PATH BASED EQUIVALENCE CHECKING OF PETRI NET REPRESENTATION OF PROGRAMS FOR TRANSLATION VALIDATION

Soumyadip Bandyopadhyay

PATH BASED EQUIVALENCE CHECKING OF PETRI NET REPRESENTATION OF PROGRAMS FOR TRANSLATION VALIDATION

Thesis submitted in partial fulfillment
of the requirements for the award of the degree

of

Doctor of Philosophy

by

Soumyadip Bandyopadhyay

Under the supervision of

Dr. Chittaranjan Mandal
and
Dr. Dipankar Sarkar

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

September 2015

APPROVAL OF THE VIVA-VOCE BOARD

Certified that the thesis entitled **"Path Based Equivalence Checking of Petri Net Representation of Programs for Translation Validation"** submitted by **Soumyadip Bandyopadhyay** to the Indian Institute of Technology, Kharagpur, for the award of the degree of Doctor of Philosophy has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

(Member of the DSC) (Member of the DSC)

(Member of the DSC) (Member of the DSC)

(Supervisor) (Supervisor)

(External Examiner) (Chairman)

Date:

CERTIFICATE

This is to certify that the thesis entitled **"Path Based Equivalence Checking of Petri Net Representation of Programs for Translation Validation"**, submitted by **Soumyadip Bandyopadhyay** to Indian Institute of Technology, Kharagpur, is a record of bona fide research work under our supervision and we consider it worthy of consideration for the award of the degree of Doctor of Philosophy of the Institute.

Chittaranjan Mandal
Professor
CSE, IIT Kharagpur

Dipankar Sarkar
Professor
CSE, IIT Kharagpur

Date:

DECLARATION

I certify that

(a) The work contained in the thesis is original and has been done by myself under the general supervision of my supervisors.

(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 writing the thesis.

(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, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

(f) Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Soumyadip Bandyopadhyay

ACKNOWLEDGEMENT

While I complete the thesis, I express my deep gratitude to my supervisors Prof. Dipankar Sarkar and Prof. Chittaranjan Mandal, they have been source of clarity from my dilemmas, encouragement, critical thinking, and was a constant guidance at all stages of my thesis. I am indebted to them for taking much passion to read each and every sentence carefully; that has helped me a lot to shape this thesis.

Equally, I acknowledge my sincere gratitude to Tata Consultancy Service for their fellowship that partially funded my research work and provided generous travel grants and stipend throughout the Ph.D period.

I owe my thank to my friends Kunal, Sudakhina di, Soumyajit da, Chandan da, Debjit and others for making the lab most enjoyable place in Kharagpur. I thank Debjit for all his system and LaTeX related supports.

My stay in Kharagpur was extremely pleasurable for the constant moral support of my friends Rohan, Soumen, Arindam, Anupam, Tuhin da, Dhole, Saptak, Pradip da, Russel, Joy da, Gopal da, Surojit, Karati, Sumit (Hyda), Tiru Da, Manjira, Ritwika, Sayan da, Sandipan (Jinta), Bappa, Shubu Da, Prasun da, Bishu, Pralay da and many others. They are the source of my continuous motivation.

I acknowledge google.com, which has helped me in searching most of the research papers. A Special thank to JSTOR which is the source of another (re)search in social science during my stay in Kharagpur.

Last but not the least, I am indebted to my Parents, Prof. Sabyasachi Sengupta, Deben Samui, Nilanjana, Late Sailen Jha, Prof.Santonu Sarkar, Prateek, Prof. Deshpandey,Prof. Ashwin Srinivashan, Prof. A Baskar, Prof.Joshi and the other members of my family. They have believed in me through all the stages of my thesis. They have been my strength and support system in each and every step of my academic career.

Soumyadip Bandyopadhyay

ABSTRACT

A user written application program goes through significant optimizing and parallelizing transformations, both (compiler) automated and human guided, before being mapped to an architecture. Formal verification of these transformations is crucial to ensure that they preserve the original behavioural specification. PRES+ model (Petri net based Representation of Embedded Systems) encompassing data processing is used to model parallel behaviours more vividly. Being value based with a natural capability of capturing parallelism, PRES+ models depict such data dependencies more directly; accordingly, they are likely to be more convenient as the intermediate representations (IRs) of both source and transformed codes for translation validation than strictly sequential, variable-based IRs like Finite State Machines with Data path (FSMDs) (which are essentially sequential control and data flow graphs (CDFGs)). This thesis presents two translation validation techniques for verifying optimizing and parallelizing code transformations by checking equivalence between two PRES+ models, one representing the source code and the other representing its optimized and (or) parallelized version.

Any path based (symbolic) program analysis method consists in introducing cut-points in the loops so that each loop is cut in at least one cut-point; this step permits us to visualize any computation of a program as a sequence of finite paths. Once computations are posed in terms of paths in the above manner, a path based equivalence checking strategy consists in finding equivalent paths in the models. Unlike sequential CDFG models like FSMDs, for PRES+ models, such a sequence is expected to have parallel paths. It is, however, found that *apparently* cutting only the loops is not adequate to capture a computation as a sequence of parallel paths. The dissertation first describes a method of introducing cut-points so that a computation can be posed as a sequence of parallel paths. This method is referred to as *dynamic cut-point induced path based equivalence checking* – "dynamic" because additional cut-points over and above those introduced to cut the loops are needed for the purpose and the method entails a symbolic execution of the model keeping track of the tokens and disregarding the symbolic values. Subsequently, we also reveal that it is possible to have a valid path based equivalence checking strategy even when the conventional approach of introducing cut-points only to cut the loops is followed. This method is referred to as *static cut-point induced path based equivalence checking*.

Correctness and complexity of the two methods have been treated formally. The methods have been implemented and tested and compared on several sequential and parallel benchmarks. While underscoring the effectiveness of equivalence checking as a method for verification of machine independent optimizing and parallelizing passes of compilers, the dissertation discusses some limitations of the work and identifies some future directions in which it can be enhanced.

Keywords: Translation Validation, Equivalence Checking, PRES+ Model (Petri net based Representation of Embedded Systems), Path Based Program Analysis, Finite State Machine with Datapath (FSMD), Control and Data Flow Graph (CDFG).

Contents

List of Symbols

List of Figures

List of Tables

Chapter 1

Introduction

Recent advancement of multi-core and multi-processor systems has enabled incorporation of concurrent applications through extensive optimizing transformations for better time performance and resource utilization [56]. Several code transformation techniques such as, code motions, common sub-expression elimination, dead code elimination, etc., several loop based transformation techniques such as, un-switching, reordering, skewing, tiling, unrolling, etc., and several thread level parallelizing transformations such as, loop distribution, loop parallelizing [6], etc., may be applied on the application programs at the prepossessing stage of system synthesis. These transformations are carried out by some compilers or design experts. Even for the former case, if such optimizations are carried out by untrusted compilers, they can result in software bugs. Validation of a compiler ensuring correct by construction property is a very difficult task. Instead, using behavioural verification techniques, it is possible to verify whether the optimized output of each run of the compiler faithfully represents the functionality captured in the input source code.

For behavioural verification, it is necessary to represent a program into a formal computational model. A comprehensive list of models proposed to represent programs for various application areas and the analysis mechanisms around these models can be found in [7, 8, 45, 99]. Petri nets have long been popular for modeling concurrent behaviours [23, 105, 126, 137]. The untimed PRES+ model (Petri net based Representation for Embedded Systems), reported in [37, 38], *enhances the classical Petri net model to capture natural concurrency present in programs; they have well* *defined semantics of computations over integers, reals and general data structures*. In essence, the enhancement of Petri net models to PRES+ involves permitting the places to hold tokens with data values and the transitions to have associated data transformations and conditions of executions. Analyses of dependencies among the operations in a program lie at the core of many optimizing transformations. Being value based with an inherent scope of capturing parallelism, the PRES+ models depict such data dependencies more directly; accordingly, they are likely to be more convenient intermediate representations (IRs) of both source and transformed codes for translation validation than strictly sequential variable-based IRs like all types of control data-flow graphs, communicating sequential processes [65], etc. Accordingly, in the present work this modelling paradigm has been chosen.

Behavioural verification involves demonstrating the output equivalence of all computations represented by the original behavioural description with those of the transformed behavioural description on identical inputs. From the success of path based equivalence checking of CDFG models, designated as Finite State Machine with data paths (FSMD) [20, 74], it is perceived that a similar approach is worth pursuing for PRES+ models. Path structures in PRES+ models, however, are far more complex than those in CDFG models due to the presence of parallel threads of computations in the former. The present thesis identifies certain issues arising out of the complexity of path structures in PRES+ models and presents some mechanisms to address them in course of devising two path based equivalence checkers for PRES+ models.

1.1 Literature Survey

Behavioural transformation techniques are used extensively on the source program in the code optimization phase of any optimizing and parallelizing compiler to obtain optimal performance in terms of execution time, energy, etc. In this section, we briefly describe several such behavioural transformations that are commonly applied by the compilers and the different verification approaches adopted for their validation.

1.1.1 Code motion transformations

Code motion is an optimization technique to improve the effectiveness of a program by avoiding unnecessary re-computations [81]. The other objective is the minimization of lifetimes of the temporary variables to avoid unnecessary resource allocation. This can be achieved moving the operations beyond the basic block boundaries. The code motion based transformation techniques can be classified into the following categories as reported in [117]. (1) *Duplicating down* — In this code motion technique, the operations are moved from a basic-block (BB) preceding a conditional block (CB) to both the BBs following the CB. *Reverse speculation* [57] and *lazy execution* [117] in this category. (2) *Duplicating up* — It involves moving operations from a BB in which conditional branches merge to its preceding BBs in the conditional branches. *Conditional speculation* [57] and *branch balancing* [54] fall in this category. (3) *Boosting up* — In this code motion technique, operations move from a BB within a conditional branch to the BB preceding the CB from which the conditional branch sprouts. Code motion techniques such as *speculation* [57] lie in this category. (4) *Boosting down* — In this code motion technique, operations move from BBs within some conditional branches to a BB following the merging of the conditional branches [117]. (5) *Useful move* — It refers to moving an operation to a control and data equivalent block. When an operation moves from a BB preceding a CB to only one of the conditional branches, or vice-versa, then such type of code motion is called *non-uniform* code motion. On the other hand, a code motion is said to be *uniform* when an operation moves from both the conditional branches to a BB before or after the CB or vice-versa. Therefore, duplicating up and boosting down are uniform code motions type whereas *duplicating down and boosting up can be of uniform as well as non-uniform code motion type.*

1.1.2 Loop transformations

The loop transformation techniques are used to increase instruction level parallelism, improve data locality and reduce overheads associated with executing loops of both scalar and array-intensive applications [10]. The execution of any scientific program is mostly spent on loops. Thus, a lot of compiler analyses and compiler optimization techniques have been developed to make the execution of loops faster. *Loop* *fission/distribution/splitting* for scalar programs attempt to break a loop into multiple loops each comprising statement of codes which are independent of each other. The inverse transformation of loop fission is loop *fusion/jamming*. *Loop unrolling* reproduces the body of a loop by some number of times called unrolling factor. Unrolling improves performance by reducing the number of times the loop condition is tested and by increasing instruction level parallelism. *Loop interchange/permutation* exchanges two loops. Such a transformation can improve the locality of reference. *Loop unswitching* moves a conditional from inside a loop to outside by duplicating the loop body. Some other important loop transformations are loop *reversal, spreading, peeling*, etc. [10].

1.1.3 Parallelizing transformations

Several applications like multimedia, image processing, signal processing and bioinformatics must achieve a high computational power with minimal energy consumption. Given these constraints, multiprocessor implementations not only deliver the necessary power of computation, but also provide the efficiency of power. However, the performance gain achieved is dependent on how well the compiler can parallelize the given program and generate code for the same so that it can be mapped to the architecture [2, 24]. There are three types of parallelism of the sequential programs: (i) Loop-level parallelism: The iterations of a loop are distributed over multiple processors. (ii) Data-parallelism: data parallelism is achieved when each processor performs the same task on different pieces of distributed data. (iii) Task-level parallelism: Task parallelism focuses on distributing sub-tasks of a program across different parallel processors. The sub-tasks can originate from different subroutine, independent loops, or even independent basic blocks. This is the most used parallelizing technique.

A parallel behaviour is obtained from a sequential behaviour using a parallelizing code transformation. Parallel transformations manipulate the concurrency level of the parallel programs [63]. The concurrency level of a program may not match the parallelism of the hardware. The transformations like *loop merging and splitting, process merging and splitting and computation migration, etc.*, are commonly used for this purpose.

1.1.4 Techniques for verification of behavioural transformations

Application of code motion techniques during the pre-processing phase of embedded system design increases verification challenges significantly. Some verification techniques have been reported in [96, 111] for transformations where a code never moves beyond the basic block boundary. Some recent works [128], [78], [84] target verification of such code motions. For example, a path recomposition based FSMD equivalence checking method has been proposed in [78] to verify speculative code motions. In [84], code motions are verified using a translation validation approach. The equivalence checking method for scheduling verification reported in [74] works well even when the control structure of the input behaviour is modified by the scheduler. The method can also verify uniform code motion techniques (whereupon code preceding a conditional block are moved to both the branches emanating from the block).

A bisimulation-based translation validation approach capable of verifying structure preserving and reordering loop transformations has been introduced by Pnueli et al. [109, 110]. This method were demonstrated by Necula in [106] and Rinard et al. [118]. This method is further enhanced by Kundu et al. [84] to verify the high-level synthesis tool SPARK capturing parallel execution of statements. A bisimulation method for concurrent programs is reported in Milner et al. [99]. The basic idea of bisimulation method is that the number of iterations of some corresponding loops in the source and the target programs must be the same but their order may be different. So, there must be a one-to-one correspondence between the iterations of the source and the transformed programs made available to the validation method. Based on that, a set of permutation rules [25], which establish that the transformed code satisfies all the implied data dependencies necessary for the validity of certain transformations, is presented. The theorem prover CVC or Z3 [5, 125] is used to check the permutation rules. The method relies on inputs from the optimizing compiler indicating the transformation rules that have been applied to decide which inference rules to apply. Some recent works [22], [67] following this approach focus on defining permutation rules for other transformations. These methods apparently fail when two loop bodies have been merged into one, or a single loop is split into two, such as, loop peeling, loop merging or loop spreading. In case of loop unrolling, the numbers of iterations of the loop in the source and the transformed programs are different. Using separation logic,

$C11$ compiler is verified reported in [131]. Mateev et al. [98] proposed a technique called fractal symbolic analysis for verification of loop transformations. Their idea is to reduce the difference between two programs by repeatedly applying simplification rules until two programs become close enough to allow a proof by symbolic analysis.

All the above mentioned works use sequential model of computation (MoC) like CDFG models for translation validation. Not many works have been reported on transformation validation using PRES+ models. Cortes et al.[38] have introduced the notion of functional and time equivalence of PRES+ models. According to this work, two PRES+ models are defined to be functionally and time equivalent if and only if after their execution on the same inputs, the token values and the token times at the output ports are the same. Being simulation based, it is not a formal verification method.

1.1.5 Objective of the work

Unlike variable based models, PRES+ models are value based; instead of storing the values of a variable obtained at various points of a computation in the specific location designated for the variable, each newly computed value is held in a place; if such a value (of the variable) is used k times before a new value is computed, then k such places are used to hold these values. This aspect along with the underlying structure of a Petri net enables a PRES+ model to capture the inherent scope of parallelism among data independent operations of a given program more vividly through its structure. Analysis of data dependence lies at the heart of most of the code optimizing and parallelizing transformations. Hence it is felt that if both source program and its transformed version are represented using PRES+ models, then they are likely to become structurally similar making the task of equivalence checking easier. Accordingly, the objective of this work is set to devise path based equivalence checking methods for PRES+ models for validating several optimizing and parallelizing transformations. In course of pursuing the work we have identified two ways to achieve this objective. In the first one, the path boundaries are so ascertained that any computation can be syntactically decomposed as a concatenation of parallel paths of the model. In the second method, in keeping with the convention used for analyzing sequential CDFGs, the path boundaries are ascertained so that each single iteration of any loop is captured by a path.

1.2 Contributions of the thesis

The primary aim of this dissertation work is to devise path based equivalence checking strategies for two PRES+ models, one representing a source code and the other representing its transformed version, obtained by application of some optimizing and parallelizing transformations on the source code. Any path based (symbolic) program analysis method consists in introducing cut-points in the loops so that each loop is cut in at least one cut-point; this step permits us to visualize any computation of a program as a sequence of finite paths. Once computations are posed in terms of paths in the above manner, a path based equivalence strategy consists in finding equivalent paths in the models. Unlike sequential CDFG models like FSMDs, for PRES+ models, such a sequence is expected to have parallel paths. It is, however, found that *apparently* cutting only the loops is not adequate to capture a computation as a sequence of parallel paths. The dissertation first describes a method of introducing cut-points so that a computation can be posed as a sequence of parallel paths. This method is referred to as *dynamic cut-point induced path based equivalence checking* – "dynamic" because additional cut-points, over and above those introduced to cut the loops, are needed for the purpose and the method entails a symbolic execution of the model keeping track of the tokens and disregarding the symbolic values, hence referred to as *token tracking execution*. We then reveal that it is possible to have a valid path based equivalence checking strategy even when the conventional approach of introducing cut-points only to cut the loops is followed. This second method is referred to as *static cut-point induced path based equivalence checking*. In the following subsections, we describe the formal vocabulary developed by us for devising the two equivalence checking methods and then present the respective contributions of the work in these two directions.

1.2.1 Dynamic Cut-point Induced Path Based Equivalence Checking Method

The basic steps of a path based equivalence checking procedure are as follows: (1) In the first step, a PRES+ model is partitioned into several fragments which are called *paths*; the paths are obtained by cutting a loop in at least one cut-point which is adopted from [50]; any computation of the model can now be represented as a concatenation of these paths. (2) It is then checked whether for all paths in the PRES+ model N_0 corresponding to the source program, there exists a path in the PRES+ model N_1 corresponding to the transformed program such that the two paths are equivalent, i.e., their data computations and conditions of execution are identical and their input and output places have correspondence. (3) Steps 1 and 2 are then repeated with N_0 and N_1 interchanged. The major challenges of the task of establishing equivalence between two PRES+ models are as follows:

1. Devising a path construction procedure for a PRES+ model
2. Devising an equivalence checking method for PRES+ models.

Formally, a path in a PRES+ model is a sequence of sets of parallelisable transitions having all the pre-places of the first set and the post-places of the last set as cut-points and having no other intermediate sets of transitions with their post-places as cut-points. It has been identified that in order to capture a computation by a sequence of parallelisable paths, we need additional cut-points over and above those which only cut the loops. Accordingly, the path construction mechanism consists in first introducing *static* cut-points at the entry points of the loops. Then additional cut-points are introduced; this step involves a *token tracking execution* of the model. In course of such an execution, whenever a set of token holding places are reached containing at least one cut-point, all the other places in the set are also marked as *dynamic* cut-points. We have devised an algorithm, implemented it in *C*, experimented with some hand fabricated examples and examples taken from [56]. We provide formal proofs of the following facts: (i) The set of paths obtained using the set of static and dynamic cut-points covering all transitions is unique, (ii) Any computation of the model can be viewed as a sequence of sets of parallelisable paths taken from the set, (iii) static and dynamic cut-point induced path based equivalence checking mechanism is valid, and (iv) the path construction is sound and complete. The complexity analysis of the algorithm has also been carried out. This work has been accepted for publication in [17].

The equivalence checking method consists in identifying for any path α in the PRES+ model N_0 of the source program, an equivalent path β in the model N_1 of the transformed version of N_0 with identical data transformations and condition of execution and the sets of pre-places of α and β having *correspondence* with each other. The correspondence among the sets of pre-places of the paths of the two models is defined in course of identifying the equivalent paths starting with the sets of input ports and those of the output ports having correspondence with each other, respectively given by the bijections f_{in} and f_{out}. Because of code motion transformations used to obtain the transformed programs, a need arises for *extending* paths. The correctness of the algorithm has been treated by showing its termination and soundness; it may be noted that the problem of equivalence checking being not even semi-decidable [65], the algorithm providing a partial procedure cannot be complete. A complexity analysis has been carried out. This work has been published in [16, 17].

1.2.2 Static Cut-point Induced Path Based Equivalence Checking

In this part of the work, we identified that even with static cut-points cutting the loops at the at their respective entry points, we can obtain a set of paths which captures a computation; more precisely, such a set has the property that any model computation μ can be represented as a sequence of paths which is *computationally equivalent to μ although, unlike the case of dynamic cut-point induced paths, such a sequence does not maintain a syntactic identity with μ; this follows from the* serializability of parallelisable transitions and parallelisability and commutativity of independent transitions. This necessitated a change in the definition of a path as a sequence of parallelisable transitions having at least one cut-point among the pre-places of the first set and one among the post-places of the last set with no cut-points in the post-places of any intermediate sets. The definition of equivalence of paths and the correspondence relation over the sets of places of the two models remain the same. The fact that static cut-point induced path based equivalence checking strategy is a valid one has been proved. The equivalence checking algorithm did not need any path extension because the paths in this case have wider expanses encompassing the moved code in the cases of code motion transformations even across loops. The algorithm has been proved correct by showing its termination and soundness. This work has been reported in [18, 19].

Both the equivalence checking procedures have been implemented in *C*. We have carried out experimentation along two courses. The first one has used hand constructed models and the second course of experimentation has been carried out using an automated model constructor which has been completed subsequently (and described in [122]). The automated model constructor ensures that the constructed models always preserve the one-safe property. Likewise, our hand constructed models also ensure that the model is one-safe; we have satisfactorily tested both the methods on several sequential [14] and parallel examples [14]. Translation is carried out by one HLS (high level synthesis) compiler, i.e., SPARK [56] and two thread level parallel compilers PLuTo [24] and Par4All[2]. All the examples involved source to source translation of *C* programs by the compilers. For checking equivalence between two paths, a normalizer [20, 121] has been used. For sequential examples, we have compared the two methods described in this work with the path extension based FSMD equivalence checking [74] and the value propagation based FSMD equivalence checking [20]. The performance of the dynamic cut-point (DCP) induced path based equivalence checking method are found to be lower than, but within comparable limits of, those of the FSMD based equivalence checking methods. The DCP method around the PRES+ model and the path extension based equivalence checker go through path extension which is costly. The static cut-point (SCP) induced path based equivalence checking method of PRES+ models is also found to be comparable with FSMD based equivalence checking method of [20] as well as DCP induced path based equivalence checking method. Data independent loop interchanging transformations, which cannot be effectively handled by any other technique at present, can be handled by our method. This benefit accrues from a PRES+ model based method because the data flow is captured more directly in the model using as many places as the number of times a definition is used. In SCP induced path based equivalence checking method, the costly path extension step is not needed. For the parallel examples, we have only compared the two equivalence checking methods described in this work; the FSMD based method of [20] does not handle parallel programs. The SCP induced path based equivalence checking method is found to be comparable with the DCP induced method for such programs too. During experimentation with parallel examples, our equivalence checker has identified a bug of the PLuTo compiler (possibly due to faulty usage of a variable name in the source program).

1.2.3 Thesis Organization

Chapter 1 provides a background, motivations, objectives, contributions and organization of the thesis.

Chapter 2 provides a detailed literature survey on different code motions, loop transformations and parallelizing transformations and their validations approaches.

Chapter 3 presents the PRES+ model description and its computational semantics formally.

Chapter 4 describes the path construction algorithm using dynamic cut-points where a computation can be represented as a concatenation of parallel paths.

Chapter 5 describes the equivalence checking method between two PRES+ models using dynamic cut-points.

Chapter 6 describes an efficient path construction algorithm using static cut-points and also the corresponding equivalence checking mechanism.

Chapter 7 concludes by summarizing the contributions made through this work and discusses some potential future research directions.

Chapter 2

Literature Survey

This chapter presents an overview of some important research contributions on various verification techniques for Petri net based models and secondly, in the area of behavioural equivalence checking based validation of code optimization and parallelizing transformations carried out by compilers. For each transformation, we first underline its role in optimization and parallelization of the code and then present a survey of the existing verification methodologies identifying certain limitations of these methodologies which have been addressed in this thesis.

2.1 Verification techniques for Petri net based models

In the present section, we focus on the literature on formal verification of some complex systems, like embedded systems, modelled as Petri nets. Many models have been proposed to represent embedded systems [45, 111] encompassing a broad range of styles and characteristics depending upon application domains of the systems; they include extensions of finite state machines, data flow graphs, communication processes and Petri nets. In [39, 116], one-safe Petri net based MoCs coded in the language PROMELA is used for designing embedded systems. The PRES+ model is first proposed in [37, 38] where a translation technique from PRES+ models to timed automata (TA) is presented; the TA (without data variables) so obtained is then used to verify some safety properties of several embedded software using the UPPAAL verification tool [4]. Literature [94] reports a similar technique where time Petri nets (TPN) are translated to TA in UPPAAL's input format to verify several safety, reachability and liveness properties using the tool. Some compositional verification techniques for Petri net based models are reported in [46, 80]. A SAT-based bounded model checking for concurrent and asynchronous systems for the safe Petri nets is reported in [113]. Petri net models have been used in verification of distributed algorithms in [29]. Literature [36] reports a verification technique of embedded systems where the program translates from Synchronous and Interpreted Petri net (SIP-net) models to optimized PROMELA code for verification through the SPIN model checker [123]. Verification of multi-agent system behaviour modelled using Petri net is also reported in [28]. Several verification approaches regarding colour Petri nets model are reported in [70, 132]. Safety property verification using Petri net based modelling paradigm is reported in [103]. While all the above works dwell upon property verification, literature [38] provides a notion of equivalence of simulation runs for specific data inputs; no literature is available on model equivalence for symbolic simulation which is needed for formal equivalence checking.

2.2 Code motion transformations

2.2.1 Applications of code motion transformations

Various code motion techniques are applied by the optimizing compilers on programs, in general, and also during the scheduling phase in high-level synthesis and other pre-processing phases of embedded system design. Parallelizing compilers too often use code motion techniques [48, 53, 68, 81, 88, 101, 107, 120]. In the following, we study the applications of several code motion techniques during code optimization.

The works reported in the literature [42, 43] describe in generalized code motions applied during the scheduling phase in the synthesis systems, whereby the operations move globally over the input source code. These works basically identify the solution space and associate some cost with each possible solution; eventually, the solution with the least cost is adopted. For reducing the search time, the methods of [42, 43] propose a pruning technique which intelligently selects the least cost solution from a set of candidate solutions.

Speculative execution is a technique which allows a super-scalar processor to keep its functional units as busy as possible by executing the instructions before they are required; thus, some computations are carried out even before the execution of the conditional operations which decide whether the computation is actually needed. The work reported in [87] describes several techniques to integrate speculative execution into the scheduling phase of high-level synthesis. This work shows that the paths for speculation is needed and accordingly, it decides the criticality of individual operations and the availability of resources in order to obtain maximum benefits. It has been denoted to be a promising technique for eliminating performance bottlenecks imposed by control flows of programs which gives a significant gain (up to seven-fold) in terms of the execution speed. Their method has been integrated into the Wavesched tool [86].

A global scheduling technique for super-scalar and VLIW processors is reported in [100]. This technique parallelizes sequential code by removing anti-dependence (i.e., write-after-read dependence) and output dependence (i.e., write-after-write dependence) in the data flow graph of a program by renaming registers, as and when required. The code motions are applied globally to maintain a data flow attribute indicating at the beginning of each basic block the operations that are available for moving up through this basic block. A similar objective is accomplished in [35]; this work combines the speculative code motion techniques and parallelizing techniques for betterment of control flow scheduling intensive behaviours.

In [71], during the register allocation phase, the code motion methods are merged to obtain a better scheduling of instructions with minimum number of registers. Register allocation can artificially constrain instruction scheduling, while the instruction scheduler can force a weak register allocation. The method reported in this work tries to overcome this limitation by combining these two phases of high-level synthesis.

In [35], a control and data dependence graph (CDFG) is used as an intermediate representation which provides the possibility of extracting the maximum parallelism from the behaviour. This work combines the speculative code motion techniques and parallelizing techniques to improve scheduling of control flow intensive behaviours. Similar techniques have been applied in [21] during analyzing a program to identify the live range overlaps for all possible placements of instructions in the basic blocks and all orderings of instructions within the blocks; based on this information, the authors formulate an optimization problem which determine code motions and the partial local schedules that minimize the overall cost of the live range overlaps. The solutions to the formulated problem are evaluated using integer linear programming. A method for elimination of concurrent copies using code motions on data dependence graphs to optimize register allocation can be found in [26].

The efficacy of traditional compiler techniques employed in high-level synthesis of synchronous circuits is studied for asynchronous circuit synthesis in [136]. It has been shown that the several transformations like speculations, loop invariant code motions and condition expansion, are applicable in decreasing the mass of handshaking circuits and intermediate modules.

Several benefits of applying code motions to improve results of high-level synthesis have also been reported in [55, 57, 58], where a set of speculative code motion transformations that enable movement of operations through, beyond, and into conditionals to maximize performance. The authors also introduced some sophisticated transformations such as speculation, reverse speculation, early condition execution, conditional speculation techniques in [57, 60, 61].

Literature [54, 55] present two novel strategies which increase the scope for application of speculative code motions: (i) Dynamically adding scheduling steps to schedule the conditional branches with fewer scheduling steps; this increases the opportunities to apply code motions, such as conditional speculation, that duplicate operations into the branches of a conditional block. (ii) Determining if an operation can be conditionally speculated into multiple basic blocks either using some existing idle resources or by creating new scheduling steps; this strategy leads to balancing of the number of steps in the conditional branches without increasing the longest path through the conditional block. Classical common sub-expression elimination (CSE) technique fails to eliminate several common sub-expressions in control-intensive designs due to the presence of a complex mix of control and data flow. Aggressive speculative code motions are used to schedule control intensive designs which often re-order, speculate and duplicate operations, changing thereby the control flow between the operations with common sub-expressions. This gives some new opportunities for applying CSE dynamically. This scenario is utilized in [59] to devise a new approach called *dynamic common sub-expression elimination*. The code motion techniques and heuristics described in this paragraph have been implemented in the high-level-synthesis compiler, namely, SPARK [56].

Energy management is an important concern to both hardware and software designers. An energy-aware code motion framework for a compiler is reported in [135] which tries to cluster accesses to input and output buffers, thereby expanding the time period during which the input and output buffers are clocked or power gated. The method [92] attempts to change the data access patterns in the memory blocks by introducing code motions in order to improve the energy efficiency and performance of STT-RAM which is basically a hybrid cache. Some insights into how code motion transformations may aid in the design of embedded reconfigurable computing architectures can be found in [41].

2.2.2 Verification of code motion transformations

Recently, a proof construction [104] mechanism has been proposed to verify some transformations performed by the LLVM compiler [1]; these proofs are then checked for validity using the theorem provers such as Z3 [5] and PVS [3]. Formal verification of single assignment form based optimizations for the LLVM compiler has been reported in [138]. Now, we shall focus on the verification strategies targeting validation of several code motions as mentioned in Section 2.2.1.

A formal verification of the scheduling phase of high level synthesis using the FSMD model is reported in [77]. In this paper, path covers for the two FSMD models are constructed introducing cut-points in the models. Then, for each path in the path cover of one FSMD, the method searches for an equivalent path in the other FSMD. The major requirement of this work is that the control structure of the input FSMD is not disturbed by the scheduling algorithm and no code can be moved beyond the basic block boundaries. This implies that the respective path covers obtained from the cut-points are essentially bijective. The limitation of this method is that such a bijective correspondence does not necessarily hold because the scheduler may merge some paths of the original specification into one path of the implementation or distribute operations of a path over several paths for optimization of time steps.

A Petri net based verification method for checking the correctness of algorithmic transformations and scheduling process in high-level synthesis is proposed in [31].

The initial behaviour is converted first into a Petri net model which is expressed by a Petri net characteristic matrix. Based on the input behaviours, they extract the initial firing pattern. If there exists at least one candidate who can allow the firing sequence to execute legally, then the high-level synthesis result is claimed as a correct solution.

All these validation approaches, however, are well suited for basic block based scheduling [69, 91], where the operations cannot move beyond the the basic block boundaries. and the path-structure of the input behaviour does not change due to scheduling. These techniques are not applicable to the verification of code motion techniques if the code is moved beyond the basic block boundaries.

Some recent work [74, 78, 84] target verification of several code motion techniques. Specifically, a path recomposition based FSMD equivalence checking has been reported in [78] which can verify only the speculative code motions. The conditions for correctness are formulated in higher-order logic and verified using the PVS theorem prover [3]. Recomposition of paths over conditional blocks fails if the non-uniform code motion transformations are applied by the scheduler. A translation validation approach for high-level synthesis is reported [84, 85] where bisimulation relation based approach is used to prove equivalence. This method automatically establishes a bisimulation relation that states which points in the initial behaviour are related to which points in the scheduled behaviour. This method is incapable of finding the bisimulation relation if a code segment preceding a conditional block is not moved to all the branches of that block. The major limitation of this work is that it can fail if the control structure of the initial program is transformed by the path-based scheduler [27]. A path based equivalence checking method has been reported in [75] for uniform code motion validation using FSMD models. The method has been further enhanced in [74] to handle non-uniform code motions as well. The work reported in [89] has identified some false negative cases reported by the algorithm in [75] and proposed an algorithm to overcome these limitations. The path based mechanism of [74] has been modified using a notion of value propagation [20] to widen the scope of the former to cover code motion across loops.

The above verification techniques for translation validation for optimizing compilers fall under two categories namely, path based equivalence checking method, and bisimulation based method. Path based equivalence checking was first proposed by Karfa et. al. [75], whereby the source and the transformed programs are represented as FSMD models which are then segmented into paths; the method consists in showing that for every path in the original FSMD, there exists an equivalent path in the other FSMD, and vice-versa; on successful identification of all pairs of equivalent paths, the two FSMD models are asserted to be equivalent. This method can handle significant modifications of the control structures introduced by path based schedulers [27] as well as dynamic loop scheduling (DLS) [112]. This method is further enhanced in [20, 74] to increase the power of the FSMD equivalence checker for handling diverse code motion transformations.

In the bisimulation based method, transition systems are used to model hardware and software at various abstraction levels. In the lower abstraction level, more implementation details are present thereby permitting less number of computations than the specification. It is, therefore, tried to verify that all the computations of the refinement of a given specification are by some computation permitted by the specification. Thus, the aim of bisimulation equivalence is to identify transition systems with the same branching structure so that they can simulate each other step by step [13]. Bisimulation equivalence establishes the possibility of mutual, step-wise simulation. Bisimulation equivalence is first proposed by Milner et. al. [99] as a binary relation between two communicating systems over the same set of atomic propositions.

Bisimulation method is then modified for labeled transition system which is reported in [47]. This method is also applicable for timed systems [127], well-structured graphs [44], probabilistic processes [11, 12], etc. Scalability issues of bisimulation based approaches are reported in [32, 49, 133].

