-m}, then T_i can be parallelized with T_{i-m} but clause 6 indicates that both T_{i-m} and T_i are maximally parallelizable within the path. The set ${}^\circ T_1$ of places is called the pre-places of the path α, denoted as ${}^\circ\alpha$; similarly, the post-places α° of the path α is T_n°. We can synonymously denote a path $\alpha = \langle T_1, T_2, \ldots, T_n\rangle$ as the sequence $\langle {}^\circ T_1, {}^\circ T_2, \ldots, {}^\circ T_n, T_n^\circ\rangle$ of the sets of places from the place(s) ${}^\circ T_1$ to the place(s) T_n°.

A path is obtained by constructing an inverted cone having as its vertex a cut-point which is either an out-port or a place with a back edge. The construction proceeds by traversing backward from the vertex encompassing sets of parallel transitions in a reverse sequence till the most recently included set have pre-places which are all cut-points; it is also ensured that no cut-points or co-places of a cut-point are included. The detailed algorithm and its correctness and complexity issues are available in [7].

Similar to the *succeeds* relation $\succ$ over the set of transitions, we can define *succeeds* relation (denoted again as $\succ$) over the set of paths and also we can define parallelizable pairs of paths and set of parallelizable paths.

Definition 14 (Concatenation of paths[8]) A path α' can be concatenated with a set $Q_p = \{\alpha_1, \ldots, \alpha_k\}$ of parallelizable paths if $\forall i, 1 \leq i \leq k, \alpha_i^\circ \subseteq {}^\circ\alpha'$. Such a concatenation of paths α, say, is denoted as $(\alpha_1 \| \ldots \| \alpha_k).\alpha'$.

D Model Computations as Concatenations of Paths

Proofs of Theorems 1 and 2 of subsection 3.4 (reproduced here for ready reference as Theorems 4 and 5, respectively) are as follows.

Theorem 4 *Let $\alpha_1 = \langle T_{1,1}, T_{2,1}, \ldots, T_{m,1}\rangle$ and $\alpha_2 = \langle T_{1,2}, T_{2,2}, \ldots, T_{n,2}\rangle$ be two paths such that their last sets of transitions $T_{m,1}$ and $T_{n,2}$ are parallelizable. Then, α_1 and α_2 are parallelizable.*

Proof Let it not be so. From Definition of parallelizable pairs of paths [8], we have the following cases:

- Case 1: $\alpha_1 \succ \alpha_2$. From Definition of successor relation between two paths [8], there exist at least one set of paths $\alpha_{k_1}, \alpha_{k_2}, \ldots, \alpha_{k_n}$ and a set of places $p_1 \in {}^\circ\alpha_1$ and $p_{k_m} \in {}^\circ\alpha_{k_m}, 1 \leq m \leq n$, such that $\langle$ last(α_2), $p_{k_1}\rangle$, $\langle$ last(α_{k_1}), $p_{k_2}\rangle, \ldots, \langle$ last(α_{k_n}), $p_1\rangle \in O \subseteq T \times P$, $n \geq 0$, and none of them is a back edge. Therefore, using the fact that last(α) $\succ$ first(α), for any path α, and reading the above sequence of edges backward, we have last(α_1)=$T_{m,1} \succ$ first(α_1) $\succ$ last(α_{k_n}) $\succ$ first(α_{k_n}) $\succ$ last($\alpha_{k_{n-1}}$) $\succ \ldots \succ$ first(α_{k_2}) $\succ$ last(α_{k_1}) $\succ$ first(α_{k_1}) $\succ$ last(α_2) = $T_{n,2}$. Hence, $T_{m,1} \succ T_{n,2} \Rightarrow T_{n,2} \nsucc T_{m,1}$ (Contradiction).
- Case 2: $\alpha_2 \succ \alpha_1$. Following the same argument, as for Case 1, by symmetry with α_1 and α_2 interchanged, we again obtain the refutation of the hypothesis $T_{m,1} \asymp T_{n,2}$.
- Case 3: $\exists \alpha_k, \alpha_l, (\alpha_k \neq \alpha_l \wedge \alpha_1 \succeq \alpha_k \wedge \alpha_2 \succeq \alpha_l \wedge {}^\circ\alpha_k \cap {}^\circ\alpha_l \neq \emptyset)$. ${}^\circ\alpha_k \cap {}^\circ\alpha_l \neq \emptyset \Rightarrow \exists t_{i,k} \in \alpha_k, \exists t_{j,l} \in \alpha_l$ such that ${}^\circ t_{i,k} \cap {}^\circ t_{j,l} \neq \emptyset$. Let the last transitions of the paths α_k and α_l be $t_{r,k}$ and $t_{s,l}$, respectively. Since $\alpha_1 \succeq \alpha_k$, $T_{m,1} \succeq t_{r,k} \succeq t_{i,k}$; (recall that $T_{m,1} = \{t_{m,1}\}$). Similarly, since $\alpha_2 \succeq \alpha_l$, $T_{n,2} \succeq t_{s,l} \succeq t_{j,l}$. Thus, $T_{m,1} \succeq t_{i,k}$, $T_{n,2} \succeq t_{j,l}$ and ${}^\circ t_{i,k} \cap {}^\circ t_{j,l} \neq \emptyset$. Therefore, $T_{m,1}$ is not parallelizable with $T_{n,2}$ (from Definition 7). □

Theorem 5 *Let Π be the set of all paths of a CPN model obtained from a set of static cut-points. For any computation μ_p of an out-port p of the model, there exists a reorganized sequence μ_p^r of paths of Π such that $\mu_p \simeq \mu_p^r$.*

Proof **Construction of a sequence μ_p^r of (concatenation of) paths from μ_p:**

Algorithm 1 (`constructPathSequence`) describes a recursive function for constructing from a given computation μ_p and a set Π of paths the desired reorganized sequence μ_p^r of paths of Π such that $\mu_p^r \simeq \mu_p$. If μ_p is not empty, then a path α is selected from Π such that last(α)$\cap$ last(μ_p) $\neq \emptyset$; if all its transitions are found to occur in μ_p, then it is put as the last member in the reorganized sequence; the member transitions of α are deleted from μ_p examining the latter backward; the transitions in the last member of α are always deleted from μ_p; each of the other transitions of α is deleted from μ_p only if it does not occur in any other path in Π. If $|\text{last}(\mu_P)| > 1$, then each member transition of last(μ_p) will result in one path which has to be processed separately through above steps. Once all these paths are processed, the last(μ_p) will get deleted from μ_p. The resulting μ_p is then reordered recursively; the process terminates when the input μ_p becomes empty.

Proof ($\mu_p^r \simeq \mu_p$): We first prove that Algorithm 1 terminates; this is accomplished in two steps; first, it is shown that each invocation comprising four loops terminate; we next show that there are only finitely many recursive invocations.

