Construction of Petri net based models for C programs

Kulwant Singh

Construction of Petri net based models for C programs

Thesis submitted to
Indian Institute of Technology, Kharagpur
for the award of the degree

of

MASTER OF TECHNOLOGY
(2014-16)

by

Kulwant Singh

14CS60R20

under the guidance of

Prof. Dipankar Sarkar

Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

May 2016

DECLARATION

I certify that

a. the work contained in this report is original and has been done by me under the guidance of my supervisor.

b. the work has not been submitted to any other institute for any degree or diploma.

c. I have followed the guidelines provided by the institute in preparing the report.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, figure, and text) from other sources, I have given due credit to them by citing them in the text of the report and giving their details in the references.

Kulwant Singh

CERTIFICATE

This is to certify that the project report entitled *"Construction of Petri net based models for C programs"*, submitted by *Kulwant Singh (14CS60R20)*, of the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India, for the award of the degree of *Master of Technology*, is a record of an original research work carried out by him under my supervision and guidance. The report fulfills all the requirements as per the regulations of this institute. Neither this report nor any part of it has been submitted for any degree or academic award elsewhere to the best of my knowledge .

Prof. Dipankar Sarkar
Department of CSE
IIT Kharagpur

Acknowledgments

Though only my name appears on the cover of this dissertation, a great many people have contributed to its completion. I owe my gratitude to all those people who have made this dissertation possible and because of whom my graduate experience has been one that I will cherish forever.

My deepest gratitude is to my adviser, Prof. Dipankar Sarkar. His patience, immense knowledge and support helped me finish this dissertation. I feel fortunate to have him as my adviser.

Besides my adviser, I would like to thank Prof. Chittaranjan Mandal for his guidance at various junctures.

I would also like to thank Soumyadip Bandyopadhyay, Ph.D. student under the supervision of my adviser, for his guidance and help.

A special thanks to my family. Their support and prayers for me has sustained me thus far. I would also like to thank all my friends who supported me during the course of this degree.

Kulwant Singh

Abstract

Compilers carry out extensive optimizing transformations on the source programs exploiting the data independence of operations. Their validation is achieved by establishing behavioural equivalence between the source and the transformed programs. Models capturing data independence of operations by parallelism are most suitable for the purpose. Accordingly, Petri net based models of programs are more suitable then *cfg*s (Control Flow Graphs) like `FSMD`s (Finite State Machines with Datapaths) and `CSP`s (Concurrent Sequential Processes). If suitably constructed, the Petri net based models of the source and the transformed programs often become structurally similar thereby making the task of establishing equivalence between them easier. Experience with equivalence checking using such a Petri net based models, called `PRES+` (Petri net based Representation of Embedded Systems), have been encouraging. No tools for automated construction of the `PRES+` models from high level language programs are available. In this paper we present a tool for the automated construction of `PRES+` models from `C` programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Conversion or translation of a program to another program is a process that is performed very often in a compiler. To verify that two programs are equivalent the conventional method is to show that the algorithms yield identical results for all possible inputs. Proving correctness of programs is extremely complex and undecidable in general. Hence, instead of proving that the compiler always produces a target code which correctly implements the source code (compiler verification), each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source program. Even if a compiler is verified with suitable abstraction and (or) modularisation, every change in the compiler (even minor revisions) requires redoing the proof; thus, compiler verification tends to stall the compiler design and discourages improvements and revisions. This drawback is avoided in the translation validation approach as it compares the input and the output of the compiler for each individual run independently of how the output is generated from the input.

A fully automatic translation validation process requires a common semantic framework for the representation of the source code and the generated target code. Models capturing data independence of operations by parallelism are most suitable for the purpose because they become structurally similar. As a result, the Petri net based models are more suitable then *cfg*s (Control Flow Graphs) like `FSMD`s (Finite State Machines with Datapaths) and `CSP`s (Concurrent Sequential Processes). 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 [2]. The mechanism uses an extension of the Petri net based models called *Petri net based Representation of Embedded Systems* abbreviated as `PRES+`. No open source tool or mechanism is available for automated construction of `PRES+` models from high level language programs.

1.1 Aim of the work

Aim of the thesis is to develop methods for automated construction of the `PRES+` models for `C` programs. In the thesis `PRES+` model and `PRES+` net are used synonymously.

1.2 Organization of the thesis

The mechanism for construction of the PRES+ model for a C program is explained in the chapter 3. The explanation covers the data structures used for representing the input C programs and the PRES+ models, the mechanism for constructing PRES+ for programs without loops and arrays, enhancement of the mechanism to handle loops and arrays and finally, the various optimizations performed on the PRES+ in that order. In chapter 4 a summary of the experimental results is provided followed by conclusion and future work in chapter 5. The algorithms explained in the chapter 3 are summarized in the appendix A

Chapter 2

PRES+ model

In this chapter first a definition of the PRES+ (Petri net based Representation of Embedded Systems) model is provided. Next, an example of a PRES+ model is given.

2.1 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 f_{pv} 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`. Each transition operates on input tokens and produces zero or more output tokens.

2.2 An example of PRES+ model

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 the 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

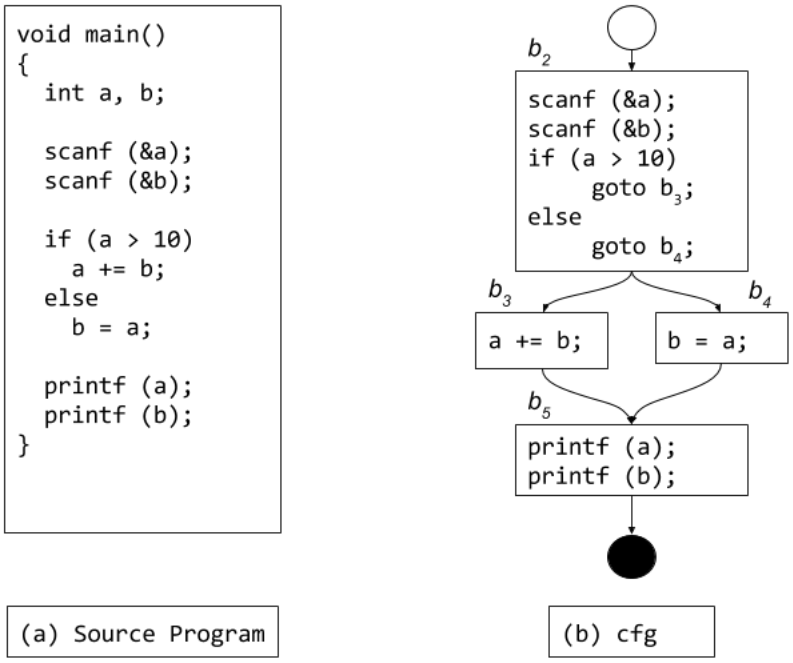

Figure 2.1: An example C program and the corresponding PRES+.

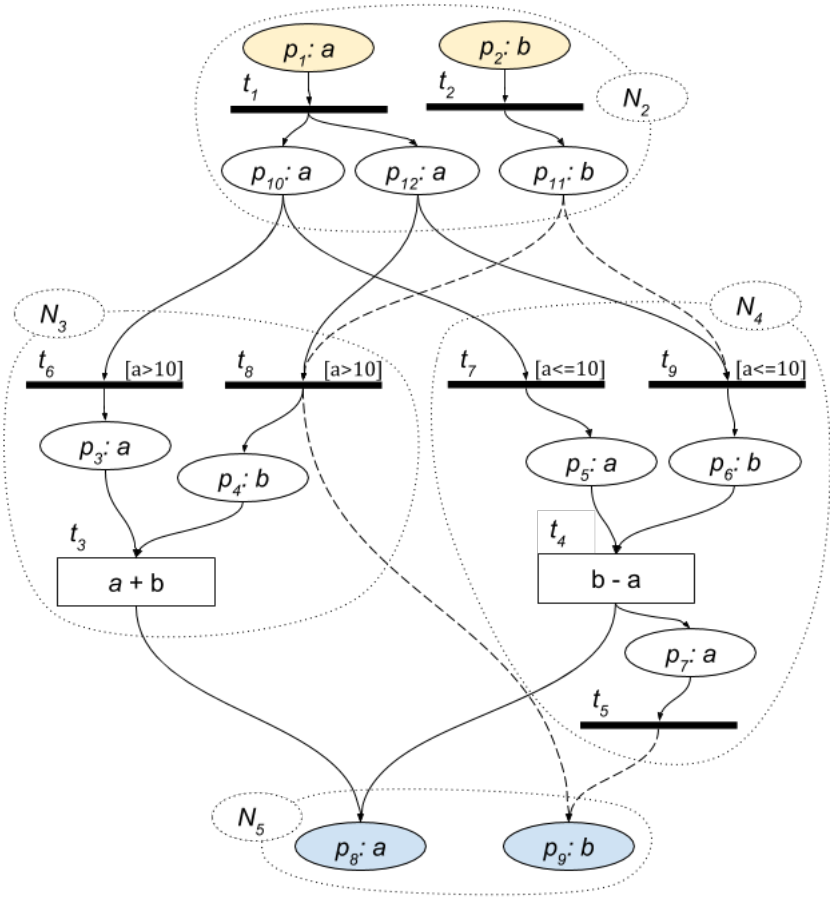

Figure 2.2: The PRES+ model for the program shown in the figure 2.1.

$a \leq 10$) 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 are produced in the places p_6 and p_7.

In the given and the following examples in the thesis, *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 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 (`[]`).

Chapter 3

Implementation

The block diagram for the construction of a PRES+ net is shown in figure 3.1. The input to the whole module is a C program and the output is the corresponding PRES+ net in a text file; the specifications of the input and the output files are given in the sections 3.1 and 3.2 respectively.

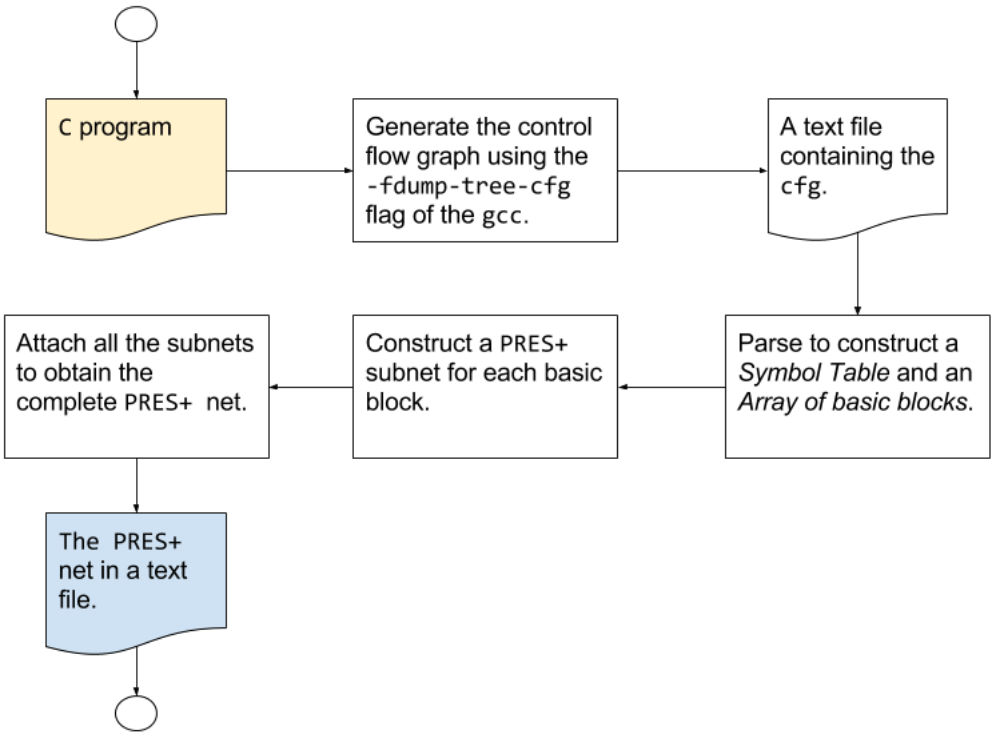

Figure 3.1: PRES+ construction block diagram

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 two data structures are explained in the section 3.3.

The given PRES+ construction methodology is able to construct PRES+ models for programs containing loops and arrays. The mechanism for constructing a subnet corresponding to a basic block is explained in section 3.5.1 followed by the mechanism for attaching the subnets to obtain the overall PRES+ in the section

3.5.2. To start with, the discussion is restricted to the simple cases of basic blocks involving only assignment and conditional statements without any loops. The subnet construction and attach mechanism is enhanced for programs involving loops and arrays in sections 3.6 and 3.7 respectively.

The model construction mechanism consists in constructing first the PRES+ models for all the basic blocks in isolation; accordingly, they appear as disconnected sub-graphs, referred to as subnets; subsequently, they are attached to each other to obtain the entire digraph for the PRES+ model of the complete program.

3.1 Input specification

Any input `C` program should satisfy the following conditions:

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

3.2 Output specification

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

1. An `xml` file which can be used as input to the `CPN Tool` [1]. The extension of the file is `cpn` in accordance to the `CPN Tool`.
2. An image of the constructed `PRES+` in `png` format. The image is produced using the `Dot` tool.
3. A text file containing the `PRES+` in the format represented by the grammar given below.

   Grammar:

```
S → PVMAP pvmap TRANS transitions
    INPUT inputPlaces OUTPUT outputPlaces
  ;
/* Where PVMAP, TRANS, INPUT and OUTPUT are keywords
 * "pvmap:", "Transitions:", "Input:" and "Output:"
 * respectively.*/

pvmap → pvmap PLACE_INDEX VARIABLE SEMI_COLON
```

[1] www.cpntools.org

```
        | PLACE_INDEX VARIABLE SEMI_COLON
        ;
/* Where PLACE_INDEX, VARIABLE and SEMI_COLON are an
 * integer (id of the place), a string (name of the
 * variable) and the character ';' respectively.*/

transition → transition OP_CUR_BR
               TRANS_INDEX  SEMI_COLON
               EXPR         SEMI_COLON
               GUARD        SEMI_COLON
               PRIORITY     SEMI_COLON
               preset       SEMI_COLON
               postset      SEMI_COLON
               CL_CUR_BR
           |   OP_CUR_BR
               TRANS_INDEX  SEMI_COLON
               EXPR         SEMI_COLON
               GUARD        SEMI_COLON
               PRIORITY     SEMI_COLON
               preset       SEMI_COLON
               postset      SEMI_COLON
               CL_CUR_BR
           ;
/* OP_CUR_BR and CL_CUR_BR are '{' and '}' respectively
   .
 * TRANS_INDEX is an integer (id of the transition).
 * EXPR and GUARD are the transition function and the
 * guard associated to the transition respectively;
 * PRIORITY is either ''HIGH'' or ''NORMAL'';
 * needed by the model constructor for simulation in
 * the CPN Tool as explained later. */

preset → preset PLACE_INDEX COMMA
       |  PLACE_INDEX SEMI_COLON
       ;
/* Where COMMA is ','.*/

postset → postset PLACE_INDEX COMMA
        |  PLACE_INDEX SEMI_COLON
        ;

inputPlaces → inputPlaces PLACE_INDEX COMMA
            |  PLACE_INDEX SEMI_COLON
            ;

outputPlaces → outputPlaces PLACE_INDEX COMMA
             |  PLACE_INDEX SEMI_COLON
             ;
```

Grammar 3.1: Grammar for the output text file containing a PRES+ net

Example: 3.2.1

Figure 3.2(a) shows a text file containing the encoding for the PRES+ net shown in the figure 3.2(b). First all the places and the associated variables are written delimited by semicolons. Then all the transitions of the PRES+ are given. In the end of the file, the input and output places of the PRES+ are given. □

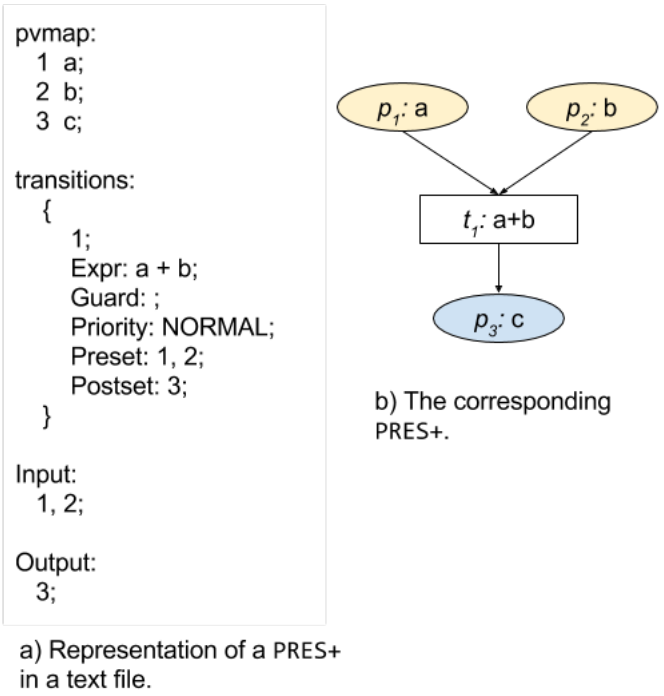

Figure 3.2: Illustration of the encoding of a PRES+ in a text file.

3.3 Data Structures for the control flow graph

Figure 3.3 illustrates a C program and the corresponding control flow graph (cfg) produced by the gcc. As observed from the figure, a cfg is a collection of *basic blocks* with *goto* statements governing the control flow from one basic block to another basic block. All the variables used in the cfg are listed at the beginning of the cfg, before the very first basic block.

This section explains the two data structures used to store a cfg. The first data structure is Symbol Table; it stores all the variables declared at the beginning of a cfg. The second data structure is an array of basic blocks; an element of this array stores a basic block. Following is an explanation of these two data structures.

3.3.1 Symbol Table

It has the following attributes:

1. total_symbols: Total number of symbols/variables used in the cfg.

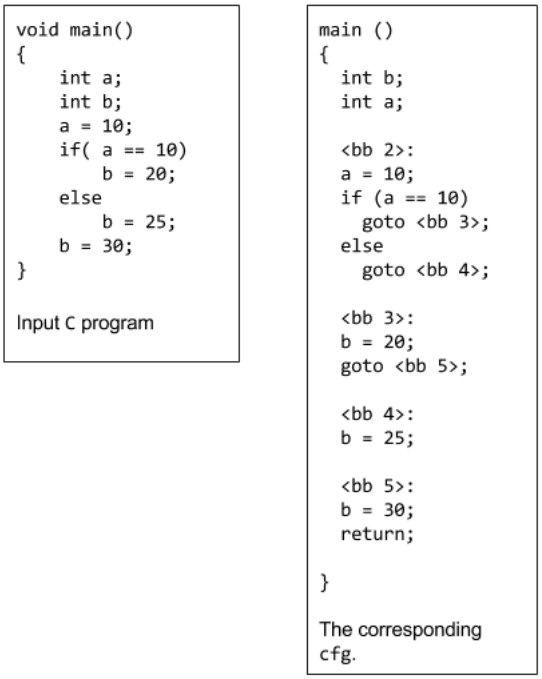

Figure 3.3: A `C` program and the corresponding `cfg`.

2. `symbols_array[MAX_VARIABLES]` An array of size `MAX_VARIABLES` where each element contains:
   - `var_name`: Name of the variable as a string.
   - `def:` A bit vector of size `MAX_STATEMENTs` (maximum statements in a *cfg*) in which i^{th} bit is set iff the statement with *unique index* (explained in section 3.6.6) i defines the variable. This bit vector is used in *live variable analysis* as explained in section 3.6.4.
   - `dimension:` An integer to store the dimension of an array, if the symbol represents an array. Otherwise, it is assigned -1 to indicate that the symbol does not represent an array.

3.3.2 `BB_array` (Array of basic blocks)

It has the following attributes:

1. `total_BBs:` Total number of basic blocks stored in the basic blocks array.
2. `BB[MAX_BBs]:` An array of size `MAX_BBs` in which each element stores a basic block. This array is of type `Basic_block`. Description of the `Basic_block` data structure follows.

3.3.3 Basic_block

This data structure represents the contents of a basic block and has the following attributes:

1. `type:` Type of a basic block `i.e. NORMAL, CONDITIONAL` or `LAST`.

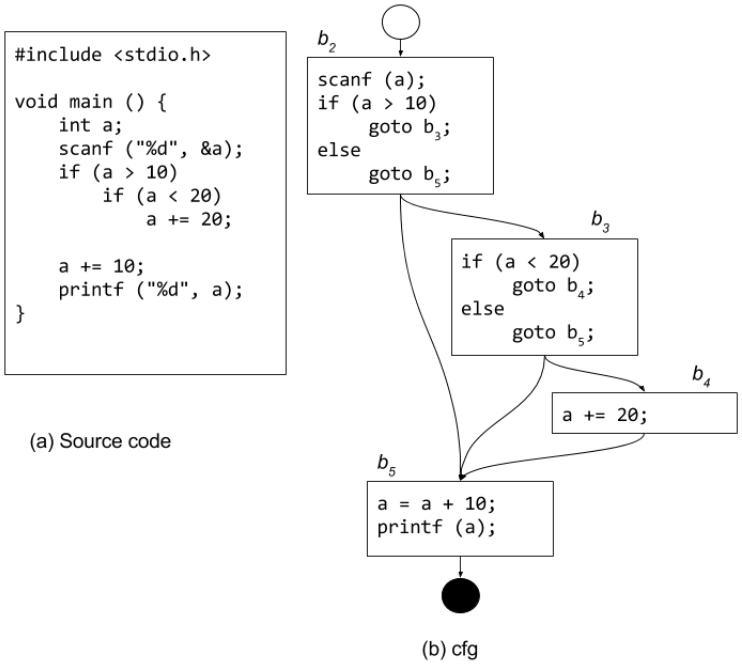

Figure 3.4: A `C` program (a) and the corresponding `cfg` (b).