Translation validation for an optimizing compiler by obtaining simulation relations between programs and their translated versions was first proposed in [110]; such a method is demonstrated by Necula et. al. [106] and Rinder et. al. [118]. The procedure mainly consists of two algorithms – an inference algorithm and a checking algorithm. The inference algorithm collects a set of constraints (representing the simulation relation) using a forward scanning of the two programs and then the checking algorithm checks the validity of these constraints. Depending on this procedure, validation of high-level synthesis procedures are reported in [84, 85]. Unlike the method of [106], their procedure considers statement-level parallelism since hardware can capture natural concurrency and high-level synthesis tools exploit the parallelization of independent operations. Furthermore, the method of [84, 85] uses a general theorem prover, rather than the specific solvers (as used by [106]). On a comparative basis, a path based method always terminates; however, some sophisticated transformations, like loop shifting, remains beyond the scope of the state of the art path-based methods. The loop shifting [40] can be verified by the method reported in [84, 85]. A major limitation the reported bisimulation based method is that the termination is not guaranteed [84, 85, 106]. Also, it cannot handle non-structure preserving transformations by path based schedulers [27, 112]; in other words, the control structures of the source and the target programs must be identical. The authors of [93] have studied and identified what kind of modifications the control structures undergo on application of some path based schedulers; based on this knowledge, they try to establish which control points in the source and the target programs are to be correlated prior to generating the simulation relations. The ability to handle control structure modifications which are applied by [112], however, still remain beyond the scope of the currently known bisimulation based techniques.

A Petri net based verification strategy is described in [15] for sequential high-level synthesis benchmarks for several code motions. In this method, the Petri net representations of an original behaviour and its transformed version are translated into equivalent FSMD models and fed as inputs to the FSMD equivalence checker of [74]; no correctness proof, however, is given for this method; moreover, in the presence of more than one parallel thread, the method fails to construct the equivalent FSMD models.

None of the above mentioned techniques has been demonstrated to handle efficiently code motions across loops as well as code motions for parallel programs [82]. All the methods work only for the sequential MoCs. Hence, it would be desirable to have an equivalence checking method that encompasses parallelism and has the ability to verify efficiently code motions across loops along with uniform and non-uniform code motions transformations where the control structure of the program is altered.

2.3 Several parallelizing transformations

Parallelizing code transformation techniques partition the sequential code into concurrent tasks. Many techniques are reported in [51, 64, 134]. In all of these reported methods, the data parallelism as well as thread level parallelism are mentioned. In [24, 51, 82] some methods have been presented to parallelize a number of divide and conquer algorithms using communication channels.

A tool, called SPRINT, is reported in [34] where a sequential code is automatically transformed to a concurrent SystemC model. While most of the reported techniques exploit data parallelism, SPRINT exploits the functional parallelism in the behaviour to yield parallel tasks where each task implements a different subset of statements. In contrast, the data parallelism consists in executing the same code in parallel on different subsets of data. In [52], an automated mechanism for enhancing the functional parallelism from ordinary programs is presented. In this method, a hierarchical task graph (HTG) is constructed from the initial sequential behaviour. The HTG provides a powerful mechanism for representation of intermediate version which encapsulates parallelism of different types and scope levels leading to generation and optimization of parallel programs. Turjan et al. [129, 130] described an automatic transformation mechanism of nested-loop programs to Kahn process networks (KPNs).

2.3.1 Verification of parallelizing transformations

In [124], an approach of symbolic model checking of process networks is introduced for validating them using their binary decision diagram based models. In this literature, the authors also introduce a representation of multi-valued functions appearing in the process network called interval decision diagrams (IDDs) which can be conventionally used by the symbolic model checker.

Some non-semantic preserving parallelizing transformations of process networks during refinement steps of embedded system design are proposed in [115]. These transformations involve lower level implementation details. In this method, a set of verification properties for every non-semantic-preserving transformation is defined as CTL^* formulae. The verification tasks are divided into two steps: (i) the local correctness of the non-semantic-preserving transformation is checked by preserving properties using a model checking tool, and (ii) the global influence of the refinement to the entire system is studied through static analysis. In [30], the designs at different abstraction levels are automatically translated into PROMELA description and verified using SPIN model checker [66].

When a sequential behaviour is transformed into a parallel behaviour, it is required to ensure that (i) the transformed behaviour must satisfy data-flow properties [62] of the systems and (ii) it is functionally equivalent to the initial behaviour [102]. For the first task, model checking is used as the initial verification approach. The verification models are mechanically generated from both the input and transformed behaviours and then the properties are checked using some model checker like, SPIN [66], NuSMV [33], etc. For the second task, however, model checking cannot be used as it is more appropriate for property verification but not for behavioural verification.

2.4 Conclusion

The literature survey carried out in this chapter clearly reveals that between property verification and functional equivalence checking, the research emphasis is much more in favour of the former compared to the latter. The task of validation of compiler transformations necessitates that the original program and the transformed program should be functionally equivalent. All the computational aspects of a program cannot be captured as some liveliness and safety properties only; such properties capture only certain aspects of a computation at a high abstraction level.

Dependences among the computation steps of a program lie at the heart of transforming programs towards more efficient performance. PRES+ models built upon Petri nets capture such dependences more vividly. As indicated in Section 1.1.5, this aspect of the model apparently makes it a convenient paradigm for optimizing and parallelizing transformation validation; in spite of this apparent potential, the survey reveals that there has not been any transformation validation work using this modelling framework. In the subsequent chapters of this dissertation, we shall study how such mechanisms can be devised and whether they would indeed score favourably over the methods reported around other model(s) of computation.

Chapter 3

PRES+ models and their computations

For formal analysis of a program, it is necessary to represent the program using some equivalent formal modelling paradigm. As the main target of this work is to validate code optimizing and several parallelizing transformations, a parallel model of computation (MoC) is necessary. In this work, the PRES+ model, whose underlying structure is a one-safe Petri net model with tokens being capable of holding values, is selected as the parallel MoC. In this chapter, we first describe the PRES+ model and its computational semantics. We then describe the notion of computational equivalence between two PRES+ models.

3.1 The PRES+ model

A PRES+ model [38] is an eight tuple $N = \langle P, V, f_{pv}, T, I, O, inP, outP \rangle$, where the members are defined as follows. The set $P = \{p_1, p_2, \ldots, p_m\}$ is a finite non-empty set of *places*. The set V is the set of variables of the program which N seeks to model. The mapping $f_{pv} : P \rightarrow V \cup \{\delta\}$ depicts an association of the places of N to the program variables V; the role of the co-domain element δ for f_{pv} is explained shortly. The variable $f_{pv}(p)$ is denoted as v_p in short; v_p assumes values from a domain D_p. Thus, depending upon the type of the variable v_p, the token value at the place p may be of type Boolean, integer, etc., or a user-defined type of any complexity (such as, a structure or a set). In this dissertation, we consider only integer type variables. The set $T = \{t_1, t_2, \ldots, t_n\}$ is a finite non-empty set of *transitions*; the relation $I \subseteq P \times T$ is a finite non-empty set of input edges which define the flow relation from places to transitions; a place p is said to be an *input place of a transition t* if $(p,t) \in I$. The relation $O \subseteq T \times P$ is a finite non-empty set of output edges which define the flow relation from transitions to places; a place p is said to be an *output place of a transition t* if $(t,p) \in O$. A place $p \in P$ is said to be an *in-port* if and only if $(t,p) \notin O$, for all $t \in T$. Likewise, a place $p \in P$ is said to be an *out-port* if and only if $(p,t) \notin I$, for all $t \in T$. The set *inP* $\subseteq P$ is the non-empty set of in-ports and the set *outP* $\subseteq P$ is the non-empty set of out-ports. The *pre-places* $^\circ t$ *of a transition* $t \in T$ is the non-empty set of *input places* of t. Thus, $^\circ t = \{p \in P \mid (p,t) \in I\}$. Similarly, the *post-places* t° *of a transition* $t \in T$ is the non-empty set of *output places* of t. So, $t^\circ = \{p \in P \mid (t,p) \in O\}$; a place p_1 is said to be a co-place of a place p_2 if $p_1, p_2 \in t^\circ$ for any transition t. For any set T of transitions, $^\circ T\ \left(= \bigcup_{t \in T}^{\circ} t\right)$ represents all the pre-places of the transitions in T. Similarly, for any set T of transitions, $T^\circ\ \left(= \bigcup_{t \in T} t^\circ\right)$ represents all the post-places of the transitions in T. The pre-transitions $^\circ p$ and the post-transitions p° of a place $p \in P$ are given by $^\circ p = \{t \in T \mid (t,p) \in O\}$ and $p^\circ = \{t \in T \mid (p,t) \in I\}$, respectively; for any set P of places, $^\circ P\big(= \bigcup_{p \in P}{}^\circ p\big)$ represents all the pre-transitions of the places in P. Similarly, for any set P of places, $P^\circ\big(= \bigcup_{p \in P} p^\circ\big)$ represents all the post-transitions of the places in P. If the post-places t° of a transition t contains n places, then all these places are associated with identical token type and token value and therefore, the domain of all the post-places of a transition are identical; this property is consistent with the firing rules of Petri net transitions.

A transition t is associated with a *guard condition* $g_t : D_{p_1} \times D_{p_2} \times \ldots \times D_{p_{n_t}} \rightarrow \{\top, \bot\}$ and a *function* $f_t : D_{p_1} \times D_{p_2} \times \ldots \times D_{p_{n_t}} \rightarrow D$, where $^\circ t = \{p_1, p_2, \ldots, p_{n_t}\}$ and $D = D_{p'_1} = D_{p'_2} = \ldots = D_{p'_{m_t}}$ such that $t^\circ = \{p'_1, p'_2, \ldots, p'_{m_t}\}$. The guard condition g_t specifies the condition that must hold over the token values in $^\circ t$ for the transition t to be executed. The function f_t captures the functional transformation that takes place on the token values in $^\circ t$ to produce the *same token value* at all the places in t°. For example, for an assignment statement of a high level language of the form $x := y + (c/d) * 4$, the transition t will have $^\circ t = \{p_1, p_2, p_3\}$, $t^\circ = \{p\}$ and f_t will be maintained as $v_{p_1} + (v_{p_2}/v_{p_3}) * 4$, where v_{p_1} is y, v_{p_2} is c, v_{p_3} is d and v_p is x. A transition t corresponding to an initialization operation of some variable(s) with a constant value has the associated function f_t as the corresponding constant function with a pre-place associated with δ; some places are also introduced as synchronizing pre-places of certain transitions to ensure that all the operations of an iteration is completed before the next iteration starts; these synchronizing places are also associated with δ; all places associated with δ are called dummy places. The model is deterministic denoted symbolically by the following conditions:

(i) The PRES+ model is one-safe, i.e., at any point, a place may hold at most one token [108].

(ii) For any place p, for any two transitions $t_i, t_j \in p^\circ$, if $^\circ t_i \cap {}^\circ t_j \neq \emptyset$, then $g_{t_i}(f_{pv}(^\circ t_i)) \wedge g_{t_j}(f_{pv}(^\circ t_j)) \equiv \bot(false)$, where $f_{pv}(^\circ t)$, for any transition t, indicates the image of the set $^\circ t$ of places under f_{pv}; likewise for $f_{pv}(t^\circ)$.

Also the model is completely specified denoted symbolically by the following condition: $\bigvee_{t \in p^\circ} g_t(f_{pv}(^\circ t)) \equiv \top(true)$.

It is to be noted that the transitions may also have delay and deadline time parameters; models having these features are called timed PRES+ models. We deal with only untimed PRES+ models. Henceforth, by a PRES+ model we only mean a *one-safe, untimed, deterministic, completely specified* PRES+ model.

3.2 Computations in a PRES+ model

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

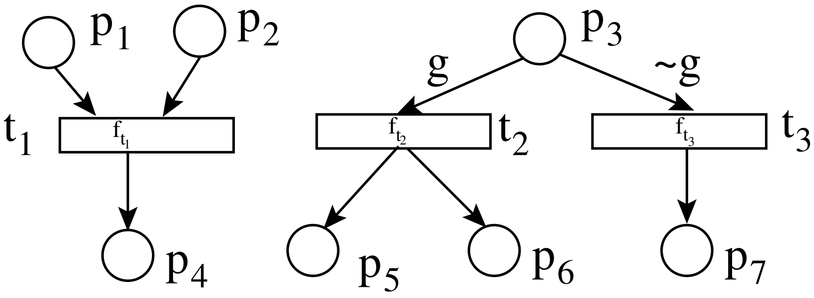

Figure 3.1: Places and transitions in a PRES+ model.

In a PRES+ model, a transition $t \in T$ is said to be *bound* for a given marking $M : \langle P_M, val_M \rangle$ if and only if all its input places are marked, i.e., $^\circ t \subseteq P_M$. A bound transition $t \in T$ for a given marking M is said to be *enabled* if and only if $g_t(f_{pv}(^\circ t))\{val_M(^\circ t)/f_{pv}(^\circ t)\} \equiv \top$ where $^\circ t = \{p_1, \ldots, p_{n_t}\}$ and $\{val_M(^\circ t)/f_{pv}(^\circ t)\}$ indicates the substitution of the variables in $f_{pv}(^\circ t)$ by $val_M(^\circ t)$. The set of enabled transitions for a marking M is denoted as T_M. For any $t_1, t_2 \in T_M$, $f_{pv}(t_1^\circ) \cap f_{pv}(t_2^\circ) = \emptyset$ and $f_{pv}(^\circ t_1) \cap f_{pv}(t_2^\circ) = \emptyset$ ensuring that there are no shared variables with write after write and read after write dependences, respectively. For untimed PRES+ models, all the enabled transitions are fired simultaneously provided they satisfy the above (freedom of) dependency requirements. After firing of all enabled transitions from a given marking M, the successor marking M^+ of M is obtained. The definition of successor marking is as follow:

Definition 1 (Successor marking of a marking)**.** *A marking $M^+ = \langle P_{M^+}, val_{M^+} \rangle$ is said to be a successor of the marking $M = \langle P_M, val_M \rangle$, if*

(i) the first component P_{M^+} referred to as, successor place marking of P_M, contains all the post-places of the enabled transitions of M and also all the places of M whose post-transitions are not enabled; symbolically, $P_{M^+} = \{p \mid p \in t^\circ$ and $t \in T_M\} \cup \{p \mid p \in P_M$ and $p \notin {}^\circ T_M\}$, and

(ii) $\forall p \in P_{M^+}$, if $p = t^\circ$ for some $t \in T_M$, then $val_{M^+}(p) = f_t(val_M(^\circ t))$ and if $p \notin {}^\circ T_M$, then $val_{M^+}(p) = val_M(p)$.

Example 1. *For illustration of the role of guard conditions of transitions in determining the enabled transitions as a subset of the bound transitions and the successor marking of a marking, let us consider the situation depicted in Figure 3.1. Let M be* *such that $P_M = \{p_1, p_2, p_3\}$; $val_M(p_3)$ denotes the value of the token at p_3. The set of bound transitions are $\{t_1, t_2, t_3\}$. Depending on the guard conditions associated with the bound transitions t_2 and t_3, the set T_M of enabled transitions will be either $\{t_1, t_2\}$ or $\{t_1, t_3\}$. If $g_{t_2}(val_M(p_3))$ is true, then the concurrent transition set $\{t_1, t_2\}$ fires and leads to the marking M_1^+ where, $P_{M_1^+} = \{p_4, p_5, p_6\}$; otherwise, the set $\{t_1, t_3\}$ fires and leads to the marking M_2^+ such that $P_{M_2^+} = \{p_4, p_7\}$.* ■

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

Definition 2 (Back edge)**.** *An edge $\langle t, p\rangle$ from a transition t to a place $p \in t^\circ$ is said to be a back edge with respect to an arbitrary DFS traversal of the PRES+ model, if p is an ancestor of t in that traversal.*

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

Definition 3 (Successor relation between two transitions)**.** *A transition t_i succeeds a transition t_j, denoted as $t_i \succ t_j$, if $\exists t_{k_1}, t_{k_2}, \ldots, t_{k_n} \in T$, and $p_1, p_2, \ldots p_{n+1} \in P, n \geq 1$ such that*

(i) $\langle t_j, p_1\rangle, \langle t_{k_1}, p_2\rangle, \ldots, \langle t_{k_{n-1}}, p_n\rangle, \langle t_{k_n}, p_{n+1}\rangle \in O \subseteq T \times P$,

(ii) $\langle p_1, t_{k_1}\rangle, \langle p_2, t_{k_2}\rangle, \ldots, \langle p_n, t_{k_n}\rangle, \langle p_{n+1}, t_i\rangle \in I \subseteq P \times T$ *and*

(iii) *none of* $\langle t_j, p_1\rangle$ *or* $\langle t_{k_m}, p_{m+1}\rangle, 1 \leq m \leq n$*, is a back edge.*

The expression $t_i \not\succ t_j$ is used as a shorthand for $\neg(t_i \succ t_j)$.

Definition 4 (Set of parallelizable transitions)**.** *Two transitions t_i and t_j are said to be parallelisable, denoted as $t_i \asymp t_j$, if (i) $t_i \not\succ t_j$, $t_j \not\succ t_i$ and (ii) $\forall t_k, t_l (t_k \neq t_l \wedge t_i \succeq t_k \wedge t_j \succeq t_l \rightarrow {}^\circ t_k \cap {}^\circ t_l = \emptyset)$, where $t_i \succeq t_k$ holds if and only if t_i succeeds t_k or t_i is the same as t_k. A set $T = \{t_1, t_2, \ldots, t_k\}$ of transitions is said to be parallelisable if $\forall t_i, t_j \in T, t_i \neq t_j \rightarrow t_i \asymp t_j$ holds.*

To ensure that shared variables are read-only for all parallelisable transitions, we need the following two properties: $(i) f_{pv}(t_i^\circ) \cap f_{pv}(t_j^\circ) = \emptyset$ and $f_{pv}({}^\circ t_i) \cap f_{pv}(t_j^\circ) = \emptyset$, (ii) $f_{pv}(t_k^\circ) \cap f_{pv}(t_l^\circ) = \emptyset$ and $f_{pv}({}^\circ t_k) \cap f_{pv}(t_l^\circ) = \emptyset$.

Definition 5 (Parallelizable sets of parallelizable transitions)**.** *Let $T_1, T_2, \ldots, T_k$ be k sets of parallelisable transitions. They are said to be parallelisable if $\bigcup_{i=1}^{k} T_i$ is a set of parallelisable transitions.*

Definition 6 (Set of maximally parallelizable transitions)**.** *A set T is said to be maximally parallelisable if there is no set T' of parallelisable transitions which contains T.*

Definition 7 (Succeed Relation over Parallelizable Transitions)**.** *Given two sets of parallelisable transitions T_1 and T_2, T_1 is said to succeed T_2, denoted as $T_1 \succ T_2$, if $\exists t_1 \in T_1$ and $\exists t_2 \in T_2$ such that $t_1 \succ t_2$.*

Since parallelisable transitions are all independent of each other, any maximally parallelisable set T of transitions can be partitioned arbitrarily and the members of the partition can be executed in any arbitrary order.

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

(i) There exists a sequence of markings of places $\langle P_{M_0}, P_{M_1}, \ldots, P_{M_{l-1}} \rangle$ such that

(a) $P_{M_0} \subseteq inP$,

(b) $\forall i, 1 \leq i < l$, P_{M_i} is a successor place marking of $P_{M_{i-1}}$, ${}^\circ T_i \subseteq P_{M_{i-1}}$ and $T_i^\circ \subseteq P_{M_i}$.

(ii) $p \in T_l^\circ$.

We can represent the computation alternatively as a sequence of place markings $\langle {}^\circ T_1, T_1^\circ \cup {}^\circ T_2, T_2^\circ \cup {}^\circ T_3, \ldots, \{p\} \subseteq T_l^\circ \rangle$; thus in this alternative representation, a computation is a sequence of markings of places which starts with the pre-places of the first set of transitions, ends with the unit set $\{p\} \subseteq T_l^\circ$, the post-places of the last (unit set of) transition and the remaining members in the sequence are the union of the post-places of a set of transitions with the pre-places of its next set of transitions.

If there are k out-ports, then for each initial marking M_0, there are at most k computations, one for each out-port. (We drop the subscripts of μ when they are clear from

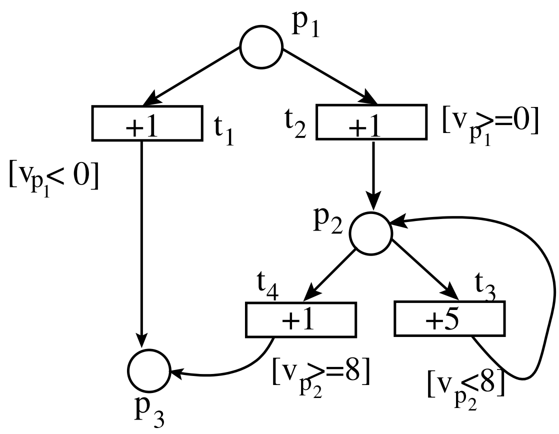

Figure 3.2: Computation in a PRES+ model.

the context. Thus, more specifically, when there is no other PRES+ model we use the symbol μ_p.)

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

We characterize the set of all traces of a given computation μ_p starting with the transition set T_1 using the characteristic predicate $R_{\mu_p}(f_{pv}(^\circ T_1))$ of the set of all valuations of $^\circ T_1$ for which μ_p results. Similarly, the data values of the variable v_p associated with p produced by all the computation traces represented by μ_p is characterized by a symbolic arithmetic expression $r_{\mu_p}(f_{pv}(^\circ T_1))$. More specifically, therefore, we have the following two entities characterizing a given computation μ_p:

1. The condition of execution is $R_{\mu_p}(f_{pv}(^\circ T_1))$.
2. The data transformation is $r_{\mu_p}(f_{pv}(^\circ T_1))$.

The entities R_{μ_p} and r_{μ_p} for μ_p can be obtained using the conventional backward substitution method [95], or alternatively by forward substitution method [79] (used extensively in program verification literature) along μ_p. Now, we illustrate computations in a PRES+ model through the following example.

Example 2. *Let us now examine how various sequences of sets of transitions of the model, given in Figure 3.2, satisfying Definition 8 represent different computations.*

Consider the sequence $\mu_{p_3}^{(1)} = \langle\{t_1\}\rangle$ *and the sequence of place markings* $\langle P_{M_0} = \{p_1\}\rangle$*; the latter satisfies clause 1(a) of Definition 8 because* $\{p_1\} = inP$*; clause 1(b) is satisfied vacuously; clause 2 is satisfied because* $p_3 \in \{t_1\}^\circ$*. The condition of execution* $R_{\mu_{p_3}^{(1)}}(f_{pv}(\{p_1\}))$ *is* $v_{p_1} < 0$ *and the data transformation* $r_{\mu_{p_3}^{(1)}}(f_{pv}(\{p_1\})) = v_{p_1} + 1$*. The computation traces for all negative token values at* p_1 *such as* $-7, -9$*, etc., belong to* $\mu_{p_3(1)}$ *and obviously,* $R_{\mu_{p_3}^{(1)}}(-7) \equiv R_{\mu_{p_3}^{(1)}}(-9) \equiv \top$*;* $r_{\mu_{p_3}^{(1)}}(-7) = -6$ *and* $r_{\mu_{p_3}^{(1)}}(-9) = -8$*. Consider next the sequence* $\mu_{p_3}^{(2)} = \langle\{t_2\},\{t_3\},\{t_3\},\{t_4\}\rangle$*; for a choice of sequence of place markings* $\langle P_{M_0} = \{p_1\}, P_{M_1} = \{p_2\}, P_{M_2} = \{p_2\}, P_{M_3} = \{p_2\}\rangle$*, clause 1(a) is satisfied because* $P_{M_0} = \{p_1\} = inP$*. Clause 1(b) is satisfied as follows; for* $i = 1$*, successor place marking* $P_{M_0^+} = \{p_1\}^+$ *has two choices, one is* $\{p_3\}$ *and the other is* $\{p_2\}$ *depending on the token value at* p_1*; thus* $P_{M_1} = \{p_2\}$ *in the chosen sequence is indeed a successor marking of* P_{M_0}*; so,* ${}^\circ T_1 = {}^\circ\{t_2\} = \{p_1\} = P_{M_0}$ *and* $T_1^\circ = \{t_2\}^\circ = \{p_2\} = P_{M_1}$*. Similarly, for* $i = 2, 3$*, it can be shown that clause 1(b) is satisfied for the chosen sequence; clause 2 is satisfied because* $p_3 \in \{t_4\}^\circ$*. The condition* $R_{\mu_{p_3}^{(2)}}$ *is computed as follows. The condition associated with the transition* t_4 *is* $v_{p_2} \geq 8$*. Starting with the predicate* $\top$ *at* $\{p_3\}$*, using the weakest pre-condition calculation, we get* $R_{\mu_{p_3}^{(2)}}(f_{pv}(\{p_1\}))$ *as* $v_{p_1} + 1 + 5 + 5 \geq 8 \wedge v_{p_1} + 1 + 5 < 8 \wedge v_{p_1} + 1 < 8 \wedge v_{p_1} \geq 0$ *which simplifies to* $v_{p_1} \geq 0 \wedge v_{p_1} \leq 1$ *and* $r_{\mu_{p_3}^{(2)}}(f_{pv}(\{p_1\})) = v_{p_1} + 1 + 5 + 5 + 1 = v_{p_1} + 12$*. It is to be noted that* $\mu_{p_3}^{(2)}$ *is executed for token values* 0 *and* 1 *at* p_1 *(for which* $R_{\mu_{p_2}}$ *is* $\top$*). Similarly, for the sequence* $\mu_{p_3}^{(3)} = \langle\{t_2\},\{t_3\},\{t_4\}\rangle$*, Definition 8 will be satisfied; hence* $\mu_{p_3}^{(3)}$ *will represent a non-empty set of valid computation traces. Specifically,* $R_{\mu_{p_3}^{(3)}}(f_{pv}(\{p_1\}))$ *is* $v_{p_1} \geq 2 \wedge v_{p_1} \leq 6$ *and* $r_{\mu_{p_3}^{(3)}}(f_{pv}(\{p_1\})) = v_{p_1} + 1 + 5 + 1 = v_{p_1} + 7$*. Thus, each of the sequences* $\mu_{p_3}^{(1)}, \mu_{p_3}^{(2)}$ *or* $\mu_{p_3}^{(3)}$ *represents more than one computation trace. Now consider the sequence* $\mu_{p_3}^{(4)} = \langle\{t_1\},\{t_4\}\rangle$*. We argue that we cannot construct any sequence of place markings satisfying the clauses of Definition 8. In order to satisfy clause 1(a), the first member in the place marking sequence must be* $P_{M_0} \subseteq inP = \{p_1\}$*. Hence,* $P_{M_0} = \{p_1\}$*. For satisfying clause 1(b), for* $i = 1$*, the following properties of the constructed sequence must hold:*

(i) P_{M_1} *must be the successor place marking of* P_{M_0}*,*

(ii) ${}^\circ T_1 \subseteq P_{M_0}$*,*

(iii) $T_1^\circ \subseteq P_{M_1}$*.*

Property (ii) is satisfied because ${}^\circ T_1 = {}^\circ\{t_1\} = \{p_1\} = P_{M_0}$*. For property (iii), there* *are two choices for $P_{M_0^+}$; accordingly, we have the following two cases:*

- *Case 1: $P_{M_0^+} = \{p_2\}$— This does not satisfy property (iii) for $i = 1$ because $T_1^\circ = \{t_1\}^\circ = \{p_3\} \nsubseteq P_{M_0^+} = \{p_2\}$.*
- *Case 2: $P_{M_0^+} = \{p_3\}$— For $i = 1$, properties (i), (ii) and (iii) are satisfied; so the sequence can be extended to $\langle\{p_1\},\{p_3\}\rangle$. Now for $i = 2$, the property $(i)(b)$ becomes ${}^\circ T_2 = {}^\circ\{t_4\} = \{p_2\} \nsubseteq P_{M_1} = \{p_3\}$. Hence property $(i)(b)$ is violated.*

So, no sequence of place markings can be computed for $\mu^{(4)}_{p_3}$; hence the latter is not a computation. ■

Now we describe the Petri net fragments corresponding to some important constructs of both sequential and parallel programs schema.

For each of these program constructs, a corresponding typical PRES+ subnet is given in Figure 3.3. The subnet corresponding to a simple assignment statement of the program is given in Figure 3.3(a) where the right hand side (rhs) function is associated with the transition. The pre-place(s) of the transition represents the value(s) of the used variable(s) and the post-place(s) represent the copies of the left hand side (lhs) variable value defined through the assignment statement.

A sequence of assignment statements makes a normal basic block. In normal basic block, the data dependency analysis is carried out over the sequence of assignment statements and parallel threads are accordingly installed; a typical subnet structure corresponding to such a basic block is given in Figure 3.3(b).

For the if-else construct, there is a bifurcation from a set of places corresponding to the variable values over which the condition g, say, of the if-else block is defined; the bifurcation leads to the pre-places of two different sets of transitions representing the start of the if-then block and that of the else block; all the (parallelisable) transitions in the first set are associated with the condition g and those in the second set are associated with $\neg g$. A typical subnet corresponding to the if-else construct is given in Figure 3.3(c). It may be noted that if the value of some variable v prior to the if-then-else block is used in both then-block and else-block, as is the case for the variable v in Figure 3.3(c), then bifurcation also takes place from the place holding this prior value of v; specifically, for Figure 3.3(c), therefore, two model arcs $\langle p_2, t_1 \rangle$ and $\langle p_2, t_2 \rangle$ are put.

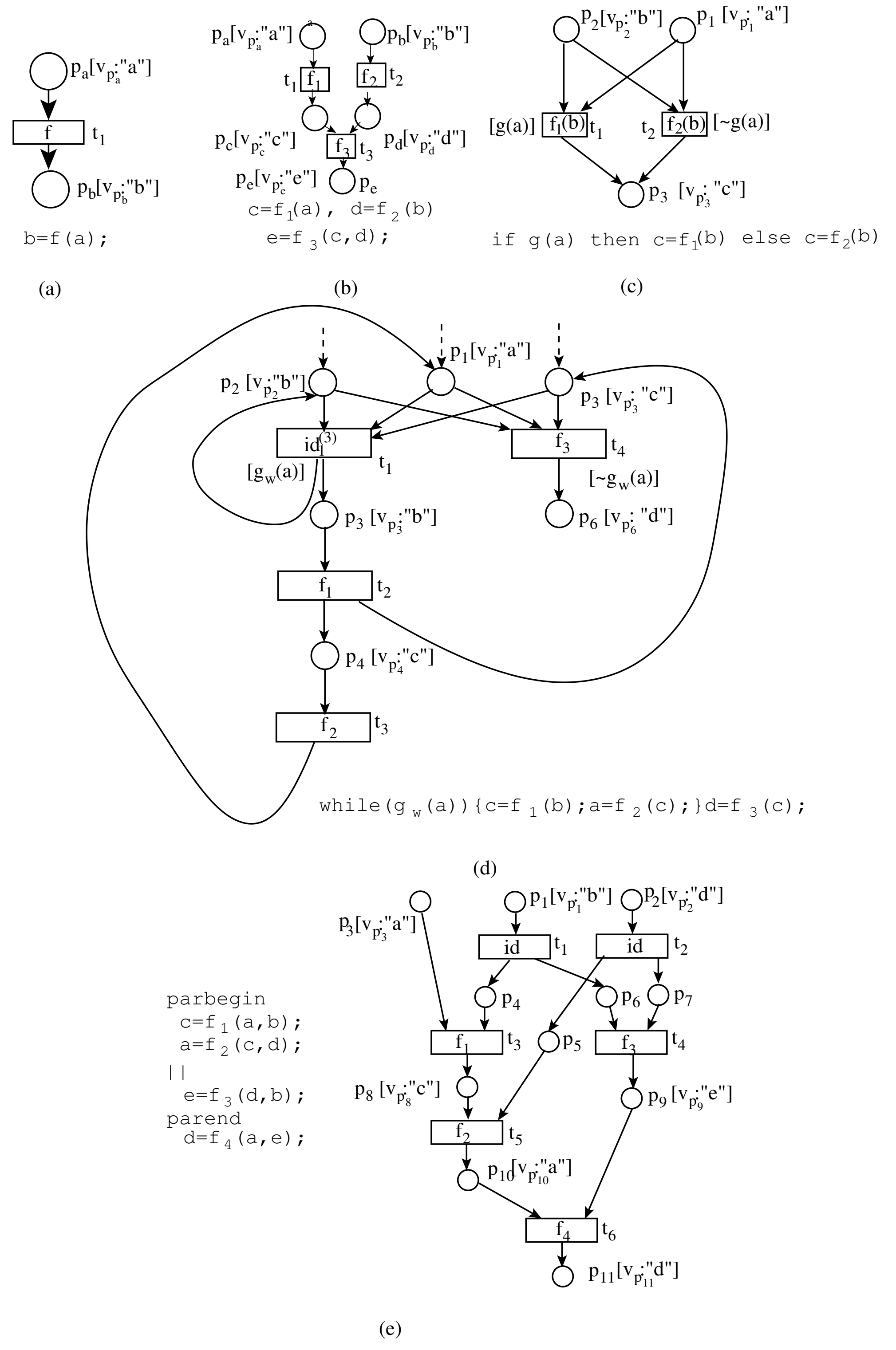

Figure 3.3: PRES+ model snapshots for various high level language constructs.

For a while-loop construct, for the condition g_w, say, again there will be a set of pre-places corresponding to the loop control variable values over which the condition g_w is defined; From these loop control places, there will be bifurcations leading to a set of transitions corresponding to the start of the while body and to another set of transitions corresponding to the start of the block reached through the exit of the while loop; all the transitions in the first set are associated with g_w and those in the second set are associated with $\neg g_w$. Finally, the values of the loop control variables updated in the loop body are returned to the corresponding loop control pre-places. A typical subnet corresponding to a while-loop construct is given in Figure 3.3(d). The body of the while-loop contains the sequence of transitions $\langle t_1, t_2, t_3 \rangle$; the unit set $\{t_1\}$ constitutes the start of the while-body and the unit set $\{t_4\}$ constitutes the start of exit from the while-loop. Hence, these transitions are made to have the place p_1 with $v_{p_1} = a$ as their common pre-place and are associated with the guard condition $g_w(a)$ and $\neg g_w(a)$, respectively. The variable a is modified in the loop body through the transitions t_3; hence the place p_1 is made its post-place. The variable b is not modified in the loop body but reused over the iterations; hence t_1 has p_2 with $v_{p_2} = b$ as one of its post-places; the other post-place p_3 provides the value of b for the transition t_2; the function f_{t_1} associated with t_1 is the identity function of arity 3. There are two issues associated with such a variable v whose values are reused over iterations; the first one is about synchronization with the start of every new iteration; for Figure 3.3(d), this task is accomplished by the loop control place p_1 itself because unless p_1 acquires new token at the end of an iteration, t_1 does not fire. Without such a synchronization mechanism in place, the one-safe property of the PRES+ model may get violated. The second issue arises if the variable b is not live at the exit point of the loop. In such a case the token put at p_2 by the last iteration of the loop body may remain unutilized. To ensure that such unutilized tokens are flushed out, the exit transition t_4 has been made to have place p_2 as one of its pre-places. The variable c is used as well as redefined in the loop body; it is also live at the exit point. Thus, the place p_3 with $v_{p_3} = c$ is kept as a pre-place of t_1 and t_4 and also as a post-place of t_2. Although c is not live at the entry point of the loop body (because it is defined anew at each iteration through t_2), the initial token has to be flushed out at the start of every iteration to preserve one-safeness of the model; this is achieved by having p_3 as a pre-place of t_1 which actually computes the identity function on b.

