Program equivalence checking using PetriNets

Under the supervision and guidance of :

Prof. Abhishek Singh

Prepared by :

- **Tanish Agrawal** 2018A7PS0202G
  (f20180202@goa.bits-pilani.ac.in)
- **Rishit Patel** 2018A7PS0189G
  (f20180189@goa.bits-pilani.ac.in)

Acknowledgement

We would like to thank Prof. Abhishek Singh for his constant guidance and teaching support throughout the project which made it possible for us to understand and discuss the relevant works published on petri nets.

Table of contents

Petri Nets

- Introduction and Structure of Petri Nets
- Analysis of Petri Nets
- Extensions
- Subclasses
- Colored Petri Nets

Translation Validation of Coloured Petri Net Models of Program on Integers

- Abstract
- Introduction
- Equivalence Checking
- Dynamic Cut-points
- Workflow
- A Motivating Example

Construction of Petri net based models for C programs

- Introduction
- Definition Of PRES+ Net
- An Example Of PRES+ Model
- Implementation

Petri Nets

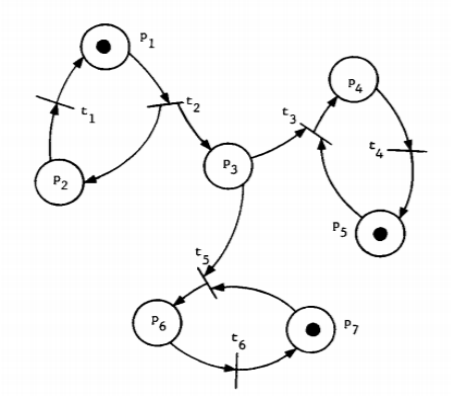
Introduction and Structure of Petri Nets:

A **Petri net**, also known as a place/transition net, is one of several mathematical modeling languages for the description of distributed systems and represents information flow.This is used to analyze the flow of information and control in systems, specially systems that can show asynchronous and concurrent behaviour.

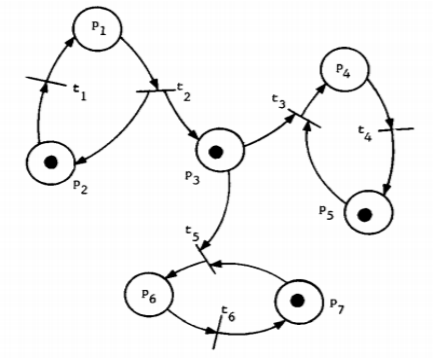
A **Petri net graph** is a **bipartite directed graph**. since the arcs are directed. Its nodes can be partitioned into two sets (places and transitions) such that each arc is directed from an element of one set (place or transition) to an element of the other set (transition or place).

The execution of a Petri net is controlled by the position and movement of **markers (called tokens)** in the Petri net.Tokens, indicated by black dots, reside in the circles representing the places of the net.Tokens are moved by the firing of the transitions of the net. A transition must be enabled (**when all of its input places have at least one token in them)** in order to fire.The transition fires by removing the enabling tokens from their input places and generating new tokens which are deposited in the output places of the transition.

Example: The transition t2 is enabled since it has a token in its input place P1.Transition t3, on the other hand, is not enabled since one of its inputs (P3) does not have a token.



When t2 fires =>



A Petri net C is defined as the four-tuple C = (P, T, I,O) [**P**- set of places , **T**-set of transitions , **input function I**- set of output places for each transition , **output function O**- set of output places for each transition.]

A **marking (μ)** of a Petri net is an assignment of tokens to the places in that net. The number and position of tokens in a net may change during its execution. The vector $\mu=(\mu_1,\mu_2,...,\mu_n)$ gives, for each place in the Petri net, the number of tokens in that place. The number of tokens in place p_i is μ_i , i=1, 2,.., n. If there are two possible options, and firing either of transitions, disables the other; they are said to be **in conflict**.

The **state** of a Petri net is defined by its marking.The **state space** of a Petri net with n places is the set of all markings.The change in state caused by firing a transition is defined by a partial function ∂, called the **next-state function**.