Termination of the while loop comprising lines 16-18 is obvious; either i becomes less than one or a member $\mu_p.T_i$ is found to contain last(α') (for some $i > 1$). The for loop comprising lines 22-26 iterates only finitely many times because the number of transitions in any member set of a path (and hence $\mu_p'.T_i$) is finite; the while loop comprising lines 15-29 terminates, because in every iteration, it is examined whether the computation μ_p' contains the last member of α'; if so, α' loses this member in line 27 and the next iteration of the loop executes with α' having one member less. Finally, the for loop comprising lines 10-35 terminates because the set last(μ_p) of transitions (before entering the loop), and hence the set $\Pi_{last(\mu_p)}$ of paths are finite.

The second step follows from the fact that in each recursive invocation, μ_p has one member (namely, its last member) less than the previous invocation (line 40 in the if statement comprising lines 37-41). Hence, if μ_p has n members, then there are n total invocations ($n - 1$ of them being recursive).

Now, for proving $\mu_p^r \simeq \mu_p$, let the first parameter μ_p for the k^{th} invocation be designated as $\mu_p^{(k)}$, $1 \leq k \leq n$; the second parameter Π remains the same for all invocations; let the value returned by the k^{th} invocation be $\mu_p^{r(k)}$; specifically, $\mu_p = \mu_p^{(1)}$; $\mu_p^{(n-1)}$ comprises just one member and $\mu_p^{(n)} = \langle\rangle$; $\mu_p^{r(n)} = \langle\rangle$ and $\mu_p^{r(1)}$ is the final reordered sequence of paths μ_p^r.

We prove $\mu_p^{(n-m)} \simeq \mu_p^{r(n-m)}, 0 \leq m \leq n-1$, by induction on m. Note that specifically for $m = n-1$, $\mu_p^{(n-m)} = \mu_p^{(1)} = \mu_p$ and $\mu_p^{r(n-m)} = \mu_p^{r(1)} = \mu_p^r$ (by line 41 of the first invocation). Hence, the inductive proof would help us establish that $\mu_p^r \simeq \mu_p$.

Basis $m = 0$: $\mu_p^{(n)} = \langle\rangle = \mu_p^{r(n)}$ (by line 2 of the n^{th} invocation)
Induction Hypothesis: Let for $m = k-1$, $\mu_p^{(n-k+1)} \simeq \mu_p^{r(n-k+1)}$
Induction step: Let m be k. Let us assume that
$\mu_p^{(n-k)} \simeq \mu_p^{(n-k+1)}.\mu_l^{r(n-k)}$ (Lemma 1 – proved subsequently)
$\simeq \mu_p^{r(n-k+1)}.\mu_l^{r(n-k)}$ (by induction hypothesis)
$\simeq \mu_p^{r(n-k)}$ (by line 40 (return statement) for the $(n-k)^{th}$ invocation) □

Lemma 1 $\mu_p^{(n-k)} \simeq \mu_p^{(n-k+1)}.\mu_l^{r(n-k)}$

Proof We mould the lemma for the k^{th} invocation directly as

$$\mu_p^{(k)} \simeq \mu_p^{(k+1)}.\mu_l^{r(k)} \simeq \mu_p^{(k+1)}.\langle\alpha_{1,k}, \alpha_{2,k}, \ldots, \alpha_{s,k}\rangle, 1 \leq k \leq n, \tag{3}$$

assuming that $\langle\alpha_{1,k}, \alpha_{2,k}, \ldots, \alpha_{s,k}\rangle$ is what is extracted as $\mu_l^{r(k)}$ from $\mu_p^{(k)}$ in line 40 of the *kth* iteration. Now, by step 6, the last transition of all the paths in the sequence $\mu_l^{r(k)}$ are parallelizable; hence, from Theorem 4, the paths of $\mu_l^{r(k)}$ are parallelizable. We prove that their transitions can be suitably placed in the member sets of $\mu_p^{(k+1)}$ (as larger sets of parallelizable transitions) to get back $\mu_p^{(k)}$.

In the k^{th} invocation, $\mu_p^{(k)}$ is the value of μ_p before entry to the for-loop comprising lines 10-35 and $\mu_p^{(k+1)}$ is the value of μ_p at the exit of this loop. Since we are speaking about only the k^{th} invocation, we drop the superfix k for clarity. Instead, we depict $\mu_p^{(k)}$ as μ_p^-, $\mu_p^{(k+1)}$ as μ_p^+ and $\mu_l^{r(k)}$ as μ_l^r. So we have to prove that $\mu_p^- \simeq \mu_p^+.\mu_l^r$.

Let $\mu_l^r = \langle\alpha_1, \alpha_2, \ldots, \alpha_s\rangle$ before line 37 just after the end of the for-loop comprising lines $10-35$, where s is the cardinality $|\Pi_{last}(\mu_p)|$ before entry to the loop (because any path has only one unit set of transitions as its last member). Thus, the for-loop comprising lines $10-35$ executes s times visiting step 33; let $\mu_p^{-(i)}$, $\mu_p^{+(i)}$ respectively denote the values of μ_p before and after the i^{th} iteration of the loop. Let $\mu_l^{r(i)}$ be the value of μ_l^r after the i^{th} execution of the loop. We have the following boundary conditions: $\mu_p^- = \mu_p^{-(1)}$, $\mu_p^+ = \mu_p^{+(s)}$, $\mu_l^{r(1)} = \langle\alpha_1\rangle$ and $\mu_l^r = \mu_l^{r(s)} = \langle\alpha_1, \alpha_2, \ldots, \alpha_s\rangle$. The i^{th} iteration of the for-loop comprising lines $10-35$ starts with $\mu_l^{r(i-1)}$ and obtains $\mu_l^{r(i)}$, $1 \leq i \leq l$; so let $\mu_l^{r(0)} = \langle\rangle$ be the value of μ_l^r with which the first execution of the loop takes place. We prove that $\mu_p^{-(i)} \simeq \mu_p^{+(i)}.\alpha_i, 1 \leq i \leq s$. If this relation indeed holds, then specifically for $i = 1$, $\mu_p^{-(1)} \simeq \mu_p^{+(1)}.\alpha_1$; for $i = 2$, $\mu_p^{-(2)}(= \mu_p^{+(1)}) \simeq \mu_p^{+(2)}.\alpha_2$. Combining these two, therefore, $\mu_p^- = \mu_p^{-(1)} \simeq \mu_p^{+(1)}.\alpha_1 \simeq (\mu_p^{+(2)}.\alpha_2).\alpha_1 \simeq \mu_p^{+(2)}.(\alpha_2.\alpha_1) \simeq \mu_p^{+(2)}.(\alpha_1.\alpha_2) \simeq \mu_p^{+(2)}.\mu_l^{r(2)}$. Proceeding this way, we have $\mu_p^- = \mu_p^{-(1)} \simeq \ldots \simeq \mu_p^{+(l)}.\mu_l^{r(l)} = \mu_p^+.\mu_l^r$. Now, let $\mu_p^{-(i)} = \langle T_{1,i}, T_{2,i}, \ldots, T_{k_i,i}\rangle$, $\alpha_i = \langle T'_{1,i}, T'_{2,i}, \ldots, T'_{l_i,i}\rangle$ and $\mu_p^{+(i)} = \langle T^+_{1,i}, T^+_{2,i}, \ldots, T^+_{n,i}\rangle$. Note that $\{\alpha_i \mid 1 \leq i \leq s\} \subseteq \Pi_{last(\mu_p)}$ and unless all the paths are extracted out, $T_{k_i,i}$ does not become empty and hence $\mu_p^{-(i)}, 1 \leq i \leq s$, do not change in length. For each transition set $T'_{j,i}$ of α_i, $1 \leq j \leq n$, there exists some transition set $T_{k,i}$ of $\mu_p^{-(i)}$, $1 \leq k \leq k_i$, such that $T'_{j,i} \subseteq T_{k,i}$. Specifically, for $j = l_i$, $T'_{l_i,i} \subseteq T_{k_i,i}$, since $\alpha_i \in \Pi_{last(\mu_p^-)}$ as ensured in step 6. For other values of $j, 1 \leq j < l_i$, the while-loop in steps 16-18, identifies proper $T_{k,i}$ in $\mu_p^{-(i)}$ such that $T'_{j,i} \subseteq T_{k,i}$; note that since α_i has figured in μ_l^r, step 32 is surely executed for