The PRES+ model fragment for a typical parbegin-parend block is shown in Figure

3.3(e). Both the parallel blocks within the construct use the values of the variables b and d computed prior to this parallel block; hence, their token values at the place p_1 (with $v_{p_1} = b$) and p_2 (with $v_{p_2} = d$) are copied in the places p_4 and p_6 (with $v_{p_4} = v_{p_6} = b$) and the places p_5 and p_7 (with $v_{p_5} = v_{p_7} = d$) through the respective transitions t_1 and t_2 with f_{t_1} and f_{t_2} as the identity mapping. They also serve the purpose of parallel bifurcation for this example. The variable a is live at the entry point of only one of the basic block; hence its value need not be copied. The merging of the two parallel blocks is accomplished by the transition t_6 realizing the assignment statement $d = f_4(a,e)$. We now illustrate a PRES+ model for a simple program and its computation through the following example.

```
int i=0,k,m,n;
while (i<=10){
       m=m+10;
       n=n+10;
       i++;
}
k=m+n;
```

Figure 3.4: A simple program

Example 3. *Figure 3.4 represents a simple program and Figure 3.5 represents the corresponding PRES+ model. In Figure 3.5, the place p_2 holds the value of the variable i initialized to 0 through the transition t_1 associated with the constant function 0; since every transition should have some pre-place, an in-port p_1 is kept as ${}^\circ t_1$. The other two in-ports p_3 and p_4 respectively hold the input values of the variables m and n. It is to be noted that such declarations (or statements), i.e., "Input m,n" create only places without creating transitions; the while-loop entry point is captured by the place p_2. The transition t_6 captures the right-hand side expression of the assignment statement "$m = m+10$". Similarly, the transition t_7 captures the assignment statement "$n = n+10$". The output place of t_6 (t_7) therefore should hold the modified value of the variable m (n); although the in-port p_3 (p_4) is already designated to hold the input value of the variable m (n), it cannot be t_6° because in-ports have no incoming transitions. Hence, a different place p_5 (p_8) is used as the place holding the values of the variable m (n) updated once corresponding to the each iteration of the while-loop. Hence, this place serves as both ${}^\circ t_6({}^\circ t_7)$ and $t_6^\circ(t_7^\circ)$; it is made to hold the initial input value of m (n) through a separate transition t_4 (t_5) with ${}^\circ t_4 = \{p_3\}$ $({}^\circ t_5 = \{p_4\})$ and $t_4^\circ = \{p_5\}(t_5^\circ = \{p_8\})$. Note that since the statements "$m = m+10$" and "$n = n+10$"*

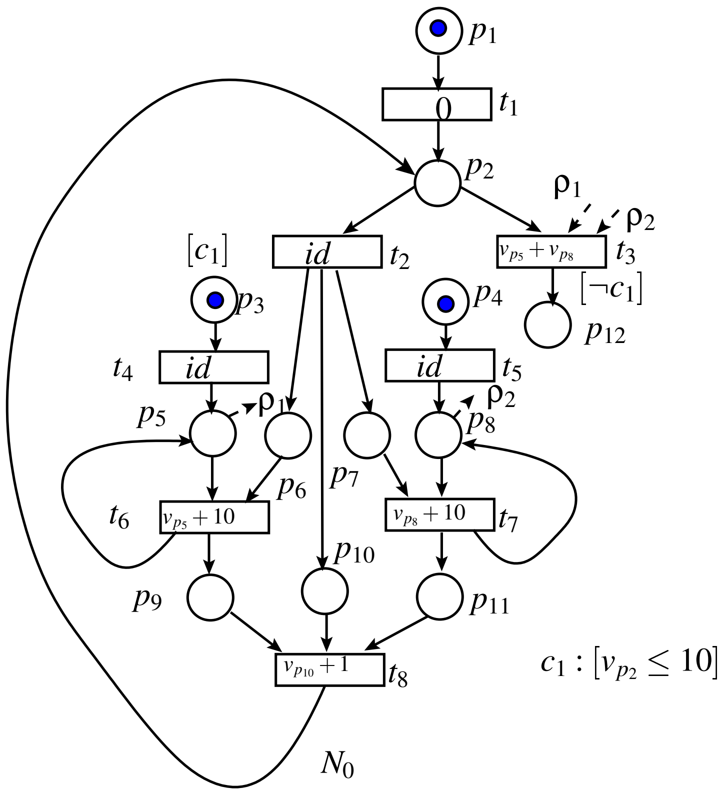

Figure 3.5: A PRES model.

have no data dependency between themselves, the associated transitions t_6 and t_7 are kept as parallelisable transitions. However, their firing have to be after entry to the loop takes place (i.e., after p_2 acquires token either from some segment external to the loop or after execution of each iteration of the loop. Hence, two synchronizing places p_6 and p_7 are required as pre-places of the transitions t_6 and t_7 respectively; these synchronizing places should acquire token through firing of a transition t_2 having p_2 as its pre-place. At this stage, since the transition t_2 only serves the purpose of synchronization, its associated function f_{t_2} is of no concern; however, subsequently we can see why f_{t_2} is the identity function. The transition t_2 therefore can be called the entry transition to the loop initiating the execution of all the parallel threads of the loop body. Since it is the entry transition, it is associated with the loop condition $i \leq 10 = c_1$, say; obviously, there has to be a loop exit transition associated with the condition $\neg c_1$; the transition t_3 serves this purpose. The transition t_8 captures the update operation "$i++$" of the loop control variable i. Although this update operation has no data dependency with the other two statements in the loop body, it is made to have control dependence with the transitions t_6 and t_7 for synchronization;

hence, synchronizing places p_9, p_{11} are used as the pre-places of t_8 and post-places of t_6 and t_7; the place p_{10} holds the value of the loop control variable i which is the only pre-place of t_8 that gets updated through its execution. Now, we can see that t_2 can provide the copy the loop control variable i held in the loop entry place p_2; therefore, the transition t_2 is made to have the associated function f_{t_2} as the identity function (id). Now, the segment reached after exit from the loop comprising the statement "$k = m+n$" would require a transition having two pre-places holding values of the variables m and n and a post place corresponding to the variable k. The already conceived exit transition t_3 can serve this purpose with its pre-places p_5 (holding the latest value of m) and p_8 (holding the latest value of n) in addition to its already conceived pre-place p_2. This immediately underlines the need of associating the transitions t_6, t_7 in the loop body with the loop entry condition; in general, all the transitions in a loop body are associated with the loop entry condition conservatively. The place p_{12} corresponding to variable k is placed as the post-place of t_3 and it is designated as an out-port because k is an output variable. So the place to variable mapping $f_{pv} : \{\{p_1, p_6, p_7, p_9, p_{11}\} \mapsto \delta, \{p_2, p_{10}\} \mapsto i, \{p_3, p_5\} \mapsto m, \{p_4, p_8\} \mapsto n, p_{12} \mapsto k\}$.

Let us now examine how the computation trace of the program, given in Figure 3.4, for the inputs $m = 7$ and $n = 11$ is represented by a computation $\mu_{p_{12}}$ of the out-port p_{12} of the model of Figure 3.5. In the following, for any marking M encountered along the computation, the second component val_M is listed using the same ordering in which the first component P_M is listed. For the inputs $m = 7$ and $n = 11$, the initial marking $M_0 = \langle\{p_1, p_3, p_4\}, \langle\omega, 7, 11\rangle\rangle$, where ω stands for any integer. Now, the first set of maximally parallelisable transitions in the computation is $T_1 = T_{M_0} = \{t_1, t_4, t_5\}$, where T_{M_i} stands for the set of enabled transitions for the marking M_i; note that ${}^\circ T_1 \subseteq inP = \{p_1, p_3, p_4\}$. After firing of (all the members of) T_1, the successor marking M_0^+ becomes $\langle\{p_2, p_5, p_8\}, \langle 0, 7, 11\rangle\rangle = M_1$; at this stage the bound transitions are $\{t_2, t_3\}$; the condition $c_1 = v_{p_2} \leq 10$ associated with t_2 is satisfied and the one $(\neg c_1)$ associated with t_3 is false; so the next set in the computation becomes $T_2 = T_{M_1} = \{t_2\}$. After firing of T_2, the successor marking $M_1^+ = M_2 = \langle\{p_5, p_6, p_7, p_8, p_{10}\}, \langle 7, 0, 0, 11, 0\rangle\rangle$. So the next set of transitions $T_3 = T_{M_2} = \{t_6, t_7\}$. After firing of T_3, the successor marking $M_2^+ = M_3 = \langle\{p_5, p_8, p_9, p_{10}, p_{11}\}, \langle 17, 21, 17, 0, 21\rangle\rangle$; $T_4 = T_{M_3} = \{t_8\}$. After firing of T_4, the successor marking $M_3^+ = M_4 = \langle\{p_2, p_5, p_8\}, \langle 1, 17, 21\rangle\rangle$ and $T_5 = T_{M_4} = \{t_2\} = T_2$. So the sub-sequence of the sets of transitions $\langle T_3, T_4, T_2\rangle$ captures one iteration of the loop. Hence, it repeats another ten times. The prefix of the computation

at this stage becomes $\langle T_1, T_2, (T_3, T_4, T_2)^{11} \rangle$ *and the resulting marking will be* $M_{44} = \langle \{p_2, p_5, p_8\}, \langle 11, 117, 121 \rangle \rangle$. *At this stage,* $T_{M_{44}} = \{t_3\} = T_6$ *because the condition associated with* t_3 $(= \neg v_{p_2} \leq 10)$ *holds. So the computation in terms of a sequence of transitions is* $\langle T_1, T_2, (T_3, T_4, T_2)^{11}, T_6 \rangle$ *and in terms of places:*
$\langle \langle {}^\circ T_1, T_1^\circ \cup {}^\circ T_2, \left(T_2^\circ \cup {}^\circ T_3, T_3^\circ \cup {}^\circ T_4, T_4^\circ \cup {}^\circ T_2\right)^{11}, T_2^\circ \cup^\circ T_6, T_6^\circ \rangle, \langle \overline{val_{M_0}}({}^\circ T_1) \rangle \rangle =$
$\langle \langle \{p_1, p_3, p_4\}, \{p_2, p_5, p_8\}, (\{p_5, p_6, p_7, p_8, p_{10}\}, \{p_5, p_8, p_9, p_{10}, p_{11}\}, \{p_2\})^{11},$
$\{p_2, p_5, p_8\}, \{p_{12}\} \rangle, \langle \omega, 7, 11 \rangle \rangle$. *The condition of execution:*
$R_{\mu_{p_{12}}}(f_{pv}({}^\circ T_1))\{\langle \omega, 7, 11 \rangle / f_{pv}({}^\circ T_1)\} \equiv \top$
and the data transformation :
$r_{\mu_{p_{12}}}(f_{pv}({}^\circ T_1))\{\langle \omega, 7, 11 \rangle / f_{pv}({}^\circ T_1)\} = 238.$

■

3.3 Computational equivalence between two PRES+ models

Two PRES+ models N_0 and N_1 will not be equivalent unless they are input-output compatible (i/o-compatible, in short), that is, there is a bijection $f_{in} : inP_0 \leftrightarrow inP_1$ between their in-ports and a bijection $f_{out} : outP_0 \leftrightarrow outP_1$ between their out-ports; both the associations f_{in} and f_{out} are consistent with the respective place-to-variable association f_{pv}^0 of N_0 and f_{pv}^1 of N_1. In other words, if $\langle p, p' \rangle \in f_{in}(f_{out})$, then $f_{pv}^0(p) = f_{pv}^1(p')$.

Let $N_0 : \langle P_0, V, f_{pv}^0, T_0, I_0, O_0, inP_0, outP_0 \rangle$ and $N_1 : \langle P_1, V, f_{pv}^1, T_1, I_1, O_1, inP_1, outP_1 \rangle$ be two i/o-compatible PRES+ models with in-port correspondence f_{in} and out-port correspondence f_{out}; we now define the notions of the computational containment and computational equivalence of two PRES+ models.

Definition 9 (Equivalence of two computations of two Models)**.** *Let* $\mu_{0,p}$ *be a computation of an out-port p of* N_0 *of the form* $\langle T_{0,1}, T_{0,2}, \ldots, T_{0,n_p} \rangle$ *and let* $\mu_{1,f_{out}(p)}$ *be a computation of the out-port* $f_{out}(p)$ *of* N_1 *of the form* $\langle T_{1,1}, T_{1,2}, \ldots, T_{1,n_{f_{out}(p)}} \rangle$. *The computations* $\mu_{0,p}$ *and* $\mu_{1,f_{out}(p)}$ *are said to be equivalent (represented as* $\mu_{0,p} \simeq \mu_{1,f_{out}(p)}$*), if*

1. $R_{\mu_{0,p}}(f_{pv}^0({}^\circ T_{0,1})) \equiv R_{\mu_{1,f_{out}(p)}}(f_{pv}^1({}^\circ T_{1,1}))$ *and*

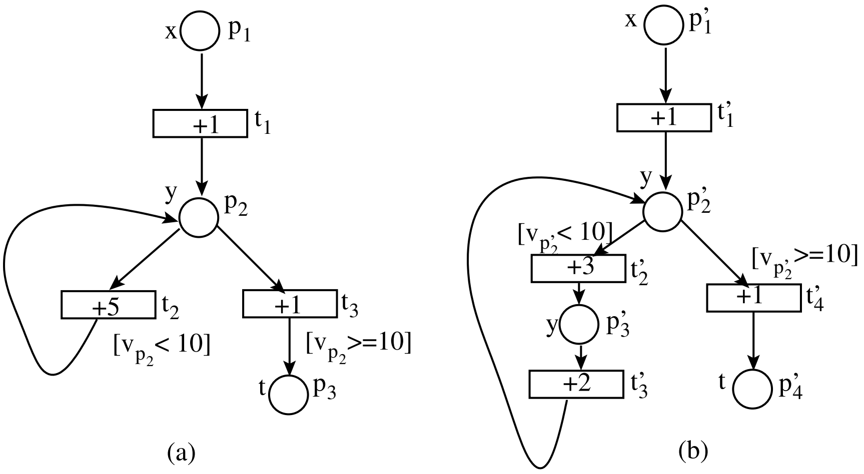

Figure 3.6: Computational equivalence of two PRES+ models.

2. $r_{\mu_{0,p}}(f^0_{pv}(^\circ T_{0,1})) = r_{\mu_{1,f_{out}(p)}}(f^1_{pv}(^\circ T_{1,1}))$

Definition 10 (Computational Containment of Models). *The PRES+ model N_0 is said to be contained in the PRES+ model N_1, represented as $N_0 \sqsubseteq N_1$, if, $\forall p \in outP_0$, for any computation $\mu_{0,p} = \langle T_{0,1}, T_{0,2}, \ldots, T_{0,n_p} \rangle$ of the out-port of N_0, there exists a computation $\mu_{1,f_{out}(p)} = \langle T_{1,1}, T_{1,2}, \ldots, T_{1,n_{f_{out}(p)}} \rangle$ of the out-port of N_1 such that $\mu_{0,p} \simeq \mu_{1,f_{out}(p)}$.*

Definition 11 (Computational Equivalence of Models). *The PRES+ models N_0 and N_1 are said to be computationally equivalent if $N_0 \sqsubseteq N_1$ and $N_1 \sqsubseteq N_0$.*

Now we illustrate how the equivalence of two given computations of two PRES+ models is resolved using the above definition. In the process, we underline the fact that such equivalence of two given computations of the PRES+ models are to be resolved symbolically; such symbolic analyses need the respective place to variable associations f^0_{pv} and f^1_{pv} to resolve the equivalence of conditions of executions and equalities of the data transformations.

Example 4. *Let Figures 3.6(a) and (b) depict two PRES+ models N_0 and N_1, respectively. The variable set $V = \{x, y, t\}$. $f^0_{pv} : \{p_1\} \mapsto x, \{p_2\} \mapsto y, \{p_3\} \mapsto t$ and $f^1_{pv} : \{p'_1\} \mapsto x, \{p'_2\} \mapsto y, \{p'_3\} \mapsto y, \{p'_4\} \mapsto t$. The set of in-ports of the model N_0 is $inP_0 = \{p_1\}$ and that of out-port $outP_0 = \{p_3\}$. Similarly, for N_1, the $inP_1 = \{p'_1\}$ and $outP_1 = \{p'_4\}$. Let $f_{in} : inP_0 \leftrightarrow inP_1$ be $p_1 \mapsto p'_1$; let $f_{out} : outP_0 \leftrightarrow outP_1$ be $p_3 \mapsto p'_4$. Let a computation $\mu^{(1)}_{p_3}$ of N_0 be $\langle \{t_1\}, \{t_2\}, \{t_2\}, \{t_3\} \rangle$; the condition of execution is*

$R_{\mu^{(1)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) \equiv R_{\mu^{(1)}_{p_3}}(f^0_{pv}(\{p_1\})) \equiv x+1+5+5 \geq 10 \wedge x+1+5 < 10 \wedge x+1 < 10 \equiv x+11 \geq 10 \wedge x+6 < 10 \wedge x+1 < 10 \equiv x < 4 \wedge x \geq -1$ *and the data transformation is* $r_{\mu^{(1)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) = r_{\mu^{(1)}_{p_3}}(f^0_{pv}(\{p_1\})) = x+1+5+5+1 = x+12$. *Let a computation* $\mu^{(1)}_{p'_4}$ *of* N_1 *be* $\langle\{t'_1\},\{t'_2\},\{t'_3\},\{t'_2\},\{t'_3\},\{t'_4\}\rangle$*; the condition of execution is* $R_{\mu^{(1)}_{p'_4}}(f^1_{pv}(^\circ\{t'_1\})) \equiv R_{\mu^{(1)}_{p'_4}}(f^1_{pv}(\{p'_1\})) \equiv x+1+2+3+2+3 \geq 10 \wedge x+1+2+3 < 10 \wedge x+1 < 10 \equiv x+11 \geq 10 \wedge x+6 < 10 \wedge x+1 < 10 \equiv x < 4 \wedge x \geq -1$ *and the data transformation is* $r_{\mu^{(1)}_{p'_4}}(f^1_{pv}(^\circ\{t'_1\})) = r_{\mu^{(1)}_{p'_4}}(f^1_{pv}(\{p'_1\})) = x+1+2+3+2+3+1 = x+12$. *Therefore,* $R_{\mu^{(1)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) \equiv R_{\mu^{(1)}_{f_{out}(p_3)}}(f^1_{pv}(^\circ\{t'_1\}))$ *and* $r_{\mu^{(1)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) = r_{\mu^{(1)}_{f_{out}(p_3)}}(f^1_{pv}(^\circ\{t'_1\}))$. *Hence,* $\mu^{(1)}_{p_3} \simeq \mu^{(1)}_{p'_4}$. *Suppose another computation* $\mu^{(2)}_{p_3}$ *of* N_0 *be* $\langle\{t_1\},\{t_2\},\{t_3\}\rangle$*; the condition of execution is* $R_{\mu^{(2)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) \equiv R_{\mu^{(2)}_{p_3}}(f^0_{pv}(\{p_1\})) \equiv x+1+5 \geq 10 \wedge x+1 < 10 \equiv x+6 \geq 10 \wedge x+1 < 10 \equiv x \geq 4 \wedge x < 9$ *and the data transformation is* $r_{\mu^{(2)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) = r_{\mu^{(2)}_{p_3}}(f^0_{pv}(\{p_1\})) = x+1+5+1 = x+7$. *For* N_1*, let* $\mu^{(2)}_{p'_4}$ *be the computation which is of the form* $\langle\{t'_1\},\{t'_2\},\{t'_3\},\{t'_4\}\rangle$*; the condition of execution is* $R_{\mu^{(2)}_{p'_4}}(f^1_{pv}(^\circ\{t'_1\})) \equiv R_{\mu^{(2)}_{p'_4}}(f^1_{pv}(\{p'_1\})) \equiv x+1+2+3+ \geq 10 \wedge x+1 < 10 \equiv x \geq 4 \wedge x < 9$ *and the data transformation is* $r_{\mu^{(2)}_{p'_4}}(f^1_{pv}(^\circ\{t'_1\})) = r_{\mu^{(2)}_{p'_4}}(f^1_{pv}(\{p'_1\})) = x+1+2+3+1 = x+7$. *Therefore,* $R_{\mu^{(2)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) \equiv R_{\mu^{(2)}_{f_{out}(p_3)}}(f^1_{pv}(^\circ\{t'_1\}))$ *and* $r_{\mu^{(2)}_{p_3}}(f^0_{pv}(^\circ\{t_1\})) = r_{\mu^{(2)}_{f_{out}(p_3)}}(f^1_{pv}(^\circ\{t'_1\}))$. *Hence,* $\mu^{(2)}_{p_3} \simeq \mu^{(2)}_{p'_4}$. *However, when we try to resolve the question whether* $N_0 \sqsubseteq N_1$*, we have to consider all computations of the out-port* p_3*. Let* $\mathcal{M}_{0,p_3}$ *be the set of all computations of* p_3*. The set* $\mathcal{M}_{0,p_3}$ *is infinite because of the presence of the loop* $p_2 \rightarrow p_2$*. Unlike its individual members, the entire set* $\mathcal{M}_{0,p_3}$ *cannot be characterized by any symbolic expressions. Hence, the containment (and equivalence) of PRES+ models cannot be established in the same manner in which equivalence of individual computation can be resolved. In the subsequent chapters, we shall introduce the notion of finite paths through which infinite sets such as* $\mathcal{M}_{0,p_3}$ *can be captured.*

■

3.4 Restrictions of the model and their implications

The PRES+ model used in this work is essentially a restricted subset of the PRES+ model described in [38]. In the following, we identify the differences. In the article [38], a PRES+ model is a five tuple; in our context, a PRES+ model is an eight tuple. The four entities P, T, I and O are common for both the representations. The initial marking M_0 is a member of the PRES+ model tuple reported in [38] which is absent in our model description because an initial marking (involving values for the tokens) pertains to a particular invocation of the model. In contrast, the additional members namely, *inP*, *outP*, f_{pv} and V, have been introduced by us in the model tuple; literature [38] introduces *inP*, *outP* in the context of model equivalence. However, it is felt that *inP* and *outP* constitute static features of the model and accordingly can be included in the model tuple itself. For the remaining two new members f_{pv} and V, it is important to note that the main objective of the present work is translation validation of compiler optimization techniques through behavioural equivalence checking of source programs with their transformed versions. So an association of places with program variables exist which gets revealed in a natural way while constructing the model of the program, both manually or automatically. This natural association has been captured by the function f_{pv} and fruitfully utilized in the subsequent chapters in establishing the computational equivalence between two i/o-compatible PRE+ models. The type $\tau(p)$ of tokens occupying the place p appears in our model as the set D_p of values assumed by such tokens; the time components of the $\tau(p)$ values are ignored because our PRES+ models (being targeted at compiler optimization transformation validation) are untimed.

For [38], the marking M is a function with the set P of places as its domain and the set of all token values including the empty set as its range. For us, a marking M is an ordered pair $\langle P_M, val_M \rangle$, where $P_M \subseteq P$ is a subset of places containing tokens and $val_M : P_M \to \sqcup_{p \in P_M} D_p$. Thus, if $P = \{p_1, p_2, p_3, p_4\}$ and $M = \langle \{p_1, p_3\}, \{p_1 \mapsto 14, p_3 \mapsto -9\} \rangle$ in our representation, then according to [38], $M(p_1) = \{\langle 14, r_1 \rangle\}$ and $M(p_3) = \{\langle -9, r_2 \rangle\}$ where $r_1, r_2 \in \mathbb{R}^+$ are the time stamps of the tokens (which are totally absent in our *untimed* PRES+ model) and $M(p_2) = M(p_4) = \emptyset$. Similarly, in [38], if $M(p_1) = \{\langle 15, 2.3 \rangle\}, M(p_2) = \{\langle -11, 1.7 \rangle\}, M(p_4) = \{\langle 5, 9.7 \rangle\}$ and $M(p_3) = \emptyset$, then in our representation $M = \langle \{p_1, p_2, p_4\}, \{p_1 \mapsto 15, p_2 \mapsto -11, p_3 \mapsto 5\} \rangle$ (time stamps are ignored). However, the marking M', represented according to [38] as

$M'(p_1)$ =$\{\langle 10, 1.2\rangle, \langle 5, 2.5\rangle\}$ and $M'_{p_2} = M'_{p_3} = M'_{p_4} = \emptyset$, cannot be represented in our model which is strictly one-safe permitting no more than one token in a place at any point (when the marking M' involves two tokens in p_1). Hence our representation of markings is synonymous to that in literature [38] for the one-safe models shorn of the time values.

The enabled transitions in our work, however, are different from the those in literature [38]. The model [38] permits non-determinism. Consider, for example, a place marking $P_M = \{p_1, p_2\}$ with $p_1^\circ = \{t_1, t_2\}$ and $p_2^\circ = \{t_3\}$. For both transitions t_1 and t_2, let g_{t_1}, g_{t_2} be identically true (independent of token values at p_1 and p_2 according to [38]). In this case, according to the model of [38], the enabled transitions are the same as the bound transitions, i.e., $\{t_1, t_2, t_3\}$. Our model does not permit non-determinism. Hence, $g_{t_1} \wedge g_{t_2}$ should be unsatisfiable. Accordingly, we could have two mutually exclusive sets of enabled transitions corresponding to this place marking namely, $T_{M_1} = \{t_1, t_3\}$ and $T_{M_2} = \{t_2, t_3\}$.

For a given marking M, literature [38] permits firing of any one of the transitions enabled under M; newer marking results for each firing step. For the present work, the enabled transitions are fired simultaneously in parallel because they cannot have any data dependency on each other. Thus, for the above example scenario, the immediately reachable place markings in [38] can be $P_{M_1^+} = \{t_1^\circ, p_2\}$ (disabling t_2 here), $P_{M_2^+} = \{t_2^\circ, p_2\}$ (disabling t_1 here) or $P_{M_3^+} = \{t_3^\circ, p_1\}$. This is how only one of the non-deterministic choices regarding firing of t_1 and t_2 is exercised at the expense of other. Since for both the immediately reachable place markings $P_{M_1^+}, P_{M_2^+}$, t_3 remain enabled, the next step of firing can yield $P_{M_1^{++}} = \{t_1^\circ, t_3^\circ\}$ and $P_{M_2^{++}} = \{t_2^\circ, t_3^\circ\}$ as two mutually exclusive immediately reachable place markings. For our model, one of t_1 or t_2 being chosen deterministically based on their guard conditions, we reach in one step $P_{M_1^{++}}$ or $P_{M_2^{++}}$ (mutually exclusive of one another) by simultaneous firing of $\{t_1, t_3\}$ or $\{t_2, t_3\}$, respectively.

The notion of successor marking used in the present work corresponds to the "immediately reachable" marking (Definition 3.1) of literature [38]. The difference is due to simultaneous firing of all the enabled transitions in our model versus their firing based on time parameters and/or exercising of some non-deterministic choice in [38]. It is obvious that for our models, since there is no data dependency among the enabled transitions, any interleaving of their firing creates the same effect in terms of the variables values as that produced by their simultaneous firing. We have provided the formal definition of successor marking (Definition 1) in a form suitable for devising the definition of computations in a PRES+ model which, in turn, permits us to address the theoretical issues of equivalence checking mechanisms described in this work.

The original PRES+ model reported in [38] is k-safe which necessitates non-determinism to be accommodated. We have considered deterministic PRES+ models because our objective is to use PRES+ models for representing programs written in some conventional high level C like languages; such programs having no writable shared variables among the parallel threads are inherently deterministic.

The notion of "function equivalence" [38] is what is relevant for our work. The definition of "functional equivalence" of literature [38] considers initial markings to have identical token values at the input places but may also have token values at various non-input places; all reachable markings from such initial markings, which have no tokens in the input places and have identical tokens in their non-input places, should have identical tokens in the output places. In the domain of application of the present work, we have not identified any situation where initial marking need to have tokens in the non-input places. Since input places do not have any incoming arc (as also in literature [38]), no reachable marking for a given initial marking can put token again in the input places. Other than the difference in permitting the initial markings to have tokens in some non-input places, there is no other fundamental difference between the definition of functional equivalence presented in [38] and that of computational equivalence presented here. The notion of functional equivalence, however, does not concern itself with any symbolic analysis mechanism which is inevitable for establishing functional equivalence in an absolute sense (i.e., independent of the specific computation corresponding to a given initial marking). The original literature on PRES+ models [38] concentrates on property verification and not on functional equivalence checking. In contrast, for our work, we need definitions which can be used to devise equivalence checking methods and validate them. So, for our work, the definitions of computation and computational equivalence had to be devised anew in a form that permits us to treat the theoretical issues regarding the equivalence checking mechanisms described in the thesis.

3.5 Conclusion

In this chapter we have introduced the PRES+ models formally and given a formal definition of the computational semantics of a PRES+ model, the computational containment and computational equivalence of two PRES+ models. These fundamental notions are used subsequently throughout the thesis.

Chapter 4

Dynamic Cut-point induced path construction method

When a PRES+ model contains loops, the number of traversals through such a loop depends on the in-port data. Since the in-ports can assume infinite number of combinations of input values, the number of computations of any out-port can be infinite. To establish computational equivalence of two models, all such computations must be accounted for. For this reason, the notion of *finite* computation paths, henceforth referred to simply as paths, is used so that any computation of an out-port can be captured in terms of these paths. To do so, we need to cut the loops designating some of the places as cut-points so that each loop contains at least one cut-point. A path originates from a set of places which contains cut-points and ends with a single cut-point. In this chapter, we discuss a mechanism of inserting cut-points so that the resulting paths capture any computation of the model; we then describe a path construction procedure using such cut-points.

4.1 Computation paths of a PRES+ model

For establishing equivalence between two PRES+ models, for any of their out-ports, p say, the set $\mathcal{M}_p$ of all possible computations of p should be covered. In the previous chapter, it has already been pointed out that while any individual computation μ_p in $\mathcal{M}_p$ can be characterized by two symbolic expressions R_{μ_p} and r_{μ_p}, the entire set $\mathcal{M}_p$

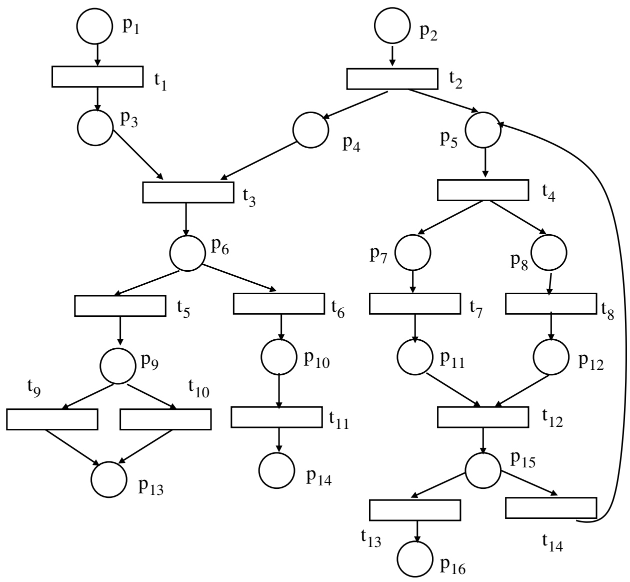

Figure 4.1: Need of Paths of a PRES+ model.

cannot be characterized in the same manner when loops are present. A conventional approach in such scenarios is to use the concept of finite paths such that any computation can be represented in terms of these paths. We illustrate the mechanism of capturing computations in terms of paths using the following example.

Example 5. *Let us consider the PRES+ model given in Figure 4.1. The set of all computations of the out-port p_{16} is given by $\mathcal{M}_{p_{16}} = \{\langle M_0, (m_s)^n, M_5\rangle, n \geq 0\}$, where $P_{M_0} \supseteq \{p_2\}$ and m_s is the sub-sequence $\langle M_1, M_2, M_3, M_4\rangle$ of markings with $P_{M_1} \supseteq \{p_5\}, P_{M_2} \supseteq \{p_7, p_8\}, P_{M_3} \supseteq \{p_{11}, p_{12}\}$, $P_{M_4} \supseteq \{p_{15}\}$, $P_{M_5} \supseteq \{p_{16}\}$. Obviously, we cannot obtain a single symbolic expression for condition of execution and a symbolic data transformation expression for the set $\mathcal{M}_{p_{16}}$ as a whole, in the same way we obtain them for any of its individual members.*

For this reason, we introduce the notion of finite paths so that any member of $\mathcal{M}_{p_{16}}$ can be considered as a finite concatenation of the paths. In particular, let the sequence of places $\langle\{p_2\},\{p_5\}\rangle$ be designated as a path α_1, the sequence $\langle\{p_5\},\{p_8\}\rangle = \langle\{p_5\},\{p_7\}\rangle$ be designated as α_2, the sequence $\langle\{p_7, p_8\},\{p_{11}, p_{12}\},\{p_{15}\},\{p_5\}\rangle$ be designated as α_3 and the sequence $\langle\{p_7, p_8\},\{p_{11}, p_{12}\},\{p_{15}\},\{p_{16}\}\rangle$ be designated as α_4. Then, the set $\mathcal{M}_{p_{16}}$ can be represented as $\{\alpha_1.(\alpha_2.\alpha_3)^n.\alpha_4, n \geq 0\}$. It may now be noted that for establishing the equivalence of all computations of the out-port p_{16} of the above model N_0, say, with those of $f_{out}(p_{16})$ in another i/o-compatible model N_1, we may similarly identify a finite number of paths in N_1 to capture $\mathcal{M}_{f_{out}(p_{16})}$ and *try to establish a path level equivalence among these sets of paths. The set of paths* $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ *have been obtained by cutting the loop from* p_5 *to* p_5 *by introducing a cut-point at* p_5*. (Although for this example, we depict a path as a sequence of sets of places, it becomes more convenient to represent a path primarily as a sequence of sets of transitions.)* ■

The above example demonstrates how the notion of *finite* paths can be used to capture any computation of an out-port. To do so, we need to designate some places as cut-points so that each loop contains at least one cut-point. Such cut-points can be identified utilizing the concept of back edges as defined below.