Let's say there is a transition t. Since t can fire only if it is enabled, $\partial(\mu, t)$ is undefined if t is not enabled in marking μ. If t is enabled, then $\partial(\mu, t) = \mu'$, where μ' is the marking that results from removing tokens from the inputs of t and adding tokens to the outputs of t.

We say that μ' is **immediately reachable** from μ if we can fire some enabled transitions in the marking μ resulting in the marking μ'. A marking μ' is **reachable** from μ if it is immediately reachable from μ or is reachable from any marking which is immediately reachable from μ.The **reachability set** R(M) for a marked Petri net M = (P, T, I, O,μ) as the set of all markings which can be reached from μ.

Petri nets were designed to model a class of problems, where there are discrete and concurrent events. Transitions are instantaneous and atomic (technical term: *primitive events*). Conditions are represented by Places/Circles and Events are represented by Transitions/Bars. Many kinds of problems, like software problems (critical section problem, concurrency), resource allocation, computational biology, hardware modeling and workflow management can be modeled via Petri Nets.

Analysis of Petri Nets:

Important questions/properties for analysis:

- **Safe net** - At most one token is present in each place of the petri net.
- **Boundedness** - Depicts the maximum number of tokens that can be present in each place of the petri net(K-bounded petri nets). Bound on the number of tokens is important in cases like implementation on hardware(capacity)
- **Conservation of tokens** - Total number of tokens in the petri net is constant for conservative petri nets. In other words, each fireable transition has an equal number of input and output places.
- **Dead transition** - The transition cannot be enabled by any sequence of firing of transitions starting from the present marking .
- **Potentially fireable transition** - The transition can be enabled eventually by a sequence of transition firings starting from the present marking.
- **Liveness** - This property is often analysed when we are aiming to model operating systems. eg:- Dead Transition might indicate potential deadlock

  There are four forms of liveness definitions for a petri net with given initial marking μ and transition t :-

  - ➢ $\delta(\mu', t)$ is defined for some μ' in R(M)
  - ➢ For every n>0 , there exists a sequence of transitions σ for which t occurs at least n times in σ and $\delta(\mu,\sigma)$ is defined
  - ➢ There exists an infinite sequence of transitions σ for which t occurs infinite times in σ and $\delta(\mu,\sigma)$ is defined
  - ➢ t is a live transition

- **Reachability problem** - Given a petri net M with an initial marking μ , determine whether the given marking μ' is part of R(M) (The special case of set reachability problem).

The reachability problem is particularly important as a lot of important questions regarding correctness and analysis of systems modelled by petri nets can be reduced to/are equivalent to this problem like :

- ➢ **zero reachability** problem (Check if an all zero marking is reachable)
- ➢ **subset reachability** problem (Check if there exists a reachable marking with a given subset of places matching that of a given initial marking)
- ➢ **liveness problem** (Check if all transitions are live)

There exists an algorithm ,which is quite complex in implementation and cost,to solve the reachability problem and hence ,the other equivalent problems.

Techniques/Approaches for Analysis:

The most important and basic technique used in analysis of petri nets is generating a finite depiction of its reachability set R(M) because many properties of petri nets are closely related to its reachability set.

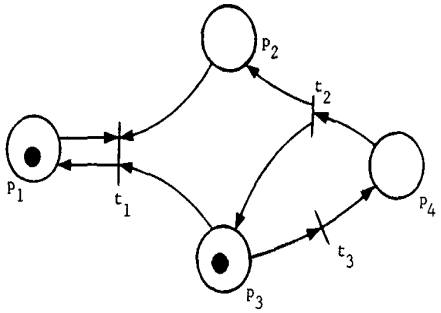
Reachability tree is a tree structure with the edges/arcs denoting the transitions that changes the petri net state on firing . The nodes represent the current marking of the petri net.
Let's see how this tree is created : -