α_i; so α' has been rendered empty ($\langle\rangle$) through execution of step 27 and hence the while-loop in steps 16-18 does not exit with $i = 0$. Now, step 13 and the for-loop in steps 22-26 ensure that $T^+_{k,i} \cup T_{j,i} = T_{k,i}$.

Let $T'_{j,i} \subseteq T_{n_j,i}, 1 \leq j \leq l_i$. So, $T_{k,i} = T^+_{k,i}$, for $k \neq n_j$, for any $j, 1 \leq j \leq l_i$.

$\mu^+_p(i).\alpha_i = \langle T^+_{1,i}, T^+_{2,i}, \ldots, T^+_{n,i}\rangle.\langle T'_{1,i}, T'_{2,i}, \ldots, T'_{l_i,i}\rangle$

$= \langle T_{1,i}, \ldots, (T^+_{n_1,i} \parallel T'_{1,i}), \ldots (T^+_{n_2,i} \parallel T'_{2,i}), \ldots, (T^+_{n_{l_i-1},i} \parallel T'_{l_i,i}), \ldots, (T^+_{n_i,i} \parallel T'_{l_i,i})\rangle$ (by commutativity of independent transitions)

$= \langle T_{1,i}, \ldots, T_{n_1,i}, \ldots, T_{n_2,i}, \ldots, T_{n_{l_i-1},i}, \ldots T_{n,i}\rangle$

$= \mu^{-(i)}_p$. □

□

Corollary 1 *If μ^r_p is of the form $\langle\alpha_1, \alpha_2, \ldots, \alpha_k\rangle$, then for all $j, 1 \leq j \leq i-1, \alpha_j \not\asymp \alpha_i$.*

E Validity of Path Based Equivalence Checking

Before describing our path based equivalence checking method, we first prove the validity of any path based equivalence checking method. Proof of Theorems 3 of subsection 3.4 (reproduced here for ready reference as Theorem 6) is as follows.

Theorem 6 *For any two models N_0 and N_1, if there exists a finite path cover $\Pi_0 = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ of N_0 for which there exists a set $\Psi_1 = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_m\}$ of sets of paths of N_1 such that for all $i, 1 \leq i \leq m$, $\alpha_i \simeq \beta$, for all $\beta \in \Gamma_i$, then N_0 is contained in N_1 ($N_0 \sqsubseteq N_1$).*

Proof Let Π_0 be the set of SCP induced paths of N_0. From Theorem 5, Π_0 is a path cover of N_0. Consider any computation $\mu_{0,p}$ of an out-port p of N_0. From Theorem 5, corresponding to $\mu_{0,p}$, there exists a reorganized sequence $\mu^r_{0,p} = \langle\alpha^p_1, \alpha^p_2, \ldots, \alpha^p_n\rangle$, say, of not necessarily distinct paths of N_0 such that (i) $\alpha^p_j \in \Pi_0$, $1 \leq j \leq n$, (ii) for each occurrence of a transition t in $\mu_{0,p}$, there exists exactly one path in $\mu^r_{0,p}$ containing that occurrence, (iii) $p \in (\alpha^p_n)^\circ$ and (*iv*) $\mu_{0,p} \simeq \mu^r_{0,p}$.

Let us now construct from the sequence $\mu^r_{0,p}$, a sequence $\mu^r_{1,p'} = \langle\Gamma^{p'}_1, \Gamma^{p'}_2, \ldots, \Gamma^{p'}_n\rangle$ of not necessarily distinct sets of paths of N_1, where $\Gamma^{p'}_n = \{\beta_l\}$ and $p' \in \beta^\circ_l$, and for all $j, 1 \leq j \leq n$, for each $\beta \in \Gamma^{p'}_j$, $\beta \simeq \alpha^p_j$. It is required to prove that (1) $p' = f_{out}(p)$ and (2) there exists a computation $\mu_{1,p'}$ of N_1 such that $\mu_{1,p'} \simeq \mu^r_{1,p'}$.

The proof of (1) is as follows. Since $p' \in \beta^\circ_l$ and $\beta_l \simeq \alpha_n$, from hypothesis (*iii*) of the theorem, p' has correspondence with p; since the place $p \in P_0$ is an out-port and the place $p' \in P_1$, p' must be an out-port of N_1 and $p' = f_{out}(p)$ (because an out-port of N_0 has correspondence with exactly one out-port of N_1 specifically, its image under the bijection f_{out}).

For the proof of (2), we first give a mechanical construction of $\mu_{1,p'}$ from $\mu^r_{1,p'}$; we then show that they are equivalent; finally, we argue that $\mu_{1,p'}$ is a computation of p' in N_1.

Construction of $\mu_{1,p'}$ from $\mu^r_{1,p'}$:

Algorithm 2 describes the construction method of $\mu_{1,p'}$ from $\mu^r_{1,p'}$ (and hence will be invoked with its input μ^r_p instantiated with $\mu^r_{1,p'}$). The parallelized version of the input μ^r_p is computed in $\mu_\parallel$ which is to be assigned to $\mu_{1,p'}$ on return. In the initialization step (step 1), a working set Γ of paths is initialized to the first member of μ^r_p and the latter is removed from μ^r_p. In step 2, some path β is taken from Γ and put into $\mu_\parallel$. In the outermost while-loop (steps 3-26), member sets of Γ are taken one by one (in steps 4-6) from μ^r_p; for each chosen set, its member paths are taken in the loop comprising steps 7-25; for each chosen path β, its member sets (of maximally parallelizable transitions) are examined one after another and checked against the members of $\mu_\parallel$ from the beginning for fusion with them to construct larger sets of parallelizable transitions (steps $9-24$). For each chosen set T_p of transitions of β, one of the following two situations may arise:

Algorithm 1 SEQUENCE **constructPathSequence** (μ_p, Π)

Inputs: μ_p: computation of an out-port p and Π: set of paths
Outputs: A sequence of paths equivalent to μ_p.