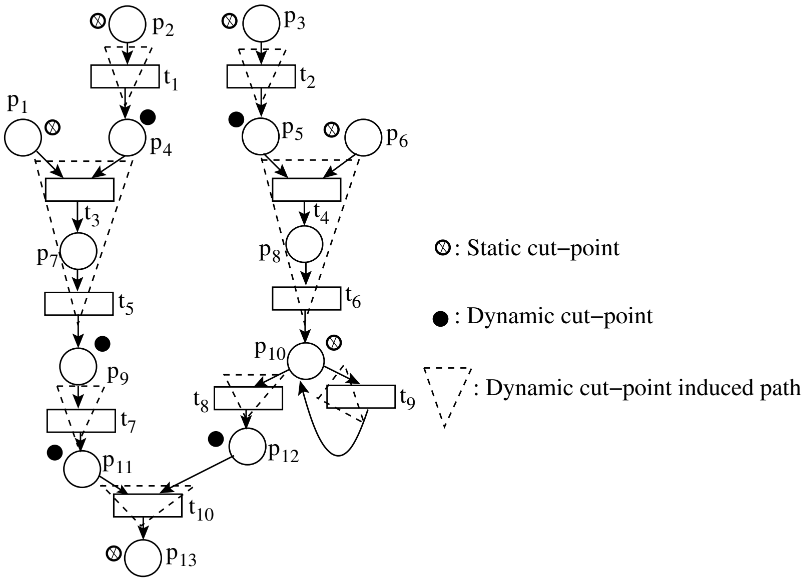

Figure 4.2: Paths of a PRES+ model.

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

It is to be noted that since there may be more than one DFS traversal for a given graph, there may be different sets of back edges corresponding to these traversals; thus, the set of static cut-points may differ from one DFS traversal to another; however, it is unique for any particular one. We need just one such set with respect to a single

DFS traversal for obtaining the cut-points to cut each of the loops in at least one cut-point (not necessarily, minimally).[1]

In Figure 4.2, for example, the in-ports p_1, p_2, p_3 and p_6 are cut-points. A DFS traversal of the graph (model) identifies the edge $\langle t_9, p_{10}\rangle$ as a back edge; hence, p_{10} is a cut-point. Place p_{13} being an out-port is also a cut-point.

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

(i) *All the members of* ${}^\circ T_1$ *are cut-points.*

(ii) *All the members of* T_n° *are cut-points.*

(iii) *There is no cut-point in* T_m°*,* $1 \le m < n$.

(iv) $\forall i, 1 < i \le n, \forall p \in {}^\circ T_i$*, if* p *is not a cut-point, then* $\exists k, 1 \le k \le i-1, p \in T_{i-k}^\circ$*; thus, any pre-place of a transition which is not a cut-point must be a post-place of some preceding transition in the path.*

(v) *There do not exist two transitions* t_i *and* t_l *in* α *such that* ${}^\circ t_i \cap {}^\circ t_l \neq \emptyset$.

(vi) $\forall i, 1 \le i \le n$*,* T_i *is maximally parallelisable within the path, i.e.,* $\forall l \neq i, \forall t \in T_l$ *in the path,* $T_i \cup \{t\}$ *is not parallelisable.*

(vii) *There exists a computation (of some out-port) having a sub-sequence of markings of places* $\langle P_{M_i}, P_{M_{i+1}}, \ldots, P_{M_{i+n}}\rangle$ *such that*

(a) ${}^\circ T_{1+j} \subseteq P_{M_{i+j}}, 0 \le j < n,$

(b) $\forall j, 1 \le j \le n, P_{M_{i+j}}$ *is a successor place marking of* $P_{M_{i+j-1}}$ *and*

(c) $T_n^\circ \subseteq P_{M_{i+n}}$.

(viii) $\forall t, 1 \le i < n, |T_i^\circ| = |T_i|$.

The set ${}^\circ T_1$ of places is called the set of pre-places of the path α, denoted as ${}^\circ\alpha$; similarly, the set T_n° is called the set of post-places of the path α, denoted as α°. We can

[1]Why they are designated as *static* cut-points becomes clear shortly.

synonymously denote a path $\alpha = \langle T_1, T_2, \ldots, T_n \rangle$ as the sequence $\langle {}^\circ T_1, {}^\circ T_2, \ldots, {}^\circ T_n, T_n^\circ \rangle$ of the sets of places from the place(s) ${}^\circ T_1$ to the place(s) T_n°.

The clauses in Definition 13 of paths have the following meaning. Clauses (i)–(iii) ensure that no sequence of sets of parallelisable transitions that constitutes a loop segment can be a proper sub-sequence of a path. Thus, for any combination of values at the input places ${}^\circ\alpha$, the computation of path α involves execution of all its transitions exactly once. Clause (iv) ensures that any computation of the path α is completely defined in terms of the token values at ${}^\circ\alpha$; more specifically, each transition uses token values either available at ${}^\circ\alpha$ or computed in some preceding transition within the path. Clause (v) ensures that a path does not involve mutually exclusive transitions. In other words, negation of clause (v) implies existence of transitions t_i, t_l having common pre-place which means that either t_i or t_l (and not both) can execute for any token value at this common pre-place. Clause (vi) ensures that between two distinct sets of transitions of a path, there is always a strict sequencing. Clause (vii) ensures that the sequence depicted in the path must appear as a sub-sequence of an overall computation of the model. Clause $(viii)$ ensures that paths resulting out of forking of parallel threads do not have any common prefix.

Example 6. *To examine how finite paths can capture a computation involving an unknown number of loop traversals, let us consider the example of Figure 4.2. By Definition 12, the set C of static cut-points is* $\{p_1, p_2, p_3, p_6, p_{10}, p_{13}\}$ *and the paths will be* $\alpha_1 = \langle \{t_1\}, \{t_3\}, \{t_5\}, \{t_7, t_8\}, \{t_{10}\} \rangle$, $\alpha_2 = \langle \{t_2\}, \{t_4\}, \{t_6\} \rangle$ *and* $\alpha_3 = \langle \{t_9\} \rangle$ *respectively. Let us now try to express a computation* $\mu_{p_{13}}$ *of the out-port* p_{13} *in terms of paths, where* $\mu_{p_{13}} = \langle T_1 = \{t_1, t_2\}, T_2 = \{t_3, t_4\}, T_3 = \{t_5, t_6\}, T_4 = \{t_7, t_9\}, T_5 = \{t_9\}, T_5 = \{t_9\}, T_6 = \{t_8\}, T_7 = \{t_{10}\} \rangle$*; the computation, however, cannot be expressed in terms of the paths* $\{\alpha_1, \alpha_2, \alpha_3\}$ *because the member* $\{t_7, t_8\}$ *of the path* α_1 *gets fragmented and combines in parallel with the path* α_3 *in the member* T_4 *in* $\mu_{p_{13}}$*. If we had* p_4, p_5, p_9, p_{11} *and* p_{12} *also as cut-points, the path-set would have been* $\alpha_1' = \langle \{t_1\} \rangle, \alpha_2' = \langle \{t_2\} \rangle, \alpha_3' = \langle \{t_3\}, \{t_5\} \rangle, \alpha_4' = \langle \{t_4\}, \{t_6\} \rangle, \alpha_5' = \langle \{t_7\} \rangle, \alpha_6' = \langle \{t_9\} \rangle, \alpha_7' = \langle \{t_8\} \rangle$ *and* $\alpha_8' = \langle \{t_{10}\} \rangle$ *(shown by dotted triangles in Figure 4.2). Now, intuitively, the computation* $\mu_{p_{13}}$ *could be depicted as the sequence* $(\alpha_1' \parallel \alpha_2').(\alpha_3' \parallel \alpha_4').(\alpha_5' \parallel \alpha_6').\alpha_6'.\alpha_6'.\alpha_7'.\alpha_8'$ *of concatenation of parallelisable paths from the set* $\{\alpha_1', \alpha_2', \alpha_3', \alpha_4', \alpha_5', \alpha_6', \alpha_7', \alpha_8'\}$*, where* $(\alpha_1 \parallel \alpha_2)$ *means parallel execution of* α_1 *and* α_2 *and* $(\alpha_1.\alpha_2)$ *means sequential execution* α_1 *followed by* α_2*.* ■

The above example underlines the need for introducing further cut-points and the notion of parallel paths and their concatenation for capturing computations. For the former, a notion of *token tracking* execution is necessary which is described as follows. The notion of parallel paths is introduced subsequently.

A token tracking execution essentially captures all computations of the model with the token values abstracted out and every loop traversed exactly once. Therefore, in the context of token tracking execution, the term marking means only place marking. Thus, a token tracking execution starts with an initial marking comprising tokens at the in-ports and tracks the progress of the tokens through the successor markings avoiding repetitions of sub-sequences of markings. If a given marking involves a token holding place with more than one outgoing transition, then firing of such transitions will be mutually exclusive of each other; hence there may be more than one alternative set of successor markings all of which are covered in a DFS manner by the token tracking execution mechanism. Note that the number of times a loop is executed varies from one execution to another depending on the input token values. During progression of tokens, if any marking contains at least one static or dynamic cut-point, mark all places in the marking as dynamic cut-points and if any transition contains more than one post places, all of these post places are also marked as dynamic cut-points. If there are parallel threads with at least one of them involving a loop, then the places encountered along all such threads may all become dynamic cut-points. We refer to such a scenario as a *degenerate case*, whereupon dynamic cut-points are introduced exhaustively in all the places of the markings, both in the loop body as well as in other parallel threads. Algorithms 1 and 2 depict the procedure of token tracking execution (These algorithms occur embedded in Algorithms 5 and 3 of the main module). The definition of *degenerate case* is as follows.

Definition 14 (Degenerate phase of token tracking execution)**.** *The degenerate phase of a token tracking execution sets in when the latter encounters a place marking P_M such that $|P_M| > 1$ and P_M contains at least one static cut-point having a back edge leading to itself. The generation phase gets over when the token tracking execution encounters a place marking P_M having just one post-transition, i.e., $|P_M^{\circ}| \leq 1$ or P_M contains only out-ports.*

Thus, for the starting of each degenerate phase of the token tracking execution, the condition $|P_M| > 1$ indicates that the token tracking execution is traversing through parallel threads; the clause "P_M contains at least one static cut-point having a back edge leading to itself" indicates that at least one of the parallel threads contains a loop. The if-statement 7-9 in the function `tokenTrack` (Algorithm 2) captures the setting in of the degenerate phase. The ending of the degenerate phase is indicated by the condition $|P_M^\circ| = 1$ capturing the fact that all the parallel threads have merged. The if-statement 4-6 of the function `tokenTrack` (Algorithm2) captures the termination of the degenerate phase. We illustrate the scenario by the following example.

Algorithm 1 SETOFDCP **initTokenTrack** (N)

Inputs: The input parameter is the PRES+ model N.
Outputs: The set C_d of dynamic cut-points

1: $M_h \Leftarrow inP$; /* Place − marking at hand − initialized to in-ports*/
   $C_d = \emptyset$ /* set of dynamic cut-points − initially empty */ degenerate = false;
2: $\mathcal{T}$ = **compAllSetsOfConcurTrans** (M_h, N);
3: **for** $T \in \mathcal{T}$ **do**
4:     $C_d = C_d \cup$ **tokenTrack**$(C_d, M_h, T,$ degenerate, $N)$
5: **end for**
6: **return** C_d

Algorithm 2 SETOFDCP **tokenTrack** (C_d, M_h, T_e, degenerate, N)

Inputs: The first parameter is set C_d of dynamic cut-points. The second parameter is a marking M_h. The third parameter is a set T_e of enabled maximally parallelisable transitions. The fourth parameter *degenerate* is a flag value. The fifth parameter is the PRES+ model N.
Outputs: C_d

1: $C_d = \emptyset$;
2: $M_{new} \Leftarrow T_e^\circ$; /* post-places of T_e acquire tokens */
3: $M_h \Leftarrow (M_h - {}^\circ T_e) \cup M_{new}$; /* modify M_h by deleting the pre-places of the concurrent transitions and adding their post-places */
4: **if** $(|M_h^\circ| \leq 1$ and degenerate = true) **then**
5:     degenerate = false;
6: **end if**
7: **if** (there exists a back edge leading to some p in M_h and $|M_h| > 1$) **then**
8:     degenerate = true;
9: **end if**
10: **if** (degenerate = true or at least one p in M_h is a cut-point or $|T_e^\circ| > |T_e|$) **then**
11:     $C_d = C_d \cup M_h$
12: **end if**
13: $\mathcal{T}$ = **compAllSetsOfConcurTrans** (M_h, N);
    /* unmark all the marked transitions */
14: **for** each $T_e \in \mathcal{T}$ **do**
15:     $C_d = C_d \cup$ **tokenTrack** $(C_d, M_h, T_e,$ degenerate, $N)$ //call itself recursively;
16:     **return** C_d;
17: **end for**

Example 7. *Let us consider the designation procedure of dynamic cut-points for the PRES+ model given in Fig. 4.3. The token tracking execution starts with the set* $\{p_1, p_2, p_3, p_6\}$ *as the initially marked places. After firing of the enabled transitions*

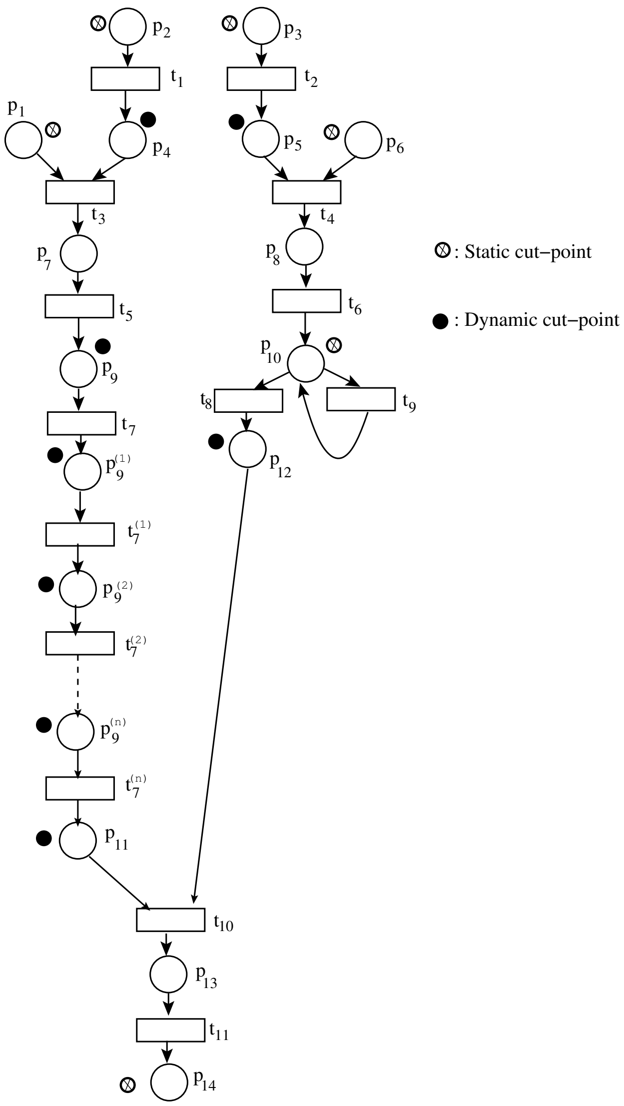

Figure 4.3: Dynamic cut-point introduction

t_1 and t_2, the next set of marked places becomes $\{p_1, p_4, p_5, p_6\}$; as p_1 and p_6 are already designated as (static) cut-points, the places p_4 and p_5 are designated as (dynamic) cut-points. From the set $\{p_1, p_4, p_5, p_6\}$ of marked places, the next set of enabled transitions is found as $\{t_3, t_4\}$ from which the following alternating subsequence of places and transitions is obtained: $\{p_7, p_8\} \rightarrow \{t_5, t_6\} \rightarrow \{p_9, p_{10}\}$. At this point, since p_{10} is a (static) cut-point, p_9 is designated as a (dynamic) cut-point. Also since p_{10} has a back edge, the degenerate phase sets in. The place p_{10} has two out-transitions t_8 and t_9. Therefore, two alternative sets of enabled transitions are obtained for the marking, namely, $\{t_7, t_8\}$ and $\{t_7, t_9\}$. These two alternatives are explored in a DFS manner. For the set $\{t_7, t_9\}$ of transitions, the next marking becomes

$\{p_9^{(1)}, p_{10}\}$; since p_{10} is a static cut-point, $p_9^{(1)}$ becomes a dynamic cut-point. Now it can be clearly seen that since the number of traversals through the loop from p_9 to itself is not known, all of the places $p_9^{(2)}$ through $p_9^{(n)}$ and also p_{11} would become dynamic cut-points. As long as the loop executes, the marking remains as $\{p_{10}, p_{11}\}$. Finally, the loop terminates resulting in the marking $\{p_{11}, p_{12}\}$ whereupon p_{12} also becomes a dynamic cut-point. At this stage the enabled transition becomes the singleton set $\{t_{10}\}$ indicating merging of the parallel threads; under this situation, the degenerate case ceases to exist. After firing of $\{t_{10}\}$, the next marking becomes $\{p_{13}\}$. At this stage the enabled transition becomes $\{t_{11}\}$ and the next marking is $\{p_{14}\}$. It is to be noted that p_{14} is an out-port. Therefore, token tracking execution cannot proceed any further. Now the token tracking execution backtracks up to the set $\{p_9, p_{10}\}$ of places and takes the other alternative of enabled transitions, i.e., $\{t_7, t_8\}$. For the set $\{t_7, t_8\}$ of transitions, the successor marking becomes $\{p_9^{(1)}, p_{12}\}$; at this stage, it is to be noted that the places $p_9^{(1)}$ and p_{12} are already designated as dynamic cut-points, therefore, no further cut-point designation takes place till the marking reaches p_{14}. When the marking is $\{p_{14}\}$, the token tracking execution ends because it has covered all the alternatives and p_{14} is an out-port. If the DFS traversal pursues $\{t_7, t_8\}$ as the first alternative, then the token tracking execution proceeds without introducing any further cut-points till the marking reaches p_{14}. However, when the execution takes the second alternative, i.e., $\{t_9, t_7\}$, the degenerate phase sets in and, therefore, it designates the places $p_9^{(1)}$ to p_{11} and p_{12} as dynamic cut-points. Similarly, when the set of enabled transitions is $\{t_{10}\}$, the degenerate case ceases to exist. Therefore, the set of dynamic cut-points is computed as $\{p_4, p_5, p_9, p_9^{(1)}, \ldots, p_9^{(n)}, p_{11}, p_{12}\}$. ■

We can now formalize the definition of dynamic cut-points as follows.

Definition 15 (Dynamic cut-point)**.** *A place p is designated as a dynamic cut-point if during a token tracking execution of the model (with static cut-points already incorporated), a place marking P_M containing p is encountered such that one of the following three conditions is satisfied:*

(i) P_M contains at least one (static or dynamic) cut-point, or

(ii) P_M contains more number of places than its pre-transitions, i.e., $|P_M| > |^\circ P_M|$; this indicates that the token tracking execution has reached a point of creation of some parallel threads; or

(iii) token tracking execution is in the degenerate phase.

It is to be noted that a cut-point is designated as static based on some static structural features of the PRES+ model (namely, in-ports, back edges and out-ports). In contrast, the dynamic cut-points are determined by a token tracking execution of the model (hence the qualifier *dynamic*). The definition of path (Definition 13) is modified with *cut-points* read as both static and dynamic cut-points. Also, from now onwards, by cut-points we will mean both static and dynamic cut-points.

The set of static cut-points is not unique; as explained earlier, it depends on the DFS traversal used to identify the back edges. Since the set of static cut-points is used during token tracking execution, the set of dynamic cut-points is also not unique; thus, the set of cut-points is not unique. However, given a set of static cut-points, the set of dynamic cut-points is unique irrespective of the order in which the choices are exercised during DFS traversal of the token tracking execution.

4.1.1 Characterization of a path

We associate with a path α two entities namely, $R_\alpha(f_{pv}(^\circ\alpha))$, the condition of execution of the path α, and $r_\alpha(f_{pv}(^\circ\alpha))$, the data transformation along the path α. For any computation μ_α of the form $\langle T_1, T_2, \ldots \rangle$ of the path α, for any marking $M = \langle P_M, val_M \rangle$, the predicate R_α depicts the condition that must be satisfied by $val_M(^\circ\alpha)$ so that α is executed for that marking. The data transformation r_α depicts the token value obtained in α° after execution of α. Thus, the places in α° contain the value $r_\alpha(f_{pv}(^\circ\alpha))$ after execution of the path α.

Example 8. *Figure 4.4(a) depicts a PRES+ model having p_0, p_1, p_2, p_3 and p_7 as cut-points following the static and dynamic cut-point introduction rules given in the previous section. So the corresponding paths are $\alpha_1 = \langle \{t_0\} \rangle$, $\alpha_2 = \langle \{t_1\} \rangle, \alpha_3 = \langle \{t_2, t_3\}, \{t_4\}, \{t_5\} \rangle$ and $\alpha_4 = \langle \{t_2, t_3\}, \{t_4\}, \{t_6\} \rangle$, respectively. Figure 4.4(b) depicts how the data transformation (r_{α_3}) and the condition of execution (R_{α_3}) for the path α_3 are computed. A forward traversal along the forward direction of the edges of the path α_3 from p_2, p_3 to p_1 is used for this purpose. Instead, we may use backward traversal (along the edges in the reverse direction) as an alternative. In Figure 4.4(b), let the token values at both p_2 and p_3 be v_{p_2} and v_{p_3} respectively, and the conditions*

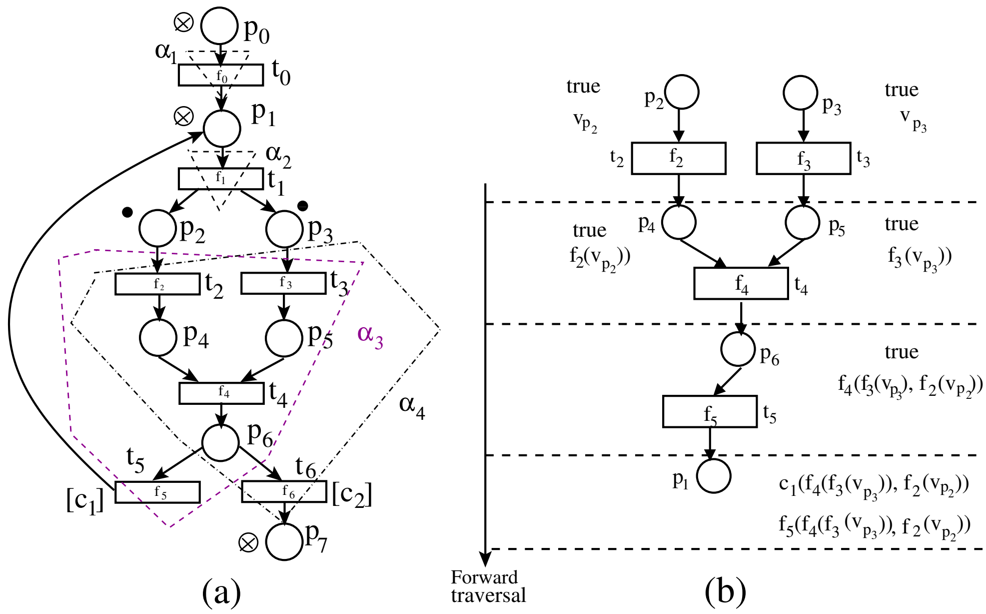

Figure 4.4: Computation of the characteristics of a path.

g_{t_2} and g_{t_3} associated with the t_2 and t_3 are true. The token values at both p_4 and p_5 after firing of both t_2 and t_3 becomes $v_{p_4} = f_2(v_{p_2})$ and $v_{p_5} = f_3(v_{p_3})$, respectively and the condition remains true. After firing of t_4, the token value at p_6 becomes $v_{p_6} = f_4(v_{p_4}, v_{p_5}) = f_4(f_2(v_{p_2}), f_3(v_{p_3}))$ and the condition still remains true. When the condition c_1 associated with the transition t_5 is satisfied by v_{p_6}, t_5 fires. After firing of t_5, the token value at p_1 becomes $v_{p_1} = f_5(v_{p_6}) = f_5(f_4(f_2(v_{p_2})), f_3(v_{p_3}))$ which is to the data transformation r_{α_3} of the path α_3 and the condition of execution R_{α_3} is $c_1(f_4(f_2(v_{p_2}), f_3(v_{p_3})))$. ■

4.1.2 Computation in terms of concatenation of parallel paths

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

Definition 16 (Successor relation between two paths)**.** *A path α_i succeeds a path α_j, denoted as $\alpha_i \succ \alpha_j$, if there exists a set of paths $\alpha_{k_1}, \alpha_{k_2}, \ldots, \alpha_{k_n}$, a place $p_i \in {}^\circ\alpha_i$ and a set of places $\{p_{k_m} \in {}^\circ\alpha_{k_m}, 1 \leq m \leq n\}$ such that $\langle\, last(\alpha_j), p_{k_1}\rangle$, $\langle\, last(\alpha_{k_1}), p_{k_2}\rangle, \ldots,$ $\langle\, last(\alpha_{k_n}), p_i\rangle \in O \subseteq T \times P$, $n \geq 0$, and none of them is a back edge. The expression $\alpha_i \not\succ \alpha_j$ is used as a shorthand for $\neg\alpha_i \succ \alpha_j$.*

Now we define the notion of parallelisable paths and concatenation of paths.

Definition 17 (Parallelizable pairs of paths)**.** *Two paths α_i and α_j are said to be parallelisable, denoted as $\alpha_i \asymp \alpha_j$, if*

(i) $\alpha_i \not\succeq \alpha_j$ *and* $\alpha_j \not\succeq \alpha_i$ *and*

(ii)
$$\begin{aligned}\forall \alpha_k, \alpha_l, [&\alpha_k \neq \alpha_l \wedge \alpha_i \succeq \alpha_k \wedge \alpha_j \succeq \alpha_l \rightarrow \\ &{}^\circ\alpha_k \cap {}^\circ\alpha_l = \emptyset \vee \\ &\exists \alpha_m, \alpha_n (\alpha_m \neq \alpha_n \wedge \alpha_m \succeq \alpha_k \wedge \alpha_i \succ \alpha_m \wedge \\ &\qquad \alpha_n \succeq \alpha_l \wedge \alpha_j \succ \alpha_n \wedge \alpha_m^\circ \cap \alpha_n^\circ \neq \emptyset)].\end{aligned}$$

When $\alpha_i \asymp \alpha_j$, their parallel combination is denoted as $\alpha_i \parallel \alpha_j$.

In clause(ii) of the above definition, the first disjunct in the consequent necessitates that the path α_i, α_j or any other paths preceding α_i and α_j should have no common pre-places; the second disjunct necessitates that (even if the first disjunct does not hold,) there should be some intervening paths preceding α_i and α_j having some common post-places (i.e., following these paths the control flow must have merged). It is to be noted that a parallelisable pair of paths is not a path.

Definition 18 (Set of parallelizable paths)**.** *A set $Q_P = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ of paths is said to be parallelisable if $\forall i, j, 1 \leq i \neq j \leq k$, $\alpha_i \asymp \alpha_j$ holds. Alternatively it is also denoted as $(\alpha_1 || \alpha_2 || \ldots || \alpha_k)$. It is to be noted that the members of any set of parallelisable paths can be executed in any arbitrary order.*

Definition 19 (Concatenation of a path to a set of parallelizable paths)**.** *A path α is said to be a concatenated path obtained by concatenation of a path α' to a set $Q_P = \{\alpha_1, \ldots, \alpha_k\}$ of parallelisable paths if $\forall i, 1 \leq i \leq k, \alpha_i^\circ \subseteq {}^\circ\alpha'$. The path α is denoted as $(\alpha_1 \parallel \ldots \parallel \alpha_k).\alpha'$. The intermediary cut-points ($\bigcup_{1 \leq i \leq k} \alpha_i^\circ$) lose their cut-point designation so that the concatenated path α does not have any intermediary cut-points.*

The characterization of a concatenated path is as follows: Let $\alpha_1, \ldots, \alpha_k$ be parallelisable paths having a successor path α'; that is, α_i, $1 \leq i \leq k$, satisfies the relation $\alpha_i^\circ \cap {}^\circ\alpha' \neq \emptyset$. Let α be the concatenated path $(\alpha_1 \parallel \alpha_2 \parallel \ldots \parallel \alpha_k).\alpha'$. In the following we describe the method of obtaining the condition of execution R_α and the data transformation r_α of the path α from R_{α_i}, r_{α_i}, $1 \le i \le k$, $R_{\alpha'}$ and $r_{\alpha'}$. Let $f_{pv}(^\circ\alpha_i)$ $(f_{pv}(\alpha_i^\circ))$ be the vector of variable (names) associated with the places of $^\circ\alpha_i(\alpha_i^\circ), 1 \le i \le k$, respectively. Let $\overline{v}$ be the vector of variables associated with the output places of $\alpha_i^\circ, 1 \le i \le n$. Obviously $\overline{v} = \langle r_{\alpha_1}(f_{pv}(^\circ\alpha_1)),\ r_{\alpha_2}(f_{pv}(^\circ\alpha_2)), \ldots, r_{\alpha_k}(f_{pv}(^\circ\alpha_k))\rangle$. Hence $R_\alpha = \bigwedge_{i=1}^{k} R_{\alpha_i}(f_{pv}(^\circ\alpha_i)) \wedge R_{\alpha'}(f_{pv}(^\circ\alpha'))\{\overline{v}/f_{pv}(^\circ\alpha')\}$ and $r_\alpha = r_{\alpha'}(f_{pv}(^\circ\alpha'))\{\overline{v}/f_{pv}(^\circ\alpha')\}$ where, $\{\overline{v}/f_{pv}(^\circ\alpha')\}$ is a substitution of $f_{pv}(^\circ\alpha')$ by $\overline{v}$. We illustrate the computation of R_α and r_α through the following example.

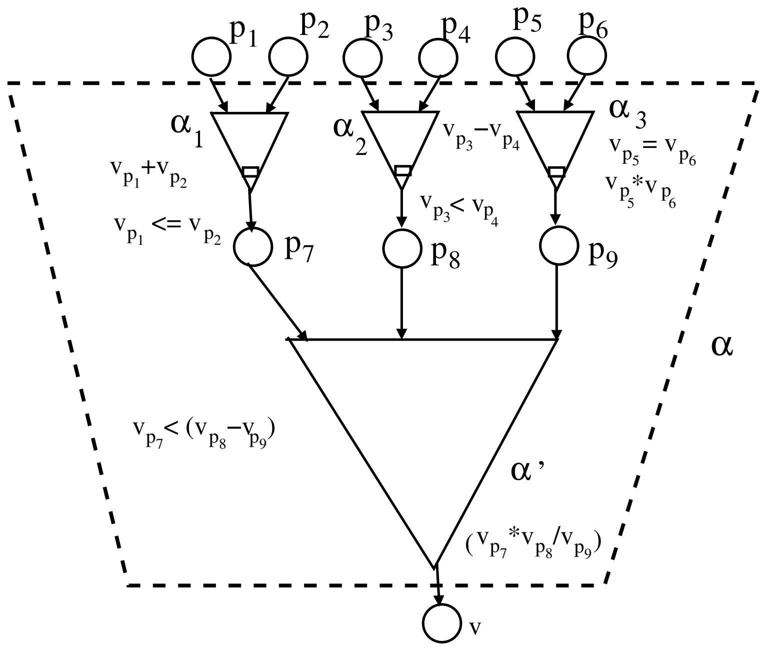

Figure 4.5: Concatenated Path of a PRES+ model.

Example 9. *Fig 4.5 depicts three paths* α_1, α_2 *and* α_3 *having a successor path* α'. *So, the concatenated path* α *is of the form* $(\alpha_1 \parallel \alpha_2 \parallel \alpha_3).\alpha'$. *Let* R_{α_1} *be* $v_{p_1} \le v_{p_2}$, r_{α_1} *be* $v_{p_1} + v_{p_2}$, R_{α_2} *be* $v_{p_3} < v_{p_4}$, r_{α_2} *be* $v_{p_3} - v_{p_4}$, R_{α_3} *be* $v_{p_5} = v_{p_6}$, r_{α_3} *be* $v_{p_5} * v_{p_6}$, $R_{\alpha'}$ *be* $v_{p_7} < v_{p_8} - v_{p_9}$, $r_{\alpha'}$ *be* $v_{p_7} * v_{p_8}/v_{p_9}$. *Let* $\overline{v}$ *be* $\langle v_{p_7}, v_{p_8}, v_{p_9}\rangle$ *and* $\overline{v'}$ *be* $\langle r_{\alpha_1}(f_{pv}(^\circ\alpha_1)), r_{\alpha_2}(f_{pv}(^\circ\alpha_2)), r_{\alpha_3}(f_{pv}(^\circ\alpha_3))\rangle = \langle v_{p_1} + v_{p_2}, v_{p_3} - v_{p_4}, v_{p_5} * v_{p_6}\rangle$. *Therefore, the condition of execution*

$R_\alpha = R_{\alpha_1}(f_{pv}(^\circ\alpha_1)) \wedge R_{\alpha_2}(f_{pv}(^\circ\alpha_2)) \wedge R_{\alpha_3}(f_{pv}(^\circ\alpha_3)) \wedge R_{\alpha'}(\overline{v})\{\overline{v'}/\overline{v}\}$, *i.e.*, $(v_{p_1} \le v_{p_2}) \wedge (v_{p_3} < v_{p_4}) \wedge (v_{p_5} = v_{p_6}) \wedge (v_{p_7} < v_{p_8} - v_{p_9})$

$$\{\langle v_{p_1} + v_{p_2},\ v_{p_3} - v_{p_4},\ v_{p_5} * v_{p_6}\rangle / \langle v_{p_7}, v_{p_8}, v_{P_9}\rangle\}$$

$= (v_{p_1} \le v_{p_2}) \wedge (v_{p_3} < v_{p_4}) \wedge (v_{p_5} = v_{p_6}) \wedge ((v_{p_1} + v_{p_2}) < (v_{p_3} - v_{p_4}) - (v_{p_5} * v_{p_6}))$.

The data transformation is

$r_\alpha = r_{\alpha'}(\overline{v})\ \{\overline{v'}/\overline{v}\}$,
i.e., $(v_{p_7} * v_{p_8}/v_{p_9})\{\langle v_{p_1} + v_{p_2}, v_{p_3} - v_{p_4}, v_{p_5} * v_{p_6}\rangle / \langle v_{p_7}, v_{p_8}, v_{P_9}\rangle\}$
$= (v_{p_1} + v_{p_2}) * (v_{p_3} - v_{p_4})/(v_{p_5} * v_{p_6})$. ■

From a given set P of places, there can be more than one set of maximally parallelisable transitions because one or more places in P might feature multiple outgoing transitions; let $T_1, T_2, \ldots, T_k$ be all the sets of maximally parallelisable transitions *from* P. They satisfy the following properties:

Prop 1: $P \cap {}^\circ T_i \neq \emptyset, 1 \leq i \leq k$.