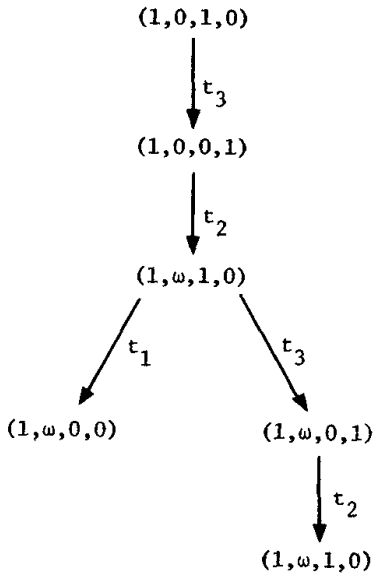
- Label the first node/root with the initial marking
- For all the fireable transitions in the current marking μ, create a new arc directed from the current node to a new node
- This new node will have the marking - $\delta(\mu,t)$ as its label and the new arc will have t as its label
- Repeat the above process for all new nodes

Two important steps to ensure that the reachability tree is finite :-

- If the newly created node is labelled with a marking which matches a previously existing marking on the path from the root to this new node , then the new node is a terminal node .
- If the newly created node is labelled with a marking which is greater than a previously existing marking on the path from the root to this new node, then replace the strictly greater numbers/components in the new node marking by ω .
- ω represents an arbitrarily large value with properties like $\omega \pm a = \omega$, $a < \omega$ for all $a > 0$

Example:





Analysis using reachability tree:

- Both K-bounded petri nets and conservative petri nets have a finite reachability set and ofcourse, a finite reachability tree
- Presence of ω in any node of the tree implies that the reachability set is infinite, the petri net is unbounded and the petri net is non conservative
- Absence of ω in any node of the tree implies that the reachability set is finite and the petri net is bounded . The conservative property is checked via inspection.

- Coverability problem - Check if there exists a marking μ' in R(M) which is greater than or equal to the initial marking μ of a petri net M
- Most analysis problems are solved via inspection of the tree structure including the coverability problem

Unsolvable problems and complexity:

- Few problems related to petri nets are not solvable or are undecidable like:
  Subset problem : Check if reachability set of one petri net is subset of another
  Equality problem : Check if reachability set of one petri net is equal to another

- Complexity of reachability problem is said to have an exponential time lower bound and exponential space lower bound similar to the coverability problem

- Thus, even though Petri nets are instrumental in modelling and analysis of systems, solving even the basic problems for analysis might be quite expensive

Extensions:

We need modeling power and decision power for a successful model. Petri nets have more modelling power than FSMs but it is still difficult or impossible to model some events or conditions by Petri nets. Therefore, we need to extend petri nets to increase their modeling power. Petri nets seem to be just below turing machines in modeling power.

In terms of modeling power Petri nets seem to be just below Turing machines, so any significant extension results in Turing-machine equivalence.

- **Generalized Petri nets**: A place may contribute or receive more than one token from the firing of a transition. They are modeled by allowing multiple arcs between transitions and places, signifying the number of tokens needed.These nets are equivalent to ordinary Petri nets.
- **Zero Testing**: Arcs, known as inhibitor arcs, from a place to a transition which allow the transition to fire only if the place has zero tokens in it. It has been shown that such Petri nets have the modeling power of a Turing machine. Therefore we can show that many of the decidability problems are unsolvable for such petri nets.

Subclasses:

The limitations on the modeling power of Petri nets relative to Turing machines are balanced out by a compensating increase in decision power.
For Petri nets many decision problems are equivalent to the reachability problem, which is decidable. However, the reachability problem is very difficult to solve. Thus, from a practical point of view, Petri nets may be too powerful to be analyzed.

- **State Machines**: Each transition has exactly one input and one output. These nets are obviously conservative, which means they have a finite reachability set and hence are finite-state. In fact, they are exactly the class of finite-state machines.
- **Marked Graphs**: A marked graph is a Petri net in which each place has exactly one input transition and one output transition. Marked graphs have high decision power. But, they have limited modelling power as they are only able to model systems without branching. This solves the problem of conflicting transitions which are difficult to analyse.
- **Free Choice Nets**: Each arc from a place is either the unique output of the place, or the unique input to a transition.If there is a token in a place, then

the token will remain in that place until its unique output transition fires.If there are multiple outputs for the place, then there is a free choice as to which of the transitions is fired. Liveness and safeness are decidable and we have necessary and sufficient conditions for these properties.

Colored Petri Nets:

Coloured Petri nets are a backward compatible extension of the mathematical concept of Petri nets.