`NORMAL, CONDITIONAL` and `LAST` are basic blocks with *one, two* and *zero* successor(s) respectively. A basic block can have at most one `if-then-else` statement and that too as the last statement. As a result a basic block can have either zero, one or two successor(s).

Example: 3.3.1

The cfg for the program shown in figure 3.4(a) is shown in the figure 3.4(b). In the cfg the basic blocks b_2 and b_3 have type `Conditional`; Basic blocks b_4 and b_5 have types `Normal` and `Last` respectively. □

2. `pred_count:` Number of the predecessors of a basic block. For a program having n basic blocks, the number of predecessors can be anything in the range $[0, n-1]$; the first basic block of a program has zero predecessors.

   Example: 3.3.2

   Figure 3.4(b) is the cfg corresponding to the program shown in the figure 3.4(a). In the cfg the basic block b_5 has three $(n-1)$ predecessors. □

3. `predecessors[MAX_PREDECESSORs]:` An integer array of size `MAX_PREDECESSORs` to store the indices of the predecessors of a basic block.

4. `stm_count:` Number of statements in a basic block, excluding the `if-else` statement in `CONDITIONAL` basic blocks.

5. `statements[MAX_STATEMENTs]:` An array of type `Statement` and size `MAX_STATEMENTs` to store the statements of a basic block. The `Statement` data structure is explained in the section 3.3.4.

6. `cond:` A pointer to the expression tree of the expression in the `if-else` statement of a conditional basic block. Expression tress are used to store expressions as explained in section 3.3.5.

7. `visited:` A boolean flag for DFS and (or) BFS, as and when needed.

8. `true_successor:` Index of the unique successor for a `NORMAL` basic block and true successor for a `CONDITIONAL` basic block. `-1` for the `LAST` basic block.

9. `false_successor:` Index of the false successor for a `CONDITIONAL` basic block and `-1` for `NORMAL` and `LAST` basic blocks.

10. `live_in:` A bit vector to store the variables *live* at the beginning of a basic block. This bit vector is obtained from *live variable analysis* (*lva*) as explained in section 3.6.4. The need for *lva* is established in the section 3.6.2. i^{th} bit of this bit vector is set iff the variable at index i in the symbol table is live at the beginning of a basic block.

11. `live_out:` A bit vector similar to `live_in` but stores the variables *live* at the end of a basic block.

12. `rd_in:` A bit vector to store the *reaching definitions* at the beginning of a basic block. The role of the *reaching definitions* is explained in section 3.6.3 and the mechanism itself is explained in section 3.6.6. i^{th} bit of this bit vector is set iff the statement with *unique index* (explained in section 3.6.6) i is a *reaching definition* at the beginning of a basic block.

13. `rd_out:` A bit vector similar to *rd_in*$_b$ut stores the *reaching definition* at the end of a basic block.

3.3.4 Statement

This data structure is used to store the attributes of a statement in a basic block. Its attributes are:

1. `type:` Type of a statement `i.e. scanf, printf,` $\texttt{Asng}_{\texttt{id}}$`,` $\texttt{Asgn}_{\texttt{unary}}$ or $\texttt{Asgn}_{\texttt{binary}}$

   `scanf:` is a statement corresponding to a `scanf()` function call.

   `printf:` is a statement corresponding to a `printf()` function call.

   $\texttt{Asgn}_{\texttt{id}}$: An assignment statement which assigns the value of a variable to another variable without any operation, for `e.g.` $a = b$;.

   $\texttt{Asgn}_{\texttt{unary}}$: An assignment statement which has a literal or a unary operator associated to a variable on its right hand side (RHS). Examples of such statements are $a = -b$;, $a = 10$; etc.

   $\texttt{Asgn}_{\texttt{binary}}$: An assignment statement with an expression involving a binary operator on its RHS. Examples of such statements are $a = b + c$;, $a = b - 10$; etc.

2. `var:` Index (to the Symbol Table) of the variable defined in the `scanf` and the `assignment` statements. Index of the output variable for the `printf` statements.

3. `expr:` Pointer to the expression tree on the RHS of an `assignment` statement. `NULL` for the `scanf` and `printf` statements.

3.3.5 Expression Tree

It is a binary tree used to store the expression on the RHS of an assignment statement. Expression trees for various statements are shown in the figure 3.5. The attributes of a node of an expression tree are:

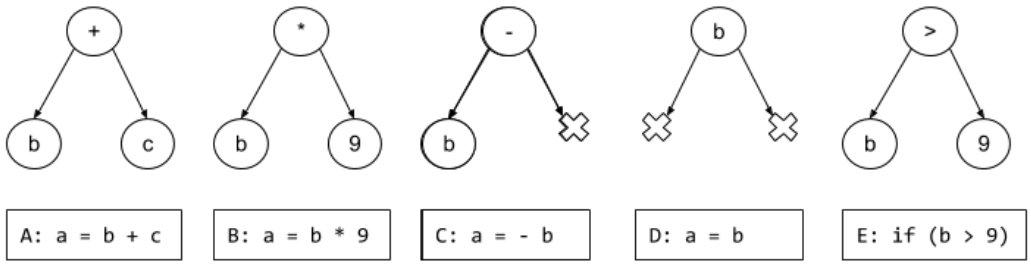

Figure 3.5: Expression trees for various types of statements.

1. `type:` Type of a node `i.e. Binary_operator, Unary_operator, Variable` or `Literal`.

   `Binary_operator:` Root node of an expression tree representing a binary operator. Left and right operands are stored in the left and the right child respectively.

   `Unary_operator:` Root node of an expression tree representing a unary operator. Left operand is stored in the left child and the right child is `NULL`.

   `Variable:` A node containing index (to the Symbol Table) of a variable.

   `Literal:` A node containing value of an integer literal, since only integers are allowed.

2. `identifier:` Depending on the `type` field, this attribute specifies an operator $\in \{+, -, *, /, \%, <, \leq, >, \geq, =, ==\}$ (for type `Binary_ operator` and `Unary_operator`), index of a variable (for type `Variable`) or a constant value (for type `Literal`).

3. `left:` Pointer to the node corresponding to the left operand.

4. `right:` Pointer to the node corresponding to the right operand.

3.4 Data Structures for the PRES+ model

In this section the data structures used to represent the overall PRES+ model and the PRES+ model corresponding to each basic block are explained.

3.4.1 Pres_Plus

It is the data structure used to store the overall PRES+ net. Its various attributes are:

1. `places_count:` An integer to store the total number of places in the PRES+.
2. `places[MAX_PLACEs]:` An array of type `Place` and size `MAX_PLACEs` used to store the places of the PRES+. The data type `Place` is explained subsequently.
3. `trans_count:` An integer to store the total number of transitions in the PRES+.
4. `trans[MAX_TRANSITIONs]:` An array of type `Transition` and size `MAX_TRANSITIONs` used to store the transitions of the PRES+. The data type `Transition` is explained subsequently.
5. `input_places_count:` An integer to store the total number of the input places of the PRES+.
6. `input_places[MAX_PLACEs]:` An integer array of size `MAX_PLACEs` to store the indices to the array `places[]` of the input places of the PRES+.
7. `output_places_count:` An integer to store the total number of the output places of the PRES+.
8. `output_places[MAX_PLACEs]:` An integer array of size `MAX_PLACEs` to store the indices to the array `places[]` of the output places of the PRES+.

3.4.2 Place

This data structure is used to represent a place in the PRES+. Its various attributes are:

1. `type:` The type of a place - one of {`Input_port, Output_port, Var, Dummy`}

   `Input_port` is a place which takes input tokens for a PRES+.

   `Output_port` is a place which provides output tokens of a PRES+.

   `Var` is an intermediate place which provides flow of tokens between transitions.

   `Dummy` is also an intermediate place providing flow of tokens between transitions. These places are used for synchronization and have no variables associated with them.

   `Input_port`, `Output_port` and `Var`, all have a variable associated with them.

2. `var:` If the type is `Input_port`, `Output_port` or `Var`, then it is the index to the Symbol Table of the variable associated with the place; if the place type is `Dummy`, then it is -1.

3. `preset_count:` Stores the total number of pre-transitions of a place.

4. `preset[MAX_PRESET_PLACE]:` Stores the indices of the pre-transitions of the place to the array `transitions[]` of all transitions of the `PRES+`.

5. `postset_count:` Stores the total number of post-transitions of a place.

6. `preset[MAX_POSTSET_PLACE]:` Stores the indices of the post-transitions of the place to the array `transitions[]` of all transitions of the `PRES+`.

3.4.3 Transition

This data structure is used to represent a transition in the `PRES+`. It has the following attributes:

1. `type:` Type of a transition - one of {`Identity, Expression`}.

   `Identity` transitions represent the variable copy function `i.e.` $a = b$. They forward the token associated with the *first* pre-place (with index value 0). They do not have any expression associated with them.

   `Expression` transitions perform some operation on the input tokens before producing output and have an expression tree associated with them. The transition corresponding to the statement $a = b + c$; belongs to this category.

2. `expr:` Pointer to the expression tree associated with the `Expression` transitions. `NULL` for the `Identity` transitions.

3. `guard:` If the transition has a guard condition associated with it, then it is a pointer to the expression tree representing the guard condition associated with the transition; if the transition does not has a guard condition, then it is `NULL`.

4. `preset_count:` Stores the number of pre-places of the transition.

5. `preset[MAX_PRESET_TRANS]:` Stores the indices of the pre-places of the transition to the array `places[]` of all the places of the `PRES+`.

6. `postset_count:` Stores the number of post-places of the transition.

7. `postset[MAX_POSTSET_TRANS]:` Stores the indices of the post-places of the transition to the array `places[]` of all the places of the `PRES+`.

3.4.4 Subnets[MAX_BBs]

An array to identify the PRES+ subnet corresponding to a basic block in the overall PRES+ net. An element of the array, Subnet[i], has the following attributes and identifies the subnet corresponding to the i^{th} basic block in the basic blocks array.

1. places_count: Number of places of the i^{th} subnet.
2. places[MAX_PLACEs]: Stores the indices of the places of the i^{th} subnet to the array places[] of all the places of the overall PRES+.
3. transitions_count: Number of transitions of the i^{th} subnet.
4. transitions[MAX_TRANSITIONs]: Stores the indices of the transitions of the i^{th} subnet to the array transitions[] of all the transitions of the overall PRES+.
5. input_places[MAX_PLACEs]: Stores the indices of the the *local input* places of the i^{th} subnet to the array places[] of all the places of the overall PRES+.
6. input_places_count: Number of the input places in the array input_places[] described above.
7. ldt[MAX_VARIABLEs]: An array to store the *latest defining transitions* in the i^{th} subnet of the variables of the variables occurring in the subnet. The entry ldt[v] is the transition in the i^{th} subnet which provides the latest token for the variable v. If a variable v is not defined in the i^{th} subnet then the ldt[v] is -1 and if v is defined twice then the ldt[v] is the transition generating the latest token.
8. maximum_transition: An integer to store the *maximum* transition of the i^{th} subnet. The role of the *maximum* transitions is explained in section 3.6.2.
9. maximum_tr_sync_pl: An integer to store the index of the place to the array places[] in the overall PRES+ used to synchronize the maximum transition of the i^{th} subnet with that of its predecessors. The type of this place is Dummy.
10. maximal_trans_count: Stores the number of maximal transitions of a subnet.
11. maximal_trans[MAX_TRANSITIONs]: Stores the indices of the *maximal* transitions of the i^{th} subnet to the array transitions[] of all the transitions of the overall PRES+. The role of *maximal transitions* is explained in section 3.6.2.
12. loop: A boolean variable having a value True if the i^{th} subnet belongs to a loop body; False otherwise.
13. interface_variables[MAX_VARIABLEs]: An integer array of size MAX_VARIABLEs; it stores the indices of places to the array places[] in the overall PRES+. *Interface places* are Dummy places used in successors of Conditional subnets in loop bodies to capture the conditional execution of such subnets; they are further explained in the section 3.6.2. An element v of this array is -1 if there exists no interface place in the i^{th} subnet for the variable v.

3.5 Construction of PRES+ for programs without loops

In this section the construction of PRES+ models for C programs containing only input (scanf), output (printf), assignment and if-else statements i.e., for programs free from loops and arrays is explained. The model construction mechanism consists of constructing first the PRES+ models for all the basic blocks in isolation; accordingly, they appear as disconnected sub-graphs, referred to as subnets; subsequently, they are attached to each other to obtain the entire digraph for the PRES+ model of the complete program.

3.5.1 Construction of a PRES+ subnet

This section explains the construction of a PRES+ subnet for a basic block. The approach is to construct the respective places and transitions for each statement of the basic block in the overall PRES+ itself. The statements of the basic block are processed in the order of their appearance in the basic block and the constructed places and transitions are then attached to the PRES+ subnet corresponding to the preceding statements of the basic block but not to the overall PRES+ net.

Construction of the PRES+ for a basic block consists of constructing the PRES+ subnet for each statement in it followed by its attachment to the already constructed PRES+ subnet constructed for the preceding statements in the basic block. The given procedure repeated for all the statements of a basic block in the order of their occurrence constructs the PRES+ subnet for a basic block.

Note that although processed sequentially, the step of attaching the newly created PRES+ subnet ensures that sequentiality is incorporated only for read after write (RAW) dependencies; thus, even for a sequence of assignment statements the corresponding subnet may have disconnected components depicting parallelism (due to dependence) among the components and sequentiality only inside each component. The inputs and outputs of this module are:

Inputs:

1. bba: A pointer to the array of basic blocks. Recall that BB_Array is a structure containing the array of basic blocks along with a count of the total number of basic blocks in the cfg (control flow graph).
2. st: A pointer to the symbol table.
3. PP: A pointer to the structure Pres_Plus representing the overall PRES+.
4. Subnets[]: A pointer to the subnets array.
5. *k:* Index of the present basic block for which the PRES+ subnet should be constructed to the basic blocks array.

Outputs:

1. The overall PRES+ net PP is modified to contain the PRES+ subnet corresponding to the given basic block.

2. The k^{th} element of the `Subnets[]` array is modified to identify the subnet corresponding to the k^{th} basic block in the overall `PRES+` net `PP`.

Construction of the `PRES+` subnet for each statement depends on the type of the statement and is explained below.

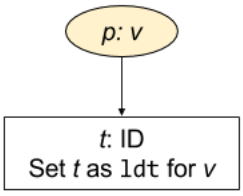

Figure 3.6: `PRES+` for a `scanf` statement

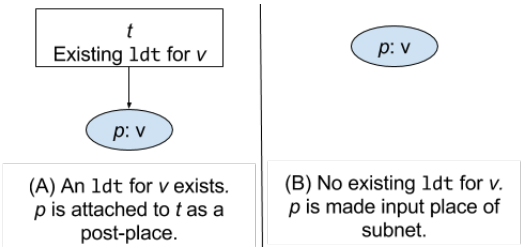

Figure 3.7: `PRES+` of a `printf` statement

`scanf` statement

For a `scanf` statement reading a variable v two places p and p' are constructed and assigned the types `Input_port` and `Dummy` respectively. The place p is assigned v as the associated variable and added to the `input_places` array of the overall `PRES+ PP`; the place p' is added to the `input_places` array of the k^{th} subnet (`Subnets[k]`). The purpose of the `Dummy` place p' is to provide synchronization if the `scanf` statement belongs to an `if-then-else` clause.

An `identity` transition t is constructed and attached to the places p and p' as a post-transition in that order; t is set as the latest defining transition (`ldt`) of the variable v in `Subnets[k]`. The places p and p' are added to the `places` array of the `Subnets[k]`; the transition t is added to the `transitions` array of `Subnets[k]`. The `PRES+` for a `scanf` statement is illustrated in figure 3.6. It may be noted that *output places* of any transition are constructed only when the corresponding value produced by the transition is used subsequently.

`printf` statement

For a `printf` statement printing a variable v a place p is constructed, assigned the type `Output_port` and v as the associated variable. The place p is added to the `output_places` array of the overall `PRES+ PP`. If in `Subnets[k]` an `ldt` exists for the printed variable then p is attached to the `ldt` as a post-place, as shown in figure 3.7(A); otherwise p is added to the `input_places` array of the `Subnets[k]` as shown in figure 3.7(B).

$\texttt{Asgn}_{\texttt{id}}$ statement

These are statements of type $v_l = v_r$, `i.e.` the value of one variable is assigned to another variable without any operation. If an `ldt` exists for v_r in `Subnets[k]`, the variable on the right hand side, then the `ldt[`v_r`]` is set as the `ldt` for v_l also where v_l is the variable on the LHS, as shown in figure 3.8(A).

Otherwise, an `ldt` does not exist for v_r in `Subnets[k]`. A place p and an `identity` transition t are constructed. The variable v_r is variable associated

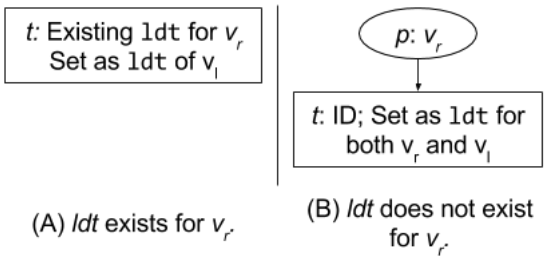

Figure 3.8: PRES+ of an identity statement

with the place p. The place p is attached to t as a pre-place and added to the input_places array of Subnets[k]. The transition t is marked as the ldt for both v_r and v_l in Subnets[k], as shown in figure 3.8(B).

$\texttt{Asgn}_{\texttt{unary}}$ statement

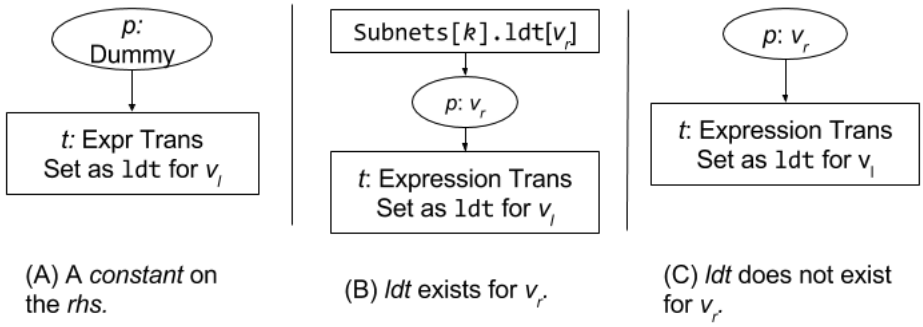

Figure 3.9: PRES+ of a unary assignment statement.

The type $\texttt{Asgn}_{\texttt{unary}}$ stands for *unary assignment* statements. Such statements are assignment statements with a literal or a unary operator on the RHS; e.g. a = 10 and a = -b. Literals 10, etc. are captured as unary constant functions because of the requirement that every transition has at least one pre-place.

A transition t of type Expression is constructed. The expression tree stored in the corresponding statement is assigned to the transition t and t is set as the ldt for v_l, the variable on the LHS. If v_r is a literal, a Dummy place p is constructed. p is joined to t as a pre-place and added to the input_places array of the Subnets[k], as shown in figure 3.9(A).

Otherwise, v_r is a variable with an associated unary operator on the RHS. A place p is constructed for the variable and attached to the transition t as a pre-place. If an ldt exists for v_r in Subnets[k], then p is attached to the ldt as a pre-place, as shown in figure 3.9(B); Otherwise, the place p is added to the input places array of the Subnets[k], as shown in figure 3.9(C).

$\texttt{Asgn}_{\texttt{binary}}$ statement

This type stands for *binary assignment* statements. Such statements are assignment statements with a binary operator and two operands on RHS, e.g. a = c

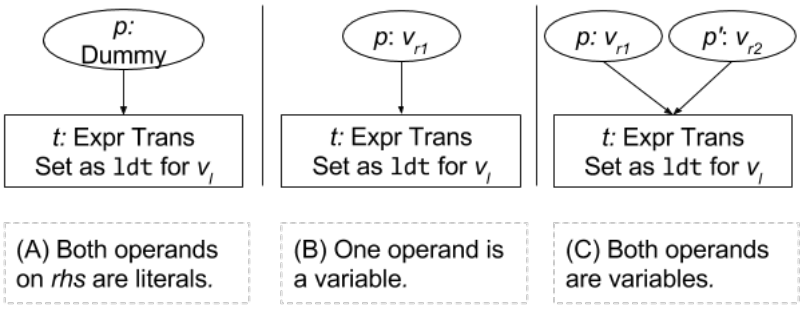

Figure 3.10: `PRES+` of `binary assignment` statement. The pre-places of the transition t are attached to their `ldt`s as post-places, if the `ldt`s exist.