Prop 2: ${}^\circ T_i \cap {}^\circ T_j \neq \emptyset, 1 \leq i \neq j \leq k$, because if we have T_i, T_j, $i \neq j$ such that ${}^\circ T_i \cap {}^\circ T_j = \emptyset$, then $T_i \cup T_j$ is parallelisable and $T_i, T_j \subset T_i \cup T_j$ are not maximal.

Prop 3: For any $i, 1 \leq i \leq k$, let R_i be the conjunction of conditions associated with the members of T_i such that T_i is fired only if $R_i(f_{pv}(P))$ holds. From property (Prop 2) and the fact that the model is deterministic, it follows that for all i, j, $1 \leq i \neq j \leq k$, $R_i(f_{pv}(P)) \wedge R_j(f_{pv}(P))$ is unsatisfiable. Also, since the model is completely specified, $\bigvee_{i=1}^{k} R_i(f_{pv}(P))$ is valid. Hence, given any set P of places, there is a unique collection of sets of maximally parallelisable transitions which cover all the post-transitions of P.

The following theorem captures the uniqueness of the set of paths obtained from a given set of cut-points.

Theorem 1. *For any PRES+ model N, for a set of cut-points that includes all the static cut-points, as identified through Definition 12, and all the dynamic cut-points as defined in Definition 15, the set of paths covering all the transitions is unique.*

Proof. Let there be two distinct sets Q_1 and Q_2 of paths where each of the sets covers all the transitions of the given PRES+ model N. Let $\alpha = \langle T_1, T_2, \ldots, T_n\rangle$ be a path such that $\alpha \in Q_1 - Q_2$. We argue that any member T_i of α, $1 \leq i \leq n$, represents the only way to group the transitions of T_i into a maximally parallelisable set and hence conclude that α must be in Q_2 as well. We prove it by induction on i.

Basis $(i = 1)$*:* Note that ${}^\circ T_1 = {}^\circ\alpha$ because if ${}^\circ T_1 \subset {}^\circ\alpha$, then after firing of T_1 the marking will have places of $\left({}^\circ\alpha - {}^\circ T_1\right) \cup T_1^\circ$. Hence from Definition 15 of dynamic cut-points, all the members of T_1° will be dynamic cut-points and hence the path α will not have any member other than T_1. From clause (6) of the definition of path (Definition 13), the set T_1 is given to be a maximally parallelisable set of transitions from $^\circ\alpha$ in the sequence α. From property *Prop 3* above, T_1 is unique among all the maximally parallelisable transition sets from $^\circ\alpha$.

Induction hypothesis: For any k, $1 \leq k < n$, let T_k in α be a unique way to group all its transitions into a maximally parallelisable set from a set P_k of places, where for $k = 1, P_k = {^\circ\alpha}$ and for $1 < k < n$, $P_k = \left(T_1^\circ \cup T_2^\circ \cup \ldots \cup T_{k-1}^\circ\right) - \left({^\circ T_2} \cup \ldots \cup {^\circ T_{k-1}}\right)$.

Induction step: Let T_{k+1} be one of the maximally parallelisable sets of out-going transitions from the set P_{k+1} of places; $P_{k+1} = \left(P_k - {^\circ T_k}\right) \cup T_k^\circ = P_k \cup T_k^\circ - {^\circ T_k} = \left(\bigcup_{i=1}^{i=k} T_i^\circ\right) - \left(\bigcup_{i=2}^{i=k} {^\circ T_i}\right)$ where all the sets $T_i, 1 \leq i \leq k$, are unique by induction hypothesis. It basically collects all the post-places of the previous sets $T_1, \ldots, T_{k-1}$ which are not covered as the pre-places of $T_2, \ldots, T_k$, and all the post places of T_k. By construction of the path α, T_{k+1} is a maximally parallelisable set of transitions from the set P_{k+1} of places. From property *P3*, T_{k+1} is unique among all the possible sets of maximally parallelisable transitions arising from P_{k+1}. This completes the induction.

Thus, the transitions of $T_1, \ldots, T_n$ in α cannot be covered by any other sequence of parallelisable transitions. Since Q_2 contains all paths which cover all the transitions of the model, it must cover the transitions of α (in a single collection) in the same way as α does. Hence $\alpha \notin Q_1 - Q_2$. Thus, $Q_1 - Q_2 = \emptyset$. Similarly, it can be shown that $Q_2 - Q_1 = \emptyset$ and thus $Q_1 = Q_2$. □

Definition 20 (DCP induced path cover)**.** *A finite set of paths* $\Pi = \{\alpha_0, \alpha_1, \ldots, \alpha_k\}$ *is said to be a path cover of a PRES+ model N if any computation μ of an out-port of N can be represented as a sequence of concatenations of parallelisable paths from* Π.

In Example 6, it is noted that the set $\{\alpha_1, \alpha_2, \alpha_3\}$ of paths which are obtained only from the static cut-points is not a path cover. Whereas, the set $\{\alpha_1', \alpha_2', \alpha_3', \alpha_4', \alpha_5', \alpha_6', \alpha_7', \alpha_8'\}$ of paths which are obtained from both static and dynamic cut-points is a path cover.

Theorem 2. *Let C be a set of cut-points that includes all the static cut-points, as identified through Definition 12, and all the dynamic cut-points, obtained by a token tracking execution of N as defined in Definition 15. The set of paths corresponding to the set C is a path cover of N.*

Proof. Let μ_p be a computation of an out-port p of the form $\langle T_1, T_2, \ldots, T_l \rangle$ where, ${}^\circ T_1 \subseteq inP$, $p \in T_l^\circ$, $T_i^\circ \subseteq P_{M_i}$, $1 \leq i < l$, where M_i is a marking and $M_{i+1} = M_i^+$, the successor marking of M_i, for all i, $1 \leq i < l$. The sequence μ_p can be represented as the sequence $\langle T_1, \ldots, T_{i_1}, T_{i_1+1}, \ldots, T_{i_2}, \ldots, T_{i_m}, \ldots, T_l \rangle$, where $T_{i_j}^\circ$, $1 \leq j \leq m$, and T_l° are all members of C (cut-points) and there are no other transitions in the above sequence whose post- places are members of C. Each of the sub-sequences $\langle T_1, \ldots, T_{i_1} \rangle$, $\{\langle T_{i_j+1}, \ldots, T_{i_{j+1}} \rangle, 1 \leq j < m\}$ and $\langle T_{i_m+1}, \ldots, T_l \rangle$ are parallelisable paths by Definition 18; note that whenever the cardinality of the post-place of the last member of any of the above sub-sequences is greater than 1, the sub-sequence represents a set of parallelisable paths (having the same cardinality) and is not a single path. Each of the remaining sub-sequences represents a single path and hence is a singleton set of parallelisable paths. However, $|T_l| = 1$; hence, the last sub-sequence $\langle T_{i_m+1}, \ldots, T_l \rangle$ is a single path (by Definition 13). Hence the above computation μ_p is a concatenation of parallelisable paths. If there is no such sub-sequence in μ_p, i.e., $T_{i_1}, \ldots, T_{i_m}$ do not exist in μ_p, then μ_p itself is a single path which is a trivial case of a concatenation of parallelisable paths. □

4.1.3 Equivalence checking using paths – An Example

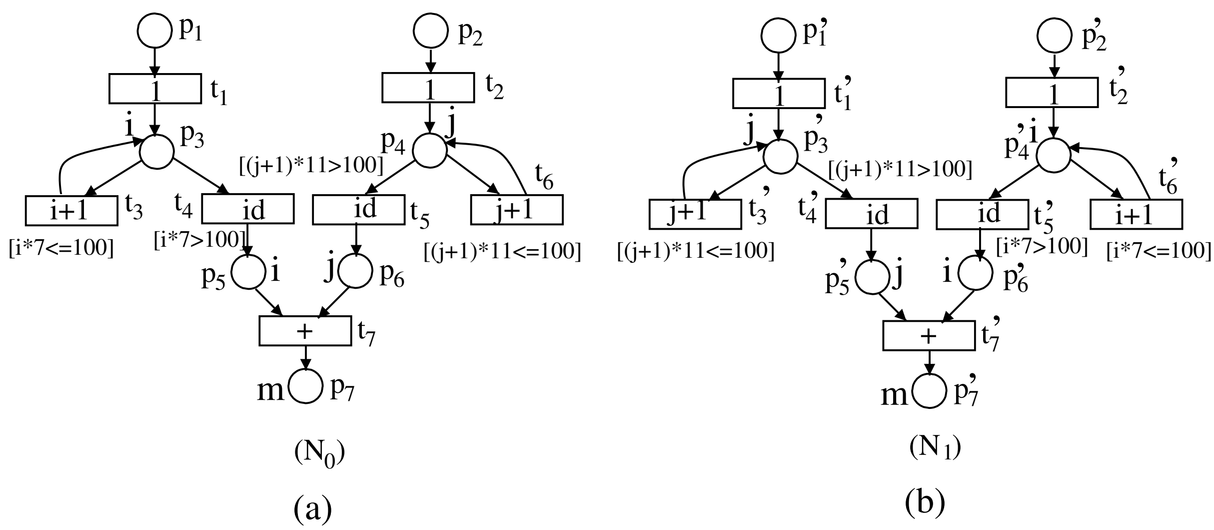

Figure 4.6: Initial and Transformed Behaviour.

Before describing the details of construction of paths, in this section we first demonstrate through an example the importance of paths in establishing equivalence between two PRES+ models. The fact that a path based equivalence checking strategy indeed accomplishes the computational equivalence of two PRES+ models is formally established in the next chapter.

Example 10. *Figure 4.6 (a) represents a PRES+ model (N_0) of an initial program which computes $\lceil \frac{100}{7} \rceil + \lfloor \frac{100}{11} \rfloor$. This initial program is transformed using loop swapping transformation whose model N_1 is given in Fig. 4.6 (b). Specifically, in Fig. 4.6(a), the fragments $\{t_1\}.(\{t_3\})^n.\{t_4\}$ computes the first term and the fragment $\{t_2\}.(\{t_6\})^m.\{t_5\}$ computes the second term for suitable values of m and n ($n \neq m$); correspondingly, in Figure 4.6(b), $\{t_1^{'}\}.(\{t_3^{'}\})^m.\{t_4^{'}\}$ computes the first term and $\{t_2^{'}\}.(\{t_6^{'}\})^n.\{t_5^{'}\}$ computes the second term for the same values of n and m. The set of variables (common to both models) $V = \{i, j, m\} \cup \{\delta\}$. The respective place to variable mappings are as follows. $f^0_{pv}(p_1) = f^0_{pv}(p_2) = f^1_{pv}(p_1^{'}) = f^1_{pv}(p_2^{'}) = \delta$, since $p_1, p_2, p_1^{'}, p_2^{'}$ are dummy places; $f^0_{pv}(p_3) = f^0_{pv}(p_5) = f^1_{pv}(p_4^{'}) = f^1_{pv}(p_6^{'}) = i$; $f^0_{pv}(p_4) = f^0_{pv}(p_6) = f^1_{pv}(p_3^{'}) = f^1_{pv}(p_5^{'}) = j$; $f^0_{pv}(p_7) = f^1_{pv}(p_7^{'}) = m$.*

In Figure 4.6 (a), suppose the static cut-points are p_1, p_2, p_3, p_4 and p_7 and the dynamic cut-points are p_5 and p_6; then the paths are $\alpha_1 = \langle\{t_1\}\rangle, \alpha_2 = \langle\{t_2\}\rangle$, $\alpha_3 = \langle\{t_3\}\rangle, \alpha_4 = \langle\{t_6\}\rangle$, $\alpha_5 = \langle\{t_4\}\rangle$ $\alpha_6 = \langle\{t_5\}\rangle$ and $\alpha_7 = \langle\{t_7\}\rangle$. Similarly, in Figure 4.6 (b), suppose $p_1^{'}, p_2^{'}, p_3^{'}, p_4^{'}$ and $p_7^{'}$ are the cut-points of which $p_5^{'}$ and $p_6^{'}$ are dynamic cut-points; then the paths are $\beta_1 = \langle\{t_1^{'}\}\rangle, \beta_2 = \langle\{t_2^{'}\}\rangle, \beta_3 = \langle\{t_3^{'}\}\rangle$, $\beta_4 = \langle\{t_6^{'}\}\rangle$, $\beta_5 = \langle\{t_4^{'}\}\rangle$, $\beta_6 = \langle\{t_5^{'}\}\rangle$ and $\beta_7 = \langle\{t_7^{'}\}\rangle$.

The condition of execution of the paths of N_0 are $R_{\alpha_1}(v_{p_1}) \equiv R_{\alpha_2}(v_{p_2}) \equiv \top$, $R_{\alpha_3}(v_{p_3}) : v_{p_3} * 7 \leq 100$, $R_{\alpha_4}(v_{p_4}) : (v_{p_4} + 1) * 11 \leq 100$, $R_{\alpha_5}(v_{p_3}) : v_{p_3} * 7 > 100$, $R_{\alpha_6}(v_{p_4}) : (v_{p_4} + 1) * 11 > 100$ and $R_{\alpha_7}(v_{p_5}, v_{p_6}) \equiv \top$ and the corresponding data transformations are $r_{\alpha_1}(v_{p_1}) = 1$, $r_{\alpha_2}(v_{p_2}) = 1$, $r_{\alpha_3}(v_{p_3}) = v_{p_3} + 1$, $r_{\alpha_4}(v_{p_4}) = v_{p_4} + 1$, $r_{\alpha_5}(v_{p_3}) = v_{p_3}$, $r_{\alpha_6}(v_{p_4}) = v_{p_4}$ and $r_{\alpha_7}(v_{p_5}, v_{p_6}) = v_{p_5} + v_{p_6}$. Likewise, in Figure 4.6 (b), the condition of execution along the paths are $R_{\beta_1}(v_{p_1^{'}}) \equiv R_{\beta_2}(v_{p_2^{'}}) \equiv \top$, $R_{\beta_3}(v_{p_3^{'}}) : (v_{p_3^{'}} + 1) * 11 \leq 100$, $R_{\beta_4}(v_{p_4^{'}}) : v_{p_4^{'}} * 7 \leq 100)$, $R_{\beta_5}(v_{p_3^{'}}) : (v_{p_3^{'}} + 1) * 11 > 100$, $R_{\beta_6}(v_{p_4^{'}}) : v_{p_4^{'}} * 7 > 100$, and $R_{\beta_7}(v_{p_5^{'}}, v_{p_6^{'}}) \equiv \top$ and the corresponding data transformations are $r_{\beta_1}(v_{p_1^{'}}) = 1$, $r_{\beta_2}(v_{p_2^{'}}) = 1$, $r_{\beta_3}(v_{p_3}^{'}) = v_{p_3^{'}} + 1$, $r_{\beta_4}(v_{p_4^{'}}) = v_{p_4^{'}} + 1$, $r_{\beta_5}(v_{p_3}) = v_{p_3^{'}}$, $r_{\beta_6}(v_{p_4^{'}}) = v_{p_4^{'}}$ and $r_{\beta_7}(v_{p_5^{'}}, v_{p_6^{'}}) = v_{p_5^{'}} + v_{p_6^{'}}$.

Let $f_{in} : inP_0 \leftrightarrow inP_1$ be $p_1 \mapsto p_2^{'}$ and $p_2 \mapsto p_1^{'}$; let $f_{out} : outP_0 \leftrightarrow outP_1$ be $p_7 \mapsto p_7^{'}$. For each path of N_0, the equivalent path of N_1 is obtained by the following steps:

For the path α_1: The path β_2 is chosen as the only candidate path for equivalence with α_1 because $^\circ\alpha_1 \in inP_0$, $^\circ\beta_2 \in inP_1$ and $\langle ^\circ\alpha_1, ^\circ\beta_2 \rangle \in f_{in}$. As $R_{\alpha_1}(f_{pv}(^\circ\alpha_1)) \equiv R_{\beta_2}(f_{pv}(^\circ\beta_2)) \equiv \top$ and $r_{\alpha_1}(f_{pv}(^\circ\alpha_1)) = r_{\beta_2}(f_{pv}(^\circ\beta_2))$, it is inferred that $\alpha_1 \simeq \beta_2$. For the purpose of equivalence checking, the condition of execution ($R_\alpha(f_{pv}(^\circ\alpha))$) and the data transformation ($r_\alpha(f_{pv}(^\circ\alpha))$) along the path α are maintained in a normalized form; one such normalized form for integers is given in [20]. Initially, the set η_t of corresponding transitions is empty and the set η_p of corresponding places is $\langle ^\circ\alpha_1, ^\circ\beta_2 \rangle$. The set η_t is updated next by putting the pair $\langle t_1, t_2^{'} \rangle$ of last transitions of α_1 and β_2 in it; the set η_p is updated to contain the pair $\{\langle \alpha_1^\circ, \beta_2^\circ \rangle\} = \{\langle p_3, p_4^{'} \rangle\}$ Similarly, it is inferred that $\alpha_2 \simeq \beta_1$ and η_t is updated to $\{\langle t_1, t_2^{'} \rangle, \langle t_2, t_1^{'} \rangle\}$ and $\eta_p = \{\langle \alpha_1^\circ, \beta_2^\circ \rangle, \langle \alpha_2^\circ, \beta_1^\circ \rangle\} = \{\langle p_3, p_4^{'} \rangle, \langle p_4, p_3^{'} \rangle\}$.

For the path α_3: Since $^\circ\alpha_3 \notin inP_0$, a different method is used to select the candidate paths of α_3. First we notice that $^\circ\alpha_3 = \{p_3\}$ which is the post-place t_1° of transition t_1. Next, we look for the corresponding transition of t_1 in the set η_t; $\langle t_1, t_2^{'} \rangle \in \eta_t$; the transition $t_2^{'}$ has one post-place, i.e., $p_4^{'}$. There are two paths β_4 and β_6 such that $^\circ\beta_4$ = $^\circ\beta_6 = \{p_4^{'}\}$. Hence all these two paths are selected as candidates. However, since $R_{\beta_4}(f_{pv}(^\circ\beta_4)) \equiv R_{\alpha_3}(f_{pv}(^\circ\alpha_3))$ and $r_{\beta_4}(f_{pv}(^\circ\beta_4)) = r_{\alpha_3}(f_{pv}(^\circ\alpha_3))$, it is inferred that $\alpha_3 \simeq \beta_4$ and η_t is updated to $\{\langle t_1, t_2^{'} \rangle, \langle t_2, t_1^{'} \rangle, \langle t_3, t_6^{'} \rangle\}$ and η_p is updated to $\{\langle \alpha_1^\circ, \beta_2^\circ \rangle, \langle \alpha_2^\circ, \beta_1^\circ \rangle, \langle \alpha_3^\circ, \beta_4^\circ \rangle\}$ Similarly, it is found that $\alpha_3 \simeq \beta_4$, $\alpha_4 \simeq \beta_3$, $\alpha_5 \simeq \beta_6$, $\alpha_6 \simeq \beta_5$ and $\alpha_7 \simeq \beta_7$.

Since each path of the original behaviour has some equivalent path in the transformed behaviour, and vice-versa, with correspondence among transitions of the respective first and the last sets of transitions of the paths, the models are asserted to be equivalent. It may be noted that the existing control data-flow graph (CDFG) oriented equivalence checking methods [84], [20], [90],[76] fail to establish the equivalence between these two programs involving loop swapping. ■

4.2 Path construction algorithm

The path construction algorithm starts by initializing the marking at hand M_h to the set of in-ports and the set Q of paths to empty. A token tracking execution is carried out from M_h by identifying at each step the enabled transitions $T_e \subseteq M_h^\circ$, removing tokens from $^{\circ}T_e \subseteq M_h$ and placing tokens in T_e°. Thus, a new marking at hand M_h is obtained. If this updated M_h contains a cut-point, then all the places in M_h are marked as (dynamic) cut-points. As explained earlier, if M_h contains a static cut-point reached through a back edge and $|M_h| > 1$, then a flag is set to designate that the token tracking execution has encountered a *degenerate* case whereupon all the places in the subsequent markings are to be designated as cut-points; the flag is reset when the post-transition of M_h is a single transition (designating join of all the parallel threads). Path construction goes hand-in-hand with dynamic cut-point introduction. Construction of a path from a cut-point, p_c say, involves moving backward from p_c through transitions already traversed till a set of places involving only cut-points is reached. Essentially, a backward cone of influence is identified from p_c in the process. The concurrency of transitions, however, cannot be identified by backward traversal. So, the algorithm keeps track of the concurrent transitions encountered during forward execution as a sequence of sets of concurrent transitions, designated as T_{sh}. To start with, T_{sh} is empty. At each step, T_e contains the concurrent transitions enabled for the marking M_h and is appended at the end of T_{sh}. While constructing paths from p_c, the cone of influence of p_c is revealed by moving backward along T_{sh}.

An intricacy arises because the forward token tracking execution does not progress linearly; whenever a place in M_h has more than one post-transition, alternative sets of concurrent transitions (guided by proper guards) result. To keep track of these alternatives, the set of all the bound transitions for M_h is partitioned into subsets of maximally parallelisable transitions having disjoint pre-places. The Cartesian product set of these subsets yields the subsets of enabled concurrent transitions. For each of these subsets, the forward progress of execution is pursued in a DFS manner. The following function modules are involved in path construction. The functional modules are depicted in Algorithms $3-6$ with Algorithm 3 being the top level module. The call graph of the path construction algorithm is given in Figure 4.7. The following example illustrates how the path construction algorithm constructs the paths in a PRES+ model.[2]

Example 11. *In Figure 4.2, the static cut-points are* $p_1, p_2, p_3, p_6, p_{10}$ *and* p_{13}. *When* M_h *is* $\{p_1, p_4, p_5, p_6\}$, p_4 *and* p_5 *are marked as dynamic cut-points as* p_1 *and* p_6 *are cut-points. The function* `obtainAllThePaths` *calls* `constOnePathDCP` *twice*

[2]Note that two of the four function module names contain the string DCP to indicate the context of "dynamic" cut-point based path construction mechanism; this would distinguish them from the static cut-point based path construction mechanism presented in Chapter 6.

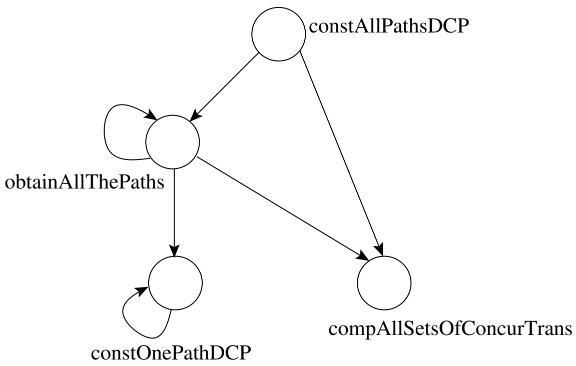

Figure 4.7: Call graph of path construction algorithm

– first with p_4 and next with p_5 as parameters. The function constructs two paths, namely, $\alpha_1 = \langle\{t_1\}\rangle$ and $\alpha_2 = \langle\{t_2\}\rangle$ using backward traversal and the cone of influence method. In the next step of token tracking execution, M_h becomes $\{p_7, p_8\}$ and neither of these is marked as a dynamic cut-point. The next M_h is $\{p_9, p_{10}\}$ and as p_{10} is a cut-point which contains a back edge and $|M_h| = 2$, the degenerate *flag is set to true and p_9 is marked as a dynamic cut-point. The function* `obtainAllThePaths` *calls* `constOnePathDCP` *twice – one with p_9 and then with p_{10} whereupon, the latter constructs two paths, namely, $\alpha_3 = \langle\{t_3\},\{t_5\}\rangle$ and $\alpha_4 = \langle\{t_4\},\{t_6\}\rangle$, using backward traversal and cone of influence method. Then, the function* `obtainAllThePaths` *calls* `compAllSetsOfConcurTrans` *function and returns the set $\mathcal{T} = \{\{t_7,t_8\},\{t_7,t_9\}\}$ of all possible mutually exclusive sets of concurrent transitions. Each of the sets is processed in a DFS manner. For the set $\{t_7,t_8\}$, M_h becomes $\{p_{11}, p_{12}\}$ and as p_{11} is a cut-point, p_{12} is also marked as a dynamic cut-point. As $p_{11}^{\circ} = p_{12}^{\circ} = t_{10}$, at this point the token tracking execution ceases to exist in the degenerate case and the two paths $\alpha_5 = \langle\{t_7\}\rangle$ and $\alpha_7 = \langle\{t_8\}\rangle$ are constructed. Similarly, in the next step, M_h becomes $\{p_{13}\}$ and as p_{13} is a cut-point, by backward traversal and cone of influence method, a path $\alpha_8 = \langle\{t_{10}\}\rangle$ is constructed. For the set $\{t_7,t_9\}$, the function* `obtainAllThePaths` *updates M_h. When $M_h = \{p_{10}, p_{11}\}$; as p_{10} is a cut-point, p_{11} is attempted to be designated as dynamic cut-point. However, p_{11} has already been designated as dynamic cut-point. So, the function* `obtainAllThePaths` *calls* `constOnePathDCP` *only once with p_{10} which constructs the $\alpha_6 = \langle\{t_9\}\rangle$ using backward traversal and cone of influence method. Therefore, the set of dynamic cut-points is computed as $\{p_4, p_5, p_9, p_{11}, p_{12}\}$ and the path set as $\{\langle\{t_1\}\rangle, \langle\{t_2\}\rangle, \langle\{t_3\},\{t_5\}\rangle, \langle\{t_4\},\{t_6\}\rangle, \langle\{t_7\}\rangle, \langle\{t_9\}\rangle, \langle\{t_8\}\rangle, \langle\{t_{10}\}\rangle\}$.* ■

The above algorithm is now analyzed for termination, complexity, soundness and completeness in the following subsections.

4.2.1 Termination of the path construction algorithm

Theorem 3. *`constAllPathsDCP` function (Algorithm 3) always terminates.*

Proof. This result is a consequence of the following lemmas.

Lemma 1. *`constOnePathDCP` function (Algorithm 6) always terminates.*

Proof. The function terminates if it invokes itself (in step 8) a finite number of times. In every invocation, in step 2, T_{sh} is updated by reducing its length. Let $|T_{sh}^{(i)}|$ be the size of T_{sh} at the i^{th} invocation. Then, $|T_{sh}^{(0)}| > |T_{sh}^{(1)}| \ldots |T_{sh}^{(i)}| > |T_{sh}^{(i+1)}| > \ldots$, is a strictly decreasing sequence bounded by zero. In other words, $\{|T_{sh}^{(i)}|, i \geq 0\} \subseteq \mathbb{N}$ and $(\mathbb{N}, <)$ is a well-ordered set [95]. Hence, `constOnePathDCP` invokes itself finitely many times. □

Lemma 2. *`compAllSetsOfConcurTrans` function (Algorithm 4) always terminates.*

Proof. There are two loops; the first one comprising steps 1-3 and the second one comprising step 4. The first one terminates as the size of M_h is finite. The output of this step is a finite collection of sets T_p's each of which is finite. Hence, step 4, which computes the Cartesian product set of these T_p's, also terminates. □

Lemma 3. *The function `obtainAllThePaths` (Algorithm 5) is invoked by itself only a finite number of times.*

Proof. There are three loops in the function `obtainAllThePaths`; the first one comprises step 5; the second one comprises steps 17-25 and the third one comprises steps 31-34; all of them execute finitely many times. The first loop in step 5 is executed $O(|T|)$ times. The second loop in step 17 is executed $O(|P|)$ times, since M_h is $O(|P|)$. Within the loop there is an invocation of the recursive function `constOnePathDCP`. This function always terminates as given in Lemma 1. Step 27 terminates by Lemma

2. Step 32 in the else-clause of step 28 involves as many recursive invocations of the function `obtainAllThePaths` as the members of $\mathcal{T}$. There are only finitely many invocations for each member of $\mathcal{T}$ as explained below. In the first invocation, $|T_{sh}| \leq |T|$ as T_{sh} does not contain any transition more than once. Step 6 expands T_{sh} by appending T_e. Hence, in every recursive invocation, the difference between the transition set T and the set T_{sh} is reduced, i.e., $|T| - |T_{sh}^{(0)}| > |T| - |T_{sh}^{(1)}| > \ldots |T| - |T_{sh}^{(i)}| > \ldots$, where $T_{sh}^{(i)}$ is the value of the parameter T_{sh} at the i^{th} recursive invocation. So, $\{|T| - |T_{sh}^{(i)}|, i \geq 0\} \subseteq \mathbb{N}$ and $(\mathbb{N}, <)$ a well-ordered set [95]. Thus, the loop in step 31 terminates. □

The proof of Theorem 3 can now be accomplished as follows. There is a single loop in step 3. The loop step 3 executes finitely many times because the sets of all concurrent transitions ($\mathcal{T}$) is finite as it is generated by the function `compAllSetsOfConcurTrans` which terminates as given in Lemma 2. Within the loop, the function calls the function `obtainAllThePaths` which always terminates as given in Lemma 3. □

Algorithm 3 SETOFPATHS **constAllPathsDCP** (PRES+ N)

Inputs: A PRES+ model N
Outputs: Set of all paths Q

1: $M_h \Leftarrow inP$; /* Place − marking at hand − initialized to in-ports*/
$Q \Leftarrow \emptyset$; /* set of all paths − initially empty */
$T_{sh} \Leftarrow \langle\rangle$; /* Transition sequence at hand − initially empty */ degenerate = false;
2: $\mathcal{T}$ = **compAllSetsOfConcurTrans** (M_h, N);
/* it takes M_h and forms all possible sets of concurrent transitions that are bound to M_h */
3: $\forall T \in \mathcal{T}$
$Q \Leftarrow Q \bigcup$ **obtainAllThePaths** (T_{sh}, M_h, T, N);
/* The function returns the set of paths corresponding to the set of cut-points in the model N*/
4: **return** Q;

Algorithm 4 CONCURRENTTRSET* **compAllSetsOfConcurTrans** (M_h, N)

Inputs: The first parameter is a marking. The second parameter is the PRES+ model N.
Outputs: The function returns all possible sets of concurrent transitions from M_h°.

1: **for** each $p \in M_h$ **do**
2: $T_p = \{p^{\circ} \mid {}^{\circ}(p^{\circ}) \in M_h\}$; /* a transition of p° is included in T_p only if all its pre-places are marked */
3: **end for**
4: $\mathcal{T} = \times_{p \in M_h} T_p$; /* Cartesian product sets of T_p, $p \in M_h$ and members of $\mathcal{T}$ are generated as ordered tuples but treated as unordered sets */
5: **return** $\mathcal{T}$;

Algorithm 5 SETOFPATHS **obtainAllThePaths** $(T_{sh}, M_h, T_e$, degenerate, $N)$

Inputs: The first parameter is sequence T_{sh} of sets of concurrent transitions. The second parameter is a marking M_h. The third parameter is a set T_e of enabled maximally parallelisable transitions. The fourth parameter *degenerate* is a flag value. The fifth parameter is the PRES+ model N.
Outputs: The function returns the set of paths corresponding to the set of cut-points in the model N.

1: SETOFPATHS $Q = \emptyset$; degenerate = false;
2: **if** $T_e == \emptyset$ **then**
3:    **return** Q;
4: **end if**
5: $\forall t \in T_e$, mark t;
6: $T_{sh} \Leftarrow T_{sh}.T_e$; /* modify T_{sh} by appending T_e */
7: $M_{new} \Leftarrow T_e^{\circ}$; /* post-places of T_e acquire tokens */
8: $M_h \Leftarrow (M_h - {}^{\circ}T_e) \cup M_{new}$; /* modify M_h by deleting the pre-places of the concurrent transitions and adding their post-places */
9: **if** ($|M_h^{\circ}| \leq 1$ and degenerate = true) **then**
10:    degenerate = false;
11: **end if**
12: **if** (there exists a back edge leading to some p in M_h and $|M_h| > 1$) **then**
13:    degenerate = true;
14: **end if**
15: **if** (degenerate = true or at least one p in M_h is a cut-points or $|T_e^{\circ}| > |T_e|$) **then**
16:    mark each place in M_h and all places in T_e° as a dynamic cut-point if it is not already a cut-point;
17:    **for** each $p' \in M_h$ **do**
18:       $\alpha =$ **constOnePathDCP** $(\{p'\}, T_{sh}, N)$; /* Traverse backward from $\{p\}$ along T_{sh} to construct a path up to some cut-points */
19:       $Q \Leftarrow Q \cup \{\alpha\}$; /* Update Q and α is a path to all the out-places of ${}^{\circ}p'$ − so delete the out-places of ${}^{\circ}p'$ to avoid repetition of effort (Steps 14, 15) */
20:       Let $S = \{p'' \mid {}^{\circ}p' = {}^{\circ}p''\}$;
21:       $M_h = M_h - S$;
22:       **if** $(|(p)^{\circ}| = 0) \vee$ (all transitions of $(p')^{\circ}$ are marked) /* first disjunct means p' is an out-port */ **then**
23:          $M_h \Leftarrow M_h - \{p'\}$; /* $(p')^{\circ}$ have already occurred in some path − this step prevents them from appearing in the subsequent set of enabled concurrent transitions */
24:       **end if**
25:    **end for**
26: **end if**
27: $\mathcal{T} =$ **compAllSetsOfConcurTrans** (M_h, N);
/* unmark all the marked transitions */
28: **if** $(\mathcal{T} = \emptyset)$ and $(M_h \neq \emptyset)$ **then**
29:    Report as invalid PRES+ Model
30: **else**
31:    **for** each $T_e \in \mathcal{T}$ **do**
32:       $Q \Leftarrow Q \cup$ **obtainAllThePaths** $(T_{sh}, M_h, T_e$, degenerate, $N)$ //call itself recursively;
33:       **return** Q;
34:    **end for**
35: **end if**

Algorithm 6 PATH **constOnePathDCP** (P, T_{sh}, N)

Inputs: The first parameter is a set P of places. The second parameter is a sequence T_{sh} of sets of concurrent transitions. The third parameter is the PRES+ model N.
Outputs: The function returns a path α.

```
1: T = last(T_sh) ∩ °P;
   /*T is earmarked. The remaining ones in last(T_sh), if any, do not fall in the cone of influence of P */
2: T'_sh = T_sh − last(T_sh); /* Ignore last(T_sh) altogether in further backward traversal */
3: P' = (P − T°) ∪ °T;
4: P' = P' − P_c, where P_c is the set of all cut-points;
   /* proceed backward only from the places which are not cut-points */
5: if P' = ∅ then
6:    return (PATH) ⟨T⟩;
7: else
8:    return append(constOnePathSCP(P', T'_sh, N), T);
      /* append T at the end of the sequence obtained by continuing backward */
9: end if
```

4.2.2 Complexity analysis of the path construction algorithm

In this subsection, we discuss the complexities of the modules used by the path construction algorithm in a bottom up manner. We show that the complexity of the overall algorithm is $O\big(\big(\frac{|T|}{|P|}\big)^{|P|}\,|T|^2\big)$ which reduces further to $O\big(|T|^2\big)$.