Coloured Petri nets preserve useful properties of Petri nets and at the same time extend the initial formalism to allow the distinction between tokens.

Coloured Petri nets allow tokens to have a data value attached to them. This attached data value is called the **token color**. Although the color can be of arbitrarily complex type, places in coloured Petri nets usually contain tokens of one type. This type is called the **color set** of the place.

Definition 4.2. A **non-hierarchical Coloured Petri Net** is a nine-tuple $CPN = (P, T, A, \Sigma, V, C, G, E, I)$, where:

1. P is a finite set of **places**.
2. T is a finite set of **transitions** T such that $P \cap T = \emptyset$.
3. $A \subseteq P \times T \cup T \times P$ is a set of directed **arcs**.
4. Σ is a finite set of non-empty **colour sets**.
5. V is a finite set of **typed variables** such that $Type[v] \in \Sigma$ for all variables $v \in V$.
6. $C : P \rightarrow \Sigma$ is a **colour set function** that assigns a colour set to each place.
7. $G : T \rightarrow EXPR_V$ is a **guard function** that assigns a guard to each transition t such that $Type[G(t)] = Bool$.
8. $E : A \rightarrow EXPR_V$ is an **arc expression function** that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)_{MS}$, where p is the place connected to the arc a.
9. $I : P \rightarrow EXPR_\emptyset$ is an **initialisation function** that assigns an initialisation expression to each place p such that $Type[I(p)] = C(p)_{MS}$.

Translation Validation of Coloured Petri Net Models of Program on Integers

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 fit the bill well. 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).

Introduction :

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. Several code transformation techniques such as, code motions, dead code elimination, etc., loop based transformation techniques such as, un-switching, reordering, and thread level parallelizing transformations such as, loop distribution, loop parallelizing etc., may be applied at the pre-processing stage of embedded system synthesis. These transformations are carried out by some compilers. 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. The operational semantics of (value based)

Petri nets permit depiction of operations having no mutual dependencies 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 compared to control data-flow graphs (CDFGs).

Equivalence Checking :

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

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.

Dynamic Cut Points :

For CPN models, 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 of paths induced by loop cut-points. This observation led to the need for additional dynamic cut-points over and above the static cut-points. 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 values. A dynamic cut-point (DCP) based analysis mechanism is known as DCPEQX. The validity of DCPEQX hinges upon the fact that any computation can be viewed as a syntactic concatenation of DCP induced paths. 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 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.

Workflow :

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.

It is important to validate the translation. For analysis of this translation, it is necessary to convert these programs into equivalent formal models. We have chosen CPN (Coloured Petri Net) as our modelling paradigm.

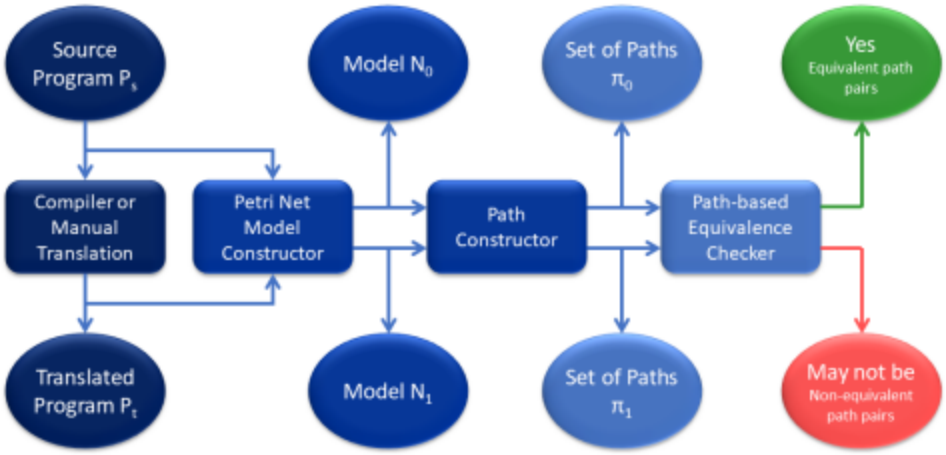


Fig. 1 Basic workflow

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. To analyze translations involving loops, it is 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 conditions 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 are 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.