`- b` and `a = 10 + 30`. The operands can be any combination of variables and literals. If both the operands are literals, a dummy place p is constructed and added to the `input_places` array of the `Subnets[k]`. The construction of the transition for the statement is explained subsequently.

Otherwise, at least one of the operands on the RHS is a variable. Places are constructed for the operands which are variables and the ones having an `ldt` in `Subnets[k]` are joined to the `ldt` as post-places; the remaining places (not having an `ldt` in `Subnets[k]`) are added to the input places array of the `Subnets[k]`.

A transition t of type `Expression` is constructed with the expression tree stored in the statement as the expression of t. The places constructed for the statement are joined to t as pre-places and the transition t is set as the `ldt` for the variable v_l, the variable on the LHS, as shown in figure 3.10.

`if-else` statement

The `if-else` statement of a `Conditional` basic block is always the last statement of the basic block and is processed while attaching the corresponding subnet to the two successors. Therefore, the procedure for handling `if-else` statements is explained along with the explanation of the methodology for attaching subnets.

The mechanism is summarized in the algorithm 1 (page 56).

3.5.2 Attach `PRES+` subnets

In this section the procedure of attaching the individual subnets corresponding to the basic blocks to obtain an overall `PRES+` net is explained. To attach the subnets, a *depth first traversal (dfs traversal)* is performed on the *cfg* and the subnets are attached to their successors while backtracking, as illustrated in the following example.

Example: 3.5.1

With respect to the *cfg* shown in the figure 3.11, the *dfs* traversal for attach begins with the subnet N_1 which, being a `Conditional` subnet, has two successors namely, N_2 and N_3; the true successor N_2 is traversed first from which its unique successor N_4 is visited. Since N_4 does not have any

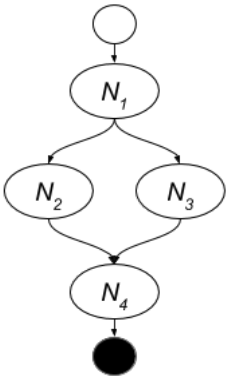

Figure 3.11: A *cfg* to illustrate the *dfs* traversal for attach.

successor, the traversal backtracks to N_2 whereupon the latter is attached to its successor N_4. Let this attached subnet be referred to as $N_2 \cdot N_4$ where the $\cdot$ (dot) operation captures the predecessor-successor relationship of the subnets involved in the attachment step. After the attachment, the traversal backtracks from N_2 to N_1. The attached subnet $N_2 \cdot N_4$ is not attached to N_1 immediately because another successor N_3 of N_1 has not yet been traversed; the *dfs* then visits N_3, the untraversed successor of N_1. From the subnet N_3, the *dfs* visits its unique successor N_4 but backtracks immediately as N_4 has already been traversed. Now N_3 is attached to $N_2 \cdot N_4$ and the traversal backtracks to N_1. It may be noted that while attaching N_3 to $N_2 \cdot N_4$, the subnet N_3 virtually gets attached to its successor N_4 and not to N_2; this fact is reflected in depicting the attached subnet as $(N_2|N_3) \cdot N_4$ underlining the feature that N_4 is the successor to both N_2 and N_3 and there is no predecessor-successor relation between N_2 and N_3. At the subnet N_1, since both of its successors have been traversed, N_1 is attached to them (depicted as $N_1 \cdot (N_2|N_3) \cdot N_4$) and the *dfs* traversal terminates. □

Recall that a basic block (and its corresponding subnet) can have either zero, one or two successors and are accordingly assigned one of the three types namely, `Last`, `Normal` and `Conditional`; the attachment procedure of a subnet to its successors is described under these three categories of the subnets. In the following sections, the attributes of a basic block such as *type*, *successor*, *predecessor*, etc., and those of its subnet are used synonymously.

During the *dfs* traversal for attaching subnets, no processing is done when the subnets of type `Last` are visited since the attachment step attaches a subnet with its successors and such subnets have no successors. For the other two types of subnets, the corresponding steps for attaching the successors are explained in the following subsections where the attachment steps are explained using the example given below.

Example: 3.5.2

Figures 3.12(a), (b) and (c) show the source program, the control flow graph of the basic blocks and the `PRES+` subnets corresponding to the individual basic blocks, respectively. There is one `Conditional` subnet N_2, two `Normal` subnets, N_3 and N_4 and one `Last` subnet N_5. In the following sections,

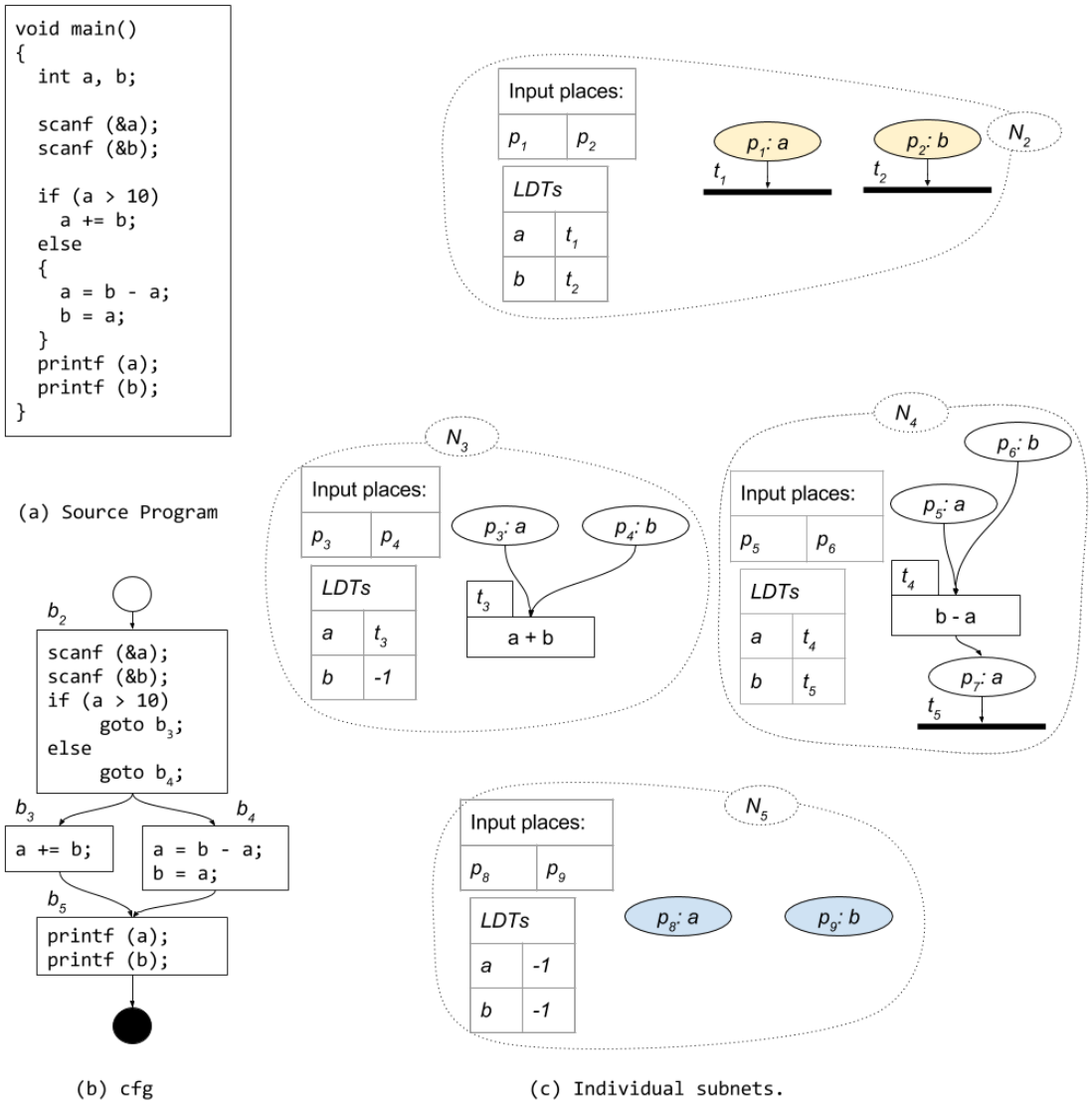

Figure 3.12: An example to illustrate the attaching of subnets

first the attachment of the subnet N_3 and N_4 to their unique successor N_5 and then the attachment of the subnet N_2 to its successors are explained. □

Attaching `Normal` subnet

To attach a `Normal` subnet N_{norm} to its unique successor N_{succ}, each input place p_{in} of N_{succ} is attached to the subnet N_{norm} based on the type of the place p_{in} as described below.

p_{in} is an `Input_port`

Recall (from section 3.5.1) that the places of the type `Input_port` are constructed only for `scanf` statements and they take input tokens for the `PRES+`. Such places are never added to the local input places array of a subnet; hence, this case does not arise.

p_{in} is of type `Var` or is an `Output_port`

Let v be the variable associated with the place p_{in}. If there exists an *ldt* (latest defining transition) for v in the subnet N_{norm}, then p_{in} is attached as a post-place to the transition. Otherwise, the place p_{in} is added to the `input_places` array of the subnet N_{norm}.

p_{in} is of type `Dummy`

As mentioned while discussing the construction methodology given in the section 3.5.1, `Dummy` places are constructed as pre-places for transitions corresponding to *constant* functions. Such transitions have no data dependence on any other transition though they may have a control dependence. However, in the present context, since the place p_{in} is being attached to a `Normal` subnet, there is no conditional control dependence. Hence, the place p_{in} is added to the `input_places` array of the subnet N_{norm}.

Example: 3.5.3

In continuation of the example 3.5.2, the present example illustrates the attachment procedure of the `Normal` subnets N_3 and N_4 to their unique successor N_5; the resultant `PRES+` is shown in the figure 3.13. To attach the `Normal` subnet N_3 to its unique successor N_5, the input places p_7 and p_8 of N_5 are considered. The variable a associated with the place p_7 has an *ldt* t_3 in the subnet N_3; hence, p_7 is attached as a post-place to t_3. Similarly, the place p_8 has b as the associated variable which does not have any ldt in N_3; as a result the place p_8 is added to the `input_places` array of the subnet N_3.

The subnet N_4 is attached to its unique successor N_5 using a similar method. The input places p_7 and p_8 of N_5 are attached as post-places to their common *ldt* t_4. □

Attaching `Conditional` subnet

To attach a `Conditional` subnet N_{cond} to its two successors — the true successor N_t and the false successor N_f — each input place p_{in} of the two successors needs to be attached to the subnet N_{cond}. The steps needed for the purpose are first described using two examples and then a generalized procedure is given.

Example: 3.5.4

In continuation of the example 3.5.2, the present example illustrates the attachment of the `Conditional` subnet N_2 to its two successors N_3 and N_4; the resultant `PRES+` obtained after attachment is shown in figure 3.14. In order to attach the conditional subnet N_2 to its successors, the input places of the successors are attached to N_2 and are processed based on the variable associated with them. In the concerned example, the set V of variables associated with the input places of the two successors consists of the variables a and b. To attach the successors N_3 and N_4 to N_2, each variable in the set V is processed one by one.

Considering the variable a first, two identity transitions t_5 and t_6 are con-

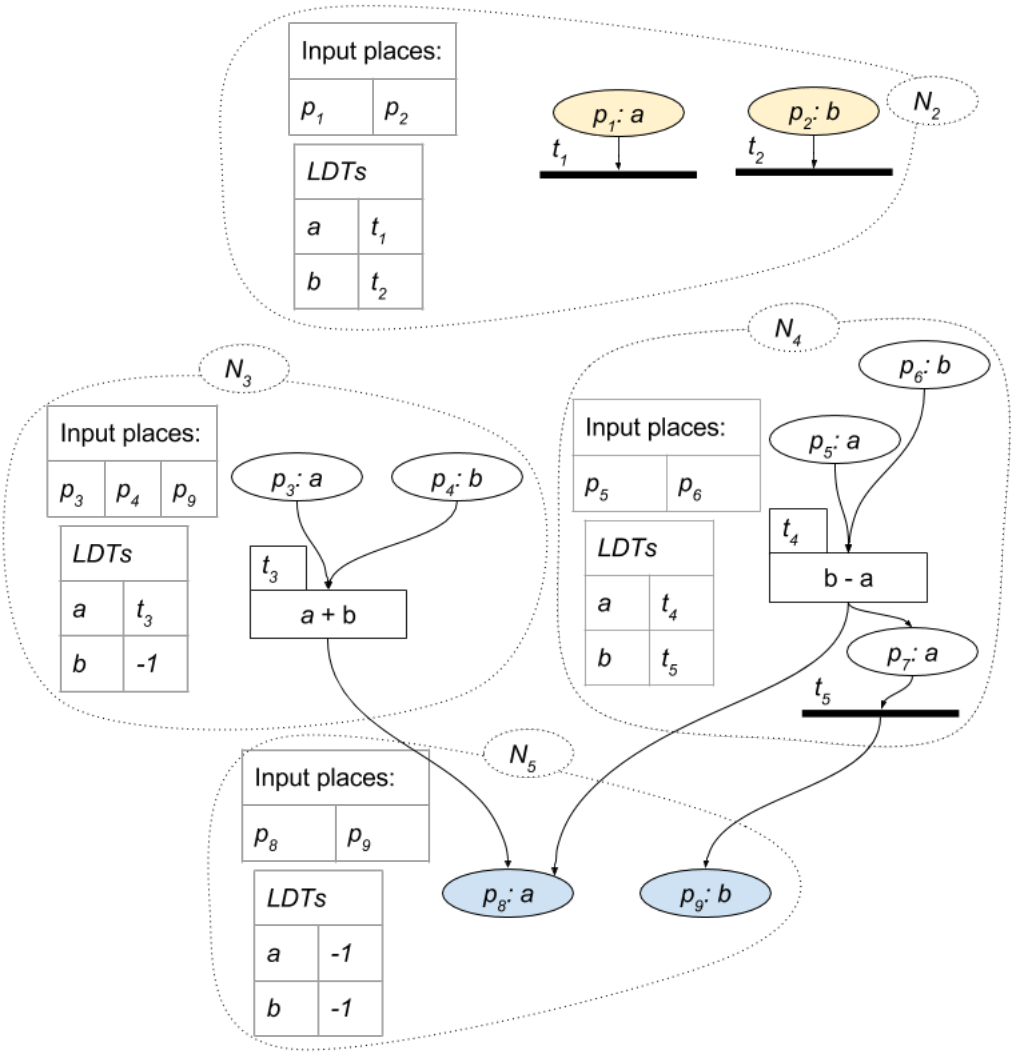

Figure 3.13: Resultant `PRES+` after attaching the `Normal` subnets to the successors for the `PRES+` shown in the figure 3.12(c).

structed. Recall that in section 3.5.1, the conditional expression c associated with the `if` clause in the subnet N_2 (or more precisely, in the corresponding basic block b_2) was ignored. The conditional expression c and its negation $\neg c$ are assigned to the transitions t_5 and t_6 as the *guards*, respectively. The input place p_3 of the true successor N_3 and p_5 of the false successor N_4 having a as the associated variable are attached as post-places to the transitions t_5 and t_6, respectively. A place p_9 for the variable a is attached to the transitions t_5 and t_6 as a common pre-place and as a post-place to the corresponding *ldt* t_1 of the variable a in the subnet N_2. Since both *transition function* and *guard* condition of the transitions t_5 and t_6 use only the variable a, no other pre-place needs to be constructed for the transitions.

For the other variable b in the set V, transitions t_7 and t_8 are constructed in a similar fashion as for the variable a. The input places p_4 and p_8 of the true successor and p_6 of the false successor having b as the associated variable are attached as post-places to the transitions t_7 and t_8, respectively. A place

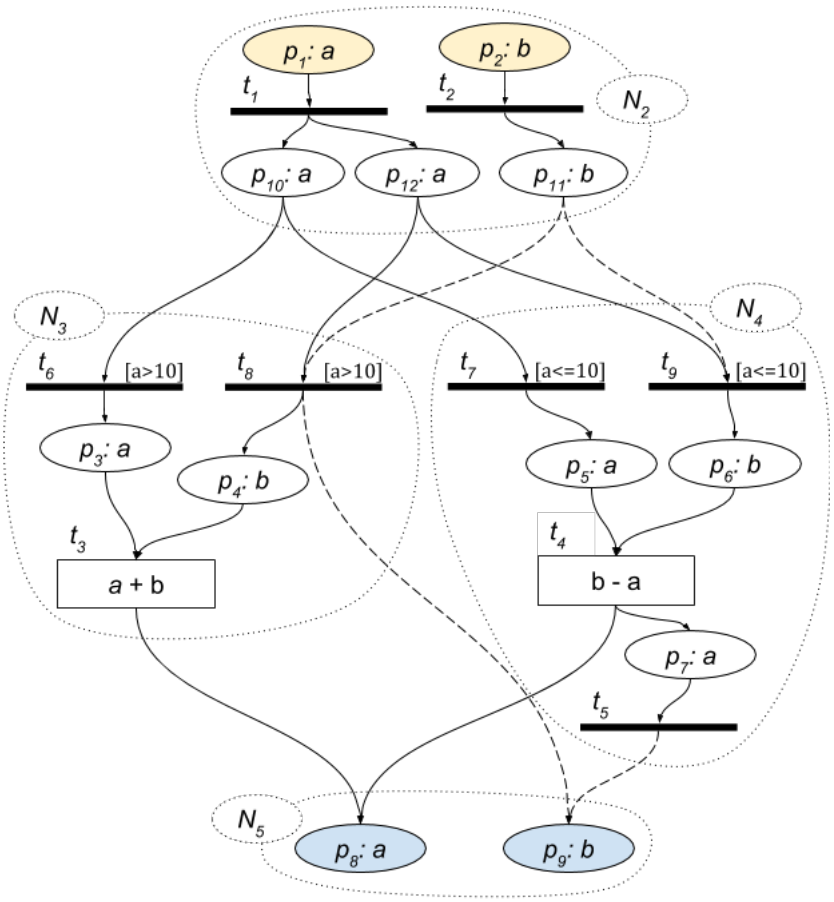

Figure 3.14: The PRES+ obtained from the one shown in the figure 3.13 after attaching the Conditional subnet to its successors.

p_{10} is constructed for the variable b and attached to the transitions t_7 and t_8 as a common pre-place. Another place p_{11} with a as the associated variable is constructed because the *guard* conditions of t_7 and t_8 use the variable and it is not used in the *transition* functions of t_7 and t_8. The place p_{11} is attached as a common pre-place to the transitions t_7 and t_8. The places p_{10} and p_{11} are attached as post-places to their respective *ldt*s t_2 and t_1 in the subnet N_2. □

In the context of the figure 3.14, a question arises — why is it necessary to create new *identity* transitions instead of adding a *guard* condition to the minimal transitions of the two successors subnets N_3 and N_4 i.e., the transitions which do not have any dependence on the transitions of N_3 and N_4. The necessity is illustrated using the following example.

Example: 3.5.5

A C program and the corresponding control flow graph are shown in figure 3.4(a) and (b) respectively. Notice the two Conditional predecessors b_2 and b_3 of the basic block b_5.