Complexity of Algorithm 6 `constOnePathDCP`: The set intersection operation in step 1 involves searching for each member of $^\circ P$ in last(T_{sh}). We keep the pre-transitions of all the places and all the members of T_{sh}, and hence last(T_{sh}), sorted in the indices of the model transitions such that binary search can be used for each member of $^\circ P$. Hence, it takes $O(|T|\log|T|)$ time. In step 2, T_{sh} is updated by deletion of the last member of T_{sh} from T_{sh}. As we use a stack for T_{sh}, the pop operation achieves this deletion step in $O(1)$ time. In step 3, the set difference and union operations (between sets of places) take place. These operations are done by binary search technique of one operand for each member of the other operand; hence it takes $O(|P|\log|P|)$ time. In step 4, the set difference operation takes $O(|P|\log|P|)$ time. Step 5 checks whether the set of places is an empty set or not in $O(1)$ time. If the set of places is empty, then the function returns $\langle T\rangle$ in $O(1)$ time. If the condition is not true, then the recursive invocation takes place at step 8 and it takes $O(|T|)$ time. Since $|T_{sh}| = |T|$ in the worst case because each transition can occur at most once, the overall complexity of this function is $O(\max(|T|\log|T|, |P|\log|P|)\,|T|)$.

Complexity of Algorithm 4 `compAllSetsOfConcurTrans`: Step 1 produces $O(|P|)$ number of T_p's, as the set M_h has size $O(|P|)$. Each T_p has $O\left(\frac{|T|}{|P|}\right)$ transitions because T_p's are disjoint. In step 4, the product set of T_p's is computed which takes $O\left(\left(\frac{|T|}{|P|}\right)^{|P|}\right)$ time.

Complexity of Algorithm 5 `obtainAllThePaths`: Step 1 initializes the set Q of paths to empty set; hence the complexity is $O(1)$. Steps 2-4 take $O(1)$ time to check whether T_e is empty. Step 5 marks all the transitions in T_e in $O(|T|)$ time. In step 6, T_{sh} is updated by appending the set T_e of enabled transitions at its end. As T_{sh} is maintained as a stack, the appending (push) operation has the complexity $O(1)$. In step 7, it obtains the post-places of the enabled transitions T_e; for each transition t, t° takes $O(1)$ and there can be $O(|T|)$ transitions in T_e; hence the complexity is $O(|T|.|P|)$. In step 8, the computation of $^\circ T_e$ takes $O(|P|)$ time, computation of $M_h - ^\circ T_e$ takes $O(|P|\log|P|)$ time, computation of union operation also takes $O(|P|\log|P|)$ time as M_h is in sorted order and the union and set difference operations are done by binary search. Steps 9 and 10 take $O(1)$ time. Step 12 takes $O(|P|)$ time. In step 18, the function calls `constOnePathDCP` routine which returns a path as a sequence of sets of transitions as output in $O((\max(|T|\log|T|, |P|\log|P|)\,|T|))$ time as explained previously. In step 19, it adds the new path α returned by `constOnePathDCP` to the set Q of all paths by union operation; since Q is maintained as an unordered set and it is ensured through steps 20 and 21 that no path is generated more than once, step 19 takes $O(|P|)$ time. Step 20 takes $O(|P|)$ time. Step 21 takes $O\left(|P|^2\right)$ because for each member of S, where $|S| = O(|P|)$, all the members of M_{new} (of size $O(|P|)$) have to be examined. Step 22 detects the condition $|(p')^\circ| = 0$ in $O(1)$ time; detecting whether all the transitions of $(p')^\circ$ are marked or not, can be found in $O(|T|)$ time since $|(p')^\circ|$ is $O(|T|)$. The operation $M_{new} = M_{new} - \{p'\}$ takes $O(1)$ time because p' is a distinguished member of M_h in each iteration of the loop involved in step 17. So the time complexity of the body (steps $18-24$) of the loop comprising steps $17-25$ is the maximum of the time complexities of steps 18 to 24, i.e., $O\left(|P|^2\right)$ time; the loop comprising steps 17-25 iterates $O(|P|)$ time; hence, the overall complexity is $O\left(|P|^3\right)$ time. Therefore, the total complexity of the then block of step 15 is $O\left(|P|^3\right)$ time. Step 27 computes the set of all concurrent transitions, which takes $O\left(\left(\frac{|T|}{|P|}\right)^{|P|}\right)$ as explained previously. In the same step, the function unmarks all the marked transitions and it takes $O(|T|)$ time. Step 28 checks whether the computed set of concurrent transitions is empty and M_h is non-empty; it takes $O(1)$ time. If the condition is true, the algorithm reports that the given model is an invalid PRES+ model. If the condition is false, the function calls `obtainAllThePaths` recursively $O(|T|)$ times as the loop in this step iterates $O(|T|)$ time; hence the overall complexity of this step is $O\left(\left(\left(\frac{|T|}{|P|}\right)^{|P|}\right)|T|\right)$.

Complexity of Algorithm 3 `constAllPathsDCP`: Step 1 initializes M_h, Q and T_{sh}. For creation of M_h, the function takes $O(|P|)$ time and for initialization of the other two entities, the function takes $O(1)$ time. Hence, the overall time complexity of this step is $O(|P|)$. In step 2, the function calls `compAllSetsOfConcurTrans` which takes $O\left(\left(\frac{|T|}{|P|}\right)^{|P|}\right)$ time as indicated above. In the same step, the function unmarks all the marked transitions and it takes $O(|T|)$ time. In step 3, for all transitions which are in some concurrent set of transitions obtained in step 2, the function updates Q by calling `obtainAllThePaths` whose complexity is $O\left(\left(\frac{|T|}{|P|}\right)^{|P|}|T|\right)$. The forward progress takes $O(|T|)$ time.

However, at each step, `compAllSetsOfConcurTrans` needs to be invoked which results in this figure. Since the loop of step 3 iterates $O(|T|)$ time, the overall complexity is $O\left(\left(\frac{|T|}{|P|}\right)^{|P|}|T|^2\right)$. Therefore, the overall complexity of the path construction algorithm is $O\left(\left(\frac{|T|}{|P|}\right)^{|P|}|T|^2\right)$. In a computation, the number of definitions for a variable is less than the number of its use. Hence the number $|T|$ of transitions is less than the number $|P|$ of places, that is, $\frac{|T|}{|P|} < 1$. (An ill-written program can violate this property by having definitions which are never used. During model construction, however, they will result in places which are not out-ports but have no post transitions; such places can be removed.) So, the complexity figure $O\left(\left(\frac{|T|}{|P|}\right)^{|P|}|T|^2\right)$ is $O\left(|T|^2\right)$.

4.2.3 Soundness of the path construction algorithm

Theorem 4. *Any member of the set Q returned by the function* `constAllPathsDCP` *satisfies the properties of the paths (as given in Definition 13).*

Proof. Let there be a path $\alpha = \langle T_1, T_2, \ldots, T_n \rangle$ in the set Q returned by the function `constAllPathsDCP` which does not satisfy all the properties of a path as listed in the definition (Definition 13) of paths. (The fact that any member of Q has such a form (as that of α) is obvious from step 18 of `obtainAllThePaths` function and steps 1, 6 and 8 of the function `constOnePathDCP` which ensure that the path α obtained comprises only a sequence of sets of parallel transitions.) Depending on which property it violates, we have the following cases:

Case 1: *There exists some member T_i, $1 \leq i \leq n$, such that T_i is not parallelisable.* The function `constOnePathDCP` appends T_i in step 8 or step 6 in each invocation. Each T_i is ensured to be a subset of some member T, say, of T_{sh} which is ensured through step 1 of the function `constOnePathDCP`. So, the member T of T_{sh} is not parallelisable. Now, the member T has been appended at the end of T_{sh} in step 6 of the function `obtainAllThePaths`. Each T is ensured to be a member of $\mathcal{T}$ by step 31 of the previous invocation of the function `obtainAllThePaths`. But all members of $\mathcal{T}$ are ensured to be parallelisable by the function `compAllSetsOfConcurTrans` invoked in step 27. Hence, T must be parallelisable. So, T_i must be parallelisable.

Case 2: *None of the members in $^\circ T_1$ is a cut-points.* The path α has been constructed through n invocations of the function `constOnePathDCP`; the first $n-1$ invocations have put the transition sets $T_n, T_{n-1}, \ldots, T_2$ in step 8; the n^{th} invocation returns a path comprising a sequence $\langle T_1 \rangle$ of length 1 in step 6. So, in this invocation, P' is found to be empty in step 5, i.e., after step 4. After step 3 of the k^{th} invocation, P' contains all the pre-places of $^\circ T_1$: Prior to step 4 P' must have been P_c. Therefore, $^\circ T_1 = P_c$.

Case 3: *There exists at least one place in T_n° which is not a cut-point.* The transition in T_n being the last transition in the path, it must have been appended at the end by the function `constOnePathDCP` in step 8 of its first invocation. It has been computed in step 1 of this invocation. Let P_1, T_{sh_1} be the first two arguments with which this invocation has taken place. So from step 1, $T_n = last(T_{sh_1}) \cap {}^\circ P_1$ which implies $T_n^\circ \subseteq P_1$. Now, the first invocation of `constOnePathDCP` takes place from step 18 of the function `obtainAllThePaths` with $P_1 = \{p'\}$ where p' is (made) a cut-point in step 16. So T_n° is a cut-point.

Case 4: *For some $m, 1 \leq m < n$, there exists at least one member T_m, say, in α such that T_m° contains a cut-point.* In this case, step 16 in `obtainAllThePaths` function would ensure that all the places in T_m° are cut-points. Step 18 should have invoked `constOnePathDCP` for each member of T_m° resulting in $|T_m^\circ|$ paths of the form $\langle T_1, T_2, \ldots, T_{m,i}\rangle$, $1 \leq i \leq |T_m^\circ|$, where $T_{m,i}$ is a transition of T_n. So the algorithm could not have returned α with T_m as its intermediate member.

Case 5: *The condition $\forall i,\ 1 < i \leq n,\ \forall p \in {}^\circ T_i$, if p is not a cut-point, then $\exists l,\ 1 \leq l \leq i-1,\ p \in T_{i-l}^\circ$ does not hold, i.e., there exists a set of concurrent transitions T_i in the path which has at least one pre-place p which is not a cut-point but is not included as a post place of any of the preceding sets T_1 through T_{i-1}.* Let T_i be the last such transition in the path with such a pre-place p. Now `constOnePathDCP` is invoked first time from step 18 of `obtainAllThePaths` with the first parameter $P = \{p_c\}$, i.e., P containing a single cut-point. There is a recursive invocation of `constructOnepath` subsequently when T_i has been earmarked for inclusion in the path with the first parameter P' containing $p \in {}^\circ T_i$ (due to the union terms in step 3). All the subsequent invocations of `constOnePathDCP` will have p in P' because of the following reasons.

1. They are not cut-points (due to step 4), and
2. They are not in T° (due to step 3), where $T =$last$(T_{sh}) \cap {}^\circ P$.

Since p satisfies both (1) and (2), all the recursive invocations have P' containing p (computed in step 8). Since, as per the premise, $p \notin T_1^\circ \cup T_2^\circ \cup \ldots \cup T_{i-1}^\circ$, the process would have gone (backward) beyond T_1 and α could not have T_1 as its first member.

Case 6: *There exist two transitions $t_i \in T_i$ and $t_j \in T_j$, $1 \leq i \neq j \leq n$, in α such that ${}^\circ t_i \cap {}^\circ t_j \neq \emptyset$.* Let $p \in {}^\circ t_i \cap {}^\circ t_j$. Hence, the function `obtainAllThePaths` constructs M_h containing p, in step 8 in some invocation. The `compAllSetsOfConcurTrans` is invoked from step 27 of this function with this M_h and it returns a set $\mathcal{T}$ of sets of concurrent transitions containing two distinct sets T_i and T_j such that $t_i \in T_i - T_j$ and $t_j \in T_j - T_i$. The steps 31 and 32 of the function `obtainAllThePaths` will proceed in a depth-first manner first with one of T_i and T_j till a path is constructed with T_i (or T_j) as one of the alternatives. Therefore, two paths are constructed one containing T_i and other T_j. Hence, t_i and t_j cannot occur in the same path α.

Case 7: *$\exists i, 1 \leq i \leq n, T_i$ is not maximally parallelisable within the path α.* Let there exist $T' \supset T_i$ such that T' is parallelisable. T_i has been put in α in the $(n-i+1)^{th}$ invocation of the function `constOnePathDCP`. In step 1 of this invocation, T_i has been defined as $last(T_{sh}) \cap {}^\circ P$. Since $T' \supset T_i, T' \nsubseteq last(T_{sh})$ or $\nsubseteq {}^\circ P$. If

$T' \not\subseteq last(T_{sh})$, then T' is not a parallelisable set (because the set T_{sh} is constructed by token tracking execution which ensures that T_{sh} has only maximally parallelisable transitions). If $T' \not\subseteq {}^\circ P$, then T' is not maximally parallelisable *within the path* α because it is not within the *cone of foci (influence)* from T_n°.

Case 8: *No computation (of any out-port) has a sub-sequence of markings of places* $\langle P_{M_i}, P_{M_{i+1}}, \ldots P_{M_{i+n}} \rangle$ *such that all clauses (vii)(a)–(vii)(c) are satisfied.* It may be noted that the i^{th} recursive invocation of `constOnePathDCP` which puts T_{n-i-1}, $1 \le i \le n$, into the path, satisfies through step 1 that $T_{n-i-1} \in$last(T_{sh_i}), where T_{sh_i} is the value of T_{sh} with which i^{th} invocation takes place. T_{sh} is constructed in step 6 of the function `obtainAllThePaths` by adding T_e where ${}^\circ T_e$ is a marking obtained by the token tracking execution (steps 7-16 and 27-33) which, in turn, ensures that these markings are obtained as successor markings of a computation. Hence there indeed exists a computation which has the desired subsequence depicted in clause (vii).

Case 9: $\exists i, 1 \le i < n$ *such that* $|T_i^\circ| > |T_i|$. In this case, the step 16 of the function `obtainAllThePaths` ensures that T_i° are all cut-points. Therefore, the path could not have contained the sets T_{i+1} to T_n.

□

4.2.4 Completeness of the path construction algorithm

Theorem 5. *The set of paths returned by the function* `constAllPathsDCP` *is a path cover of the model.*

Proof. Let μ_p be a computation of some out-port p, which is not covered by the set of paths returned by the path construction algorithm. From Theorem 2, μ_p must be of the form $\langle T_1, \ldots, T_{k_1}, T_{k_1+1}, \ldots, T_{k_2}, \ldots, T_{k_r}, \ldots, T_{l-1}, T_l \rangle$ such that all the places in $T_{k_i}^\circ$, $1 \le i \le r$, are cut-points and no other intermediary marking has any cut-point. Proof of Theorem 2 also established that each of the sub-sequences $\mu_{s_1} = \langle T_1, \ldots, T_{k_1} \rangle, \ldots, \mu_{s_{r+1}} = \langle T_{k_r}, \ldots, T_l \rangle$ results in a a set of parallelisable paths by definition of paths so that μ_p can be represented as a concatenation $((\alpha_{1,1} \parallel \alpha_{1,2} \parallel \ldots \parallel \alpha_{1,k_1'}).(\alpha_{2,1} \parallel \alpha_{2,2} \parallel \ldots \parallel \alpha_{2,k_2'}). \ \ldots \parallel \alpha_{r+1,1})$ of parallelisable paths, where $k_1' = |T_{k_1}|$,

$k_2^{'} = |T_{k_2}|$ and so on. Let $\alpha_{j,i}$ for some i , $1 \leq i \leq k_j^{'}$, which is a path in the jth group in the above concatenation, be not constructed by the *path construction* algorithm. As $\alpha_{j,i}^{\circ} \in T_{k_j}^{\circ}$ are cut-points, step 18 of `obtainAllThePaths` function must call `constOnePathDCP` with $P = \{p\}$, where $p \in \alpha_{j,i}^{\circ}$ and the function returns the path $\alpha_{j,i}$. Hence there does not exist any path of μ_p which is not constructed by the *path construction* algorithm. □

4.3 Experimental Results

The algorithm is implemented in *C* and tested on both sequential and parallel examples on a 2.0 GHz Intel(R) Core(TM)2 Duo CPU machine (using only a single core). We have carried out the experimentation along two courses. The first one has used hand constructed models because initially, we did not have any automated model constructor at our disposal; this set up has been depicted in Figure 4.8. The second course of experimentation has been carried out using an automated model constructor which has been completed subsequently (and described in [122]); this flow has been depicted in Figure 4.9. Note that in this second course, the inputs have been taken as FSMD models rather than the *C* codes of the programs. The automated model constructor had to be enhanced with a pre-processing utility to construct *C* code corresponding to an FSMD model; the process has also been described in [122]. This has been done for providing a common set of inputs for comparing the performance of the two PRES+ model based equivalence checking methods described in this dissertation with the FSMD based equivalence checking reported in [20]. The same set of examples are used for testing the path construction modules and the equivalence checking modules of the two PRES+ model based equivalence checking approaches. In the present chapter, we describe only our experimentation with the path constructor module of the first approach where we primarily observe the number of paths and the time taken to construct them. In Chapter 5, the experimentation with the corresponding equivalence checking module is presented. In the following subsection, we discuss these two courses of experimentation in detail.

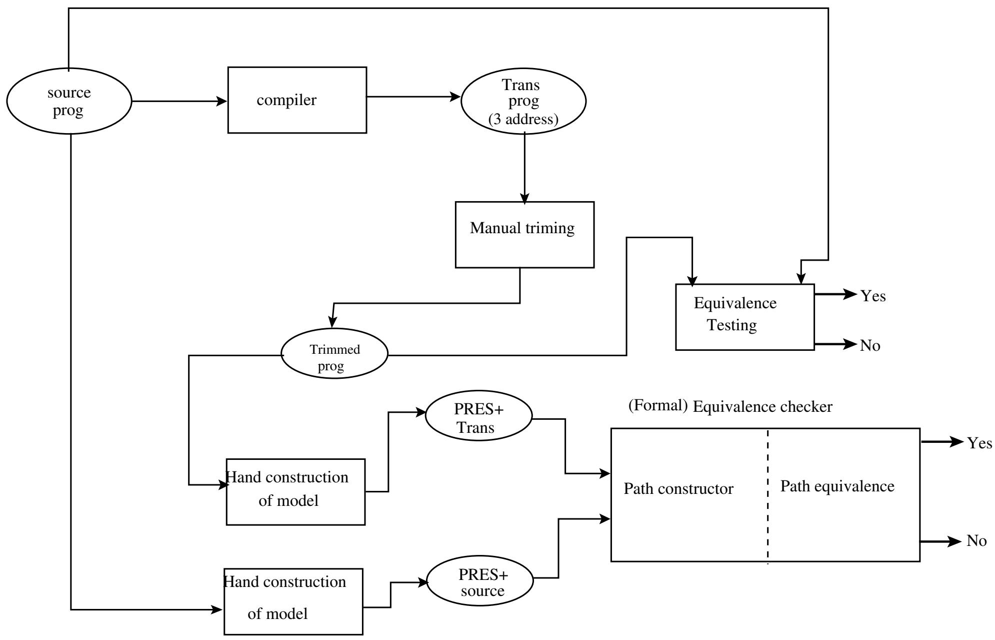

Figure 4.8: Experimentation using hand constructed models

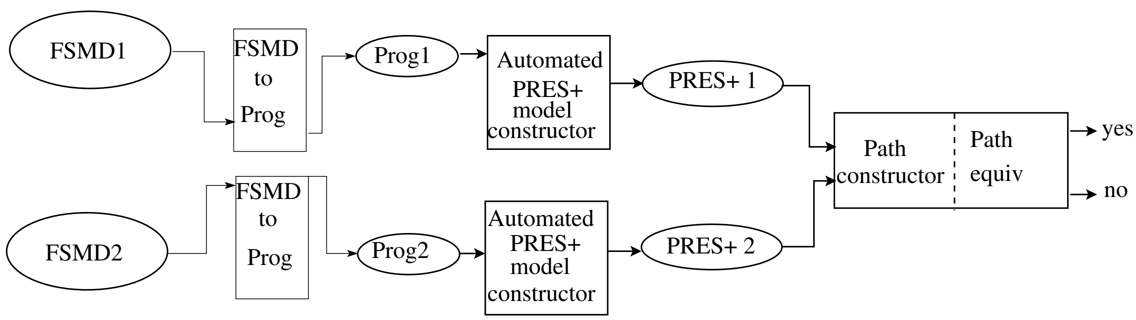

Figure 4.9: Experimentation using automated model constructor

4.3.1 Experimentation using hand constructed models

The steps for carrying out the experimentation are described stepwise, first for sequential and then for parallel transformations.

1. **Preparation of the example suite:** The list of source programs used for the experimentation and their functionalities are as follows:

   (a) *MODN:* Calculates $(a * b)$ modulo n, where $a, b < n$.

   (b) *SUMOFDIGITS:* Carries out repetitive summation of digits of the input number and of the number obtained in each iteration until the sum becomes a single digit; e.g., for the input number 12345, the iterations are to yield

Example	Transformations
MODN	Uniform and non-uniform code motion, code motion across loop (human guided)
SUMOFDIGITS	Dynamic loop scheduling (DLS) (human guided)
PERFECT	DLS, uniform and non-uniform code motion, code motion across loop
PRIMEFACS	DLS, Uniform and non-uniform code motion (SPARK), code motion across loop
GCD	Uniform and non-uniform code motion (SPARK), code motion across loop
TLC	Uniform and non-uniform code motion (SPARK), code motion across loop
DCT	Uniform and non-uniform code motion (SPARK), code motion across loop
LRU	Uniform and non-uniform code motion (SPARK), code motion across loop
LCM	Uniform and non-uniform code motion (SPARK), code motion across loop
MINANDMAX-S	Loop swapping (human guided)

Table 4.1: Experimentation with sequential transformations.

the sequence $12345 \rightarrow 15 \rightarrow 6$. A recursive definition is as follows:

$$\begin{aligned} sodTill1dig(n) &= n, \ \ if \ \ n < 10, \\ &= sodTill1dig(sod(n)), if \ \ n \geq 10 \\ where, \ \ sod(n) &= n, \ \ if \ \ \ n < 10 \\ &= sod(n/10) + n\%10, \ \ if \ \ n \geq 10. \end{aligned}$$

(c) *PERFECT:* Checks whether the input number is perfect or not.

(d) *GCD:* Calculates the GCD of two input numbers.

(e) *TLC:* Traffic light controller for a highway - farmroad crossing.

(f) *DCT:* Computes the four point discrete cosine transform.

(g) *LCM:* Calculates the LCM of two input numbers.

(h) *LRU:* Identifies the least-recently used item in a cache.

(i) *PRIMEFACS:* Sum of all the prime factors of the input number.

(j) *MINANDMAX-S:* Computes sum of the maximum of four numbers n_1, n_2, n_3, n_4 and the minimum of the four numbers n_1, n_5, n_6 and n_7 (having n_1 as the common element). This functionality has been chosen so that the corresponding PRES+ models (for both source and transformed programs) force degenerate phases to be encountered during path construction.

It is to be noted that some of these examples, such as GCD and TLC, are control intensive; some are data intensive, such as DCT, whereas some are both control and data intensive, such as LRU.

2. **Transforming the programs:** Each of the above sequential programs is then transformed using some human guided transformations or by the SPARK compiler. In Table 4.1, depicts the transformations that are applied for each of the above ten examples. It is to be noted that for testing our path construction module, we, therefore, have ten pairs of source and transformed programs.

3. **Trimming of the transformed program:** Since for both source and transformed programs the models have been constructed manually, the sizes of the programs are of importance. The transformed programs had many redundant temporary variables (due to three address code produced by the compiler); so the transformed programs are trimmed manually by removing such temporary variables. To alleviate human errors, the source *C* code and the manually trimmed version of the transformed code of each of the ten example problems are compiled using *GCC* and run on some test cases (as shown in Figure 4.8).

4. **Manual construction of models:** From both source program and the trimmed version of the transformed program, we construct two PRES+ models manually using human ingenuity extensively. To guard against human errors, each of these models is tested using the CPN simulator [70].

5. **Path Construction:** Finally, we feed these two PRES+ models as inputs to our path constructor module which is the front end of the equivalence checker.

6. **Reporting of results:** For each test case, we observe the numbers of static cut-points (SCPs) and dynamic cut-points (DCPs), the number of times the degenerate phase is encountered and the number of paths produced. For small examples like MODN, GCD, SUMOFDIGITS, PERFECT, DCT, PRIMEFACS and LCM, the paths produced are manually checked for correctness. The source and transformed codes for all examples and their corresponding PRES+ models which are depicted in Table 4.2 are given in AppendixA.

Example 12. *In this example, we describe the above experimental steps using the example* `MODN`*. The source and the trimmed versions of the transformed C code for the example is given in Figures 4.10 and 4.12. The source C program is transformed* *by the SPARK compiler; the output of the SPARK compiler is given in Figure A.1 in the appendix. Using a human guided trimming procedure, as indicated in Figure A.1, we have got the trimmed version of the transformed program which is given in Figure 4.12. The source and trimmed programs are then compiled using the* `GCC` *compiler and tested on some test inputs. If the test input is $n = 7, a = 5, b = 6$, then the outputs of both the programs are 2. The PRES+ models constructed manually for these two programs are given in Figures 4.11 and 4.13. The models are then validated using CPN simulator for the same data set. Then we have fed these two examples one by one as inputs to our path constructor module. For each of them, our path construction module gives the list of static and dynamic cut-points, indicates entries to and exit from the degenerate phase and then finally prints the set of paths. Our tool also gives the path construction time. For calculating time, we have used* `get_cpu_time()`. *A typical program output for the source* `MODN` *is given in Figure 4.14.* ■

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

Figure 4.10: Source program of MODN

Table 4.2 depicts the sizes of the original and the transformed PRES+ models in terms of numbers of their places, transitions (trans), static (SCP) and dynamic cut-points (DCP), degenerate cases (DC) and paths. Last two columns depict the path construction time for both original and transformed PRES+ models.

Experimentation with parallelizing transformations

In this step, we have transformed five sequential programs into parallel programs using two prominent thread level parallelizing compilers PLuTo [24] and Par4All [2]. The experimental set up is as follows:

1. **Preparation of the example suite:** We have taken five sequential source programs. The list of the source programs and their functionalities are as follows:

   (a) *BCM [82]:* A toy example on basic code motion without writable shared variables which illustrates computational vs. executional optimality.

   (b) *MINANDMAX-P:* The same MINANDMAX-S program used in the sequential example suites.

   (c) *LUP:* It computes "LU Decomposition with Pivoting". In this experimentation, we have only taken the pivoting routine which does not contain any array. The detailed functionality of this source program is given in PLuTo example suite [14, 24].

   (d) *DEKKER's* and *PATTERSON's algorithms:* Implementations of the classical solutions to the mutual exclusion problem of two concurrent processes. Since our mechanism does not handle writable shared variables among parallel threads, we have considered a single process in each of these cases; also we have introduced a series of dummy assignment statements within the critical section which was otherwise left unspecified in the code. Unlike the previous cases, for these two examples, the corresponding PRES+ models have been taken directly from [9, 119].

2. **Transforming the programs:** The above five sequential programs are transformed by two prominent thread level parallelizing compilers, PLuTo [24] and Par4All; the transformed versions accordingly have parallel structures. Table 4.3 depicts the type of transformations applied for each of the above examples. It is to be noted that for testing our path construction module, (in the context of parallelizing transformations) we have five sequential programs and two sets of five parallel programs – one obtained using PLuTo compiler and the other using Par4All compiler. Before submitting the sequential programs to these compilers for parallelization, the scope in the source program is designated manually using `pragma scop` such that the compiler transforms only that particular portion of the code.

3. **Selecting portion of the source and the transformed programs:** To contain the size of the hand constructed models, we have taken only those portions of the codes which are transformed by the compilers. The variables which are only used within the scope are the in-ports of the PRES+ model.

4. We construct two PRES+ models by hand–one from the original code snippet(s) present in `pragma` scope of the original program and the other from the code snippet(s) present in `CLooG scop` of the transformed program. All the above parallel examples do not contain any writable shared variables. To guard against human errors, each of these models is checked for validity using the CPN tool [70]. Finally, we feed these two PRES+ models as (two independent) inputs to our path constructor module which is the front end of the equivalence checker and observe the same set of parameters as described for Table 4.2.

Example	Original PRES+						Transformed PRES+						Time (μs)	
	Place	Trans	SCP	DCP	DC	Paths	Place	Trans	SCP	DCP	DC	Path	Original	Transf
MODN	28	21	17	8	1	17	27	20	16	10	1	17	5532	4834
SUMOFDIGITS	11	9	8	2	1	9	10	9	6	4	1	9	1051	1168
PERFECT	19	14	12	6	2	13	14	10	11	3	2	9	2929	1679
GCD	31	27	14	14	1	16	19	17	12	5	1	15	6561	3240
TLC	30	28	16	12	0	23	40	39	16	14	0	23	7355	8532
DCT	25	18	6	0	0	1	20	13	6	0	0	1	796	785
LCM	34	28	14	14	1	16	22	18	12	5	1	15	6693	3825
LRU	39	37	18	16	2	18	45	42	20	17	2	18	6345	6783
PRIMEFACS	12	10	7	5	1	10	12	10	8	4	1	10	1065	1217
MINANDMAX-S	28	21	11	16	1	21	28	21	11	16	1	21	6234	6225

Table 4.2: DCP induced path construction times for hand constructed models of sequential examples

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

Table 4.3: Transformations carried out using parallelizing compilers

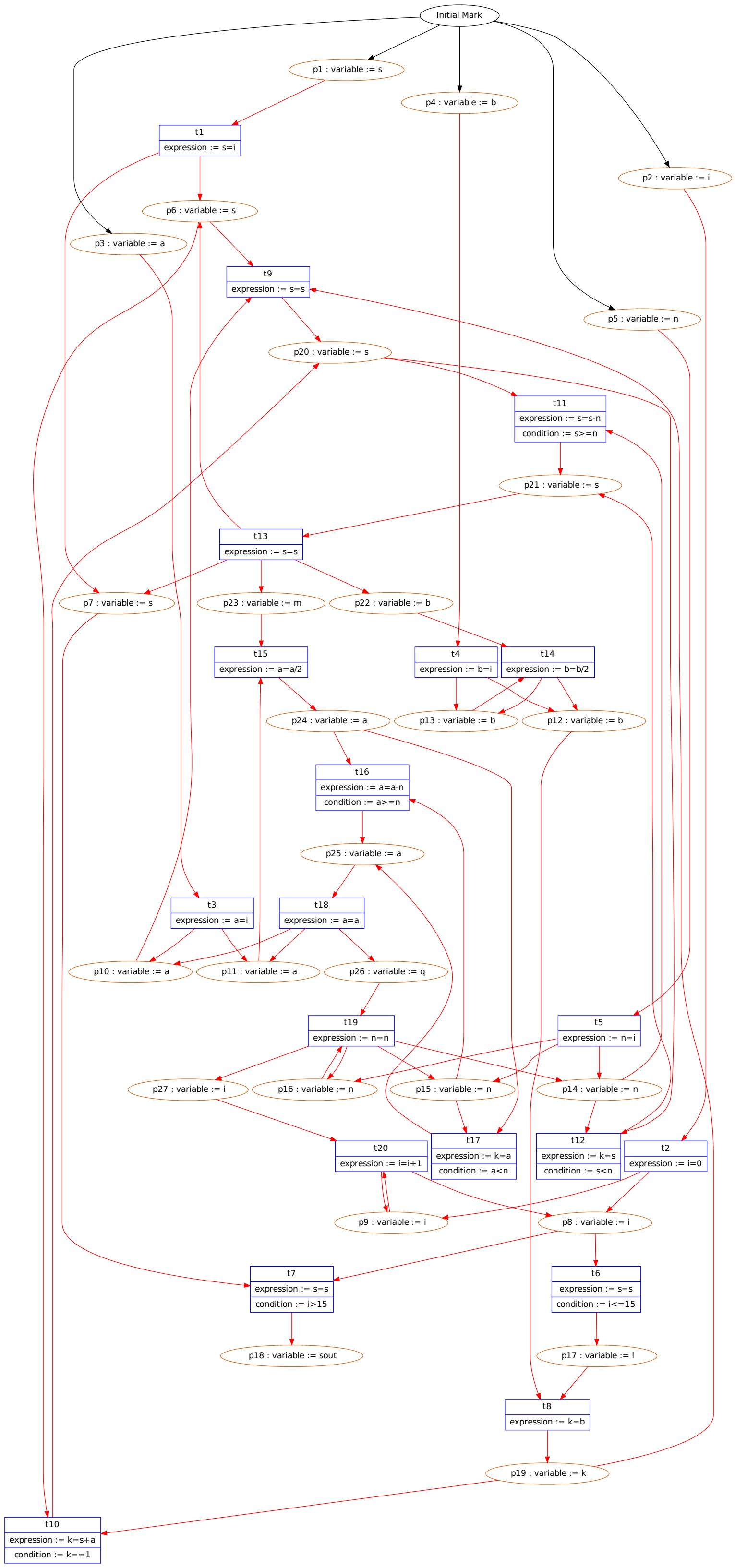

Figure 4.11: PRES+ models corresponding to MODN source program

```
/* Trimmed Version of MOD N */
#include <stdio.h>
int main(void) {
    int s, i, n = 6, b = 6, sout, a = 120, k, l, t;
    /* int  sT0_6, sT1_8 (=k), sT2_8,
       sT3_10,sT4_14,sT5_12 (=l),sT6_15 (=t);
       /* some temporary variables trimmed out -- some renamed */
// int returnVar_main; /* trimmed out */
    s = 0;
    i = 0;
    // returnVar_main = 0; /* trimmed out */
    do {
        if (i <= 15) {/* originally sT0_6= i <=15; if (sT0_6) */
           i = (i + 1);
           k = (b % 2);/* retain (renamed) Variable */
           l = (a * 2);/* retain (renamed) variable */
           //sT2_8 = (sT1_8 == 1); /* = (k==1) */
           //sT4_14 = (l >= n); /* originally sT5_12 >= n */
            b = (b / 2);
           if (k == 1) {/* originally sT2_8 == 1 */
                s = (s + a);
                t = (l - n);
                a = l;
            } else {
               t = (l - n);
               a = l;
            }
            // sT3_10 = (s >= n); /* Trimmed out this statement */
            if (s >= n) {/* originally sT3_10 = (s >= n); if (sT3_10) */
                s = (s - n);
            }
            if (l >= n) {
               a = t;
            }
        } /* end of loop condition */
       else
           break;
    } while (1);
    sout = s;
    printf("%d \n", sout);
    return 0;
}
```

Figure 4.12: Trimmed version of MODN

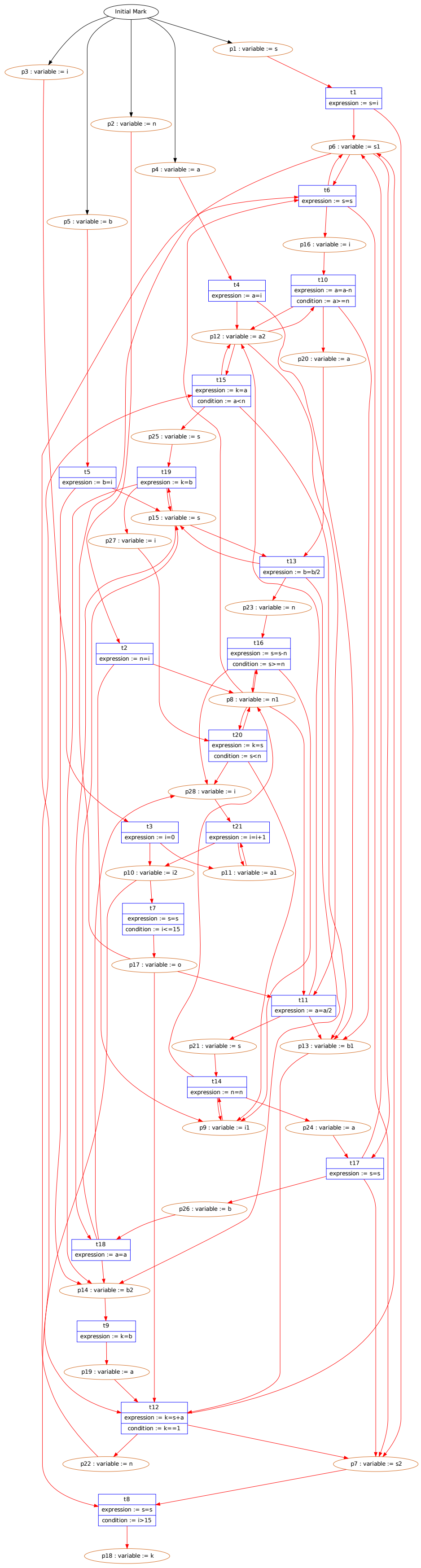

Figure 4.13: PRES+ models corresponding to MODN trimmed transformed programs

```
********************** Finding all paths of model N0 *****************************
Finding Cut-points type=0: Out-ports type=1 : In-ports, type=2: Backedge
**********************************************************************************
The cutpoint list is:-
p1(type=1) p2(type=1) p3(type=1) p4(type=1) p5(type=1) p6(type=2)
p7(type=2) p8(type=2) p9(type=2) p10(type=2) p11(type=2) p12(type=2)
p13(type=2) p14(type=2) p15(type=2) p18(type=0) p22(type=2)
**********************************************************************************
path 0: < { t1  } > path 1: <{ t2  }  > path 2 : <  { t3  }  > path 3 : <{ t4  } >
path 4 : <  { t5  }  >
**********************************************************************************
p10 (type=2) Degenerate case start...
**********************************************************************************
p16 is Dynamic cut point  p21 is Dynamic cut point p22 is Dynamic cut point
**********************************************************************************
path 5 : <  { t7  }  > path 6 : <  { t8  }  > path 7 : <  { t9  }  >
path 8 : <  { t11  }  > path 9 : <  { t12 }  >
**********************************************************************************
Degenerate case ends...
**********************************************************************************
p20 is Dynamic cut point p24 is Dynamic cut point p25 is Dynamic cut point
**********************************************************************************
path 10 : <  { t10 ,t15  }  > path 11 : <  { t14  }  >
**********************************************************************************
p26 is Dynamic cut point p27 is Dynamic cut point
**********************************************************************************
path 12 : <  { t17  }  > path 13 : <  { t13 ,t19  }  >
path 14 : <  { t16 ,t20  }  > path 15 : <  { t18  }  > path 16 : <  { t21  }  >
**********************************************************************************
###################### Path construction time #####################################
 No. of places in N0: 28  No. of transitions in N0: 21  DC in N0: 1
 No. of paths in initial path cover of N0: 17  Exec time is 0 sec and 5532 microsecs
####################################################################################
```

Figure 4.14: Output of DCP induced path construction module

In the following example, we show our experimentation procedure using a parallel example.