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.

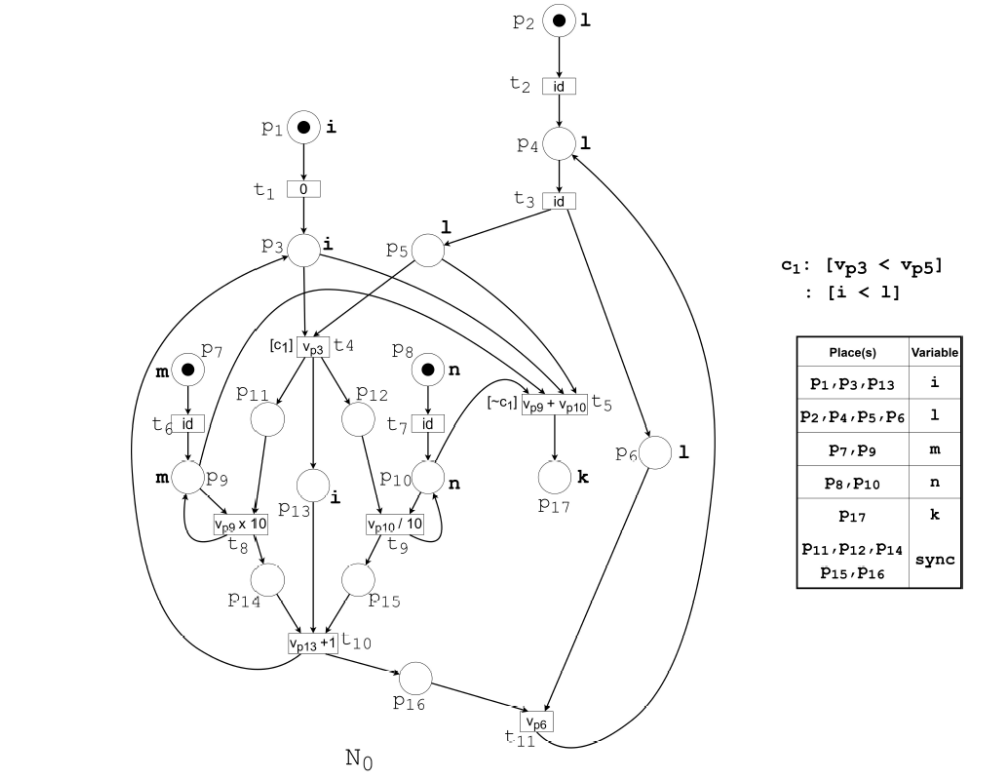
Figure 2(a) depicts the program P_s which takes three input integers m, n, l; it computes the function

$$k = m.10^l + n/10^l \text{ , } l \geq 0$$

$$= m + n, \; l < 0$$

The corresponding transformed program P_t given in Figure 2(b) is obtained by loop

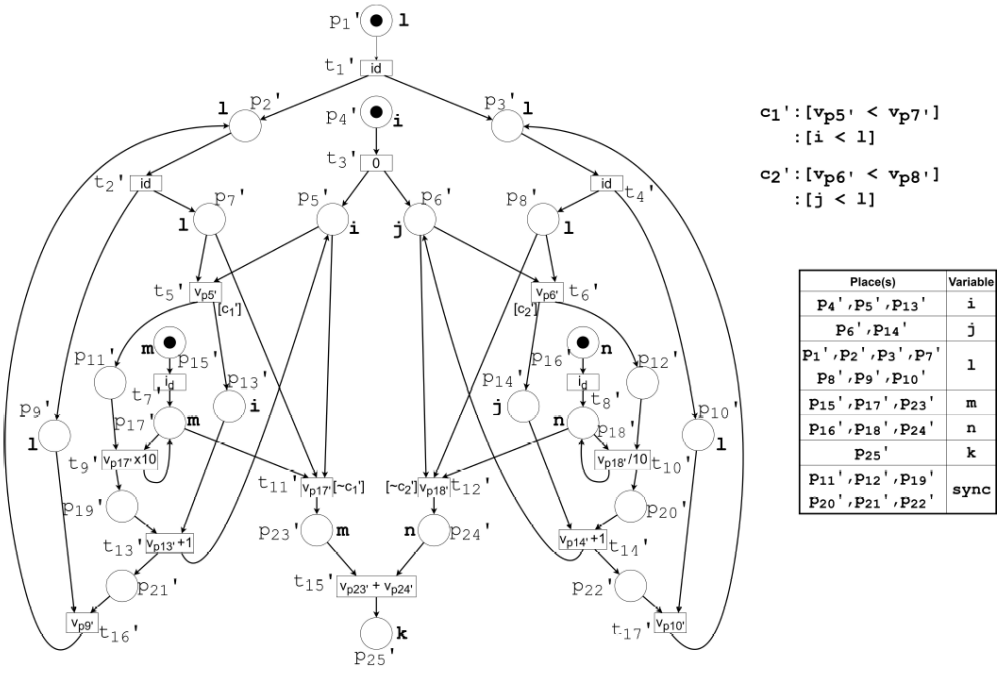
distribution followed by thread level parallelizing transformation of P_S. In Figure 2(b), the independent subexpressions $m* 10^l$ and $n/10^l$ are computed separately in two parallelized loops.