In the PRES+ net for the program shown figure 3.15 notice the transition t_3 belonging to the subnet (N_5) which is a successor of two Conditional subnets N_2 and N_3 and a Normal subnet N_4. The transition t_3 cannot be assigned

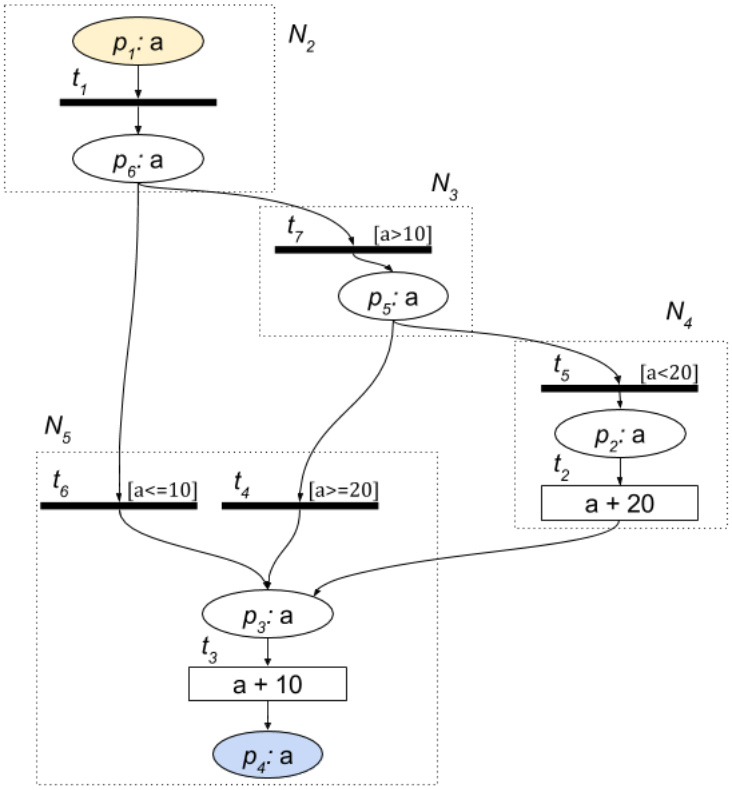

Figure 3.15: PRES+ net for the program shown in the figure 3.4.

a *guard* because it will be executed always; the only factor that needs to be governed is the transition which shall provide token to the pre-place p_3 of the transition t_3. To properly route the token to the place p_3, *guarded* transitions t_4 and t_6 are constructed and the three pre-transitions of p_3 reflect the three possible paths to reach the subnet N_5 in the control flow graph. □

The method for attaching `Conditional` subnets

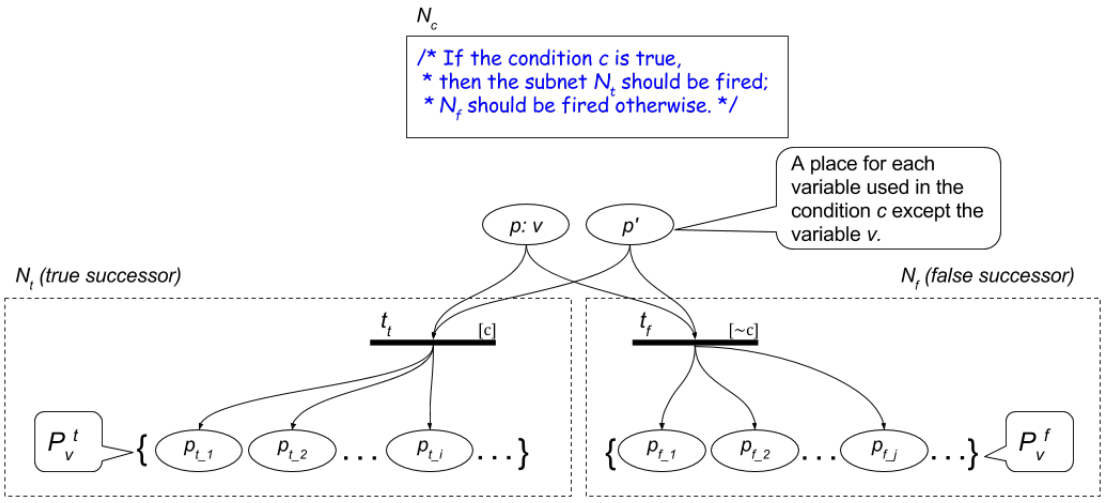

Figure 3.16: General structure of a `Conditional` subnet and its successors.

The attachment of the *input places* of the *true* (N_t) and the *false* (N_f) successors of a `Conditional` subnet N_{cond} is governed by the variables associated with the input places of N_t and N_f. Let v be the variable associated with some input places of N_t (or N_f). Let P_v^t and P_v^f be the set of the *input places* of the subnets N_t and N_f respectively with v as the associated variable, as shown in figure 3.16.

If either of the sets P_v^t or P_v^f is non-empty, then two *identity* transitions t_t and t_f are constructed with c and $\neg c$ as the *guard* condition respectively, where c is the condition in the `if` clause of the `Conditional` subnet (corresponding basic block) N_{cond}. All the places in the sets P_v^t and P_v^f are attached as post-places to the transitions t_t and t_f, respectively.

A place p is constructed with v as the associated variable and attached as a common pre-place to the transitions t_t and t_f. If an *ldt* exists for v in N_{cond}, then p is attached as a post-place to the *ldt*; otherwise, p is added to the `input_places` array of the subnet N_{cond}.

For each variable v' used in the condition c other than the variable v, a place p' is constructed and attached as a common pre-place to the transitions t_t and t_f. The place p' is attached to N_{cond} in a manner similar to the attachment of the place p.

`Dummy` input places of the successors:
The attachment procedure of the `Dummy` input places of N_t and N_f is similar to that of the places with v as the associated variable with the difference being that the place p (as shown in figure 3.16) is not constructed. Places are constructed and processed for the variables used in the guard condition as described above.

3.6 Construction of PRES+ for programs containing loops

This section explains the enhancement of the model construction mechanism to handle programs containing loops. The mechanism for detecting the subnets belonging to a loop body is explained first, followed by the steps for programs containing loops without nesting. Next, the mechanism to extract parallelism between a loop and the segments preceding and following it is explained; finally, the mechanism to handle programs containing nested loops is presented.

3.6.1 Subnets belonging to a loop body

To detect if a subnet belongs to a loop body or not a *boolean* variable called *loop* is used. It maintained for each subnet and is initialized to *false* indicating that a subnet does not belong to any loop. The value of this variable is modified while backtracking in the *dfs* traversal as described below.

First for the subnets from which back edges emanate the variable *loop* is set as *true*. For a `Normal` subnet N_n, if its successor belongs to a loop, then the subnet N_n's *loop* variable is also set to *true*. In case of a `Conditional` subnet which is not a loop header if the successors belong to a loop, then the subnet itself also belongs

to a loop. For a *loop header* subnet, it belongs to a loop if its false successor has its *loop* variable set to *true*.

3.6.2 Loops without nesting

Let N_l be the subnet corresponding to a non-nested loop body, N_h be the subnet corresponding to the loop header and N_e be the subnet corresponding to the basic block which is reached through the loop exit; thus, N_l and N_e are the two successors of the conditional subnet N_h. The loop structures captured by the present mechanism can be represented by the regular expression $N_h(N_l)^*N_e$ encompassing *for* and *while* loops; *do-while* loops (with structures represented by the regular expression $N_lN_h(N_l)^*N_e$) are not encompassed.

Corresponding to a loop, a PRES+ subnet is constructed for the entire loop body and attached to the rest of the PRES+. Thus, it may be noted that while constructing N_l it is ensured that transitions having no dependence among themselves are executed in parallel. The subnet for a non-nested loop body is obtained using the same mechanism described for programs containing no loops because a non-nested loop body is essentially a sub-program without any loop. For loops, another important property is also to be ensured namely, prevention of simultaneous execution of different loop iterations. This property guides the steps for attaching various subnets to construct N_l and also while attaching N_l to N_h. First, the enhancement necessary in the construction of N_l corresponding to a loop body (over the process used for segments outside any loop); it also underlines the enhancement needed in attaching N_l to N_h (over the process of attaching any conditional subnet to its successors) and is illustrated using the following example; the generalized procedure along with the *rationale* for various steps is then described.

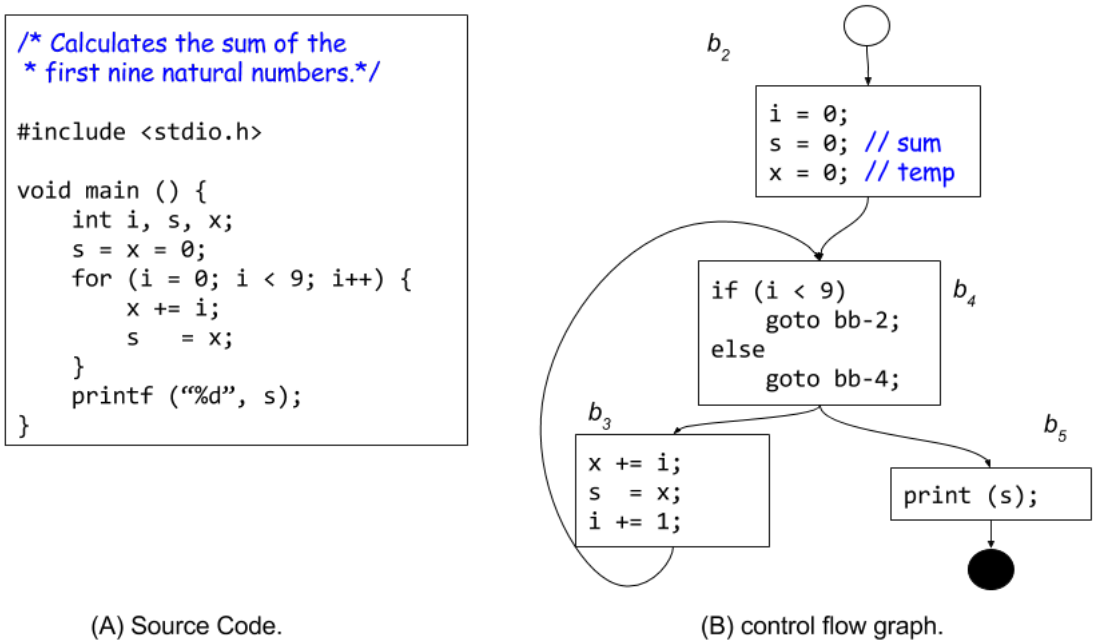

Figure 3.17: Source code and the *cfg* of an example program containing a loop.

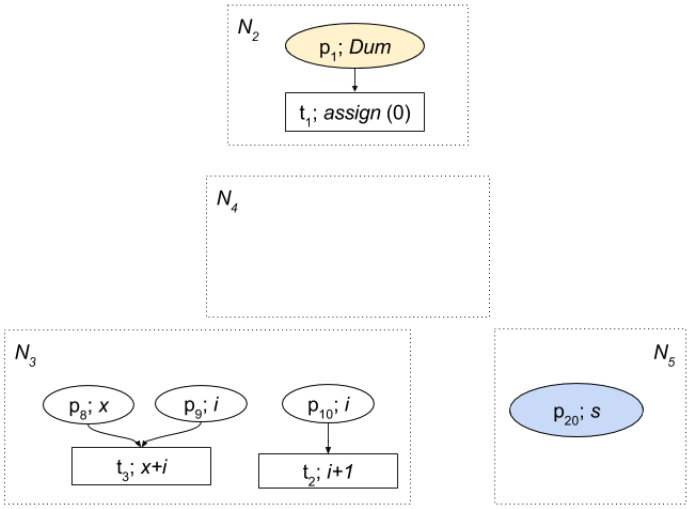

Figure 3.18: The PRES+ net consisting of unattached subnets corresponding to the program given in figure 3.17.

Example: 3.6.1

Figure 3.17(A) and (B) show a C program containing a loop and the corresponding *cfg*, respectively. The PRES+ consisting of the unattached subnets for the basic blocks and the overall PRES+ are shown in the figures 3.18 and 3.19, respectively.

Initially, all the subnets are marked as `Not_visited` and the *dfs* traversal for attaching the subnets begins with the subnet N_2. The subnets N_2, N_4 and N_3 are traversed in that order and are marked as `Visiting`. When the *dfs* traversal visits N_4 again as a child of N_3, the *dfs* backtracks because N_4 is in `Visiting` state and a loop (a back edge) is detected designating N_4 as the loop header and N_3 as the loop body.

The subnet N_3 is not only the true successor of the conditional subnet N_4 but also is a loop body; hence, the construction cannot be considered to be complete and ready for attachment with N_4 and itself, unless it is ensured that the i^{th} firing instance of all its transitions (corresponding to the i^{th} iteration) precedes the $(i+1)^{th}$ firing instance of any of its transition. To ensure such strict sequentiality between two iterations of N_3 the end of a loop iteration is captured by a notion of *maximum* transition of the loop body. The loop body, however, consists of multiple independent execution paths leading to *maximal* transitions, a *maximum* transition, therefore, needs to be specially constructed. In the concerned figure (3.19) the transition t_4 is introduced as a *maximum* transition and all the *maximal* transitions (t_2 and t_3) of the subnet N_3 are attached to t_4 via `Dummy` places (p_{11} and p_{13}). A `Dummy` place p_{18} is also constructed for synchronizing the maximum transition of the subnet N_4 with its predecessor N_3.

With the execution of the *maximum* transition, an iteration of loop body completes and the next iteration can be started. To start an iteration of a loop tokens need to be provided in the *input places* of the loop header, which do not exist right now because the loop header gets its *input places*

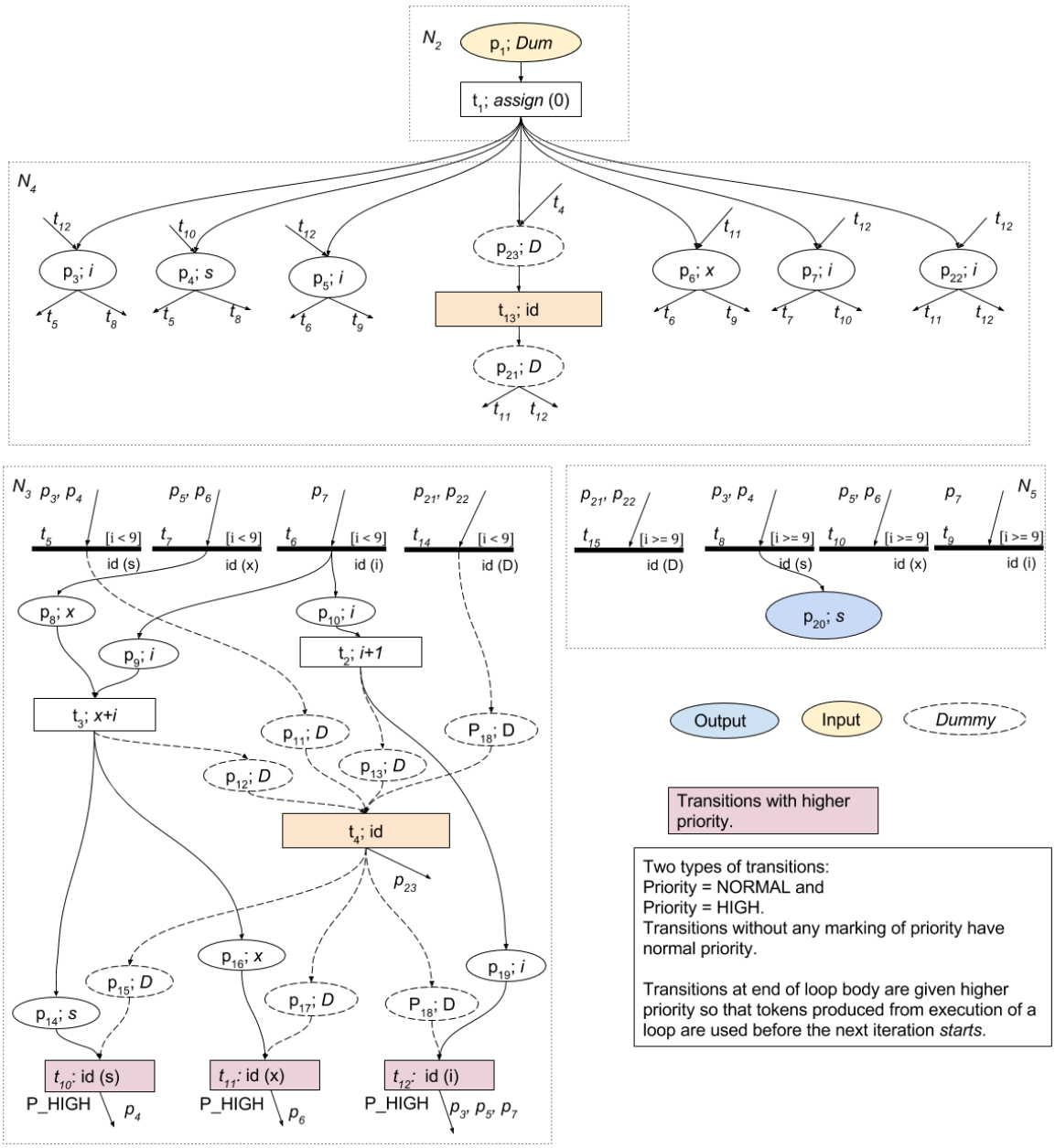

Figure 3.19: The complete PRES+ net corresponding to the program given in figure 3.17.

after its attachment to its *true* and *false* successors. To resolve this issue, *live variables* (defined in section 3.6.4) at the end of the subnet N_3 are used. Identity transitions namely t_{10}, t_{11} and t_{12}, are constructed for the variables *live* at the end of the subnet N_3 (or equivalently, at the beginning of the loop header) namely, s, x and i, such that each transition defines a variable, as shown in the figure. For the purpose of simulation using the CPN Tool, the newly constructed identity transitions are assigned higher priority over of the transitions in the rest of the PRES+ because the CPN Tool does not ensure that all enabled transitions are fired simultaneously; by assigning the transitions a higher priority, it is ensured that they are all fired prior to all their successor transitions.

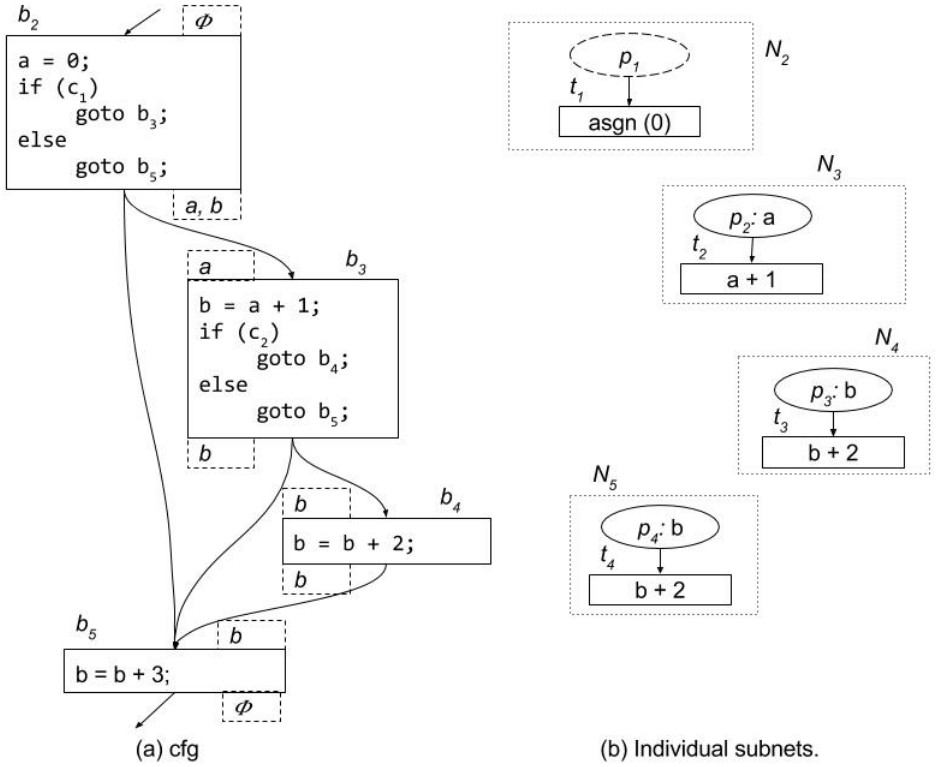

Figure 3.20: A subgraph of a *cfg* belonging to a loop body and the corresponding disconnected PRES+ net of individual subnets.

For each of the transitions t_{10}, t_{11} and t_{12}, a pre-place associated with the respective live variable and a `Dummy` place are constructed. The first set of pre-places (associated with the live variables) are the post-places of the corresponding *ldt*s in N_3; the second set of `Dummy` pre-places are post-places of the maximum transition t_4. These transitions thus get attached to N_3 and become the new set of *maximal* transitions providing values for the next iteration of N_l or the loop exit subnet N_e. Accordingly, the transitions t_{10}, t_{11} and t_{12} are set as the *ldt*s of the respective *live* variables.

After the above mentioned enhancement of N_3, the *dfs* traversal backtracks from N_3 to its predecessor N_4; the attachment of the subnet N_4 to its successor N_3, however, is deferred as the the subnet N_4 has an un-traversed successor N_5; the *dfs* traversal then visits N_5. Since the subnet N_5 is of type `Last`, the *dfs* backtracks right away to the loop header subnet N_4 again.

Having both its successors N_3 and N_5 traversed, the subnet N_4 is now attached to its successors using the procedure described for attaching a `Conditional` subnet to its successors for programs without loops. In course of attachment of N_3 to N_4, three new transitions are constructed for N_3 namely, t_5, t_6 and t_7. Among these, t_5 has an empty post-set because the variable s is not live at the beginning of N_3 and hence is attached to the *maximum* transition t_4 via the `Dummy` place p_{11}. A *maximum* transition t_{13} is also constructed for the loop header N_3. Transitions t_{14} and t_{15} are constructed for attaching the maximum transitions of the subnets as shown in the figure. Since the loop header N_3

itself does not belong to a loop body, the maximum transition synchronization place p_{23} is added to the input places array of the subnet.

Recall that the subnet N_4 has so far remained empty. The construction of the input places in the subnet N_4 comes from general attachment procedure of a conditional subnet with its successors. Now that the *loop header* subnet N_4 has been attached to its successors, it is attached to the loop body N_3 from which back edges lead to the loop header subnet N_4. For this purpose, the procedure for attaching a `Normal` subnet to its successors with a modification that the `Dummy` input places of N_4 are attached as post-places to the maximum transition of N_3 is invoked on the subnet N_3 is invoked on the subnet N_3.