1: **if** $\mu_p = \langle\rangle$ **then**
2: **return** $\langle\rangle$;
3: **else**
4: Let μ_p be $\langle T_1, T_2, \ldots, T_i, \ldots T_n\rangle$;
5: Let $\mu_l^r = \langle\rangle$;
/* a local sub-sequence of paths which, at the `return` statement 40, contains the sequence of paths with their last transitions in T_n */
6: Let $\Pi_{last(\mu_p)} = \{\alpha \mid last(\alpha) \cap last(\mu_p) \neq \emptyset\}$;
7: **if** $\Pi_{last(\mu_p)} = \emptyset$ **then**
8: $\mu_p.T_n = \emptyset$; //Ignore intermediary transitions of paths
9: **else**
10: **for** all $\alpha \in \Pi_{last(\mu_p)}$ **do**
11: $\alpha' = \alpha - last(\alpha)$;
12: $\mu_p' = \mu_p$; // work on a copy of μ_p
13: $\mu_p'.T_n = \mu_p'.T_n - last(\alpha)$;
/* Delete the last transition of α; if it occurs in any other paths (as an intermediary transition), then such a path has already been detected. Now detect whether all the remaining transitions of α are available in $\mu_p(\mu_p')$; as a transition is detected, it is deleted from μ_p' and the copy α' of α only if it does not occur in any other path in Π. If all the transitions of α do not occur in μ_p, (i.e., α' does not become empty), then α is ignored and the next path from $\Pi_{last(\mu_p)}$ is taken in the next iteration. */
14: $i \Leftarrow n - 1$; // detection of transitions proceeds backward
15: **while** $\alpha' \neq \langle\rangle$ **do**
16: **while** $(i \geq 1 \wedge \text{last}(\alpha') \notin \mu_p'.T_i = \emptyset)$ **do**
17: $i = i - 1$;
18: **end while**
19: **if** $i = 0$ **then**
20: break;
21: **else**
22: **for** all $t \in \text{last}(\alpha')$ **do**
23: **if** t does not occur in any path in $\Pi - \{\alpha\}$ **then**
24: $\mu_p'.T_i \Leftarrow \mu_p'.T_i - \{t\}$;
25: **end if**
26: **end for**
27: $\alpha' = \alpha' - \text{last}(\alpha') \cap \mu_p.T_i$; not
28: **end if**
29: **end while**
30: /* both $\alpha' \neq \langle\rangle$ and $\alpha' = \langle\rangle$ are possible */
31: **if** $\alpha' = \langle\rangle$ **then**
32: append (α, μ_l^r);
33: $\mu_p = \mu_p'$;
34: **end if**
35: **end for**
36: **end if**
37: **if** original member $\mu_p.T_n$ is not empty **then**
38: report failure with μ_p
39: **else**
40: **return** (**concatenate** (**constructPathSequence**(μ_p, Π), μ_l^r));
41: **end if**
42: **end if**

Algorithm 2 Sequence **parallelizeSeqSetsOfPaths** (μ_p^r)

Inputs: μ_p^r : a sequence of sets of paths
Outputs: $\mu_{\|}$: a sequence of maximally parallelizable sets of transitions of all the paths in μ_p^r.

1: Γ = head(μ_p^r); μ_p^r = tail(μ_p^r);
2: $\mu_{\|}$ = some path $\beta \in \Gamma$; $\Gamma = \Gamma - \{\beta\}$; // β chosen arbitrarily
3: **while** $\mu_p^r \neq \emptyset$ **do**
4: **if** $\Gamma \neq \emptyset$ **then**
5: $\Gamma = head(\mu_p^r)$; $\mu_p^r = tali(\mu_p^r)$; // except for the first iteration, if-condition holds
6: **end if**
7: **for** each $\beta \in \Gamma$ **do**
8: Let $c = 1$;
/* index to the members of $\mu_{\|} - c^{th}$ member is $\mu_{\|,c}$; for each path of μ_p^r, checking has to be from the first member of $\mu_{\|}$.*/
9: **while** $\beta \neq \emptyset$ **do**
10: $T_c = \mu_{\|,c}$;
11: T_p = head (β);
/* T_p is the maximally parallelizable set (member) of β presently being considered for fusion with T_c */
12: β = tail(β);
13: **while** $T_p \succ T_c \wedge c \leq$ length ($\mu_{\|}$) **do**
14: /* T_p succeeds T_c */
15: $c++$;
16: $T_c = \mu_{\|,c}$;
17: **end while**
18: **if** $c >$ length ($\mu_{\|}$) **then**
19: /* T_p is found to be parallelizable with none of the members of $\mu_{\|}$; so $T_p \succ T, \forall T \in \mu_{\|}$ concatenate all the members (including T_p) of β after $\mu_{\|}$ */
20: $\mu_{\|} \leftarrow$ concatenate ($\mu_{\|}, \beta$); $\beta = \emptyset$;
21: **else**
22: $\mu_{\|,c} = \mu_{\|,c} \cup T_p$; $c++$;
/* $T_c \asymp T_p$ or $T_c = T_p$ – absorb T_p in T_c */
23: **end if**
24: **end while**
25: **end for**
26: **end while**
27: **return** $\mu_{\|}$;

- Case 1: The member T_p of the chosen path β of μ_p^r is found to succeed all the members in $\mu_{\|}$, i.e., T_p is not parallelizable with any member of $\mu_{\|}$. In this case, all the remaining members (including T_p) of β is concatenated at the end of $\mu_{\|}$ [Steps 18-20] .
- Case 2: The member T_p of β is found not to succeed the c^{th} member $\mu_{\|,c}$ of $\mu_{\|}$, i.e., T_p is parallelizable with $\mu_{\|,c}$, as argued later. In this case, T_p is combined (through union) with $\mu_{\|,c}$; the successor transition sets of β need to be compared with only the subsequent members of $\mu_{\|}$, i.e., with $\mu_{\|,c+1}$ onwards [Step 22] .

Termination: The algorithm terminates because all the three while loops and the for-loop terminate as given below:

The outer loop (steps 3-26) terminates because μ_p^r is finite to start with; (step 1 outside the loop reduces its length by one;) step 5 inside the loop reduces its length by one on every iteration of the loop. The for-loop (steps 7-25) terminates because the set Γ contains a finite number of paths and loses the chosen path in each iteration as per the semantics of the for-construct. The loop comprising steps 9-24 terminates because every path β in μ_p^r contains a finite number of sets of transitions and step 12 reduces the length by one in every iteration of the loop; if, however, any of these iterations do visit steps 19-20, then in step 20, β becomes empty and hence, this will be the last iteration of the while loop comprising steps 9 to 24. The loop comprising steps 13-17

terminates because at any stage, and hence on entry to the loop, $\mu_{\|}$ has only a finite number of sets of transitions and in every iteration c increases by one; so finally, the second condition $c \leq$ length($\mu_{\|}$) is bound to become false if the first condition does not become false by then.