a)

c_1: $[v_{p3} < v_{p5}]$
: $[i < l]$

Place(s)	Variable
p_1, p_3, p_{13}	i
p_2, p_4, p_5, p_6	l
p_7, p_9	m
p_8, p_{10}	n
p_{17}	k
p_{11}, p_{12}, p_{14} p_{15}, p_{16}	sync



b)

c_1': $[v_{p5'} < v_{p7'}]$
: $[i < l]$

c_2': $[v_{p6'} < v_{p8'}]$
: $[j < l]$

Place(s)	Variable
p_4', p_5', p_{13}'	i
p_6', p_{14}'	j
p_1', p_2', p_3', p_7' p_8', p_9', p_{10}'	l
$p_{15}', p_{17}', p_{23}'$	m
$p_{16}', p_{18}', p_{24}'$	n
p_{25}'	k
$p_{11}', p_{12}', p_{19}'$ $p_{20}', p_{21}', p_{22}'$	sync

$\mathbf{N_0}$ - Colored Petri net representation for source program P_S , $\mathbf{N_1}$ - Colored Petri net representation for transformed program P_t

Let us examine the CPN model $\mathbf{N_0}$ corresponding to the program Ps. 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 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_{t3} 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^o_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 ${}^o t_8 ({}^o t_9)$ and $t^o_8 (t^o_9)$; it is made to hold the initial input value of m (n) through a separate transition t_6 (t_7) with ${}^o t_6 = \{p_7\}$ (${}^o t_7 = \{p_8\}$) and $t^o_6 = \{p_9\} (t^o_7 = \{p_{10}\})$. 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 has 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_{t4} is of no concern; however, subsequently we can see why f_{t4} 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 the 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_{t4} 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.

Construction of Petri net based models for C programs

Introduction :

A fully automatic translation validation process requires a common semantic framework for the representation of the source code and the generated target code. Petri net based models are more suitable then cfgs (Control Flow Graphs) like FSMDs (Finite State Machines with Datapaths) and CSPs (Concurrent Sequential Processes) due to parallelism. The structural similarity between the Petri net models of the source and the transformed programs makes the task of establishing equivalence between them easier. The work done indigenously to establish equivalence between programs using Petri net based models have been encouraging. The mechanism uses an extension of the Petri net based models called Petri net based Representation of Embedded Systems abbreviated as PRES+.

Definition of PRES+ Net :

A PRES+ net is an eight-tuple $N = < P, T, I, O, inP, outP, V, f_{pv}>$, where the members are defined as follows. The set P is a finite non-empty set of places. A place p is capable of holding a token having a value v_p of any data type. T is a finite non-empty set of transitions. A transition represents a function. $I \subseteq P \times T$ 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 $t \in T$. V is the set of variables used in the PRES+ and fpv is a relation capturing the association of the places of the PRES+ with variables. A transition can have a guard condition associated with it s.t. the transition is executed only if its guard condition is evaluated to True.