The *dfs* traversal then backtracks to the subnet N_2 which is then attached to its successor N_4 following the usual procedure for attaching a `Normal` subnet to its successor completing the construction of the overall `PRES+`. □

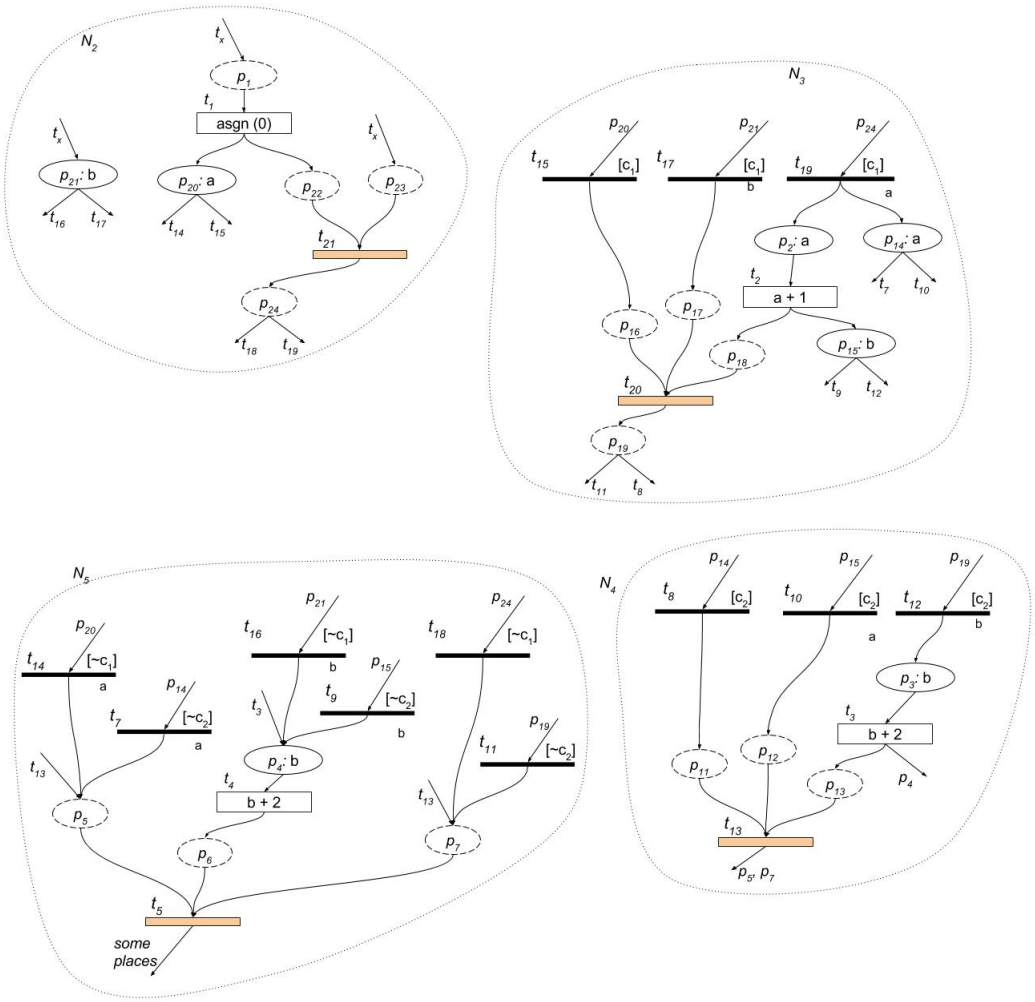

Figure 3.21: The attached `PRES+` net corresponding to the program given in figure 3.20.

Example: 3.6.2

This example illustrates the need for *interface variables* and places. In the figure 3.20(a) a part of a *cfg* is shown. Assume that the *cfg* shown in the figure

to be the body of some loop. Live variables at beginning and at the end of each basic block are also mentioned in the figure. The figure 3.20(b) shows the individual subnets corresponding to basic blocks of the *cfg*. The overall PRES+ obtained after attaching the subnets is shown in the figure 3.21. In the overall PRES+ subnet boundaries are not necessary but are shown just for clarity. The transition t_7 in the subnet N_5 would not have been constructed if only live variables (or the input places of the successors) are used for constructing the guarded id transitions. This would have halted the firing of the transition t_5 when the control would have to the subnet N_5 from N_3 (the condition c_2 being false).

Hence, it is necessary to consider all variables which are live at end of some predecessor of a subnet belonging to a loop body while constructing the *guarded identity* transitions for synchronization of such a subnet. □

Procedure for attaching the PRES+ models of programs containing non-nested loops:

The main issue for the loop is preventing simultaneous execution of different loop iterations. In this section, some properties/characteristics over the subnets of the loop body from which back edges lead to the loop header are stated first because they are used in the mechanism. The attachment mechanism is presented next. Let $\mathcal{N}_{be}$ be the set of subnets of the loop body from which back edges exist to the loop header subnet N_h. The following characteristics of the members of the loop body are used in enhancing the algorithm for handling loops:

1. The members of $\mathcal{N}_{be}$ from which back edges emanate are `Normal` subnets.

2. The control flows via exactly one subnet in the set $\mathcal{N}_{be}$ in an iteration of the loop.

The general structure of a program containing a loop is shown in figure 3.22 and is refereed to in the following explanation. After constructing the individual subnets for all the basic blocks, the *dfs* traversal for attaching the subnets is initiated from the first subnet in the *cfg*. When the *dfs* traversal reaches the loop header subnet N_h, it is marked as `Visiting`; the *dfs* traversal visits the *true* successor of N_h which is the first subnet in the loop body. The traversal eventually reaches some subnet N_j from which another traversal step leads to N_h with status "Visiting" implying that N_j is the last subnet in an execution path through the loop. The backtracking begins at this point; with an invocation of appropriate routines for attaching the subnets encountered during backtracking depending on their types. All the subnets in the loop body are either `Normal` or `Conditional`, hence, the corresponding routines presented for programs without loops to attach the two types of subnets to their successor(s) are enhanced to prevent simultaneous execution of two different loop iterations. The specific enhancement steps are described in the following sections.

Attachment of `Normal` subnets belonging to a loop body:
The `Normal` subnets belonging to a loop body are further divided into two subcategories. The first category $\mathcal{N}_{be}$ comprises subnets which are in the set $\mathcal{N}_b$ having

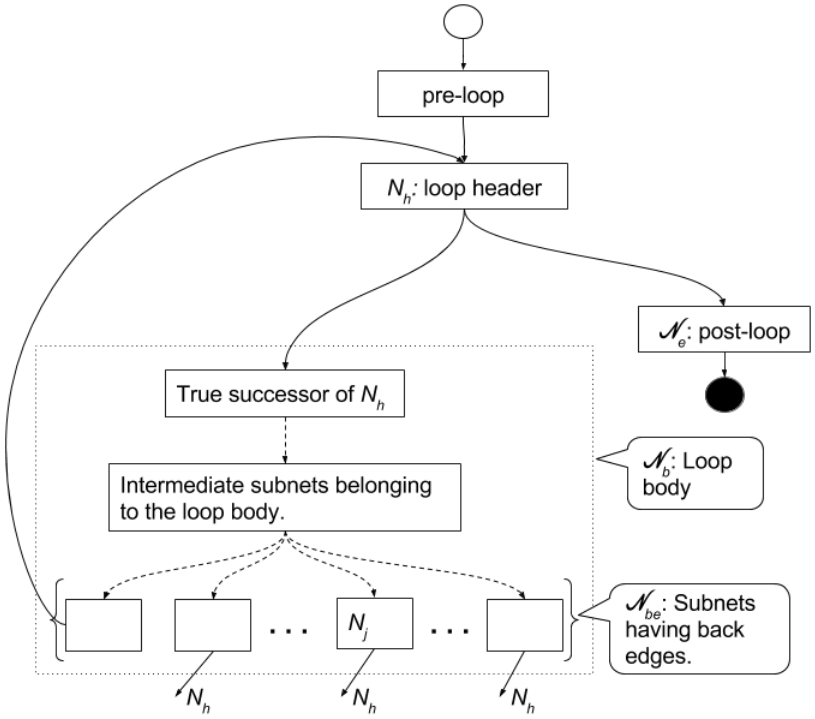

Figure 3.22: The structure of a program containing a loop.

back edges leading to the *loop header* and the second category comprises the rest of the `Normal` subnets belonging to the loop body. The mechanism for each type is described below.

a. Subnets belonging to $\mathcal{N}_{be}$

For a subnet $N_j \in \mathcal{N}_{be}$, the following operations are performed. A *maximum* transition $t_{j,\ max}$ of N_j is constructed and is attached to each *maximal* transition t' of the subnet N_j via a `Dummy` place p_d as a post-place of t' and a pre-place of $t_{j,\ max}$. The attaching procedure while backtracking is such that $t_{j,\ max}$ becomes the *last* transition in the loop body for all the execution path segments of the loop iterations which pass via the subnet N_j; hence the transition $t_{j,\ max}$ can be used as a control point to mark the completion of an iteration of the loop body.

Since a new iteration of the loop begins when the input places of the loop header get tokens, after the execution of the *maximum* transition $t_{j,\ max}$, tokens may be put into the input places of the loop header. But since the loop header has not yet been attached to its successors, its list of input places is not yet available. To resolve this issue, *live variable analysis* is used. The variables associated with the input places of the loop header are essentially the variables live at the beginning of the loop header (or equivalently, live at the end of the subnet N_j).

Hence, an *identity* transition t_v is constructed for each variable v live at the end of the subnet N_j. For each of these transitions a pre-place p_v associated with the respective *live* variable v and a `Dummy` pre-place p_d are constructed. The first set of pre-places (associated with *live* variables) are attached to the respective *ldt*s or added to the *input places* array of the subnet N_j, if the respective *ldt* does not exist; the second set of `Dummy` pre-places are post-places of the *maximum* transition $t_{j,\ max}$ of the subnet N_j. These newly constructed *identity* transitions are

now attached to N_j and become the new set of *maximal* transitions in N_j providing values for the next iteration of N_l or the loop exit subnet N_e. Thus, these *maximal* transitions are set as the *ldt*s of the respective *live* variables.

A `Dummy` place is constructed and attached as a pre-place to the maximum transition $t_{j,\ max}$; the place is maintained as a synchronization place for the *maximum* transition $t_{j,\ max}$ and will be referred as *maximum trans sync place* of the j^{th} subnet. With these steps the processing of the subnet N_j concludes. You may note that the subnet N_j is not yet attached to its successor subnet, the *loop header*.

b. `Normal` subnets not belonging to $\mathcal{N}_{be}$:
For a subnet N_i in this category, first the procedure for attaching a `Normal` subnet to its unique successor used for programs without loops is invoked and then a maximum transition t_m for the subnet is constructed. The *maximum trans sync* place of the N_i's successor is attached as a post-place to the transition t_m. A `Dummy` place is constructed as the *maximum trans sync place* of N_i and attached as a pre-place to the transition t_m.

For the unique successor N_u of the subnet N_i the following steps are also performed if there exists at-least one `Conditional` subnet of the subnet N_u. The necessity for these steps is established in the example 3.6.2. A set called *interface variables* $\mathcal{V}_{intr}$ is constructed by taking union of the *live variables* at the end of the predecessors of the unique successor. An array of size `MAX_VARIABLEs` called *interface places* is maintained for each subnet. The i^{th} index of the array stores an index of a place to the array `places[]` in the overall `PRES+` and has an initial value of -1. For each variable v in the set $\mathcal{V}_{intr}$, if the index v of the *interface places* array is -1, then a new `Dummy` place p_{intr} is constructed, stored at index v of the array and attached as a pre-place to the *maximum* transition of the successor. Otherwise, the place at the index v of the *interface places* array is set as p_{intr}. The place p_{intr} is attached as a post-place to the maximum transition of the subnet N_n.

Attachment of the `Conditional` subnets belonging to the loop body:
As stated earlier a `Conditional` subnet belonging to a loop body cannot have a back edge from itself to the *loop header*. For a subnet N_c in this category a *maximum* transition $t_{m,\ c}$ is constructed. and attached to the *maximum trans sync places* of the two successors via *guarded identity* transitions as shown in the figure 3.23. A *maximum trans sync place* is also constructed for the subnet N_c and will be used while attaching N_c to its predecessors.

Afterwards, the set *interface variables* $\mathcal{V}_{intr}$ is constructed by taking union of the *live variables* at the end of the predecessors of the two successors N_c. Let c be the condition in the *if-else* statement of the subnet N_c (or the corresponding basic block more precisely). For each variable v in the set $\mathcal{V}_{intr}$ two transitions t_v and t'_v are constructed with c and $\neg c$ as the *guards*, respectively. The transitions t_v and t'_v are attached as pre-transitions to the *input places* of the two successors of N_c having v as the associated variable. Pre-places to the transitions t_v and t'_v are constructed in the same way as mentioned for the `Conditional` subnets not belonging to a loop body. For each successor N_t and N_f, if the index v of the *interface places* array is -1, then a new `Dummy` place p_{intr} is constructed and is stored at index v of the array. Otherwise, the index v is set as p_{intr}. The place

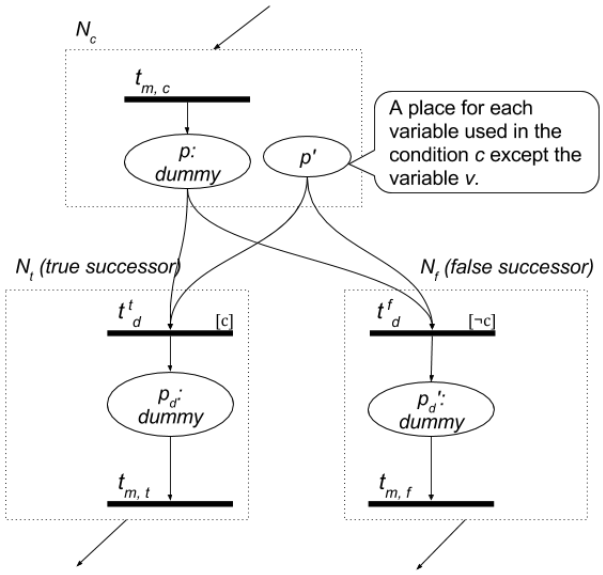

Figure 3.23: Illustration of the maximum transitions of a `Conditional` subnet and its successors'.

p_{intr} of each successor is attached as a post-place to the respective transition t_v or t'_v. If newly constructed, the place p_{intr} is attached as a pre-place to the maximum transition of the subnet it (p_{intr}) belongs to.

Attachment of the loop header:
Let N_h be the loop header subnet of a loop. First, the method for attaching a `Conditional` subnet belonging to a loop body to its successors is invoked on the loop header. Latter the method for attaching a `Normal` subnet to its successor for a program not containing any loop is invoked on each of the subnet N_j having a back edge to the loop header with one additional step `i.e.` attaching the *maximum trans sync place* as a post-place to the *maximum* trans of N_j. Each input place of the loop header is successfully attached to a transition in the predecessor because there exists a defining transition for each variable live at end of the predecessor subnet belonging to the loop body. The `Dummy` input places of the loop header are attached as post-places to the maximum transition of the the subnet N_j. The *maximum trans sync place* is added to the input places array of the subnet N_h, only if the subnet N_h does not belong to an outer loop's loop body, detected from the *loop* variable of its false successor.

3.6.3 Inter loop parallelism

If two loops have no data dependence between themselves, then the `PRES+` model should support their parallel executions, although they appear one after another in the `C` program. This is achieved over more general situations as described below. Let a basic block b_i, which itself is not necessarily a loop, follow some loop in the `C` program. If a transition belonging to the subnet N_i corresponding to the basic block b_i has no pre-place which gets a token generated in the loop body, then it must be allowed to fire independent of the loop exit condition. Hence, if a transition has no pre-place which gets a token from the loop body, then the

transition must be allowed to fire independently of the loop condition (imposed by the guarded transitions at the beginning of the false successor of a loop). *Reaching definition analysis* is used to check whether the variable v associated with an input place p_{in} of N_i is defined in the loop body or not. If the variable v is not defined in the loop body, then it is either attached to the respective *ldt* in the loop header or is added to the input places array of the loop header. This modification in the attach-step of the successors for a loop header is sufficient to incorporate the inter-loop parallelism, wherever possible.

3.6.4 Live variable analysis

Definition:

Recall from the section 3.6.2 (page 36) that *live variable* analysis is also used in the attach mechanism. Following is an explanation of the *live variable* analysis.

A variable v is said to be live at a point p in the cfg *iff* there exists a path in the cfg originating from the point p along which the current value of the variable v is used. □

Live variable analysis is another standard data-flow analysis technique described in [1]. It provides the *live variables* at the beginning and at the end of each basic block in form of bit vectors of lengths equal to the number of variables in the program associated with these points and are called $\texttt{Live}_{\texttt{in}}$ and $\texttt{Live}_{\texttt{out}}$, respectively. The i^{th} bit of the bit vectors is decoded as follows:
The variable with the index i in the Symbol Table is a live variable at the point, if and only if the bit is 1.

3.6.5 Loops with nesting

The *dfs* traversal for attaching subnets ensures that the `PRES+` subnet for the innermost loop is constructed first. The mechanism described in previous sections is able to attach the subnet corresponding to an inner loop to the outer loop by considering the inner loop as a single subnet.

3.6.6 Reaching definition analysis

Definition:

A definition d defining a variable v is said to be a reaching definition at a point p in the cfg if there exists an execution path from d to p along which the variable v is not redefined. □

Reaching definition analysis is a standard data flow analysis technique described in [1]. It provides the *reaching definitions* at the beginning and at the end of each basic block in the form of bit vectors of length equal to the number of statements in the program associated with these points and are called $\texttt{RD}_{\texttt{in}}$ and $\texttt{RD}_{\texttt{out}}$, respectively. The i^{th} bit of such a bit vector is decoded as follows:
The statement with *unique index* i is a *reaching definition*, at the point if and only if the bit is 1.

Unique index of a statement:

The unique index of a statement is computed as:

$$S_{ui} \leftarrow bb_i * \texttt{MAX\_BB\_STATEMENTs} + S_{ri} \tag{3.1}$$

Where:

S_{ui} is the *unique index* of the statement S_{ri} of the basic block bb_i and

`MAX_BB_STATEMENTs` is the maximum number of statements in a basic blocks.

With this the discussion on the attach mechanism concludes and the algorithm 5 (page 61) summarizes the attach mechanism.

3.7 Construction of PRES+ for programs containing arrays

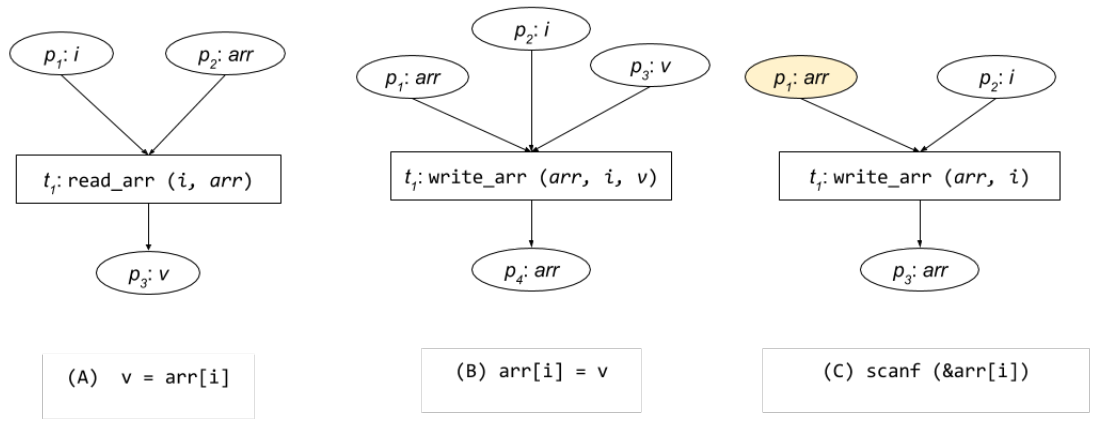

Figure 3.24: Various types of statements and transitions for arrays

In this section the enhancement of the PRES+ construction mechanism for handling arrays is explained. Figure 3.24 shows three different types of statements involving arrays that occur in a *cfg*. Note that only two new type of transitions are introduced namely, *write_array* transition for an array assignment or a scanf statement reading an array element and *read_arr* transition for an assignment statement involving an array reference at the right hand side. A *read_arr* transition has two pre-places, one holds tokens with values of the elements of an entire array and the other assumes the index values; it has a post-place which holds a token with the value of the indexed array element, as shown in the figure 3.24(A). For a statement writing a value to an array an *write_arr* transition is produced. It has three input places, the first one holding tokens with the array element values as its value, the second one with array index, and the third one containing the value to be written at the i^{th} index of the array and produces the array as output as shown in the figure 3.24(B). For a statement reading the value of an array element from input a *write_arr* statement is constructed as shown in figure 3.24(C). The

place with the array itself as the associated variable is used to get the input token and is highlighted as an input place. Note that each array in its entirety is treated as a variable and are stored in the symbol table but with a range associated to them. In the symbol table an new attribute is introduced for each symbol `i.e.` range. It is the size of an array for arrays and -1 for the variables of type `int`.