Example 13. *We describe the above experimentation steps using the* `MINANDMAX-P` *which computes the sum of the minimum among the set of four numbers and maximum among the set of four numbers where the two sets contain a common element. Figure 4.15(a) depicts the original C code which is transformed by PLuTo and Par4All compilers yielding the same output as shown in Figure 4.15(b). It may be noted that after the common element is read, the two loops can proceed independent of each other; so there is no shared variables–not even read-only ones–for this example. accordingly, they are put manually under "♯pragma scop" – "♯ pragma endscop" construct so that when fed as input, the compilers create parallel threads as given in Figure 4.15(b). Specifically, the parallel threads appear within the construct "♯ CLooG code" – "♯ CLooG code" with the "♯ Par" construct depicting the parallel thread boundaries (depicted in PLuTo version 0.2.0 version). Figure 4.16 represents schematically the whole PRES+ model corresponding to the both source code and and the transformed program. As the example* `MINANDMAX-P` *dose not contain any writable or readable shared variables, their PRES+ representations are exactly identical. Figures 4.17 represents the manually constructed PRES+ subnet corresponding to the segment which finds the maximum among the four input numbers; the subnet for finding minimum among four given numbers is identical; dotted triangles of Figure 4.17 represent the paths of the subnet. Every instance of* `scanf` *statements results in an in-port with a post transition with identity function. The models have been validated using the CPN simulator. Both these models are then fed one by one to the path construction module. In Figure 4.17, when token tracking execution reaches the places p_6, p_7 and p_8, the degenerate case sets in (as is explained in Section 4.1); the degenerate case is exited when the token tracking execution reaches the out-port of the overall net whose schematic version is in Figure 4.16. For each of the models, our path construction module gives the lists of static and dynamic cut-points, indicates entries to and exits from the degenerate phase and then finally, prints the set of paths. Our tool also gives the path construction time. For calculating time we have used* `get_cpu_time()`*.*

■

```
 int main() {
    int num, max, min, i, j, out;
    printf("Enter seven numbers:");
    scanf("%d", &num);
    max = min = num;
#pragma scop
    for (i = 0; i < 3; i++) {
        scanf("%d", &num);
        if (max < num)
            max = num;
    }
    for (j = 0; j < 3; j++)
    {
        scanf("%d", &num);

        if (min > num)
            min = num;
    }
#pragma endscop
    out = min + max;
    printf("%d ", out);
    return 0;
}
      (a)
```

```
int main() {
    int num, max, min, i,j, out;
    printf("Enter seven numbers:");
    scanf("%d", &num);
    max = min = num;
# CLooG code
    for (i = 0; i < 3; i++) {
        scanf("%d", &num);
        if (max < num)
            max = num;
\PAR
    for (j = 0; j < 3; j++) {
        scanf("%d", &num);
        if (min > num)
            min = num;
    }
# CLooG code
    out = min + max;
    printf("%d ", out);
    return 0;
}
     (b)
```

Figure 4.15: Source and transformed programs of MINANDMAX-P

Table 4.4 summarizes the observations made during our experimentation with the above examples in terms of the numbers of places, transitions (trans), static (SCP) and dynamic cut-points (DCP), degenerate cases (DC) and paths. In Table 4.5, the last three columns depict the path construction times for the PRES+ models of the original and transformed programs. It is to be noted that the paths are also examined manually to ensure that they have been constructed correctly.

Example	Original PRES+						Transformed PRES+											
							PLuTo						Par4All					
	place	trans	SCP	DCP	DC	path	place	trans	SCP	DCP	DC	path	place	trans	SCP	DCP	DC	path
BCM	10	6	6	2	0	3	11	7	6	2	0	3	11	7	6	2	0	3
MINANDMAX-P	28	21	11	16	1	21	28	21	11	16	1	21	28	21	11	16	1	21
LUP	55	53	30	17	2	35	52	50	29	17	2	34	52	50	29	17	2	34
DEKKER	34	32	20	9	1	17	30	29	18	5	1	17	30	29	18	5	1	17
PATTERSON	32	30	15	14	1	12	30	28	14	14	1	12	30	28	14	14	1	12

Table 4.4: Characterization of parallel examples

Example	Path Construction Time (μs)		
	Org	PLuTo	Par4All
BCM	965	929	929
MINANDMAX-P	6234	6234	6234
LUP	10279	9643	9640
DEKKER	14293	13876	13887
PATTERSON	8712	8199	8245

Table 4.5: DCP induced path construction times for hand constructed models of parallel examples

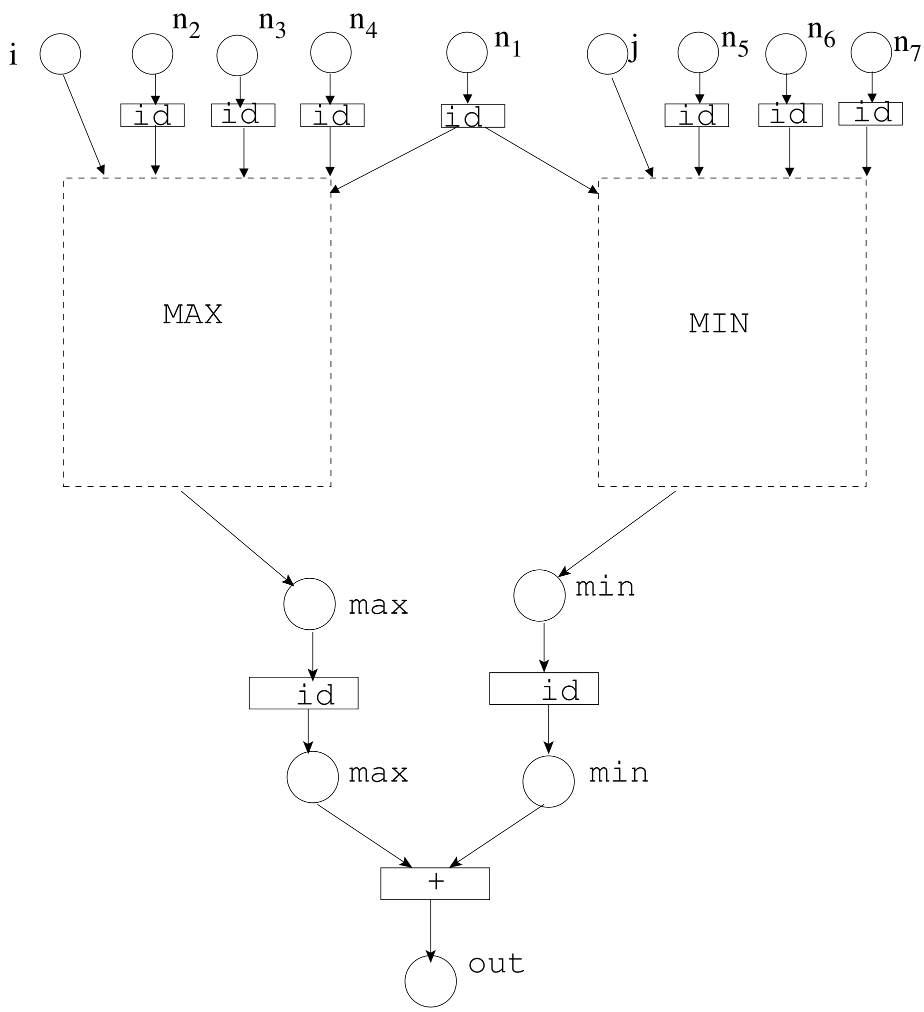

Figure 4.16: Schematic of PRES+ models for MINANDMAX-P source and transformed programs

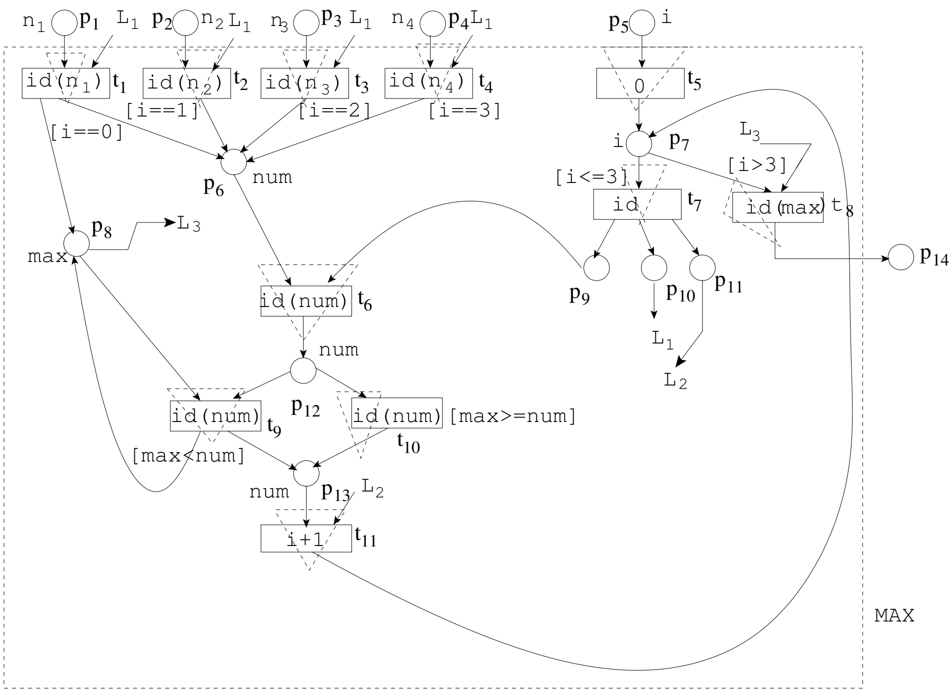

Figure 4.17: PRES+ subnet corresponding to MAX function

4.3.2 Experimentation using an automated model constructor

An automated PRES+ model constructor has been reported in [122] which constructs PRES+ models for the input *C* programs. Since it cannot handle any parallel construct, in this set up, we have only considered those programs whose both original and transformed versions are sequential in nature. The experimental set up is as follows:

1. **Preparation of the example suite:** The sequential program suite is the same as that described in subsection 4.3.1. However, unlike the first line of experimentation, in the present one, we want to compare the performances of the two PRES+ equivalence checking techniques described in this dissertation with these of the FSMD equivalence checking technique reported in [20] on a common set of inputs, i.e., the FSMD models of the C programs. Hence, the automated model constructor is equipped with a utility which takes FSMD models and creates the corresponding C codes from which the PRES+ models are automatically constructed. The FSMD models of the example suite described in subsection 4.3.1 are taken from [14] and the corresponding C codes obtained using the above mentioned utility; their PRES+ models are then obtained using the automated

model constructor. Each of the output PRES+ models is validated using the CPN tool [70] by running on some test data.

2. **Running the path constructor:** Finally, we feed these two PRES+ models as inputs to our path constructor module described in this chapter.

Table 4.6 represents the description of the sequential examples in terms of the numbers of their places, transitions (trans), static (SCP) and dynamic cut-points (DCP), degenerate phases encountered (DC) and paths. It is to be noted that the size of each model in Table 4.6 is significantly large compared to the size of the corresponding model in Table 4.2. The main reason behind this fact is that in Table 4.2, during manual model construction human ingenuity is used; however in Table 4.6, due to automated model construction, many dummy places and transitions are introduced mechanically. Moreover, the model constructor is not adequately optimized. During experimentation with hand constructed models for sequential examples, we fail to construct the PRES+ models for seven examples namely, BARCODE, PRAWN, DIFFEQ, DHRC, IEEE 754, QRS and EWF because of their large size of the code. The article on equivalence checking for code motion using value propagation [20] records the list of transformations which are applied on the seven examples. The last two columns of Table 4.6 depict the path construction times for both original and transformed PRES+ models. The path construction module is found to have worked under both the contexts with the respective observed time units being consistent with the model size. A typical output of the path construction module is given in Figure 4.14. The entire tool is available in [14].

4.4 Conclusion

In order to capture computations by finite paths, a notion of dynamic cut-points has been incorporated. The path construction method is described in detail and illustrated with an example; a detailed complexity analysis has been carried out and formal correctness proofs have been presented; an implementation of the method has been tested on the PRES+ models of some sequential programs and their transformed sequential versions obtained using manual (or SPARK compiler driven transformations) and also a separate set of sequential programs parallelized through some parallelizing compil-

Example	Original PRES+						Transformed PRES+						Path Const. Time (μs)	
	Place	Trans	SCP	DCP	DC	Paths	Place	Trans	SCP	DCP	DC	Path	Original	Transformed
MODN	78	63	11	30	4	43	76	61	18	30	4	42	11345	10863
SUMOFDIGITS	46	31	12	20	3	28	32	23	9	20	3	18	6341	5834
PERFECT	160	113	32	98	6	100	47	32	18	14	3	27	33432	10943
GCD	94	71	36	34	3	52	75	57	28	30	3	49	15534	13426
TLC	362	313	80	85	7	103	189	171	52	72	7	52	195938	86723
DCT	160	71	17	12	0	14	108	70	17	12	0	14	18913	16724
LCM	97	72	37	31	3	52	78	68	28	31	3	49	16534	14426
LRU	880	546	354	234	6	178	865	532	312	213	6	178	447174	387155
PRIMEFACS	76	56	28	30	7	49	47	33	17	20	7	26	11116	10730
MINANDMAX-S	115	56	32	83	1	56	104	51	38	66	1	51	12544	12230
DIFFEQ	82	44	23	59	1	44	72	34	23	49	1	34	16342	11652
DHRC	1953	1244	234	802	3	121	1708	944	234	534	3	107	4494567	4092345
PRAWN	1736	1582	502	429	5	782	1724	1575	502	425	5	782	7508172	7023523
IEEE 754	1261	996	312	413	10	430	1805	1492	485	363	17	415	2976048	2975124
BARCODE	1730	1099	534	842	22	884	2655	1757	610	930	28	1024	3019502	6174098
QRS	880	546	354	234	6	178	865	532	312	213	6	156	447174	387155
EWF	1123	826	415	201	8	540	1054	775	318	413	9	525	2046828	1261312

Table 4.6: DCP induced path construction times for sequential examples using automated model constructor

ers. It is to be noted that the entire experimentation is carried out along two courses. The first one has used hand constructed models and the second course of experimentation has been carried out using an automated model constructor. In the next chapter, we develop a path based equivalence checker with the dynamic cut-point based path construction module described in this chapter as its front end.

Chapter 5

Equivalence Checking Method using Dynamic Cut-points

In Chapter 4, we have discussed a dynamic cut-point based path construction method. In this chapter, we describe a dynamic cut-point induced path based equivalence checking method which uses the path construction module as its subroutine. In the sequel, we refer to this method as `DCPEQX` method. Before describing the method, it is first proved in general that any method which uses such dynamic cut-point induced paths is sound.

5.1 Validity of dynamic cut-point induced path based equivalence checking

To prove the validity of dynamic cut-point induced path based equivalence checking, we need the following definitions.

Definition 21 (Path equivalence, Transition correspondence and Place correspondence)**.** *Let N_0 and N_1 be two PRES+ models with their in-port bijection f_{in} and out-port bijection f_{out}. Equivalence of paths of N_0 and N_1, a transition correspondence relation, denoted as $\eta_p \subseteq T_0 \times T_1$, and a place correspondence relation, denoted as $\eta_p \subseteq P_0 \times P_1$, are defined as follows:*

1. *$f_{in} \subseteq \eta_p$,*

2. *Two paths α of N_0 and β of N_1 are said to be equivalent denoted as $\alpha \simeq \beta$ if $\forall p \in {}^\circ\alpha$, there exists exactly one $p' \in {}^\circ\beta$ such that $f^0_{pv}(p) = f^1_{pv}(p'), \langle p, p' \rangle \in \eta_p$, $R_\alpha(f_{pv}({}^\circ\alpha)) \equiv R_\beta(f_{pv}({}^\circ\beta))$ and $r_\alpha(f_{pv}({}^\circ\alpha)) = r_\beta(f_{pv}({}^\circ\beta))$.*

3. *For any two equivalent paths α, β, $\langle$ last (α), last (β) $\rangle \in \eta_t$ and $\forall p \in \alpha^\circ, p' \in \beta^\circ$ if $f^0_{pv}(p) = f^1_{pv}(p')$, then $\langle p, p' \rangle \in \eta_p$.*

4. *If $\forall p \in \alpha^\circ, p' \in \beta^\circ$, $f^0_{pv}(p) \neq f^1_{pv}(p')$ or there do not exist any paths α' of N_0 and β' of N_1 such that $p \in {}^\circ\alpha' p' \in {}^\circ\beta'$ and $\alpha' \simeq \beta'$, then $\langle p, p' \rangle \notin \eta_p$.*

It may be noted that the above definition provides a procedural mechanism to build the equivalence relation among paths and the correspondence relations η_t and η_p. Clause (1) depicts the f_{in}-pairs as the initial pairs of η_p. Then clause (2) can be used to identify the equivalent pairs of paths originating from the respective in-ports. Such paths, in turn, would define members of η_t and further members of η_p. Repeated applications of clause (2) followed by clause (3) would respectively introduce newer members in the path equivalence relation and in η_t and η_p. The process continues until no new member gets added to the path equivalence relation by clause (2). At this stage, clause (4) can be applied to filter out some extraneous members of η_p.

Theorem 6. *A PRES+ model N_0 is contained in another PRES+ model N_1, denoted as $N_0 \sqsubseteq N_1$, if there exists a finite path cover $\Pi_0 = \{\alpha_0, \alpha_1, \ldots, \alpha_l\}$ of N_0 for which there exists a set $\Pi_1 = \{\beta_0, \beta_1, \ldots, \beta_l\}$ of paths of N_1 such that for all $i, 0 \leq i \leq l$, (i) $\alpha_i \simeq \beta_i$, (ii) the places in ${}^\circ\alpha_i$ have correspondence with those in ${}^\circ\beta_i$ and (iii) the places in α_i° have correspondence with those in β_i°.*

Proof. Consider any computation $\mu_{0,p}$ for an out-port p of N_0. It is required to prove that for the out-port $p' = f_{out}(p)$ of N_1, there exists a computation $\mu_{1,p'} \simeq \mu_{0,p}$. Let $\mu_{0,p} = \langle T_1, T_2, \ldots, T_i, \ldots, T_l \rangle$ where, ${}^\circ T_1 \subseteq inP_0$, $p \in T_l^\circ$ and for all $i, 1 \leq i \leq l$, if $T_i^\circ \subseteq P_{M_i}$, for some marking M_i and $T_{i+1}^\circ \subseteq P_{M_{i+1}}$, for some marking M_{i+1}, then $M_{i+1} = M_i^+$. Since Π_0 is a path cover of N_0, $\mu_{0,p}$ can be captured as a concatenation $(\alpha_1^{(1)} \parallel \alpha_2^{(1)} \parallel \ldots \parallel \alpha_{n_1}^{(1)}).(\alpha_1^{(2)} \parallel \alpha_2^{(2)} \parallel \ldots \parallel \alpha_{n_2}^{(2)}) \ldots (\alpha_1^{(t)}) = \mu^c_{0,p}$, say, of parallelisable paths of Π_0 such that $\mu_{0,p} \simeq \mu^c_{0,p}$. (Note that the last member in the concatenated sequence must be a single path because the last set T_l in $\mu_{0,p}$ is a singleton.)

From $\mu^c_{0,p}$ let us construct a concatenated sequence $\mu^c_{1,p'} = (\beta^{(1)}_1 \parallel \beta^{(1)}_2 \parallel \ldots \parallel \beta^{(1)}_{n_1}).(\beta^{(2)}_1 \parallel \beta^{(2)}_2 \parallel \ldots \parallel \beta^{(2)}_{n_2}) \ldots (\beta^{(t)}_1)$ of parallelisable paths of N_1 such that for all $i, 1 \leq i \leq t$, for all $j, 1 \leq j \leq n_i, \alpha^{(i)}_j \simeq \beta^{(i)}_j$ with any place in $^\circ(\alpha^{(i)}_j)$ having a correspondence with some place in $^\circ(\beta^{(i)}_j)$ and any place in $(\alpha^{(i)}_j)^\circ$ having a correspondence with some place in $(\beta^{(i)}_j)^\circ$. From the premise of the theorem, such paths exist; also, $\mu^c_{0,p} \simeq \mu^c_{1,p'}$.

Consider the i^{th} group $(\beta^{(i)}_1 \parallel \beta^{(i)}_2 \parallel \ldots \parallel \beta^{(i)}_{n_i})$ of $\mu^c_{1,p'}$. For any $j, 1 \leq j \leq n_i$, let the j^{th} path $\beta^{(i)}_j$ in the i^{th} group be the sequence $\langle T^{(i)}_{1,j}, T^{(i)}_{2,j}, \ldots T^{(i)}_{l_j,j} \rangle$ of parallelisable transitions. For all $k, 1 \leq k \leq \max_{j=1}^{n_i}(l_j)$, we combine all the k^{th} transitions of all the paths in the i^{th} group through the union operation to form a single set of parallelisable transitions $T^{(i)}_k = \bigcup_{j=1}^{n_i} T^{(i)}_{k,j}$; obviously, the paths in the i^{th} group can be of varying lengths and those having lengths less than k will not contribute to the set T_k. Let the maximum length of the paths occurring in the i^{th} group be m_i. Then the above step of combining the transition sets of the paths groupwise results in a sequence of parallelisable transitions $\mu^{c'}_{1,p'} = \langle T^{(1)}_1, T^{(1)}_2, \ldots, T^{(1)}_{m_1}, T^{(2)}_1, \ldots, T^{(2)}_{m_2}, \ldots, T^{(t)}_1, \ldots, T^{(t)}_{m_t} \rangle$. We show that $\mu^c_{1,p'}$ is a computation of the out-port p' of N_1 as per definition of computation (Definition 8). What remains to be proved is that for any two consecutive transition sets T, T^+ in $\mu^c_{1,p'}$, if $(T^+)^\circ \subseteq P_{M_{1,i+1}}$ and $(T)^\circ \subseteq P_{M_{1,i}}$, then $M_{1,i+1} = M^+_{1,i}$, i.e., $P_{M_{1,i+1}} = P^+_{M_{1,i}}$. Recall that

$$P_{M^+_{1,i}} = \{p \mid p \in t^\circ \text{ and } t \in T_{M_{1,i}}\} \cup \{p \mid p \in P_{M_{1,i}} \text{ and } p \notin {}^\circ T_{M_{1,i}}\} \ldots (1)$$

Note that $T^+ = T_{M_{1,i}}$, the set of enabled transitions for the marking $M_{1,i}$. We give the proof of $P_{M_{1,i+1}} \subseteq P_{M^+_{1,i}}$; the proof of $P_{M^+_{1,i}} \subseteq P_{M_{1,i+1}}$ follows identically. Now, consider any $p \in P_{M_{1,i+1}}$. Either $p \in (T^+)^\circ$ or $p \notin (T^+)^\circ$.

- *Case 1:* $p \in (T^+)^\circ \Rightarrow p \in t$, for some $t \in T^+ = T_{M_{1,i}} \Rightarrow p \in P_{M^+_{1,i}}$ because $p \in$ the first subset in the definition (1) of $P_{M^+_{1,i}}$.
- *Case 2:* $p \notin (T^+)^\circ$: In this case, $p \in P_{M_{1,i+1}} \Rightarrow p \in P_{M_{1,i}}$ and and $p \notin {}^\circ t$, for any $t \in T^+ = T_{M_{1,i}} \Rightarrow p \in P_{M_{1,i}}$ and $p \notin^\circ T_{M_{1,i}} \Rightarrow p \in P_{M^+_{1,i}}$ because it belongs to the second subset in the definition (1) of $P_{M^+_{1,i}}$.

□

The above theorem leads to a method for checking equivalence between two PRES+ models consisting of the following steps:

1. Introduce static and dynamic cutpoints and hence construct the paths of N_0 and N_1.
2. Construct the initial path covers Π_0 of N_0 and Π_1 of N_1, comprising paths from a set of cutpoints to another cutpoint without having any intermediate cutpoint. Let $\Pi_0 = \{\alpha_0, \alpha_1, \cdots, \alpha_k\}$ and $\Pi_1 = \{\beta_0, \beta_1, \cdots, \beta_l\}$.
3. Show that $\forall \alpha_i \in \Pi_0$, there exists a path β_j of N_1 such that $^\circ\alpha_i$ have correspondence with $^\circ\beta_j$ and $\alpha_i \simeq \beta_j$. If all the paths of Π_0 is found to have equivalence with some paths of N_1, then it is inferred that $N_0 \sqsubseteq N_1$.
4. Let $\Pi_1^E \subseteq \Pi_1$ be the paths of N_1 which are found to have equivalence with paths of Π_0 in step 3. If $\Pi_1 - \Pi_1^E \neq \emptyset$, it is inferred that $N_1 \not\sqsubseteq N_0$. Otherwise, it is inferred that $N_1 \sqsubseteq N_0$.

Step 3 may fail because of code motion transformations where the code segments move beyond the basic block boundaries. In this situation, some paths $\alpha_i \in \Pi_0$ have no equivalent paths in N_1. In such a case, either α_i or some path of N_1 having pre-place correspondence with α_i is to be extended till equivalence of the resulting concatenated path(s) are obtained. The idea of path extension is similar to that of path based FSMD equivalence checking mechanism [20]. Intricacies, however, arise due to presence of paths parallel to the path being extended. The mechanism is illustrated using the following example.

Example 14. *Figure 5.2(a) depicts a PRES+ model N_0 which can be obtained from a simple parallel program schema given in Figure 5.1 (a). Figure 5.2(b) depicts the PRES+ model N_1 corresponding to the schema given in Figure 5.1 (b) which is obtained by moving the code segment c_3 in parallel with the segments c_0 and c_1. Transitions having $f_t = +$ with more than two pre-places indicate application of the function an appropriate number of times. The paths α_0 in N_0 and β_0 in N_1 corresponds to the code segment c_0; the paths α_1 in N_0, β_1 in N_1 corresponds to the code segment c_1; the path α_2 in N_0 corresponds to the code segment c_2; note that this code segment does not appear explicitly in the transformed schema of Figure 5.1 (b); the paths α_3 in N_0 (Figure 5.2 (a)) and β_2 in N_1 (Figure 5.2 (b)) corresponds to the code segment c_3 in*

```
#parbegin
   C0, C1;
#parend

#parbegin
   C2, C3;
#parend
    C4;
```

(a)

```
#parbegin
   C0, C1, C3;
#parend
   C'4;
```

(b)

Figure 5.1: Code motion transformation for parallel programs

Figure 5.1; the path α_4 *in* N_0 *corresponds to the code segment* c_4 *in the schema of Figure 5.1 (a); the path* β_3 *in* N_1 *(Figure 5.2 (b)) corresponds to the code segment* $c_4^{'}$ *in the transformed schema of Figure 5.1 (b).*

Steps 1 and 2 will construct the initial path cover $\Pi_0^{'}$ *of* N_0 *in Figure 5.2 (a) as* $\{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ *and the initial path cover* $\Pi_1^{'}$ *of* N_1 *in Figure 5.2(b) as* $\{\beta_0, \beta_1, \beta_2, \beta_3\}$*. (Note that the computation* $\mu_{0,p_{12}}$ *can be represented as* $(((\alpha_0||\alpha_1).\alpha_2)||\alpha_3).\alpha_4$*. Similarly,* $\mu_{1,p_{12}^{'}}$ *can be represented as* $(\beta_0||\beta_1||\beta_2).\beta_3$*). In step 3, paths* α_0 *and* α_1 *will be found to have equivalence with the paths* β_0 *and* β_1*, respectively. As the equivalent path for the path* α_2 *is tried to be obtained, all its pre-places* $^{\circ}\alpha_2 = \{p_4, p_5, p_6\}$ *are found to have correspondence with the places* $\{p_4^{'}, p_5^{'}, p_6^{'}\} \subset^{\circ} \beta_3$ *but* $\alpha_2 \not\simeq \beta_3$*. However, since* $^{\circ}\alpha_2 \subset^{\circ} \beta_3$ *there is a possibility of extending* α_2 *through its successor path (*α_4 *in this case) so that* $\alpha_2.\alpha_4$ *may have equivalence with* β_3*. This necessitates that the correspondence of all places in* $^{\circ}(\alpha_2.\alpha_4)$ *with places in* N_1 *must be available. In other words, the equivalence of all the pre-path(s) of the path* α_4 *through which the extension is sought with some paths in* N_1 *must be established before carrying out the extension. So, the equivalent path for the path* α_3 *has to be identified first. In this case,* α_3 *will be found to have equivalence with* β_2 *and now an extended path* $(\alpha_2.\alpha_4) = \alpha_e$*, say, will be obtained; subsequently,* $^{\circ}\alpha_e = \{p_4, p_5, p_6, p_{10}, p_{11}\}$ *will have correspondence with* $^{\circ}\beta_3 = \{p_4^{'}, p_5^{'}, p_6^{'}, p_{10}^{'}, p_{11}^{'}\}$*; finally, the equivalence between* $(\alpha_2.\alpha_4)$ *and* β_3 *will be established through the equivalence of their conditions of execution and equality of their data transformations.* ■

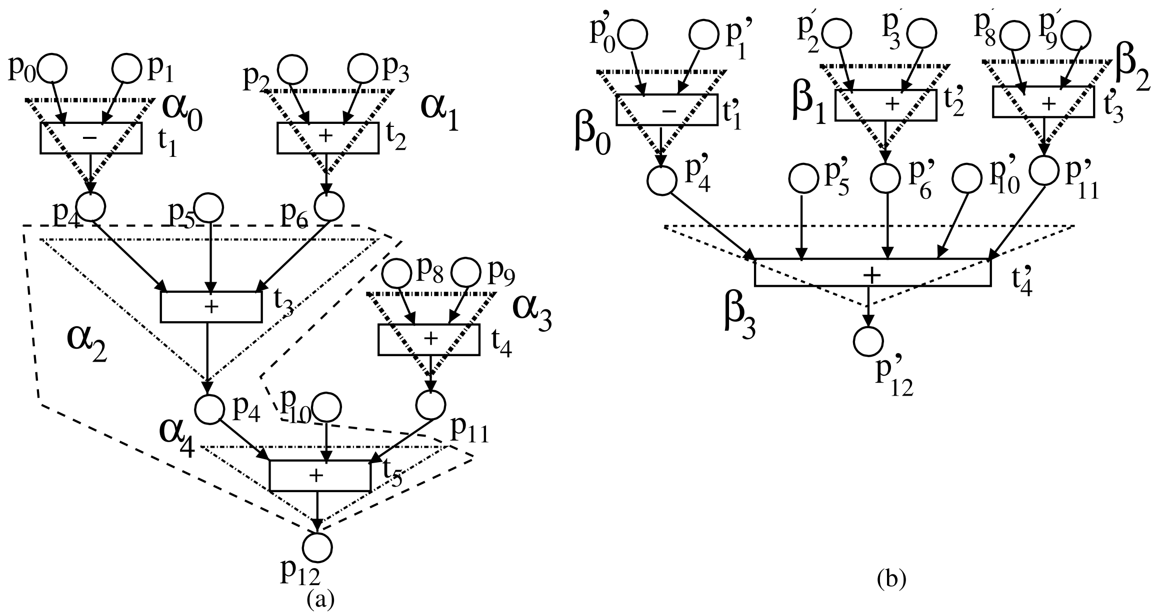

Figure 5.2: An Illustrative Example for Equivalence Checking

In the next section, we present the formal steps (incorporating path extension) of an equivalence checking algorithm.