An Example of PRES+ Model :

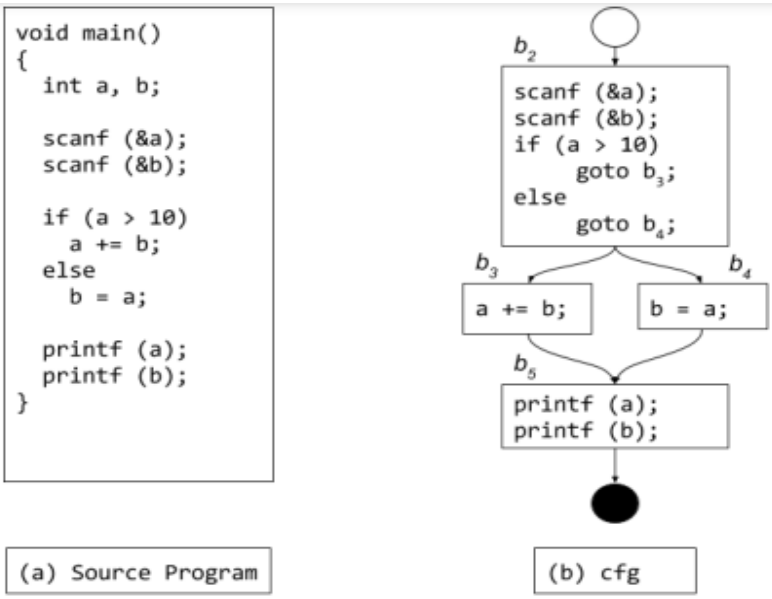


Figure 2.1: An example C program and the corresponding PRES+.

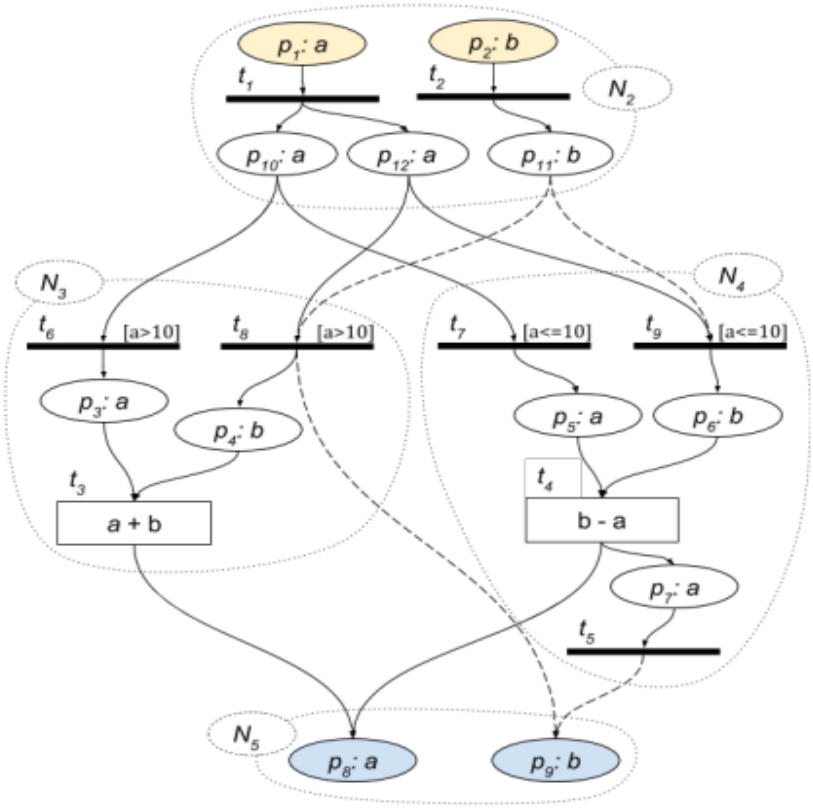
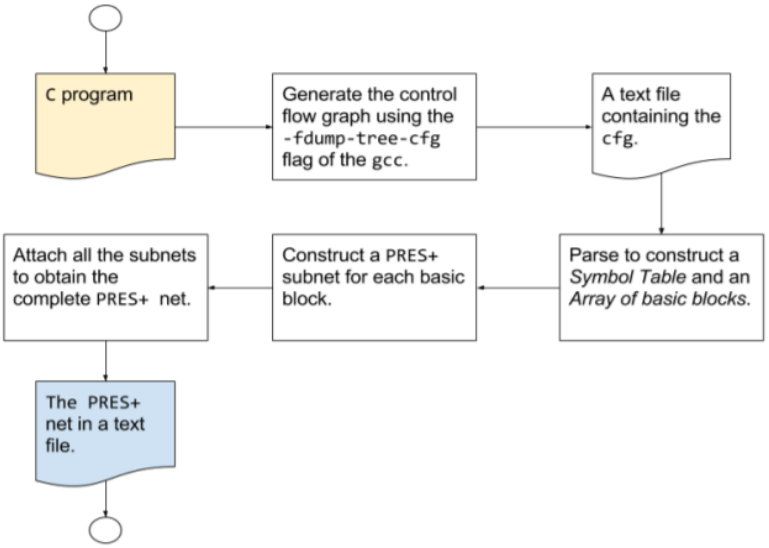