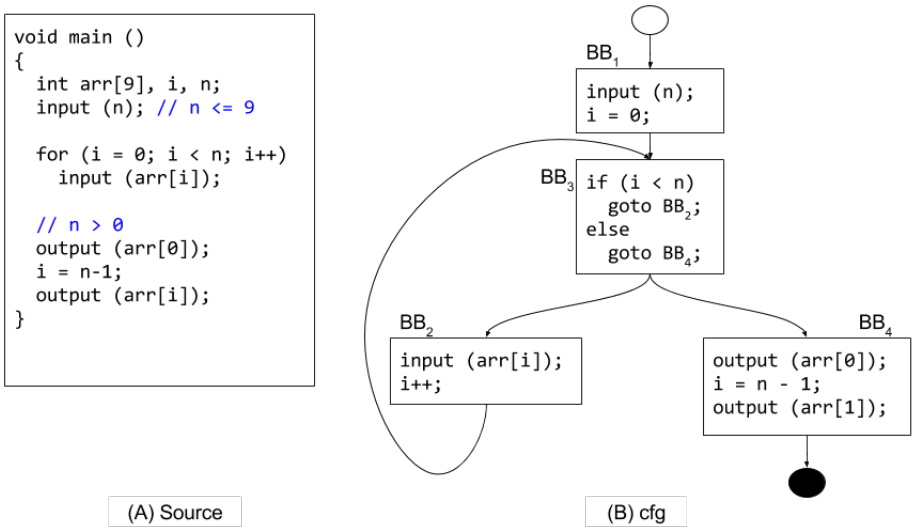

Figure 3.25: A C program and the corresponding *cfg* containing an array.

3.8 Handling parallelized programs

`C` programs containing parallelization directives make no difference to the *cfg* generated by the `gcc`. The *cfg* is essentially the same *cfg* had there been no parallelization directives. Hence, the model construction mechanism is also able to construct `PRES+` models for `C` programs containing parallelization directives. An example parallelized program and the corresponding *cfg* are shown in figures 3.27 and 3.28 respectively.

3.9 Reducing the size of the overall `PRES+`

To reduce the size of the overall `PRES+`, the following optimizations are performed.

3.9.1 Removal of identity transitions

Let t_{id} be an *identity* transition having a pre-place p_{pre} and post-places $p_{post,\ 1}$, $p_{post,\ 2}$, $\ldots$, $p_{post,\ j}$. Let $t_{pre,\ 1}$, $t_{pre,\ 2}$, $\ldots$, $t_{pre,\ i}$ be the pre-transitions of the place p_{pre}. The id-transition t_{id} and its pre-place p_{pre} are removed; for each, $1 \leq k \leq i$, the transition $t_{pre,\ k}$ is made tho have all the places $p_{post,\ 1}$, $p_{post,\ 2}$, $\ldots$, $p_{post,\ j}$ as its post-places. This step is pictorially depicted in the figure 3.29.

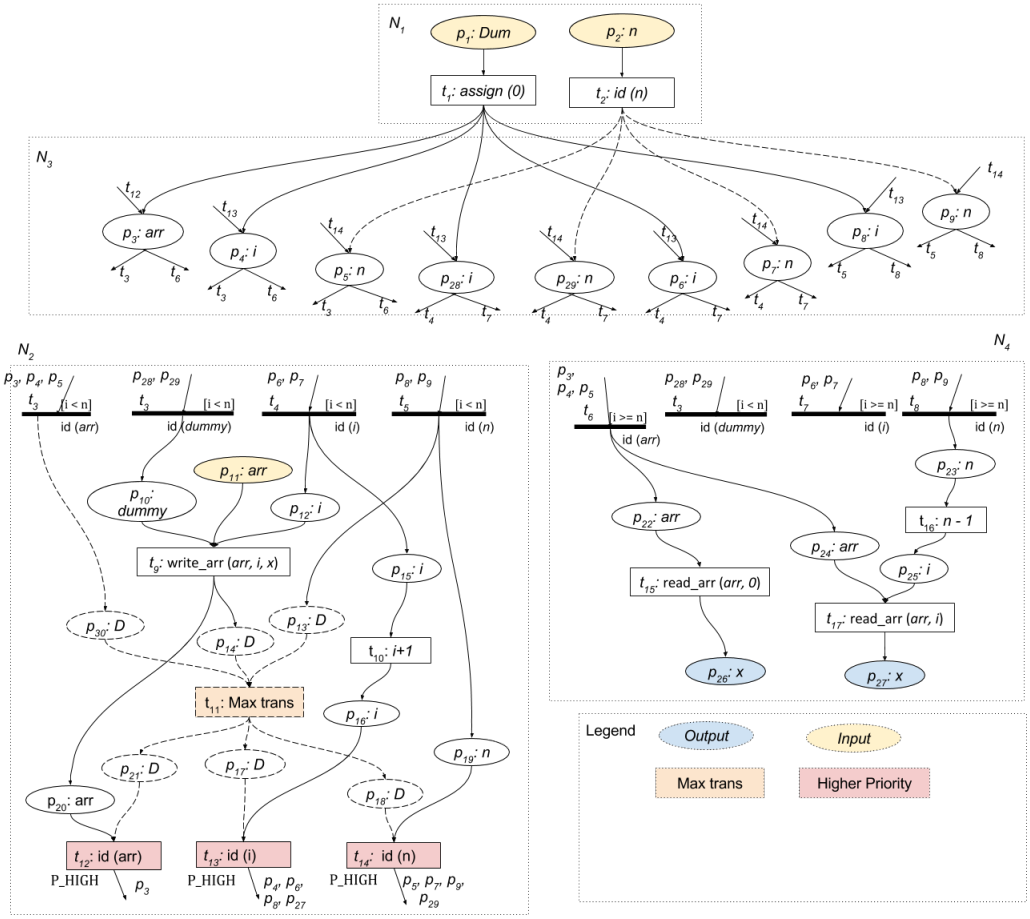

Figure 3.26: PRES+ corresponding the *cfg* shown in figure 3.25

```c
#include <stdio.h>
int main() {
    int num, max, min, i,  out;
    printf ("Enter four numbers: ");
    scanf ("%d", &num);
    max = min = num;
    #pragma scop
    for (i = 0; i < 3; i++) {
        scanf ("%d", &num);
        if (max < num)
            max = num;

    }

    for (i = 0; i < 3; i++)
    {
       if (min > num)
           min = num;
   }
   #pragma end
    out = min+max;
    printf ("%d ", out);
    return 0;
}
```

Source Program

Figure 3.27: Source code of a program containing parallelization constructs.

```
main ()
{
  int out;
  int i;
  int min;
  int max;
  int num;
  int num.1;
  int num.0;

  <bb 2>:
  scanf ("%d", &num);
  min = num;
  max = min;
  i = 0;
  goto <bb 6>;

  <bb 3>:
  scanf ("%d", &num);
  num.0 = num;
  if (max < num.0)
    goto <bb 4>;
  else
    goto <bb 5>;

  <bb 4>:
  max = num;

  <bb 5>:
  i = i + 1;

  <bb 6>:
  if (i <= 2)
    goto <bb 3>;
  else
    goto <bb 7>;

  <bb 7>:
  i = 0;
  goto <bb 11>;

  <bb 8>:
  num.1 = num;
  if (min > num.1)
    goto <bb 9>;
  else
    goto <bb 10>;

  <bb 9>:
  min = num;

  <bb 10>:
  i = i + 1;

  <bb 11>:
  if (i <= 2)
    goto <bb 8>;
  else
    goto <bb 12>;

  <bb 12>:
  out = min + max;
  printf ("%d ", out);
  num = {CLOBBER};
  return;
}
```

Note the absence of any new kind of statement corresponding to the parallelization constructs.

Figure 3.28: *cfg* for the source code given in the figure 3.27.

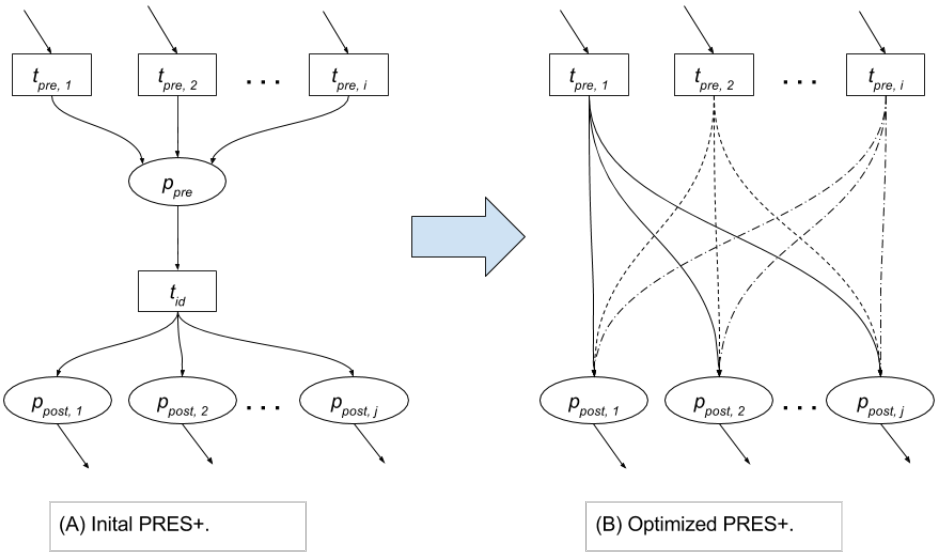

Figure 3.29: An example `PRES+` before and after removal of `identity` transitions.

3.9.2 Removal of useless computation

Recall that the place for the variable defined by a transition is constructed only when the the variable is used in some transition. In this optimization step, the *unguarded* transitions having an empty post-set are removed from the `PRES+` along with their set of the pre-places because this represents computation where values are never used. The process is repeated for the transitions for which the post-places set is reduced to empty because of the optimization until no more transitions are

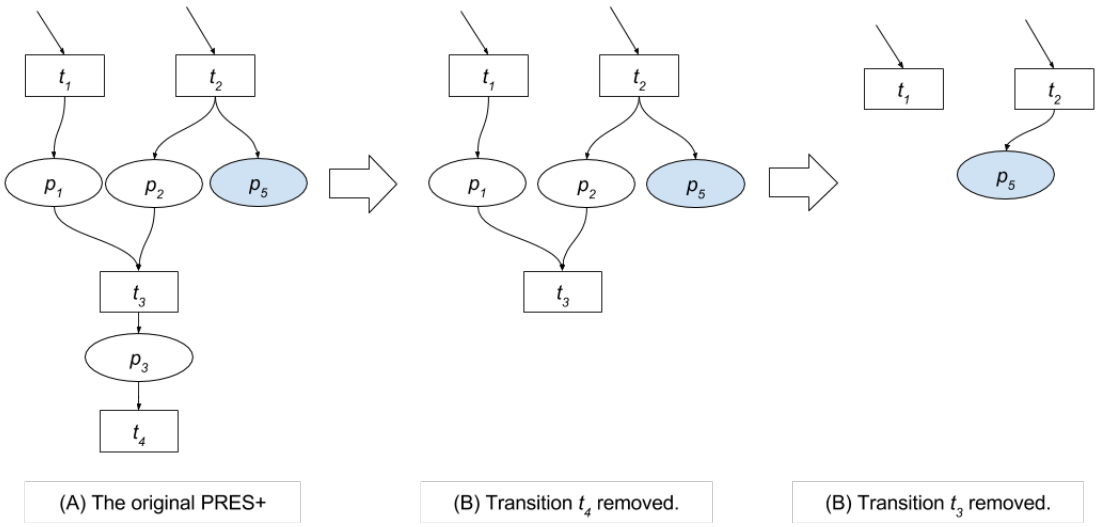

Figure 3.30: An example illustrating removal of useless computation.

removed from the PRES+. The *guarded* transitions cannot be removed even if they have empty sets of post-places because they perform the task of removing useless tokens from the PRES+ as explained earlier. The optimization step is illustrated using the following example.

Example: 3.9.1

The figure 3.30(A) shows a part of a PRES+ net. In the PRES+ net, the transition t_4 has an empty post-set and hence, it is removed from the PRES+ as shown in figure 3.30 (B). As a result, the transition t_3 gets an empty post-set. In this round the transition t_3 is removed from the PRES+ to obtain the PRES+ as shown in figure 3.30(C) and the process shall continues until no more transitions can be removed. □

Chapter 4

Experimental Results

Program	C code	Original PRES+		Optimized PRES+	
	Lines	Places	Transitions	Places	Transitions
Factorial	9	39	23	37	21
Fibonacci	30	77	53	73	49
GCD	23	82	51	72	41
Perfect Number	19	89	55	85	51
Primality Testing	18	67	42	64	39
Prime Factors	30	132	76	130	74
Sum natural numbers	11	45	29	42	26
Triple loop	18	134	70	130	66
Square Root	18	74	43	69	38

Table 4.1: Summary of simulation of the PRES+ models for some example programs in the CPN Tool.

This section provides outputs of the PRES+ construction module for some example programs. The results for some C programs tested using CPN Tool simulation are provided first. Next, the results obtained over the benchmark FSMDs are provided. Finally, PRES+ models for some C are shown.

4.0.1 C Programs

Table 4.1 summarizes the testing for various examples. Following is a description of each test case. "Factorial", "Fibonacci" and "GCD" are standard programs each producing a single integer as output. "Perfect Number" and "Primality Testing" produce the token value 1 at the output place if the input number is perfect/prime respectively. "Prime Factors" uses the property of the CPN Tool to gather multiple tokens in a place; it produces a token for each prime factor in the output place. Although the tool always ensures that no place gets more than one token but an output place inside a loop can get multiple tokens as in the concerned example. "Sum Natural Number" is the standard program to calculate the sum of first n natural numbers. "Triple Loop" is an extension of the "Sum

Natural Number" program; the one *loop* of the latter is enclosed in two loops each iterating from 0 to n. Though it does not compute anything useful, it was used as a test case for nested loops. The last program "Square root" calculates the integer square root of a number. A python script was written to convert a PRES+ model to the input (xml) format of the CPN.

4.0.2 Experimentation on the benchmark FSMDs

The table 4.2 summarizes the results obtained from execution of the benchmark FSMDs[4] over PRES+ construction module. The FSMDs were first converted to the corresponding C programs using Bison and Flex. They could not be simulated in the CPN Tool because of their size.

Program	FSMD	Original PRES+		Optimized PRES+	
	States	Places	Transitions	Places	Transitions
barcode.org	32	2268	1237	2262	1231
barcode.schd	24	3470	1950	3459	1939
dct.org	8	97	72	75	50
dct.schd	16	96	71	75	50
DHRC.org	60	2055	1050	2044	1039
DHRC.schd	57	2502	1378	2491	1367
diffeq.org	15	84	47	81	44
diffeq.schd	9	93	52	84	43
ewf.org	31	89	57	75	43
ewf.schd	23	84	52	71	39
gcdLCM.org	9	172	98	168	94
gcdLCM.schd	6	375	227	371	223
gcd.org	8	158	90	156	88
gcd.schd	5	272	170	270	168
ieee754.org	55	2119	1167	2113	1161
ieee754.schd	44	2969	1727	2956	1714
lruLCM.org	32	1162	635	1157	630
lruLCM.schd	32	1185	651	1180	646
lru.org	33	1158	633	1153	628
lru.schd	32	1177	647	1172	642
modn.org	6	168	93	166	91
modn.schd	9	164	91	163	90
perfect.org	6	74	44	74	44
perfect.schd	4	82	50	81	49
prawn.org	122	1620	1686	1586	1652
prawn.schd	122	1628	1693	1594	1659
qrs.org	50	1053	826	1018	791
qrs.schd	24	1004	775	981	752
tlc.org	13	164	172	161	169
tlc.schd	7	363	382	360	379

Table 4.2: Summary of PRES+ model construction for benchmark FSMDs [4].

4.0.3 Some example PRES+ nets

A program without loops.

```c
#include <stdio.h>

void main ()
{
     /* Simple if-else example.*/
     int a, b, c, d;
     scanf ("%d", &c);
     a = b = 0;
     if (c > 0)
          a = c + 10;
     else
          b = c - 10;

     d = a + b;

     printf ("%d", d);
}
```

Program 4.1: A simple C program containing only one `if-else` statement.

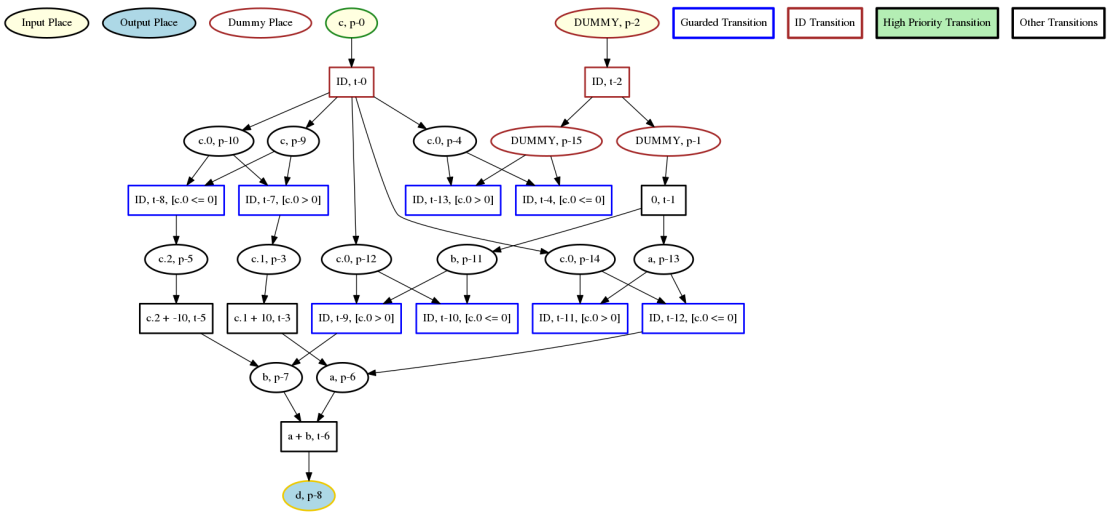

Figure 4.2: The optimized PRES+ model for the program 4.1 containing an `if-else` statement.

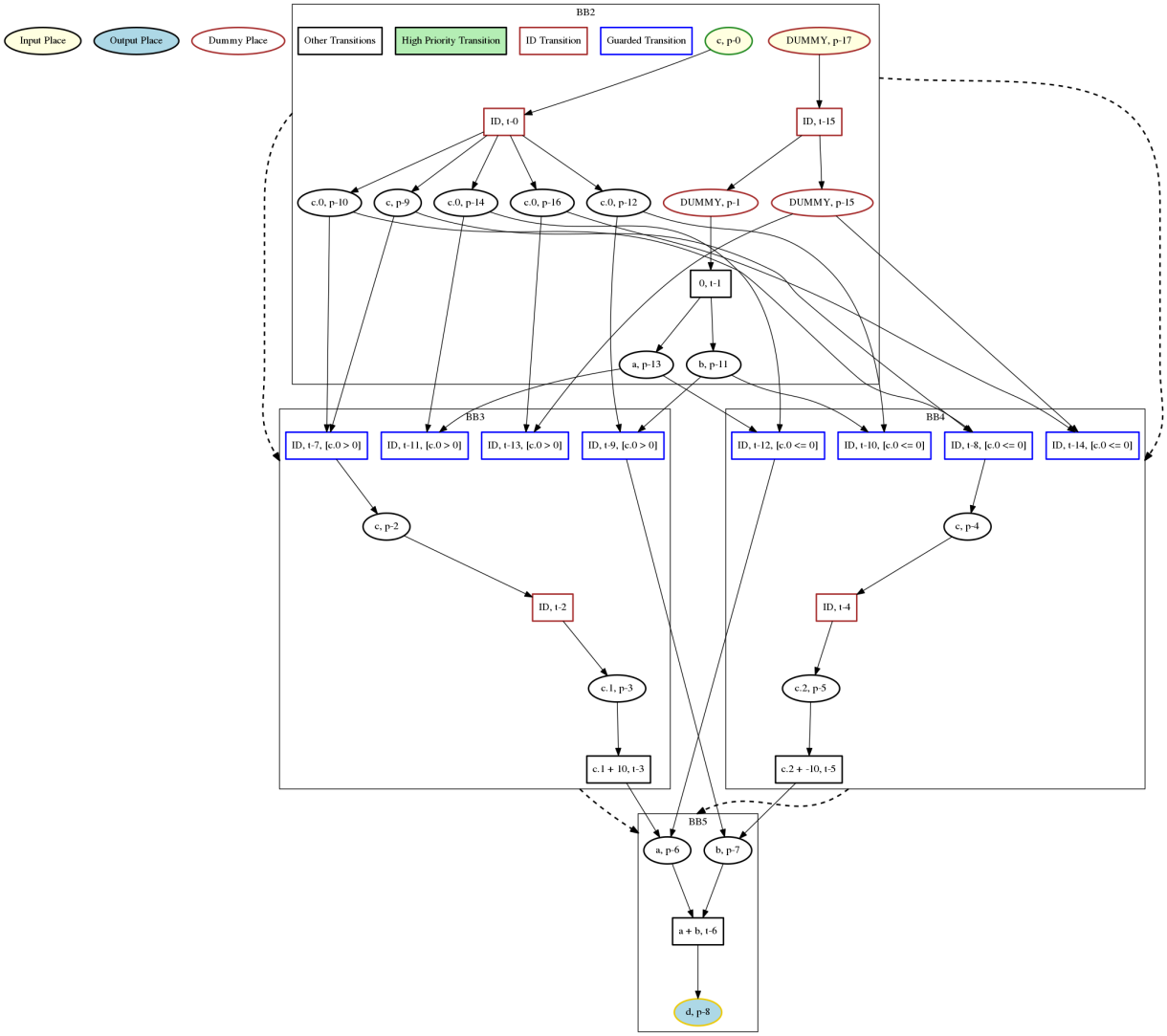

Figure 4.1: The un-optimized `PRES+` model for the program 4.1 containing an `if-else` statement.