Proof of $\mu_{1,p'} \simeq \mu^r_{1,p'}$: Let the initial value of μ^r_p (with which the function in Algorithm 2 is invoked), denoted as $\mu^r_p(-1)$, be of the form $\mu^r_p(-1) = \langle \Gamma_1, \Gamma_2, \ldots, \Gamma_n \rangle$, where, for all $i, 1 \leq i \leq n, \Gamma_i = \{\beta_{1,i}, \beta_{2,i}, \ldots, \beta_{t_i,i}\}$. So the outermost while-loop (steps 3-26) executes n times; for the $i-th$ execution of this loop, the inner for-loop executes t_i times; together, there are $t_1 \times t_2 \times \ldots \times t_n = t$, say, iterations in each of which a path $\beta_{j,i}$ is accounted for. The algorithm treats these paths identically without making any distinction among paths from the same set or different sets. Hence we can treat the members of μ^r_p as a flat sequence of paths of the form $\langle \beta_1, \beta_2, \ldots \beta_t \rangle$. Let $\mu^r_p(i)$ and $\mu_{\|}(i)$ respectively indicate the values of μ^r_p and $\mu_{\|}$ at step 8 after the $i-th$ path β_i in the above sequence has been treated. So, the first time step 8 is executed, the value of $\mu_{\|}$ is $\mu_{\|}(0)$ = the first member β_1 of $\mu^r_p(-1)$ and $\mu^r_p(0)$ contains all the remaining members $\beta_2, \ldots, \beta_t$ of $\mu^r_p(-1)$. The final value returned by the algorithm (step 27) is $\mu_{\|}(t)$ and $\mu^r_p(t) = \emptyset$ (by negation of the condition of the outermost while loop (steps 3-26)). We have to prove that $\mu_{1,p'} = \mu_{\|}(t) \simeq \mu^r_p(-1) = \mu^r_{1,p'} \simeq \mu^r_{0,p} \simeq \mu_{0,p}$. We prove the invariant

$$\mu_{\|}(i).\mu^r_p(i) \simeq \mu^r_p(-1), \forall i, 0 \leq i \leq t \ldots\ldots Inv(1) \tag{4}$$

by induction on i, where the operator '.' stands for concatenation of two sequences. Note that in this invariant, for $i = t$,

$\mu_{\|}(t).\mu^r_p(t) \simeq \mu^r_p(-1) \Rightarrow \mu_{1,p'}.\emptyset \simeq \mu^r_p \Rightarrow \mu_{1,p'} \simeq \mu^r_{1,p'}$, which would accomplish the proof as $\mu^r_{1,p'} \simeq \mu^r_{0,p}$ holds because the former has been obtained by *equivalence substitution of each member* in the latter and $\mu^r_{0,p} \simeq \mu_{0,p}$ by Theorem 5.

Basis ($i = 0$): $\mu_{\|}(0).\mu^r_p(0) = \langle \beta_1 \rangle.\langle \beta_2, \ldots, \beta_t \rangle = \langle \beta_1, \beta_2, \ldots, \beta_t \rangle \simeq \mu^r_p(-1)$.

Induction Hypothesis: Let $\mu_{\|}(i).\mu^r_p(i) \simeq \mu^r_p(-1)$, for $i = m - 1$.

Induction step ($i = m$): R.T.P $\mu_{\|}(m).\mu^r_p(m) \simeq \mu^r_p(-1)$. Let the m^{th} path chosen be $\beta_m = \langle T_{1,m}, T_{2,m}, \ldots, T_{l_m,m} \rangle$. Let $\mu_{\|}(m-1) = \langle T_1, T_2, \ldots, T_k \rangle$. For $T_{1,m} (= T_p)$, comparison starts with the first member $T_1 = T_c$ of $\mu_{\|}(m-1)$.

Now we need to consider the inner while loop comprising steps 9-24, where the members of β_m, i.e., $T_{j,m}$, $1 \leq j \leq l_m$, are taken one by one and compared with the members of $\mu_{\|}(m-1)$. Note that the inner loop need not always execute l_m times. Let it execute $n_m \leq l_m$ times. Let $\mu^r_p(m-1)(j)$, $1 \leq j \leq n_m$, represent the value of $\mu^r_p(m-1)$ after the j^{th} iteration of this loop for the path β_m. Thus, $\mu^r_p(i-1)(0)$ is the value of $\mu^r_p(i-1)$ at step 8 when no members of β_i have yet been considered. Hence, $\mu^r_p(m-1)(0) = \mu^r_p(m-1)$. Also, $\mu^r_p(i-1)(n_i) = \mu^r_p(i)$. Let $\beta_m(j)$ be the value of β_m and $\mu_{\|}(m-1)(j)$ be the value of $\mu_{\|}(m-1)$ after the j^{th} execution of the inner while loop (steps 9-24) for the path β_m. We prove the invariant

$$\mu_{\|}(m-1).\beta_m \simeq \mu_{\|}(m-1)(j).\beta_m(j), \forall j, 0 \leq j \leq n_m \ldots\ldots Inv(2) \tag{5}$$

Let us first examine how the *Inv (2)* helps us accomplish the proof of the induction step of *Inv (1)*. Putting $j = n_m$ in *Inv (2)*,

$\mu_{\|}(m-1).\beta_m \simeq \mu_{\|}(m-1)(n_m).\beta_m(n_m) = \mu_{\|}(m).\emptyset$
(since, $\mu_{\|}(m-1)(n_m) = \mu_{\|}(m)$ and $\beta_m(n_m) = \emptyset$ from the termination condition of the loop comprising steps 9-24).

Also, $\beta_m.\mu^r_p(m) = \mu^r_p(m-1)$ [when β_m is chosen at step 7]. So for the inductive step proof goal, $\mu_{\|}(m).\mu^r_p(m) = (\mu_{\|}(m-1).\beta_m).\mu^r_p(m)$
$= \mu_{\|}(m-1).(\beta_m.\mu^r_p(m))$ [by associativity of concatenation operation '.']
$= \mu_{\|}(m-1).\mu^r_p(m-1) \simeq \mu^r_p(-1)$ [by induction hypothesis]

We now carry out the inductive proof of *Inv (2)* by induction on j.

Basis ($j = 0$)**:** The basis case holds because $\mu_{\|}(i-1)(0) = \mu_{\|}(i-1)$ and $\beta_i(0) = \beta_i$.

Induction Hypothesis: Let the invariant *Inv (2)* is true for $j = k - 1$, i.e., $\mu_{\|}(m-1).\beta_m \simeq \mu_{\|}(m-1)(k-1).\beta_m(k-1)$.