Figure 2.2: The PRES+ model for the program shown in the figure 2.1.

Figure 2.1(A) and (B) show a C program and the corresponding control flow graph (cfg). The program contains one if-else statement and accordingly the cfg contains four basic blocks. The corresponding PRES+ model is shown in figure 2.2. The PRES+ model has two input places namely, p_1 and p_2. The transitions t_1 and t_2 are identity transitions and forward the token present in the first (zero-th) pre-place. The condition c in the if clause is captured using guarded transitions t_5 to t_8. The transitions t_5 and t_6 represent the condition c being true and transitions t_7 and t_8 correspond to the negation of the condition c being true. You may note that the transition t_8 has no post-place as it serves the purpose of removing tokens in the places p_9 and p_{10} when the negation of the condition c (i.e. the condition $a \leq 10$) is true. If the condition c is true, then the transition t_3 is executed. Otherwise, the transition t_4 is executed. Eventually, the output is produced in the output tokens in the places p_6 and p_7. Light yellow and light blue color is used for input and output places of the PRES+ respectively; uncolored (white) places represent intermediate places in the PRES+. Dark black transitions are identity transitions, forwarding the token in the first pre-place. The transitions having an expression associated with them are represented as rectangular boxes with the associated expression specified inside the box. The guard condition associated with a transition is written in square brackets ([]).

Implementation :

The block diagram for the construction of a PRES+ net from a C program :-



From a given C file, first the corresponding control flow graph (cfg) is obtained using the -fdump-tree-cfg flag of the gcc. A cfg consists of three-address code with basic block boundaries and is written into a text (.cfg) file by the gcc. The cfg file is first parsed using Bison and Flex to obtain a symbol table and an array of basic blocks. The given PRES+ construction methodology is able to construct PRES+ models for programs containing loops and arrays. The model construction mechanism consists in constructing first the PRES+ models for all the basic blocks in isolation; accordingly, they appear as disconnected subgraphs, referred to as subnets; subsequently, they are attached to each other to obtain the entire digraph for the PRES+ model of the complete program.

Any input C program should satisfy the following conditions:

- Only integer variables and arrays must be used.
- Every scanf() and printf() call must read and write exactly one integer variable respectively and there must not be any other function calls.
- The program must contain exactly one function i.e. main() with void as its return type.
- The program must not contain any goto statements (although the intermediate cfg may contain such statements).

The PRES+ construction module produces the PRES+ net for a given C program in following formats:

- An xml file which can be used as input to the CPN Tool
- An image of the constructed PRES+ in png format. The image is produced using the Dot tool.
- A text file containing the PRES+ represented by a grammar. Example:

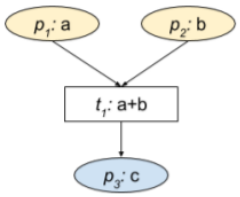
```
pvmap:
  1 a;
  2 b;
  3 c;

transitions:
  {
     1;
     Expr: a + b;
     Guard: ;
     Priority: NORMAL;
     Preset: 1, 2;
     Postset: 3;
  }

Input:
  1, 2;

Output:
  3;
```

a) Representation of a PRES+ in a text file.



b) The corresponding PRES+.