For loop

```c
#include <stdio.h>

void main ()
{
    /* Simple For example: Sum of First n OR 9 natural
       numbers.*/
    int s, x, i, n;
    scanf ("%d", &n);
    s = x = 0;

    for (i = 0; i < n; i++)
    {
        x += i;
        s = x;
    }
    printf ("%d", s);
}
```

Program 4.2: A `C` program containing only one `for` loop.

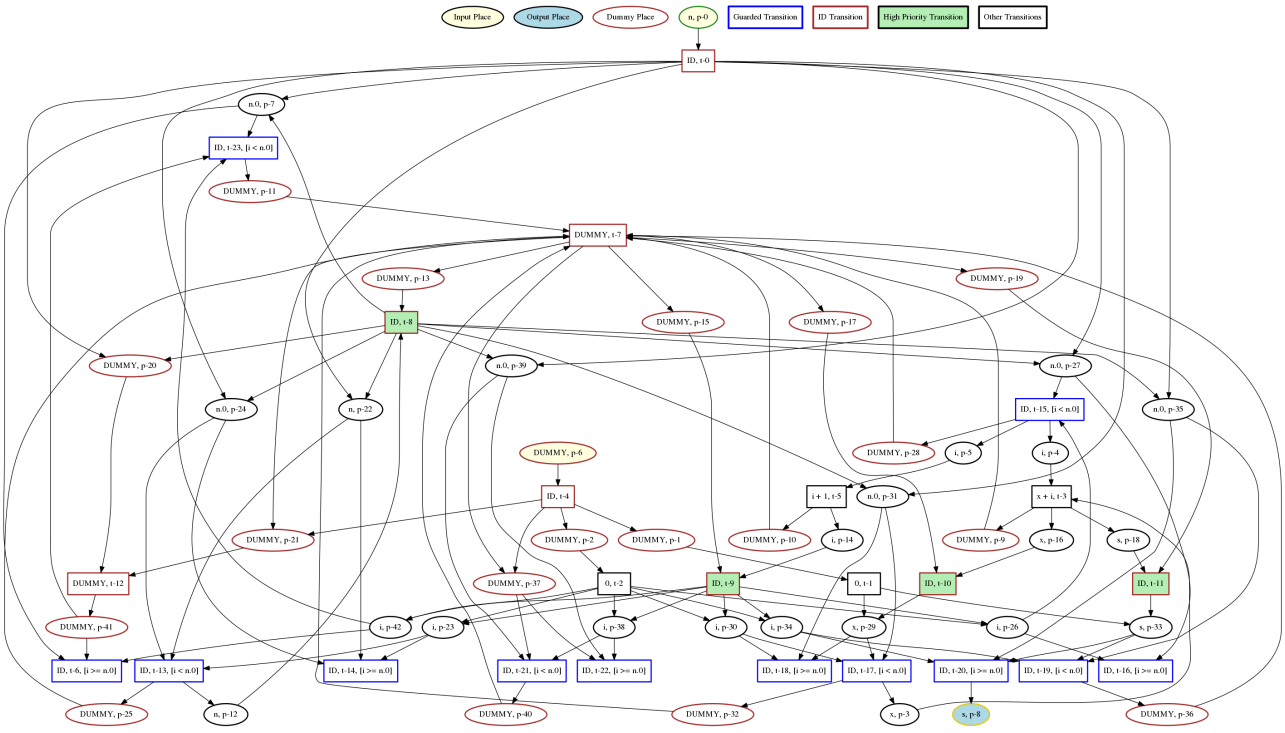

Figure 4.3: The optimized `PRES+` model for the program 4.2 illustrating a for loop.

While loop

```c
#include <stdio.h>

void main ()
{
    /* Simple while Loop example: Calculates Fibonacci
       . */
    int f1, f2, n, s, i;
    i = f1 = f2 = 1;
    scanf ("%d", &n);
    s = 0;
    while (i < n)
    {
        i++;
        s = f1 + f2;
        f1 = f2;
        f2 = s;
    }
    printf ("%d", s);

}
```

Program 4.3: A `C` program containing one `while` loop.

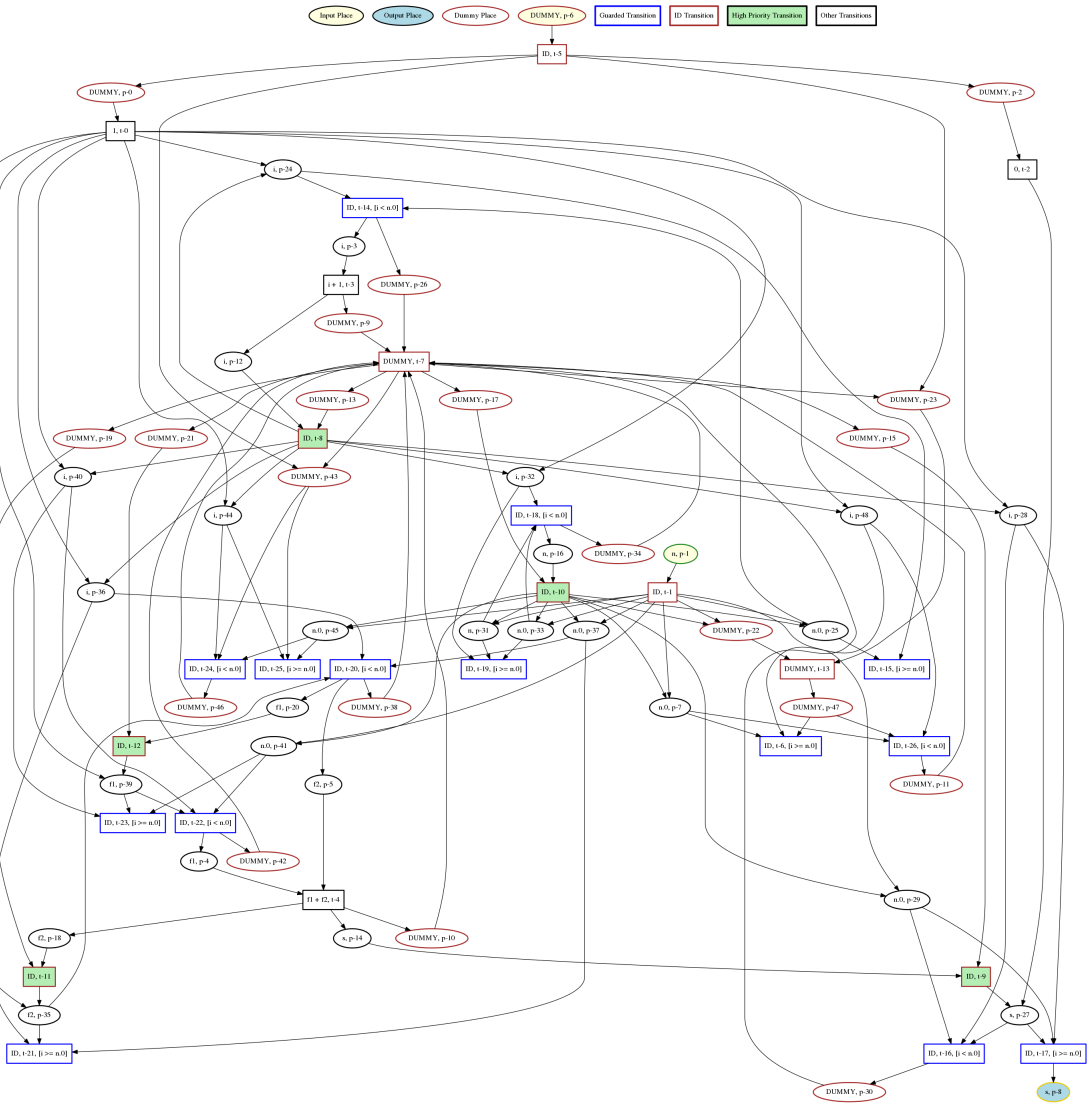

Figure 4.4: The PRES+ model for the program 4.3 illustrating a while loop.

Initialization of different arrays.

```c
#include <stdio.h>

void main ()
{
    int a1[10], a2[20], i, j;

    for (i = 0; i < 10; i++)
        a1[i] = 0;

    printf ("%d", i);

    for (j = 0; j < 20; j++)
        a2[j] = 0;
}
```

Program 4.4: A C program containing initialization of two arrays in different loops.

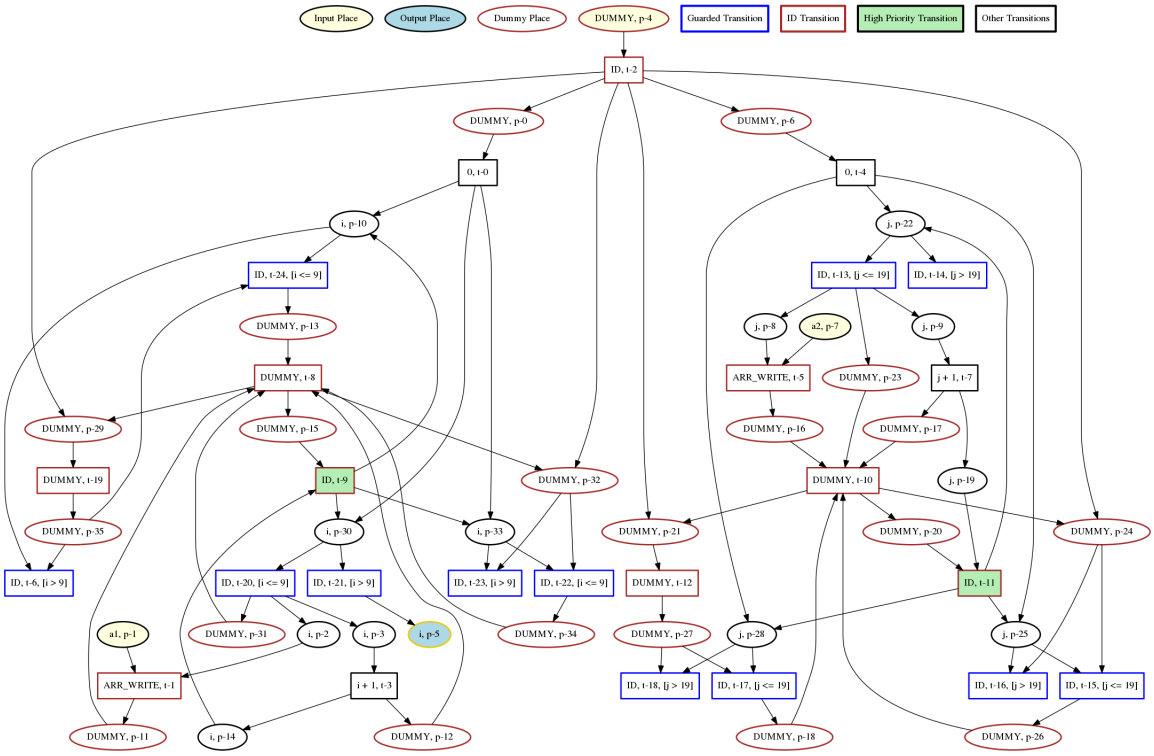

Figure 4.5: The optimized PRES+ model for the program 4.4 illustrating inter loop parallelism. The parallel execution of the two loops may be noted.

Chapter 5

Conclusion and future work

5.1 Conclusion

The model construction mechanism consists of constructing first the PRES+ models for all the basic blocks in isolation; accordingly, they appear as disconnected sub-graphs, referred to as subnets; subsequently, they are attached to each other to obtain the entire digraph for the PRES+ model of the complete program. The data structures used to store the control flow graph (*cfg*) of the input program, generated by the `gcc` and those used to represent a PRES+ model are provided first, followed by the mechanism to construct PRES+ subnets for individual basic blocks and attachment of the basic blocks to obtain the overall PRES+ model. The description of the implementation has been organized to proceed from simpler program constructs to more complex ones. Thus, the model construction mechanism for programs containing only input, output, assignment and `if-else` statements is explained first. The mechanism is then enhanced to construct models for programs containing loops, arrays and parallelization constructs. Thereafter, some optimizations are provided on the PRES+ model. In the optimizations, data transfer (identity) transitions and useless computation have been removed. Finally, experimental results are provided.

5.2 Future Work

The method presented in this work handles only integer variables; the mechanism can be extended to handle other data types also. Also, the mechanism needs to be enhanced to handle exit control loops. Another concern is that the size of the constructed PRES+ model is huge even after optimizations. An enhancement of the PRES+ model with a reduction in size is provided in [3]. The given method can be enhanced to produce this smaller PRES+ model. However, the existing analysis and simulation tool (`CPN Tool`) reported in the work, available as a free software, will also have to be updated to use such smaller sized representations of the models.

Appendix A

Algorithms

The various algorithms described in the implementation are summarized below.

A.1 Construction of PRES+ subnet for an individual basic block

Algorithm 1: *const_pp_one_bb (*`bba[], st, PP, Subnets[]`*, k)*

Inputs :

1. `bba[]`: Pointer to the array of basic blocks.
2. `st`: Pointer to the symbol table.
3. `PP`: Pointer to the overall `PRES+` net.
4. `Subnets[]`: Pointer to the array used to identify the `PRES+` corresponding to a basic block in the overall `PRES+` net.
5. k: Index of the basic block for which a `PRES+` subnet is to be constructed.

Outputs:

1. `PP`: Pointer to the overall `PRES+` net modified to contain the `PRES+` subnet for the k^{th} subnet.
2. `Subnets[]`: Pointer to the `Subnets[]` array with its k^{th} element modified to contain the indices to the arrays `places[]` and `transitions[]` of the overall `PRES+` of the places and the transitions belonging to the k^{th} subnet.

```
1  foreach statement s in the k^th basic block do
       /* For each statement of the given basic block in the order of appearance in the
          basic block:*/
2      switch s.type do
3          case scanf:
4              v ← s.var // The variable read by the scanf statement.
5              p_v ← const_new_place (PP, Subnets, k, v) /* Constructs a place
                   for v.*/
6              p_d ← const_new_place(PP, Subnets, k, −1) /* Constructs a
                   Dummy place.*/
7              t ← const_new_trans (PP, Subnets, k, NULL) /* Constructs an id
                   trans.*/
8              attach_as_pre_place (PP, p_v, t) // Attaches p_v to t as a pre-place.
9              attach_as_pre_place (PP, p_d, t)
10             add_to_local_input_places_array (Subnets, k, p_d) /* Add p_d to the
                   input places array of the k^th subnet.*/
11             add_to_global_input_places_array (PP, p_v) /* Add p_v to the input
                   places array of the overall PRES+.*/
12             Subnets[k].ldt[v] ← t // Set t as the ldt for v.
13         case printf:
14             v ← s.var // The variable printed by the statement.
15             p_v ← const_new_place (PP, Subnets, k, v)
16             add_to_global_output_places_array (PP, p_v) /* Adds the place p_v to
                   the output places array of the overall PRES+.*/
17             if Subnets[k].ldt[v] ≠ −1 then
18                 attach_as_pre_place (PP, p_v, t)
19             else
20                 add_to_local_input_places_array (Subnets, k, p_v)
21             end
22         contd...
78     endsw
79 end
```

Algorithm 2: *const_pp_one_bb ()* contd...

```
23 case Asgn_id:
24     v_l ← s.var // Variable defined by the statement s.
25     v_r ← get_first_operand (s.expr) /* The rhs operand is obtained from the
           expression tree associated to the statement s.*/
26     if Subnets[k].ldt[v] ≠ −1 then
           /* An ldt exists for the variable v_r.*/
27         Subnets[k].ldt[v_l] = Subnets[k].ldt[v_r] /* Set the ldt for the
               variable v_r as the ldt for v_l.*/
28     else
           /* An ldt does not exist for the variable v_r.*/
29         t ← const_new_trans (PP, Subnets, k, NULL) // Construct an identity
               transition.
30         p ← const_new_place (PP, Subnets, k, v_r)
31         add_to_local_input_places_array (Subnets, k, p)
32         attach_as_pre_place (PP, p, t)
           /* Set t as the ldt for both the variables v_l and v_r.*/
33         Subnets[k].ldt[v_l] = t
34         Subnets[k].ldt[v_r] = t
35     end
36 case Asgn_unary:
37     t ← const_new_trans (PP, Subnets, k, s.expr) /* Construct a new
           expression transition for the statement s.*/
38     v_l ← s.var
39     v_r ← get_first_operand (s.expr) /* The rhs operand, −1 if operand is a literal;
           index to the symbol table otherwise.*/
40     p ← const_new_place (PP, Subnets, k, v_r) /* p is a Dummy place, if
           v_r == −1; a Var place otherwise with v_r as the associated variable.*/
41     attach_as_pre_place (PP, p, t)
42     if v_r = −1 ∨ Subnets[k].ldt[v_l] = −1 then
           /* Either v_r is a literal or there exists no ldt for v_r in the k^th subnet.*/
43         add_to_local_input_places_array (Subnets, k, p)
44     else
           /* v_r is a variable and there exists an ldt for v_r.*/
45         attach_as_post_place (PP, p, Subnets[k].ldt[v_l])
46     end
47     Subnets[k].ldt[v_l] = t // Set t as the ldt for v_l.
48 contd...
```

Algorithm 3: *const_pp_one_bb ()* contd...

```
49 case Asgn_binary:
50     v_l ← s.var
51     o_1 ← get_first_operand (s.expr)
52     o_2 ← get_second_operand (s.expr)
53     t ← const_new_trans (PP, Subnets, k, s.expr) Subnets[k].ldt[v_l]
           = t // Set t as the ldt for v_l.
54     if o_1 = −1 ∧ o_2 = −1 then
55         p ← const_new_place (PP, Subnets, k, −1)
56         attach_as_pre_place (PP, p, t)
57         add_to_local_input_places_array (Subnets, k, p)
58     end
59     if o_1 ≠ −1 then
           /* First operand is a variable.*/
60         p_1 ← const_new_place (PP, Subnets, k, o_1)
61         attach_as_pre_place (PP, p_1, t)
62         if Subnets[k].ldt[o_1] = −1 then
63             add_to_local_input_places_array (Subnets, k, p_1)
64         else
65             attach_as_post_place (PP, p_1, Subnets[k].ldt[o_1])
66         end
67     end
68     if o_1 ≠ −1 then
           /* Second operand is a variable.*/
69         p_1 ← const_new_place (PP, Subnets, k, o_2)
70         attach_as_pre_place (PP, p_2, t)
71         if Subnets[k].ldt[o_2] = −1 then
72             add_to_local_input_places_array (Subnets, k, p_2)
73         else
74             attach_as_post_place (PP, p_2, Subnets[k].ldt[o_2])
75         end
76     end
77 end
```

A.2 Attaching individual subnets to obtain the overall PRES+ model

Algorithm 4: *dfs_attach_subnets_wrapper* (`bba[], pp, Subnets[], st`)

Inputs :

1. `bba[]`: The array of basic blocks (`cfg`).
2. `pp`: The disconnected `PRES+` net consisting of individual subnets for all the basic blocks.
3. `Subnet[]`: An array of type `Subnet` containing various attributes like places, transitions, input places, input transitions etc. of the individual subnets.
4. `st`: The symbol table.

Outputs: The overall `PRES+` net in the variable `pp` itself as a value-return argument.

/* Functionality: The function performs various initialization and invokes the depth first traversal for attaching disconnected subnets of the overall `PRES+`. */

```
1  foreach subnet 𝒩 in the Subnets[] array do
2      𝒩.loop ← False // Initializes all the subnets as non loop body subnets.
3      𝒩.visited ← Not_Visited // Mark each subnet as not visited.
4      𝒩.maximal_trans_count ← 0 // Initialize maximal trans count to zero.
5      𝒩.maximum_trans ← −1 // Initialize the maximum trans.
6      𝒩.mt_sync_pl ← −1 // Initialize the maximum trans sync place.
7      foreach transition t belonging to the subnet 𝒩 do
8          if t does not have a succeeding transition in the subnet 𝒩 then
               /* The transition t is a maximal transition of the subnet.*/
9              k ← 𝒩.maximal_trans_count
10             𝒩.maximal_trans[k] ← t // Add t to the maximal trans array.
11             𝒩.maximal_trans_count + = 1
12         end
13     end
14     for i ← 0 to MAX_VARIABLEs do
15         𝒩.interface_place[i] ← −1 // Initialize interface place for var i.
16     end
17 end
18 dfs_attach_subnets (bba[], PP, Subnets[], 0, st) /* Invoke the dfs traversal
       from the very first subnet.*/
```

Algorithm 5: *dfs_attach_subnets* (`bba[], PP, Subnets[],` i, `st`)

Inputs : The argument i holds the index of the current subnet. Other input arguments have same meaning as explained in the algorithm 4.

Outputs: This function does not return anything but modifies the value return arguments.