Induction step ($j = k$)**:** R.T.P $\mu_{\|}(m-1).\beta_m \simeq \mu_{\|}(m-1)(k).\beta_m(k)$. Let $\beta_m(k-1) = \langle T_{k,m}, T_{k+1,m}, \ldots, T_{l_m,m} \rangle$. Without loss of generality, let the iterations $1, \ldots, k-1$ of the loop of steps 9-24 did not visit step 20; otherwise, the loop will not be executed k^{th} time. In the k^{th} iteration of the loop, $T_{k,m}$ is compared with some $T_c \in \mu_{\|}(m-1)(k-1)$. We have the following two cases:

- Case 1: $T_{k,m}$ is found to succeed all the members of $\mu_{\|}(m-1)(k-1)$ from T_c onwards– Hence, $T_{k,m}$ is parallelizable with no members of $\mu_{\|}(m-1)(k)$. In this case, step 20 is executed resulting in concatenation of all the transition sets of $\beta_m(k-1)$ with $\mu_{\|}(m-1)(k-1)$ and $\beta_m(k)$ becomes empty. So, $\mu_{\|}(m-1)(k) = \mu_{\|}(m-1)(k-1).\beta_m(k-1)$;
  hence, $\mu_{\|}(m-1).\beta_m \simeq \mu_{\|}(m-1)(k-1).\beta_m(k-1)$ [by Induction hypothesis]
  $= \mu_{\|}(m-1)(k).\beta_m(k)$ (since $\beta_m(k) = \emptyset$)
- Case 2: $T_{k,m} \not\succ T_c$– This implies $T_{k,m} \asymp T_c$, as argued below. Note that between the two transition sets $T_{k,m}$ and T_c, there can be three mutually exclusive relations possible namely, $T_{k,m} \succ T_c$, $T_c \succ T_{k,m}$ and $T_{k,m} \asymp T_c$. It is given that $T_{k,m} \not\succ T_c$; now, let $T_c \succ T_{k,m}$. The transition set T_c in $\mu_{\|}(m-1)$ is contributed to by paths which precede the path β_m in μ_p^r. Hence T_c does not succeed $T_{k,m}$. Therefore, $T_{k,m} \asymp T_c$. Let $\mu_{\|}(m-1) = \langle T_1, T_2, \ldots, T_c, T_{c+1}, \ldots, T_k, \ldots T_{k_{m-1}}\rangle$. For all $s, 1 \le s \le k_{m-1} - c$, $T_c \not\succ T_{c+s}$. By an identical reasoning, $T_{k,m}$ does not also succeed T_{c+s} because otherwise T_{c+s} would have preceded in the path β_m. Therefore, $T_{c+s}.T_{k,m} \simeq T_{k,m}.T_{c+s}$. So, the concatenation $\langle T_{c+1}, \ldots, T_{k_{m-1}}\rangle.\langle T_{k,m}, T_{k+1,m}, \ldots, T_{l_m,m}\rangle$ is computationally equivalent to
  $\langle T_{k,m}, T_{c+1}, \ldots, T_{k_{m-1}}\rangle.\langle T_{k+1,m}, \ldots, T_{l_m,m}\rangle$. Now,
  $\mu_{\|}(m-1).\beta_m \simeq \mu_{\|}(m-1)(k-1).\beta_m(k-1)$ [by induction hypothesis]
  $= \langle T_1, T_2, \ldots T_c, T_{c+1}, \ldots T_{k_m-1}\rangle.\langle T_{k,m}, T_{k+1,m}, \ldots, T_{l_m,m}\rangle$
  $\simeq \langle T_1, T_2, \ldots T_c, T_{k,m}, T_{c+1}, \ldots T_{k_m-1}\rangle.\langle T_{k+1,m}, \ldots, T_{l_m,m}\rangle$
  $\simeq \langle T_1, T_2, \ldots T_c \cup T_{k,m}, T_{c+1}, \ldots T_{k_m-1}\rangle.\beta_m(k)$
  [since, by step 12, $\beta_m(k) = \langle T_{k+1,m}, \ldots, T_{l_m,m}\rangle$]
  $= \mu_{\|}(m-1)(k).\beta_m(k)$ [by step 22].
  Note that since $T_{k,m} \succ T_{c-1}$ in $\mu_{\|}(m-1)(k-1)$, as identified in the loop steps 13-17, T_c cannot be combined with T_{c-1} through union.

Proof of $\mu_{1,p'}$ being a computation: Recall that $\mu_{1,p'}$ is obtained from the sequence $\mu_{1,p'}^r = \langle \Gamma_1^{p'}, \Gamma_2^{p'}, \ldots, \Gamma_n^{p'}\rangle = \langle\{\beta_{1,1}, \beta_{2,1}, \ldots, \beta_{l_1,1}\}\{\beta_{1,2}, \beta_{2,2}, \ldots, \beta_{l_2,2}\}, \ldots, \{\beta_n\}\rangle$ of sets of paths of N_1, which, in turn, was constructed from the sequence $\mu_{0,p}^r = \langle \alpha_1^p, \alpha_2^p, \ldots, \alpha_n^p\rangle$ of paths of Π_0 satisfying the condition: for all $j, 1 \le j \le n$, for all $k, 1 \le k \le l_j, \beta_{k,j} \simeq \alpha_j^p$. From Definition 3 of path equivalence, the above set of equivalences imply the following properties: (i) $p' = \beta_n^\circ$, (ii) each of the places in ${}^\circ\Gamma_j^{p'}$ has correspondence with some place in ${}^\circ\alpha_j^p$ and (iii) all the places in Γ_j° have correspondence with all the places in $(\alpha_j^p)^\circ$.

Let $\mu_{1,p'}$ be $\langle T_1, T_2, \ldots, T_l\rangle$, where T_1 is the first member of $\beta_{1,1}$ (by step 1 and first time execution of steps 7 and 11 of Algorithm 2). By property (ii) above, the places in ${}^\circ\beta_{1,1} \supseteq {}^\circ T_1$ have correspondence with those in ${}^\circ\alpha_1^p \subseteq inP_0$. Since only the input places of N_1 have correspondence with the input places of N_0, ${}^\circ T_1 \subseteq {}^\circ\beta_{1,1} \subseteq inP_1$. It has already been proved that $p' = f_{out}(p) \in outP_1$. Since Algorithm 2 introduces the transition sets of the paths strictly in order from $\Gamma_1^{p'}$ to $\Gamma_n^{p'}$, T_l is a unit set containing the last transition of β_n and hence, $p' \in T_l^\circ$. Now, consider any $T_i \in \mu_{1,p'}$, $1 \le i < l$; $T_{i+1} \succ T_i$ as ensured by the condition $T_p \succ T_c$ associated with the while loop of steps 13-17. For any $i, 1 \le i < l$, let $T_{i+1}^\circ \subseteq P_{M_{i+1}}$ and $T_i^\circ \subseteq P_{M_i}$. It is required to prove that $M_{i+1} = M_i^+$, where $P_{M_i^+} = \{p \mid p \in t^\circ \wedge t \in T_m\} \cup \{p \mid p \in P_M \wedge p \notin^\circ T_m\}$, by first clause of Definition 4 of successor marking. We have the following two cases:

- Case 1: $p_1 \in T_{i+1}^\circ \subseteq P_{M_{i+1}} - p_1 \in T_{i+1}^\circ \Rightarrow \exists t_1 \in T_{i+1}$ such that $p \in t_1^\circ$. Now, $T_{i+1} = T_{M_i}$, the set of enabled transitions for the marking M_i. So, $p_1 \in t_1^\circ$ and $t_1 \in T_{i+1} = T_{M_i} \Rightarrow p_1 \in P_{M_i^+}$ by virtue of its being in the first subset of $P_{M_i^+}$.
- Case 2: $p_1 \notin T_{i+1}^\circ$ but $\in P_{M_{i+1}}$ – So, $p_1 \notin T_{i+1}^\circ = T_{M_i}$. Hence, $p_1 \in P_{M_i}$ because $p_1 \in T_{i-k}^\circ$ for some $k \ge 1$. So $p_1 \in P_{M_i^+}$ by virtue of its being in the second subset of $P_{M_i^+}$. Therefore, $M_{i+1} = M_i^+$.□

F Equivalence Checking Algorithm its Complexity and Correctness

The module-wise functional description of the equivalence checking mechanism is captured through the Algorithms 3 and 4. The `chkEqvSCP` function is the central module for the method.