/* Functionality: This function traverses the control flow graph in depth first manner. While backtracking the current subnet is attached to its successor(s). `Subnets[`i`]` is referred as $\mathcal{N}$ in the following algorithm. */

```
1 if N.status ≠ Not_Visited then
2 |   return;
3 end
4 N.status ← Visiting
```

// *Contd. . .*

Algorithm 6: *dfs_attach_subnets () Contd...*

```
   /* Based on type of the subnet 𝒩, various operations are performed on a subnet.*/
 5 if 𝒩 is a Conditional subnet then
       /* Invoke the dfs traversal for the two successors.*/
 6     dfs_attach_subnets (bba[], PP, Subnets[], 𝒩.true_succ, st)
 7     dfs_attach_subnets (bba[], PP, Subnets[], 𝒩.false_succ, st)
 8     if 𝒩.true_succ < 𝒩 then
           /* 𝒩 is a loop header.*/
 9         const_maximum_trans (PP, Subnets[], i) /* Constructs a new
               maximum transition in the subnet 𝒩.*/
10         attach_successors (PP, Subnets[], i)      /* Attaches the 𝒩 to its two
           successors.*/
11         foreach predecessor 𝒩_pred of 𝒩 do
12             if 𝒩_pred.status = VISITED then
                   /* The edge 𝒩_pred→𝒩 forms a back edge.*/
13                 attach_successors (PP, Subnets[], 𝒩_pred)
14             end
15         end
16     else if 𝒩.true_succ.loop = True then
           /* 𝒩 is a Conditional subnet belonging to a loop body.*/
17         const_maximum_trans (PP, Subnets[], i)
18         attach_successors (PP, Subnets[], i)
19     else
           /* 𝒩 is a Conditional subnet not belonging to any loop.*/
20         attach_successors (PP, Subnets[], i)
21     end
22 else if 𝒩 is a Normal subnet then
23     𝒩_true_succ ← 𝒩.true_succ
24     switch 𝒩_true_succ.status do
25         case VISITING:
               /* 𝒩 → 𝒩_true_succ is a back edge.*/
26             const_maximum_trans (PP, Subnets[], i)
27             const_defining_trans (bba[], Subnets[], PP, i) /* Constructs
                   defining transitions for all the variables live at the end of the subnet 𝒩.*/
28         otherwise
29             dfs_attach_subnets (bba[], PP, Subnets[], 𝒩.true_succ,
               st)
30             attach_successors(PP, Subnets[], i)
31         end
32     endsw
33 else
       /* 𝒩 is the last subnet, Do nothing.*/
34 end
35 bba[𝒩].status ← VISITED
```

Algorithm 7: *const_maximum_trans*(PP, Subnets[], *i*)

Inputs : The input parameters have their usual meaning as in preceding algorithms.

Outputs: Return type of this function is void.

/* This function constructs the maximum transition for the i^{th} subnet. The construction is reflected in the PRES+ PP and the Subnets[] array. Subnets[i] is referred as $\mathcal{N}$ in the following algorithm. */

```
1  t ← const_new_trans (PP, Subnets[], i, NULL) /* Constructs an id trans.*/
2  N.maximum_trans ← t /* Set t as the maximum transition for the subnet.*/
3  foreach maximal transition t' in the Subnets[i] do
       /* Maximal transitions of a subnet are stored in the array maximal_trans[].*/
4      p ← const_new_place(PP, Subnets[], i, −1) /* Constructs a Dummy
           place.*/
5      attach_as_pre_place (PP, p, t) // Attaches p to t as a pre-place.
6      attach_as_post_place (PP, p, t') // Attaches p to t as a post-place.
7  end
```

Algorithm 8: *const_defining_trans*(PP, Subnets[], *i*)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The return type of this function is void.

/* This function constructs the defining transitions for the variables live at the end of the i^{th} subnet. The overall PRES+ PP and the Subnet[] array are modified as value return arguments. Subnets[i] is referred as $\mathcal{N}$ in the following text. */

```
1  t_m ← N.maximum_trans // The maximum transition of the subnet.
2  foreach variable v live at the end of the i^th subnet do
3      p ← const_new_place(PP, Subnets[], i, v) /* Constructs a place for the
           variable v.*/
4      t ← const_new_trans (PP, Subnets[], i, NULL) /* Constructs an id
           trans.*/
5      attach_as_pre_place (PP, p, t) // Attaches p to t as a pre-place.
6      if N.ldt[v] ≠ −1 then
           /* An ldt exists for the variable v.*/
7          t' ← N.ldt[v]
8          attach_as_post_place (PP, p, t')
9      else
           /* An ldt does not exist for the variable v.*/
10         add_to_local_input_places_array (Subnets[], i, p)
11     end
       /* Attach t to the maximum transition of N as a succeeding transition.*/
12     p ← const_new_place(PP, Subnets[], i, −1) /* Constructs a Dummy
           place.*/
13     attach_as_pre_place (PP, p, t) // Attaches p to t as a pre-place.
14     attach_as_post_place (PP, p, t_m) // Attaches p to t_m as a post-place.
15 end
```

Algorithm 9: *attach_successors* (`PP, Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The overall `PRES+ PP` is modified to reflect the attachment of the i^{th} subnet with its successors.

/* In this function Subnets[i] and $\mathcal{N}$ are used synonymously. */

1 $\mathcal{N}_t \leftarrow \mathcal{N}$.true_succ // True successor of $\mathcal{N}$.
2 **if** $\mathcal{N}$ *is a* ***Normal*** *subnet* **then**
3 **if** $\mathcal{N}_t.loop =$ ***True*** **then**
 /* The subnet $\mathcal{N}_t$ belongs to a loop body.*/
4 *attach_succ_normal_loop_subnet* (`PP, Subnets[], i`)
 /* Attach the maximum trans of $\mathcal{N}$ to that of its successor's.*/
5 $t \leftarrow \mathcal{N}$.`maximum_trans` $p \leftarrow \mathcal{N}_t$.`mt_sync_pl` // Maximum trans sync place of the succ.
6 *attach_as_post_place* (`PP`, p, t)
7 **else**
 /* The subnet $\mathcal{N}_t$ does not belong to any loop body.*/
8 *attach_succ_normal_non_loop_subnet* (`PP, Subnets[], i`)
9 **end**
10 **else if** $\mathcal{N}$ *is a* ***Conditional*** *subnet* **then**
11 **if** $\mathcal{N}_t < i$ **then**
 /* $\mathcal{N}$ is a loop header.*/
12 *attach_succ_loop_header* (`PP, Subnets[], i`)
13 *attach_maximum_trans* (`PP, Subnets[],` i) /* Attaches a maximum transition to its successors' maximum trans.*/
14 **else if** $\mathcal{N}_t.loop =$ ***True*** **then**
 /* $\mathcal{N}$ is a Conditional subnet belonging to a loop body.*/
15 *attach_succ_conditional_loop_subnet* (`PP, Subnets[], i`)
16 *attach_maximum_trans* (`PP, Subnets[],` i)
17 **else**
 /* $\mathcal{N}$ is a conditional subnet not belonging to any loop.*/
18 *attach_succ_conditional_non_loop_subnet* (`PP, Subnets[], i`)
19 **end**
20 **else**
  /* $\mathcal{N}$ is the `Last` subnet, do nothing.*/
21 **end**

Algorithm 10: *attach_maximum_trans* (`PP, Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The overall `PRES+ PP` and the `Subnets[]` array are modified to reflect the attachment of the maximum transition of the i^{th} subnet to its successors' maximum transitions.

/* The two successors of the subnet $\mathcal{N}$.*/
$\mathcal{N}_t \leftarrow \mathcal{N}$.true_succ
$\mathcal{N}_f \leftarrow \mathcal{N}$.false_succ
const_guarded_id_trans (`PP, Subnets[], i`, v, $\&t_t$, $\&t_f$) /* Constructs guarded id trans for the given variable in the two successors.*/
$p \leftarrow$ *const_new_place* (`PP, Subnets`, $\mathcal{N}_f$, -1) // A new Dummy place.
attach_as_post_place (`PP`, p, $\mathcal{N}$`.maximum_trans`)
attach_as_pre_place (`PP`, p, t_t)
attach_as_pre_place (`PP`, p, t_f)
/* Attach the newly constructed transitions t_t and t_f to the respective maximum transition sync places.*/
attach_as_post_place (`PP`, $\mathcal{N}_t$`.mt_sync_pl`, t_t)
if $\mathcal{N}_f$`.mt_sync_pl` $\neq -1$ **then**
    /* There exists a maximum trans sync place in the false successor. It may not exist if $\mathcal{N}$ is a loop header.*/
    attach_as_post_place (`PP`, $\mathcal{N}_f$`.mt_sync_pl`, t_f)
end

Algorithm 11: *attach_succ_normal_non_loop_subnet* (`PP, Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The overall `PRES+ PP` and the `Subnets[]` array are modified to reflect the attachment of a `Normal` subnet **not** belonging to a loop to the successor of the subnet.

$\mathcal{N}_t \leftarrow \mathcal{N}$.true_succ // True successor of $\mathcal{N}$.
foreach *input place* p *of* $\mathcal{N}_t$ **do**
    if $p.type =$ `Dummy` $\vee\ \mathcal{N}.ldt[v] = -1$ **then**
        add_to_local_input_places_array (`Subnets`, i, p)
    else
        $v \leftarrow p.var$ // The variable associated with the place p.
        attach_as_post_place (`PP`, p, $\mathcal{N}.ldt[v]$)
    end
end

Algorithm 12: *attach_succ_normal_loop_subnet* (`PP, Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The overall `PRES+ PP` and the `Subnets[]` array are modified to reflect the attachment of a `Normal` subnet not belonging to a loop to the successor of the subnet.

```
1  N_t ← N.true_succ // True successor of N.
2  attach_succ_normal_non_loop_subnet (PP, Subnets[], i)
4  iv ← get_interface_vars (bba[], Subnets[], i) /* Returns an array
       containing the union of the live variables at the end of the predecessors of the subnet
       N.*/
5  t_m ← N_t.maximum_trans // Maximum trans of the true succ.
6  foreach variable v in the array iv do
7  |  if N_t.interface_places[v] = −1 then
9  |  |  p ← const_new_place (PP, Subnets, i, -1) // A new Dummy place.
10 |  |  N_t.interface_places[v] ← p
11 |  |  attach_as_pre_place (PP, p, t_m) // Attaches p to t_m as a pre-place.
12 |  end
13 end
```

Algorithm 13: *attach_succ_cond_non_loop_subnet* (`PP, Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The overall `PRES+ PP` and the `Subnets[]` array are modified to reflect the attachment of a `Conditional` subnet, not belonging to any loop, to the two successors of the subnet.

```
   /* The two successors of the subnet N.*/
1  N_t ← N.true_succ
2  N_f ← N.false_succ
3  c ← N.cond // Contrition associated with the subnet N.
4  foreach variable v live at the end of the subnet N do
6  |  const_guarded_id_trans (PP, Subnets[], i, v, &t_t, &t_f) /* Constructs
   |     guarded id trans for the given variable in the two successors.*/
7  |  foreach input place p of N_t having v as the associated variable do
8  |  |  attach_as_post_place (PP, p, t_t)
9  |  end
10 |  foreach input place p of N_f having v as the associated variable do
11 |  |  attach_as_post_place (PP, p, t_f)
12 |  end
13 end
```

Algorithm 14: *attach_succ_cond_loop_subnet* (`PP, Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The overall `PRES+ PP` and the `Subnets[]` array are modified to reflect the attachment of a `Conditional` subnet, belonging to a loop, to the two successors of the subnet.

/* The two successors of the subnet $\mathcal{N}$.*/
$\mathcal{N}_t \leftarrow \mathcal{N}$.true_succ
$\mathcal{N}_f \leftarrow \mathcal{N}$.false_succ
$iv_t \leftarrow$ *get_interface_vars* (`bba[], Subnets[],` $\mathcal{N}_t$) /* Returns an array containing the union of the live variables at the end of the predecessors of the subnet $\mathcal{N}$.*/
$iv_f \leftarrow$ *get_interface_vars* (`bba[], Subnets[],` $\mathcal{N}_f$)
$iv \leftarrow iv_t \cup iv_f$ // Take union of the two arrays.
$c \leftarrow \mathcal{N}$.`cond` // Condition associated with the subnet $\mathcal{N}$.
foreach *variable* v *in the array* iv **do**
    const_guarded_id_trans (`PP, Subnets[], i,` v, $\&t_t$, $\&t_f$) /* Constructs guarded id trans for the given variable in the two successors.*/
    foreach *input place* p *of* $\mathcal{N}_t$ *having* v *as the associated variable* **do**
        attach_as_post_place (`PP,` p, t_t)
    end
    foreach *input place* p *of* $\mathcal{N}_f$ *having* v *as the associated variable* **do**
        attach_as_post_place (`PP,` p, t_f)
    end
    if $\mathcal{N}_t$.*interface_places*[v] $= -1$ **then**
        $p \leftarrow$ *const_new_place* (`PP, Subnets,` $\mathcal{N}_t$, -1) /* A new Dummy place.*/
        $\mathcal{N}_t$.`interface_places`[v] $\leftarrow p$
        attach_as_pre_place (`PP,` p, $\mathcal{N}_t$.`maximum_trans`) /* Attaches p to t_m as a pre-place.*/
    else
        $p \leftarrow \mathcal{N}_t$.`interface_places`[$v$]
    end
    attach_as_post_place (`PP,` p, t_t)
    if $\mathcal{N}_f$.*interface_places*[v] $= -1$ **then**
        $p \leftarrow$ *const_new_place* (`PP, Subnets,` $\mathcal{N}_f$, -1)
        $\mathcal{N}_f$.`interface_places`[v] $\leftarrow p$
        attach_as_pre_place (`PP,` p, $\mathcal{N}_f$.`maximum_trans`)
    else
        $p \leftarrow \mathcal{N}_f$.`interface_places`[$v$]
    end
    attach_as_post_place (`PP,` p, t_f)
end

Algorithm 15: *attach_succ_loop_header* (`PP, Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: The overall `PRES+ PP` and the `Subnets[]` array are modified to reflect the attachment of a loop header to its two successors.

/* The two successors of the subnet $\mathcal{N}$.*/
1 $\mathcal{N}_t \leftarrow \mathcal{N}$.true_succ
2 $\mathcal{N}_f \leftarrow \mathcal{N}$.false_succ
3 $iv_t \leftarrow get\_interface\_vars$ (`bba[], Subnets[],` $\mathcal{N}_t$) /* Returns an array containing the union of the live variables at the end of the predecessors of the subnet $\mathcal{N}$.*/
4 $iv_f \leftarrow get\_interface\_vars$ (`bba[], Subnets[],` $\mathcal{N}_f$)
5 $iv \leftarrow iv_t \cup iv_f$ // Take union of the two arrays.
6 $c \leftarrow \mathcal{N}$.`cond` // Condition associated with the subnet $\mathcal{N}$.
7 **foreach** *variable* v *in the array* iv **do**
9 *const_guarded_id_trans* (`PP, Subnets[], i`, v, $\&t_t$, $\&t_f$) /* Constructs guarded id trans for the given variable in the two successors.*/
10 **foreach** *input place* p *of* $\mathcal{N}_t$ *having* v *as the associated variable* **do**
12 *attach_as_post_place* (`PP`, p, t_t)
13 **end**
14 **foreach** *input place* p *of* $\mathcal{N}_f$ *having* v *as the associated variable* **do**
15 **if** *is_var_defined_in_loop* (`bba[]`, i, v) = `True` **then**
17 *attach_as_post_place* (`PP`, p, t_f)
18 **else**
19 *add_to_local_input_places_array* (`Subnets`, i, p)
20 **end**
21 **end**
22 **if** $\mathcal{N}_t$.`interface_places[`v`]` $= -1$ **then**
24 $p \leftarrow$ *const_new_place* (`PP, Subnets`, $\mathcal{N}_t$, -1) /* A new Dummy place.*/
25 $\mathcal{N}_t$.`interface_places[`v`]` $\leftarrow p$
26 *attach_as_pre_place* (`PP`, p, $\mathcal{N}_t$.`maximum_trans`) /* Attaches p to t_m as a pre-place.*/
27 **else**
28 $p \leftarrow \mathcal{N}_t$.`interface_places[`$v$`]`
29 **end**
30 *attach_as_post_place* (`PP`, p, t_t)
31 **if** $\mathcal{N}_f$.`interface_places[`v`]` $= -1$ **then**
33 $p \leftarrow$ *const_new_place* (`PP, Subnets`, $\mathcal{N}_f$, -1)
34 $\mathcal{N}_f$.`interface_places[`v`]` $\leftarrow p$
35 *attach_as_pre_place* (`PP`, p, $\mathcal{N}_f$.`maximum_trans`)
36 **else**
37 $p \leftarrow \mathcal{N}_f$.`interface_places[`$v$`]`
38 **end**
39 *attach_as_post_place* (`PP`, p, t_f)
40 **end**

Algorithm 16: *get_interface_vars* (`bba[], Subnets[], i`)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: Returns an array containing the interface variables for the i^{th} subnet.

```
1 iv ← φ // Initialize to an empty array
2 if ∃ a Conditional processor of 𝒩 then
3 |   foreach processor pr of 𝒩 do
4 |   |   iv ← iv ∪ pr.live_out /* Union with the live vars at end of each
  |   |     predecessor.*/
5 |   end
6 else
7 |   iv ← 𝒩.live_in // Live vars at the beginning of 𝒩.
8 end
9 return iv;
```

Algorithm 17: *is_var_defined_in_loop* (`bba[]`, i, v)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: Returns `True` if the given variable is defined in the given subnet; `False` otherwise.

```
/* The two successors of the subnet 𝒩.*/
1 𝒩_t ← 𝒩.true_succ
2 𝒩_f ← 𝒩.false_succ
3 foreach definition d of the variable v in the reaching definitions-in set of
  the basic block (subnet) 𝒩_f do
4 |   b ← d mod MAX_BB_STATEMENTs /* b contains the index of the subnet basic
  |     block to which the definition d belongs.*/
5 |   if 𝒩_t ≤ b ≤ i then
6 |   |   return True;
7 |   end
8 end
9 return False;
```

Algorithm 18: *const_guarded_id_trans* (`PP, Subnets[], i`, v, $\&t_t$, $\&t_f$)

Inputs : The input parameters have their usual meaning as in the preceding algorithms.

Outputs: This function constructs guarded identity transitions for the variable v in the two successors of the i^{th} subnet. The overall `PRES+ PP` and the `Subnets[]` array are modified accordingly.

1 $N_t \leftarrow N_i.true\_successor$ // The true successor of N_i.
2 $N_f \leftarrow N_i.false\_successor$ // The false successor of N_i.
4 $t \leftarrow const\_new\_trans$ (`PP, Subnets,` i, `NULL`)
6 $t' \leftarrow const\_new\_trans$ (`PP, Subnets,` i, `NULL`)
8 $t.guard \leftarrow N_i.cond$ /* Set the condition c in the `if` clause of the `Conditional` subnet N_i as the *guard* of the transition t*/
9 $t'.guard \leftarrow \neg N_i.cond$ /* Set the negation of the condition c*/
10 **if** $v \neq -1$ **then**
    /* v does not represent a Dummy variable.*/
11 $p \leftarrow const\_new\_place$(`PP, Subnets,`$i$, v)
12 $attach\_as\_pre\_place$ (`PP,` p, t)
13 $attach\_as\_pre\_place$ (`PP,` p, t')
14 **if** $N_i.ldt[v] \neq -1$ **then**
15 $attach\_as\_post\_place$ (`PP,` p, $N_i.ldt[v]$)
16 **else**
17 $add\_to\_local\_input\_places\_array$ (`Subnets,` i, p)
18 **end**
19 **end**
20 $\mathcal{V} \leftarrow \Phi$ // An empty set of variables.
21 $\mathcal{V} \leftarrow \{v'$ `s.t.` v' is a variable used in the condition c $\}$
22 **foreach** *variable* x *in the set* $\mathcal{V} \setminus \{v\}$ **do**
23 $p \leftarrow const\_new\_place$(`PP, Subnets,`$i$, x)
24 $attach\_as\_pre\_place$ (`PP,` p, t)
25 $attach\_as\_pre\_place$ (`PP,` p, t')
26 **if** $N_i.ldt[x] \neq -1$ **then**
27 $attach\_as\_post\_place$ (`PP,` p, $N_i.ldt[x]$)
28 **else**
29 $add\_to\_local\_input\_places\_array$ (`Subnets,` i, p)
30 **end**
31 **end**
32 $^*t_t \leftarrow t$
33 $^*t_f \leftarrow t'$

Bibliography

[1] Alfred Aho, Jeffrey Ullman, Monica S. Lam, and Ravi Sethi. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[2] Soumyadip Bandyopadhyay, Dipankar Sarkar, and Chittaranjan A. Mandal. An efficient path based equivalence checking for petri net based models of programs. In Santonu Sarkar, Ashish Sureka, Domenico Cotroneo, Nishant Sinha, Vibha Singhal Sinha, Radhika Venkatasubramanyam, Padmaja Joshi, R. D. Naik, Pushpendra Singh, and JayPrakash Lalchandani, editors, *Proceedings of the 9th India Software Engineering Conference, Goa, India, February 18-20, 2016*, pages 70–79. ACM, 2016.

[3] Luis Alejandro Cortes, Luis Alej, Ro Corts, Petru Eles, and Zebo Peng. A petri net based model for heterogeneous embedded systems. In *in Proc. NORCHIP Conference*, pages 248–255, 1999.

[4] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. An equivalence-checking method for scheduling verification in high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):556–569, March 2008.