The inputs to this function are the CPN models, N_0 and N_1, with their in-port and out-port bijections f_{in} and f_{out}. The outputs are two sets Π_0 of N_0 and Π_1 of N_1 comprising the respective paths of N_0 and N_1 which are equivalent, a set E of ordered pairs of equivalent paths of N_0 and N_1, the set η_t of corresponding transition pairs and the sets of paths $\Pi_{n,0}$ of N_0 and $\Pi_{n,1}$ of N_1 comprising member paths for which no equivalent is found (in the other CPN model).

The function starts by initializing the set η_p of ordered pairs of corresponding places of N_0 and N_1 to the in-port bijection f_{in} and out-port bijection f_{out}; the set η_t of ordered pairs of corresponding transitions of N_0 and N_1 and the sets $E, \Pi_{n,0}$ and $\Pi_{n,1}$ are initialized to empty. It then constructs the set Π_0 of paths of N_0 and the set Π_1 of paths of N_1 by introducing static cut-points at places at which some back edges terminate. For each path α in Π_0 (of N_0), the algorithm calls `findEqvSCP` function which tries to find an equivalent path from Π_1 of N_1 starting from the place which have pairwise correspondence with those of $^\circ\alpha$. The function `findEqvSCP` returns a set Γ of paths. If the set Γ has more than one member ($|\Gamma| > 1$), then it implies that for a path α in N_0, there are more than one equivalent path in N_1, all of them originating from the same set of places and having identical conditions of execution (as that of α); the following entities are updated: For each $\beta \in \Gamma$, (1) the set η_t of corresponding transitions by adding the pair comprising the last transition of the path α and that of β; (2) the set E of ordered pairs of equivalent paths by adding the ordered pair $\langle \alpha, \beta \rangle$; (3) The set η_p of corresponding places by adding the pair comprising the post-places of the last transition of the path α and that of β. If Γ is empty, the module updates $\Pi_{n,0}$ by adding the path α to it.

When all the paths in the path cover Π_0 of N_0 have been examined exhaustively, all the paths in Π_1, which were not identified to be equivalent with any path in Π_0, are put in $\Pi_{n,1}$. The function then checks $\Pi_{n,0}$ and $\Pi_{n,1}$; we have the following four cases:
Case 1: $\Pi_{n,0}, \Pi_{n,1} = \emptyset \Rightarrow N_0 \equiv N_1$; Case 2: $\Pi_{n,0} = \emptyset, \Pi_{n,1} \neq \emptyset \Rightarrow N_0 \sqsubseteq N_1$ but may be that $N_1 \not\sqsubseteq N_0$; Case 3: $\Pi_{n,0} \neq \emptyset, \Pi_{n,1} = \emptyset \Rightarrow N_1 \sqsubseteq N_0$ but $N_0 \not\sqsubseteq N_1$; Case 4: $\Pi_{n,0}, \Pi_{n,1} \neq \emptyset \Rightarrow$ neither $N_0 \sqsubseteq N_1$ nor $N_1 \sqsubseteq N_0$.

F.1 Complexity analysis of the equivalence checking algorithm

We discuss the complexity of the equivalence checking algorithm in a bottom-up manner.

Complexity of Algorithm 4 (`findEqvSCP`): Step 1 is an initializing step which takes in $O(1)$ time. Step 2 takes $O(|\Pi_1|) = O\left(|P|^3\right)$ time, which is the complexity of path construction as explained in literature [9]. Step 4 compares the condition of execution and the data transformation for each path. Hence the complexity for each of this comparison is $O(|F|)$, where $|F|$ is the maximum of the lengths of the formulae representing the data transformations and conditions of execution of paths of N_0, N_1. Computation of such formulae is exponential in the number of variables which, in turn, is upper-bound by the number of places, i.e., $O(|F|)$ is $O\left(2^{|P|}\right)$. Step 5 is a union operation needing just $O(1)$ time with β being blindly put at the end of Γ. The loop iterates as many times as $O(|\Pi_1|) = O\left(|P|^3\right)$. Hence, the overall complexity is $= O\left(|P|^3 + 2^{|P|}.\,|P|^3\right) = O\left(2^{|P|}.|P|^3\right)$.

Complexity of Algorithm 3 (`chkEqvSCP`): In step 1, construction of η_p takes $O(|P|)$ time. In the same step, the function constructs all the paths for the two CPN models in $O\left(|P|^3\right)$ time as given in [9]. The complexity of each iteration of the loop of step 2 is as follows. Step 3 uses `findEqvSCP` function and takes $O\left(2^{|P|}.|P|^3\right)$ time as explained above. Π_1 is updated by the set minus operation in $O(|P|^3)$ time. So step 3 takes $O\left(2^{|P|}.|P|^3\right)$ time. Checking of the condition $\Gamma \neq \emptyset$ in step 4 takes $O(1)$ time. The complexity of the inner loop starting at step 5 is as follows. The update operations of η_t and E in step 6 take $O(1)$ time whereas that of η_p takes $O\left(|P|^2\right)$ time. So the body of the loop (step 6) takes $O\left(|P|^2\right)$ time; the loop executes $O(|P|^3)$ time; hence it takes $O\left(|P|^5\right)$ time. Step 9 takes $O(1)$ time. So, the `if`-statement comprising 4-10 takes $O\left(|P|^5\right)$ time. Thus, the body of the outer loop (steps 2-11) takes $O\left(2^{|P|}.|P|^3\right)$ (step 3) + $O\left(|P|^5\right) = O\left(2|P|.|P|^3\right)$ time. The loop executes $O\left(|P|^3\right)$ time. So the complexity of the loop is $O\left(\left(2^{|P|}.|P|^3\right).|P|^3\right) = O\left(2^{|P|}.|P|^6\right)$. Step 12 takes $O\left(|P|^3\right)$ time and step 13 takes $O(1)$ time. So the overall complexity of this module is $O\left(2^{|P|}.|P|^6\right)$.

Algorithm 3 STRUCT6TUPLE **chkEqvSCP**(N_0, N_1)

Inputs: The CPN models N_0 and N_1.
Outputs: A six tuple structure comprising
1. Π_0: A finite path cover of N_0.
2. Π_1: A finite path cover of N_1.
3. E : a set of ordered pairs of the form $\langle \alpha_i, \beta_j \rangle$ of paths of Π_0 and Π_1 respectively, such that $\alpha_i \simeq \beta_j$.
4. η_t: the set of corresponding transition pairs;
5. $\Pi_{n,0}$: the set of paths of N_0 for which no equivalent is found in N_1 even with extension.
6. $\Pi_{n,1}$: the set of paths of N_1 for which no equivalent is found in N_0.

1: Let $\eta_p = \{\langle p, p' \rangle \mid p \in inP_0 \wedge p' = f_{in}(inP_0) \wedge \langle p, p' \rangle \in f_{in}(p)\} \cup \{\langle p, p' \rangle \mid p \in outP_0 \wedge p' = f_{out}(outP_0)\}$;
Let η_t, the set of pairs of corresponding transitions, be $\emptyset$;
Π_0 = **constAllPathsSCP**(N_0);
Π_1 = **constAllPathsSCP**(N_1);
Let $\Pi_{n,0}$, $\Pi_{n,1}$ and E be empty;
2: **for** each $\alpha \in \Pi_0^{'}$ such that $\forall p \in^{\circ} \alpha, \exists p', \langle p, p' \rangle \in \eta_p$ **do**
3: $\Gamma \Leftarrow$**findEqvSCP** $(\alpha, \eta_p, \Pi_1^{'}, f_{in})$;
$\Pi_1 = \Pi_1 - \Gamma$;
4: **if** $\Gamma \neq \emptyset$ **then**
5: **for** each $\beta \in \Gamma$ **do**
6: $\eta_t = \eta_t \cup \{ \langle \text{last}(\alpha), \text{last}(\beta) \rangle \}$;
$E \leftarrow E \cup \{\langle \alpha, \beta \rangle\}$;
$\eta_p = \eta_p \cup \{\langle p, p' \rangle | p \in \alpha^{\circ}, p' \in \beta^{\circ}, f_{pv}^0(p) = f_{pv}^1(p')\}$;
7: **end for**
8: **else**
9: $\Pi_{n,0} = \Pi_{n,0} \cup \{\alpha\}$;
10: **end if**
11: **end for** /* $\forall \alpha \in \Pi_0^{'}$ */
12: $\Pi_{n,1} = \Pi_1$;
13: **Case 1** ($\Pi_{n,0} = \emptyset$ and $\Pi_{n,1} = \emptyset$):
Report "N_0 and N_1 are the equivalent models."
break;
Case 2($\Pi_{n,0} = \emptyset$ and $\Pi_{n,1} \neq \emptyset$):
Report "$N_0 \sqsubseteq N_1$ and $N_1 \not\sqsubseteq N_0$."
break;
Case 3($\Pi_{n,0} \neq \emptyset$ and $\Pi_{n,1} = \emptyset$):
Report "$N_1 \sqsubseteq N_0$ and $N_0 \not\sqsubseteq N_1$."
break;
Case 4 ($\Pi_{n,0} \neq \emptyset$ and $\Pi_{n,1} \neq \emptyset$):
Reports "two models may not be equivalent."
14: **return** $\langle \Pi_0, \Pi_1, E, \eta_t, \Pi_{n,0}, \Pi_{n,1} \rangle$;

F.2 Correctness of Equivalence checking algorithm

Theorem 7 *If the function* `chkEqvSCP` *(Algorithm 3) reaches step 14 and* (*a*) *returns* $\Pi_{n,0} = \emptyset$, *then* $N_0 \sqsubseteq N_1$ *and* (*b*) *if it returns* $\Pi_{n,1} = \emptyset$, *then* $N_1 \sqsubseteq N_0$.

Proof Let the function `chkEqvSCP` reach step 14 and $\Pi_{n,0} = \emptyset$. It is required to prove that $N_0 \sqsubseteq N_1$, i.e., for any computation $\mu_{0,p}$ of N_0, there exists a computation $\mu_{0,p'}$ of N_1 such that $\mu_{0,p} \simeq \mu_{1,p'}$ and $p' = f_{out}(p)$. The fact that if the function `chkEqvSCP` reaches step 14 and $\Pi_{n,1} = \emptyset$, then $N_1 \sqsubseteq N_0$ can be proved identically.

Consider any computation $\mu_{0,p}$ of N_0. Step 2 of the function `chkEqvSCP` calls the function `constAllPathsSCP` and yields the set Π_0 of paths of N_0 from the set of cut-points. From Theorem 5, there exists a reorganized sequence of $\mu_{0,p}^r$ of paths of Π_0 such that $\mu_{0,p}^r \simeq \mu_{0,p}$. Hence, Π_0 is a path cover of N_0. So, from Theorem 6, it is required to prove that for every member α in Π_0,

Algorithm 4 SETOFPATHS **findEqvSCP** (α, η_p, Π_1)

Inputs: α: a path whose equivalent has to be found. η_p: the set of corresponding places pair and Π_1: path cover of N_1 If flag = 0, it belongs to N_1; if flag = 1, it belongs to N_0.
Outputs: Set Γ of equivalent paths.

1: $\Gamma = \emptyset$;
2: $\Gamma' = \{\beta \mid \beta \in \Pi_1' \wedge (\forall p \in {}^\circ\alpha, \exists p' \in {}^\circ\beta$ s.t. $\langle p, p' \rangle \in \eta_p \vee \forall p' \in {}^\circ\beta, \exists p \in {}^\circ\alpha$ s.t $\langle p, p' \rangle \in \eta_p) \wedge$
$\forall p \in \alpha^\circ$ if $p \in outP_0$, then $\exists p' \in \beta_0$ s.t. $p' = f_{out}(p) \in outP_1\}$
/* for candidate path selection */
3: **for** each $\beta \in \Gamma'$ **do**
4: **if** $R_\beta(f_{pv}({}^\circ\beta)) \equiv R_\alpha(f_{pv}({}^\circ\alpha))$ **then**
5: **if** $r_\beta(f_{pv}({}^\circ\beta)) = r_\alpha(f_{pv}({}^\circ\alpha))$ **then**
6: $\Gamma' = \Gamma' \cup \{\beta\}$
7: **else**
8: report "β not equivalent to α in spite of having pre-place correspondence and equivalent condition of execution";
9: $\Gamma' = \emptyset$; // all not equivalent
10: **end if**
11: **end if**
12: **end for**
13: **return** Γ;

there is a path β of N_1 such that (i) $\alpha \simeq \beta$, (ii) the pre-places of α have correspondence with the pre-places of β and (iii) the post-places of α have correspondence with those of β. It may be noted that the algorithm `chkEqvSCP` finds a path β of Π_1 by calling the function `findEqvSCP` such that conditions (i) and (ii) are satisfied (as ensured by steps 2 and the loop comprising steps 3-7). Condition (iii) is satisfied by step 6 in `chkEqvSCP`. □