5.2 An Equivalence Checking Method

The `checkEqDCP` (Algorithm 14) function is the central module for `DCPEQX` method. The inputs to this function are the PRES+ models, N_0 and N_1. The outputs are the final path covers Π_0 of N_0 and Π_1 of N_1, a set E of ordered pairs of equivalent paths of N_0 and N_1 and the set $\Pi_{n,0}$ of paths of N_0 and $\Pi_{n,1}$ of paths of N_1 for which no equivalent is found (in the other PRES+ model) even with extension, if needed.

The function starts by initializing the set η_p of ordered pairs of corresponding places of N_0 and N_1 to the in-port bijection f_{in} (Definition 21); the set η_t of ordered pairs of corresponding transitions of N_0 and N_1 and the sets $E, \Pi_0, \Pi_1, \Pi_{n,0}$ and $\Pi_{n,1}$ are initialized to empty. It then constructs the set $\Pi_0^{'}$ of paths of N_0 and the set $\Pi_1^{'}$ of paths of N_1 by introducing static and dynamic cutpoints. These sets are the respective initial path covers of N_0 and N_1. It is to be noted that Π_0 and Π_1 are the final path cover which are obtained from $\Pi_0^{'}$ and $\Pi_1^{'}$, respectively.

For each path of α of $\Pi_0^{'}$ (of N_0), the function `checkEqDCP` calls `findEqvDCP` function which tries to find an equivalent path from $\Pi_1^{'}$ of N_1. The function `findEqvDCP` returns a path-flag pair, $\langle \beta, \lambda \rangle$, where β is a path of N_1 which can be considered to be a candidate for checking equivalence with α. The candidate path of α is defined as follows.

Definition 22 (Candidate path)**.** *Let N_0 and N_1 be two i/o-compatible PRES+ models. A path β of N_1 is said to be a candidate path for (checking equivalence with) a path α of N_0 if one of the following conditions holds:*

(i) $\forall p \in {}^\circ\alpha$, if $p \in inP_0$, then $\exists p' \in {}^\circ\beta$ such that $\langle p, p' \rangle \in \eta_p$.

(ii) $\forall p \in {}^\circ\alpha$, if $p \in \alpha_1^\circ$, for some $\alpha_1 \in \Pi_0$, then $\exists p_1 \in {}^\circ\beta$ such that $p_1 \in \beta_1^\circ$ and $\langle \text{last}(\alpha_1), \text{last}(\beta_1) \rangle \in \eta_t$.

The function `findEqvDCP` uses the function `findCandidate` which, in turn, either returns a candidate path or or an empty path. Depending on the value of the flag λ we have the following cases:
$\underline{\lambda = 0}$: the function has found a candidate path β in N_1 which has a stronger condition of execution than that of α, i.e., $(R_\beta(f_{pv}^1({}^\circ\beta)) \Rightarrow R_\alpha(f_{pv}^0({}^\circ\alpha))$ or $|{}^\circ\alpha| < |{}^\circ\beta|$. The second condition suggests that some code segment after α in N_0 may have been moved prior to α; hence α, possibly with other paths, parallel to α are extended.
$\underline{\lambda = 1}$: the function has found a candidate path β in N_1 which has a weaker condition of execution than that of α, i.e., $(R_\alpha(f_{pv}^0({}^\circ\alpha)) \Rightarrow R_\beta(f_{pv}^1({}^\circ\beta))$ or $|{}^\circ\beta| < |{}^\circ\alpha|$. The second condition suggests that possibly some code segment prior to α has been moved after α; hence the path β, possibly along with some parallel paths, are to be extended.
$\underline{\lambda = 2}$: there is no candidate path β in $\Pi_1^{'}$ emanating from the places corresponding to the pre-places of α such that $(R_\alpha(f_{pv}^0({}^\circ\alpha)) \equiv R_\beta(f_{pv}^1({}^\circ\beta))$ or$(R_\alpha(f_{pv}^0({}^\circ\alpha)) \Rightarrow R_\beta(f_{pv}^1({}^\circ\beta))$ or $(R_\beta(f_{pv}^1({}^\circ\beta)) \Rightarrow R_\alpha(f_{pv}^0({}^\circ\alpha))$. Hence, there is no scope of extension at all; it then updates $\Pi_{n,0}$ by adding the path α to it and also updates $\Pi_0^{'}$ by deletion of α.
$\underline{\lambda = 3}$: the candidate path β is an equivalent path of α and $|{}^\circ\beta| = |{}^\circ\alpha|$. The following entities are updated: (1) The set η_t of corresponding transitions by adding the pair comprising the last transition of the path α and that of β; (2) the set E of ordered pairs of equivalent paths by adding the ordered pair $\langle \alpha, \beta \rangle$; (3) the initial path covers $\Pi_0^{'}$ of N_0 and $\Pi_1^{'}$ of N_1 by deleting the paths α from $\Pi_0^{'}$ and β from $\Pi_1^{'}$; (4) α is added to the

final path cover Π_0 of N_0 and β to Π_1 of N_1; (5) The set η_p of corresponding places by adding the pair comprising the post-place of the last transition of the path α and that of β.

$\underline{\lambda = 4}$: β is an equivalent path of α but $|^\circ\beta| < |^\circ\alpha|$. The sets $\eta_p, \eta_t, E, \Pi_0^{'}$ and Π_0 are updated identically as in $\lambda = 3$ – case. The sets $\Pi_1^{'}$ and Π_1 are not updated because the path β may turn out to be equivalent to other paths of $\Pi_0^{'}$ as well.

$\underline{\lambda = 5}$: β is an equivalent path of α but $|^\circ\beta| > |^\circ\alpha|$. The sets $\eta_p, \eta_t, E, \Pi_1^{'}$ and Π_1 are updated identically as in $\lambda = 3$ – case. Again, the sets $\Pi_0^{'}$ and Π_0 are not updated because of similar reason as stated in $\lambda = 4$ – case.

Extension of a path α of N_0 (accomplished by the function `prepareForExtension`) involves the following steps:

1. All the post-paths of α are identified. Such paths include those which emanate from the post-places α° (under different guards) or synonymously, those which emanate from the post-places of the last transition of α.

2. For each post-path $\alpha^{'}$, all the pre-paths of $\alpha^{'}$ other than α are identified. Some of these may not execute in parallel (with α). For example, let $\{\alpha, \alpha_1, \alpha_2, \alpha_3\}$ be three such pre-paths of $\alpha^{'}$; let α_2 and α_3 have an identical post-place. Hence, α_2 and α_3 cannot execute in parallel because the models are one-safe. So the pre-paths (including α) are decomposed into two subsets $\{\alpha, \alpha_1, \alpha_2\}$ and $\{\alpha, \alpha_1, \alpha_3\}$ by the function `findSetOfSetsOfPrePaths`.

3. The function `trimPrePaths` is invoked to trim each of the subsets of pre-paths of the members (other than α) each of which is found to have equivalence (without any extension) with some path in N_1. To accomplish this task, the function `trimPrePaths` invokes `findEqvDCP` function. If it is detected that such a path may have to be extended before its equivalence is found, then no action is initiated because they are already under consideration for extension. However, if it is found that the path does not merit any further consideration (such as extension), it is put in the set $\Pi_{n,0}$ of paths of N_0 which may have no equivalent path in N_1. The set is not used for extension.

4. Finally, extended paths are created for each of the remaining pre-paths obtained from step (2). These pre-paths and the post-paths are removed from $\Pi_0^{'}$ and the extended path is included in it for normal processing. This step is achieved

in the function `extend`. Extension of paths of N_1 takes place in an identical manner.

When all the paths in the path cover Π_0' of N_0 have been examined exhaustively (i.e., Π_0' is rendered empty), all the paths remaining in Π_1' are put in $\Pi_{n,1}$. The function then checks $\Pi_{n,0}$ and $\Pi_{n,1}$; we have the following four cases:

- *Case 1:* $\Pi_{n,0}, \Pi_{n,1} = \emptyset \Rightarrow N_0 \equiv N_1$.
- *Case 2:* $\Pi_{n,0} = \emptyset, \Pi_{n,1} \neq \emptyset \Rightarrow N_0 \sqsubseteq N_1$ and $N_1 \not\sqsubseteq N_0$.
- *Case 3:* $\Pi_{n,0} \neq \emptyset, \Pi_{n,1} = \emptyset \Rightarrow N_1 \sqsubseteq N_0$ and $N_0 \not\sqsubseteq N_1$.
- *Case 4:* $\Pi_{n,0}, \Pi_{n,1} \neq \emptyset \Rightarrow N_0$ and N_1 may not be equivalent.

The functional modules are depicted in Algorithms 7 – 13 with Algorithm 14 being the top level module. The call graph of the dynamic cut-point based equivalence checking algorithm is given in Figure 5.3. During path extension in the equivalence checking phase, it is necessary to extend either a path from N_0 or one from N_1 (and not both). This fact necessitates that the function `prepareForExtension` and all the functions it calls should be symmetric. Hence, except `checkEqDCP` function, each of the other modules in the equivalence checking phase is associated with a binary flag designating which PRES+ model it has to work upon.

The following example illustrates how the equivalence checking algorithm works.

Example 15. *Consider the PRES+ models given in Figure 5.2 (a) and (b). For this example, the equivalence checking method progresses through the following steps:*

1. *Let f_{in} be $\{p_0 \mapsto p_0', p_1 \mapsto p_1', p_2 \mapsto p_2', p_3 \mapsto p_3',\ p_5 \mapsto p_5', p_8 \mapsto p_8', p_9 \mapsto p_9', p_{10} \mapsto p_{10}'\}$. The set η_p of corresponding places is initialized to f_{in}. The sets $\eta_t, E, \Pi_0, \Pi_1, \Pi_{n,0}, \Pi_{n,1}$ are initialized to $\emptyset$.*
2. *For α_0, the function `findEqvDCP` identifies (by the function `findCandidate`) the path β_0 as the candidate for examining equivalence with α_0 because both have two pre-places correlated by the function f_{in}. Since $|{}^\circ\alpha_0| = |{}^\circ\beta_0|$, the function `findEqvDCP` identifies that $R_{\alpha_0}(f_{pv}^0({}^\circ\alpha_0)) \equiv R_{\beta_0}(f_{pv}^1({}^\circ\beta_0))(\equiv \top)$ and $r_{\alpha_0}(f_{pv}^0({}^\circ\alpha_0)) = r_{\beta_0}(f_{pv}^1({}^\circ\beta_0))$ (i.e., $v_{p_0} - v_{p_1} = v_{p_0'} - v_{p_1'}$ and puts $\langle p_0, p_0'\rangle, \langle p_1, p_1'\rangle$ in η_p) and infers $\alpha_0 \simeq \beta_0$.*

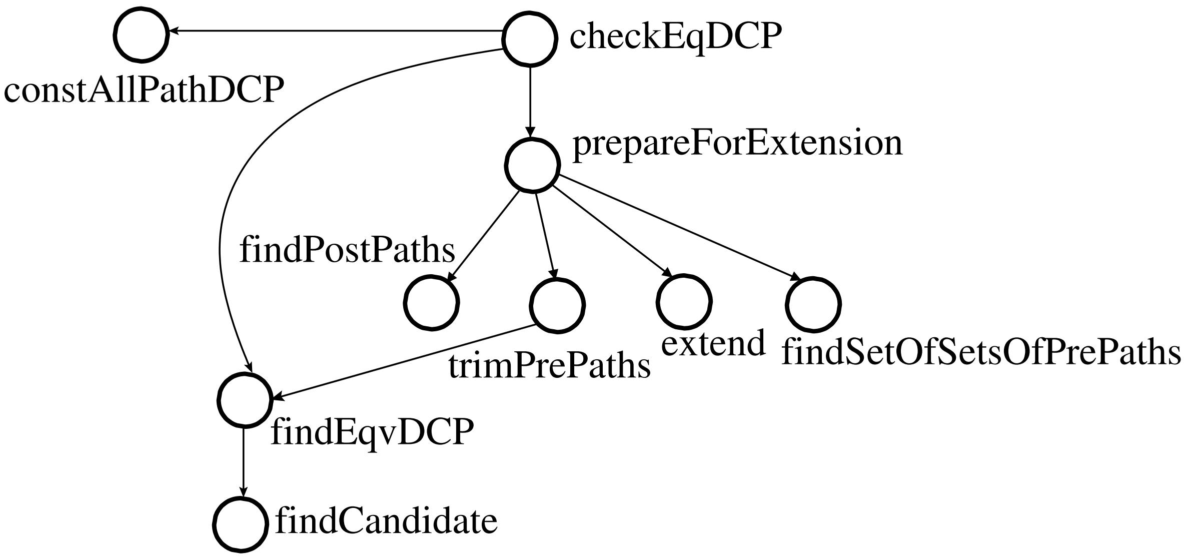

Figure 5.3: Call Graph for the Verification Algorithm

3. *Consequently, the function* `checkEqDCP` *removes the path* α_0 *and* β_0 *from* $\Pi_0^{'}$ *and* $\Pi_1^{'}$*, respectively, adds the path* α_0 *to* Π_0 *and* β_0 *to* Π_1 *and adds* $\langle\, ^\circ(\alpha_0^\circ)$*,* $^\circ(\beta_0^\circ)\,\rangle$ *in* η_t*, puts* $\langle \alpha_0^\circ, \beta_0^\circ \rangle$ $\langle p_4, p_4^{'} \rangle$ *in* η_p *and* $\langle \alpha_0, \beta_0 \rangle$ *in E.*

4. *Similarly,* α_1 *is found to have equivalence with* β_1 *and the sets* $\Pi_0^{'}, \Pi_1^{'}, \Pi_0, \Pi_1, \eta_t, \eta_p$ *and E are updated.*

5. *For* α_2*, the function* `findCandidate` *identifies the path* β_3 *as the candidate path for* α_2 *because* (i) α_0°*,* $\alpha_1^\circ \in {^\circ\alpha_2}$*,* (ii) $\langle\, ^\circ(\alpha_0^\circ),\, ^\circ(\beta_0^\circ)\,\rangle$*,* $\langle\, ^\circ(\alpha_1^\circ),\, ^\circ(\beta_1^\circ)\,\rangle$ $\in \eta_t \Rightarrow \beta_0^\circ$*,* $\beta_1^\circ \in$ *the pre-places of the candidate paths. Hence, the function* `findEqvDCP` *is invoked with* β_3*.*

6. *Since* `findEqvDCP` *finds* $|{^\circ\alpha_2}| = 3 < |{^\circ\beta_3}| = 5$ *and* $r_{\alpha_2}(f_{pv}^0({^\circ\alpha_2})) \neq r_{\beta_3}(f_{pv}^1({^\circ\beta_3}))$*, it returns* $\langle \alpha_2, \lambda \rangle$ *with* $\lambda = 4$ *ascertaining that an extension of* α_2 *is required.*

7. *The function* `checkEqDCP` *calls* `prepareForExtension`*.*

    (a) `prepareForExtension` *calls the function* `findPostPaths` *which returns the set of post-paths of* α_2*, i.e.,* $\{\alpha_4\}$*.*

    (b) *For the path* α_4*, the function* `prepareForExtension` *calls* `findSetOfSetsOfPrePaths` *to obtain the mutually exclusive subsets of pre-paths of* α_4 *as the single subset* $\{\alpha_2, \alpha_3\}$*.*

    (c) *For* $\{\alpha_2, \alpha_3\}$*, the function* `prepareForExtension` *calls* `trimPrePaths`*; the latter function finds* α_3 *to be equivalent to* β_2 *by invoking the function* `findEqvDCP` *and accordingly updates the sets* $\Pi_0^{'}, \Pi_1^{'}, \Pi_0, \Pi_1, \eta_t, \eta_p$ *and* E.

(d) *As the set of pre-paths obtained after* `trimPrePaths` *is not empty, the function* `prepareForExtension` *calls* `extend` *function which returns the extended path* α_e *as* $(\alpha_2.\alpha_4)$. *The paths* α_2 *and* α_4 *are then removed from* $\Pi_0^{'}$*; the path* α_e *is added to* $\Pi_0^{'}$ *and control is returned to* `checkEqDCP`.

8. `checkEqDCP` *finds* $\alpha_e \simeq \beta_3$ *using the function* `findEqvDCP`. *The following entities are as follows:* $\Pi_0^{'} = \emptyset, \Pi_1^{'} = \emptyset, \Pi_0 = \{\alpha_0, \alpha_1, \alpha_3, \alpha_e\}, \Pi_1 = \{\beta_0, \beta_1, \beta_2, \beta_3\}$, $\eta_t = \{\langle {}^\circ(\alpha_0^\circ), {}^\circ(\beta_0^\circ)\rangle, \langle {}^\circ(\alpha_1^\circ), {}^\circ(\beta_1^\circ)\rangle, \langle {}^\circ(\alpha_3^\circ), {}^\circ(\beta_2^\circ)\rangle, \langle {}^\circ(\alpha_e^\circ), {}^\circ(\beta_3^\circ)\rangle\}$, $\eta_p = \{ \langle (\alpha_0^\circ), (\beta_0^\circ) \rangle, \langle (\alpha_1^\circ), (\beta_1^\circ) \rangle, \langle (\alpha_3^\circ), (\beta_2^\circ) \rangle, \langle (\alpha_e^\circ), (\beta_3^\circ) \rangle\}$ *and* $E = \{ \langle (\alpha_0), (\beta_0) \rangle, \langle (\alpha_1), (\beta_1) \rangle, \langle (\alpha_3), (\beta_2) \rangle, \langle (\alpha_e), (\beta_3) \rangle \}$.

9. *Since* $\Pi_0^{'}$ *is now empty,* `checkEqDCP` *identifies that* $\Pi_{n,0} = \Pi_1^{'} = \emptyset$ *and declares that the two models* N_0 *and* N_1 *are equivalent.*

■

The above algorithm is now analysed for termination, complexity and soundness in the following subsections.

5.2.1 Termination of the equivalence checking algorithm

The path construction algorithm terminates as shown in Chapter 4. Therefore, the respective path covers $\Pi_0^{'}$ and $\Pi_1^{'}$ of N_0 and N_1 produced by this algorithm are finite and the equivalence checking phase starts with finite $\Pi_0^{'}$ and $\Pi_1^{'}$. The following lemma establishes that they remain finite in the equivalence checking phase. The termination of the equivalence checking phase hinges upon this property.

Lemma 4. *Both the initial path covers* $\Pi_0^{'}$ *and* $\Pi_1^{'}$ *of* N_0 *and* N_1, *respectively, remain finite across all the functions (Algorithms 7-14) in the equivalence checking phase.*

Proof. Only deletion takes place from $\Pi_0^{'}$ $\left(\Pi_1^{'}\right)$ in the functions `trimPrePaths` and `checkEqDCP` (Algorithms 11 and 14). If they start with finite values of $\Pi_0^{'}$ $\left(\Pi_1^{'}\right)$, the finiteness is preserved. Both deletion from and addition to the set $\Pi_0^{'} \left(\Pi_1^{'}\right)$ takes place in the function `prepareForExtension` (Algorithms 13). Step 10 of the function `prepareForExtension` (Algorithm 13) updates $\Pi_0^{'} \left(\Pi_1^{'}\right)$ by deleting the set Γ_P of trimmed pre-paths of $\gamma^{'}$, the path $\gamma^{'}$ itself and adding the extended path $\gamma_e = \Gamma_P.\gamma^{'}$. So, $\left|\Pi_0^{'}\right| \left(\left|\Pi_1^{'}\right|\right)$ decreases by $|\Gamma_P| + 1$ and increases by 1, i.e., an effective decrease by $|\Gamma_P|$. However, $|\Gamma_P| \geq 0$. If $|\Gamma_P| = 0$; then it remains the same. Hence, in each iteration (step 3 to 12) $\left|\Pi_0^{'}\right| \left(\left|\Pi_1^{'}\right|\right)$ either decreases or remains the same. Outside the loop, since in step 1 there is a decrease by 1, $\left|\Pi_0^{'}\right| \left(\left|\Pi_1^{'}\right|\right)$ decreases in every invocation of function. No other function changes $\Pi_0^{'} \left(\Pi_1^{'}\right)$. □

Theorem 7. *`checkEqDCP function` (Algorithm 14) always terminates.*

Proof. The function `checkEqDCP` (Algorithm 14) consists of a loop which depends on $|\Pi_0^{'}|$ and there is no recursive call also. Hence, the loop always terminates as given in Lemma 4. Therefore, the above mentioned function terminates. □

5.2.2 Complexity analysis of the equivalence checking algorithm

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

Functional Specification of Algorithm 7 (`findCandidate`): The function identifies the candidate paths for checking equivalence with the input path γ based on the correspondence of their pre-places decided by either the in-port bijection f_{in} or the correspondence between the last transition of the preceding paths (as recorded in η_t) or their post-places decided by the out-port bijection f_{out}.

Complexity of Algorithm 7 (`findCandidate`): Steps 1 to 6 take $O(1)$ time. Condition detection steps for Case 1 and Case 4 take $O(|P|)$ time and that for cases 2, 3 and 5 take $O(|P|\,log\,|P|)$ time assuming that pre-places of any path and the sets $inP_0, inP_1, outP_0$ and $outP_1$ are maintained in sorted order. The statements in each of the cases take $O(|T|\,log\,|T|)$ time and iterate as many times as the number of paths which is bounded by $|T|\,.\,|P|$, i.e., $O(|T|.|P|)$ times. Hence, the overall complexity of this function is $O(\{max(|T|log|T|, |P|log|P|)\}.|T|.|P|)$.

Functional Specification of Algorithm 8 (`findEqvDCP`): The function tries to find an equivalent path of γ from the candidate paths obtained from the function `findCandidate` (Algorithm 7). The function `findEqvDCP` returns a path-flag pair, $\langle\gamma^{'},\lambda\rangle$, where $\gamma^{'}$ is a candidate path of N_0 (N_1) which is closest to γ of N_1 (N_0), if not equivalent, or an empty path. If *flag* value is 0, $\gamma\ (=\alpha)$ is a path of N_0 and $\Pi^{'}\ (=\Pi_1^{'})$. Otherwise, $\gamma\ (=\beta)$ is a path of N_1 and $\Pi^{'}\ (=\Pi_0^{'})$. It returns a path-flag pair $\langle\gamma^{'},\lambda\rangle$, where $\gamma^{'}$ is a path of N_0 or N_1 (depending upon *flag*) or an empty path and λ has the following values: $\lambda = 0 \Rightarrow R_{\gamma^{'}}(f_{pv}(^{\circ}\gamma^{'})) \rightarrow R_{\gamma}(f_{pv}(^{\circ}\gamma))$ and $R_{\gamma^{'}}(f_{pv}(^{\circ}\gamma^{'})) \not\equiv R_{\gamma}(f_{pv}(^{\circ}\gamma))$: extend $\gamma(=\alpha)$, $\lambda = 1 \Rightarrow R_{\gamma}(f_{pv}(^{\circ}\gamma)) \rightarrow R_{\gamma^{'}}(f_{pv}(^{\circ}\gamma^{'}))$ and $R_{\gamma^{'}}(f_{pv}(^{\circ}\gamma^{'})) \not\equiv R_{\gamma}(f_{pv}(^{\circ}\gamma))$: extend $\gamma^{'}(=\beta)$, $\lambda = 2$ $R_{\gamma^{'}}(f_{pv}(^{\circ}\gamma^{'})) \not\Rightarrow R_{\gamma}(f_{pv}(^{\circ}\gamma))$ and $R_{\gamma}(f_{pv}(^{\circ}\gamma)) \not\Rightarrow R_{\gamma^{'}}(f_{pv}(^{\circ}\gamma^{'}))$: no scope of extension at all, $\lambda = 3 \Rightarrow$: $\gamma^{'}$ is an equivalent path. $\lambda = 4 \Rightarrow$: $\gamma^{'}$ is an equivalent path, where $(|^{\circ}\gamma^{'}| - |^{\circ}\gamma| \geq 1)$. $\lambda = 5 \Rightarrow$: $\gamma^{'}$ is an equivalent path, where $(|^{\circ}\gamma| - |^{\circ}\gamma^{'}| \geq 1)$.

Complexity of Algorithm 8 (`findEqvDCP`): Step 1 uses `findCandidate` function which takes $O(\{max(|T|log|T|, |P|log|P|)\}.|T|.|P|)$ as explained above. Testing of the respective condition of cases 1, 2 and 3 takes $O(|P|)$ time. For Case 1, we need to consider three sub-cases as given in steps 4, 7 and 10. Each of these steps compares the condition of execution and the data transformation for each path. Hence the complexity for each of this comparison is $O(|F|)$, where $|F|$ is the length of the formula. The body of the loop takes maximum among the three cases 1, 2 and 3 which is $O(|F| + |P|)$. The loop iterates as many times as the number of paths which is bounded by $|T|.|P|$. Hence, the overall complexity is $O((\{max(|T|log|T|, |P|log|P|)\} + |F|).|T|.|P|)$.

Functional Specification of Algorithm 9 (`findPostPaths`): The function computes the post-paths of γ through which γ can be extended. Such paths include those which emanate from the post-places γ° (under different guards) or those which emanate from the post-places of the last transition of γ.

Complexity of Algorithm 9 (`findPostPaths`): Step 1 takes $O(1)$ time. In Step 3, there is a conditional branch which takes $O(|P|)$ time. If the condition is true, the function updates the set of paths by union operation which takes $O(1)$ time. The loop involved in step 2 iterates $O(|T|.|P|)$ time. Hence, the complexity of this function is $O\left(|P|^2.|T|\right)$.

Functional Specification of Algorithm 10 (`findSetOfSetsOfPrePaths`): The function computes all the pre-paths of γ' other than γ. Some of these may not execute in parallel (with γ). For example, let $\{\gamma, \gamma_1, \gamma_2, \gamma_3\}$ be three such pre-paths of γ'; let γ_2 and γ_3 have an identical post-place. Hence, γ_2 and γ_3 cannot execute in parallel because the models are one-safe. So the pre-paths (including γ) are decomposed into two subsets $\{\gamma, \gamma_1, \gamma_2\}$ and $\{\gamma, \gamma_1, \gamma_3\}$.

Complexity of Algorithm 10 (`findSetOfSetsOfPrePaths`): Step 1 takes $O(1)$ time. In Step 3, there is a conditional branch which takes $O(|P|)$ time. If the condition is true, the function updates the set of paths by union operation and it takes $O(1)$. The loop involved in step 2 iterates $O(||T||.||P||)$ time. Hence, the complexity of the loop in step 2 is $O\left(|P|^2.|T|\right)$ time. Similarly, in step 8 the function computes Γ_P° and it takes $O(|P|)$ time. For each member in Γ_P°, the function checks whether $|{}^\circ p| > 1)$; if the statement is true, the function computes the mutually exclusive subsets of pre-paths and it takes $O(|T|)$ time. This step iterates $O(|P|)$ time. Hence, the complexity of this step is $O(|P|.|T|)$. In the next step the function computes the Cartesian product of mutually exclusive subsets of pre-paths and it takes $O\left(2^{(\frac{|T|}{2})}\right)$ time. Hence, the overall complexity of this function is $O\left(2^{(\frac{|T|}{2})}\right)$ time.

Functional Specification of Algorithm 11 (`trimPrePaths`) : Each of the subsets of pre-paths is trimmed of the members (other than γ) which are found to have equivalence (without any extension) with some path in $N_0(N_1)$. If it is detected that such a path may have to be extended before its equivalence is found, then no action is initiated because they are already under consideration for extension. However, if it is found that the path does not merit any further consideration (such as extension), it is put in the set $\Pi_{n,0}(\Pi_{n,1})$ of paths of $N_0(N_1)$ which may have no equivalent path in $N_0(N_1)$. The set is not used for extension.

Algorithm 7 SETOFPATHS **findCandidate** ($flag, \gamma, \eta_t, \Pi^{'}, f_{in}, f_{out}$)

Inputs: The first parameter is a flag. The second parameter γ: a path. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The third parameter η_t: the set of corresponding transition pairs. The fourth parameter $\Pi^{'}$: a set of paths remaining from the original path cover. If flag = 0, it belongs to N_1; if flag = 1, it belongs to N_0. The fifth parameter f_{in}: in-port bijections. The sixth parameter f_{out}: out-port bijections.
Outputs: The set Γ' of paths of the PRES+ model other than the model having γ from which equivalent of γ should be found.

SETOFPATHS $\Gamma^{'} = \emptyset$;
if (flag = 0) **then**
  $inP' = inP_0; outP' = outP_0$;
else
  $inP' = inP_1; outP' = outP_1$;
end if
Case 1 ($^{\circ}\gamma \subseteq inP^{'}$):
for each $\gamma^{'} \in \Pi^{'}$ **do**
  $^{\circ}\gamma^{'} = f_{in}(^{\circ}\gamma)$; // $inP' = inP_0(inP_1)$ if $flag = 0(1)$
  $\Gamma^{'} = \Gamma^{'} \cup \{\gamma^{'} \mid f_{in}(^{\circ}\gamma) \in^{\circ} \gamma^{'}\}$;
end for
return Γ';
Case 2 ($^{\circ}\gamma \cap inP^{'} \neq \emptyset$) :
for each $\gamma^{'} \in \Pi^{'}$ **do**
  $^{\circ}\gamma^{'} = f_{in}(^{\circ}\gamma \cap inP^{'})$;
  $\Gamma^{'}_1 = \{\gamma^{'} \mid f_{in}(^{\circ}\gamma \cap inP^{'}) \in^{\circ} \gamma^{'}\}$;
  $T_{pre} = \{t_c \mid \langle t, t_c \rangle \in \eta_t, t \in^{\circ} (^{\circ}\gamma)\}$;
  $\Gamma^{'} = \Gamma^{'} \cup \Gamma^{'}_1 \cup \{\gamma^{'} \mid^{\circ} \gamma^{'} = (T_{pre})^{\circ}\}$;
end for
return Γ';
Case 3 ($^{\circ}\gamma \cap inP^{'} = \emptyset$) :
for each $\gamma^{'} \in \Pi^{'}$ **do**
  $T_{pre} = \{t_c \mid \langle t, t_c \rangle \in \eta_t, t \in^{\circ} (^{\circ}\gamma)\}$;
  $\Gamma^{'} = \Gamma^{'} \cup \{\gamma^{'} \mid^{\circ} \gamma^{'} = (T_{pre})^{\circ}\}$;
end for
Case 4 ($\gamma^{\circ} \subseteq outP^{'}$): // $outP' = outP_0(outP_1)$ if $flag = 0(1)$
for each $\gamma^{'} \in \Pi^{'}$ **do**
  $(\gamma^{'})^{\circ} = f_{out}(\gamma^{\circ})$;
  $\Gamma^{'} = \Gamma^{'} \cup \{\gamma^{'} \mid f_{out}(\gamma^{\circ}) \in (\gamma^{'})^{\circ}\}$;
end for
return Γ';
Case 5 ($\gamma^{\circ} \cap outP^{'} \neq \emptyset$) :
for each $\gamma^{'} \in \Pi^{'}$ **do**
  $(\gamma^{'})^{\circ} = f_{out}(\gamma^{\circ} \cap outP^{'})$;
  $\Gamma^{'}_1 = \{\gamma^{'} \mid f_{out}(\gamma^{\circ} \cap outP^{'}) \in (\gamma^{'})^{\circ}\}$;
  $T_{pre} = \{t_c \mid \langle t, t_c \rangle \in \eta_t, t \in^{\circ} (^{\circ}\gamma)\}$;
  $\Gamma^{'} = \Gamma^{'} \cup \Gamma^{'}_1 \cup \{\gamma^{'} \mid^{\circ} \gamma^{'} = (T_{pre})^{\circ}\}$;
end for
return Γ';

Algorithm 8 Path-Flag **findEqvDCP** ($flag, \gamma, \eta_t, \Pi'$)

Inputs: The first parameter is a flag. The second parameter γ: a path whose equivalent has to be found. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The third parameter η_t: the set of pairs of corresponding transitions. The fourth parameter Π': a set of paths remaining from the original path cover. If flag = 0, it belongs to N_1; if flag = 1, it belongs to N_0.

Outputs: Path-flag pair $\langle \gamma', \lambda \rangle$, where γ' is a path of N_0 or N_1 or an empty path. If $flag = 0$, γ' is a path of N_1. If $flag = 1$, γ' is a path of N_0. The flag λ in the path-flag pair has the following values:

$\lambda = 0 \Rightarrow$ extend $\gamma' = \gamma$, the input path, because $R_{\gamma'}(f_{pv}(^\circ\gamma')) \not\equiv R_\gamma(f_{pv}(^\circ\gamma))$ and $R_{\gamma'}(f_{pv}(^\circ\gamma')) \Rightarrow R_\gamma(f_{pv}(^\circ\gamma))$;

$\lambda = 1 \Rightarrow$ extend γ', which is a path of the other PRES+ (than the PRES+ containing γ) because $R_{\gamma'}(f_{pv}(^\circ\gamma')) \not\equiv R_\gamma(f_{pv}(^\circ\gamma))$ and $R_\gamma(f_{pv}(^\circ\gamma)) \Rightarrow R_{\gamma'}(f_{pv}(^\circ\gamma'))$;

$\lambda = 2 \Rightarrow R_{\gamma'}(f_{pv}(^\circ\gamma')) \not\Rightarrow R_\gamma(f_{pv}(^\circ\gamma)), R_\gamma(f_{pv}(^\circ\gamma)) \not\Rightarrow R_{\gamma'}(f_{pv}(^\circ\gamma'))$: no scope of extension at all;

$\lambda = 3 \Rightarrow \gamma'$ is an equivalent path with same number of pre-places.

$\lambda = 4 \Rightarrow \gamma'$ is an equivalent path and $(|^\circ\gamma'| - |^\circ\gamma| \geq 1)$.

$\lambda = 5 \Rightarrow \gamma'$ is an equivalent path and $(|^\circ\gamma| - |^\circ\gamma'| \geq 1)$.

$\Gamma' =$ **findCandidate** $(flag, \gamma, \eta_t, \Pi', f_{in})$;
for each $\gamma' \in \Gamma'$ **do**
    Case 1 ($|^\circ\gamma'| - |^\circ\gamma| = 0$):
    if $((R_\gamma(f_{pv}(^\circ\gamma)) \equiv R_{\gamma'}(f_{pv}(^\circ\gamma')))$ and $(r_\gamma(f_{pv}(^\circ\gamma)) = r_{\gamma'}(f_{pv}(^\circ\gamma')))$ **then**
        return $\langle \gamma', 3 \rangle$; // equivalent path pair
    end if
    if $((R_\gamma(f_{pv}(^\circ\gamma)) \rightarrow R_{\gamma'}(f_{pv}(^\circ\gamma')))$ **then**
        return $\langle \gamma', 1 \rangle$; // extend γ'
    end if
    if $((R_{\gamma'}(f_{pv}(^\circ\gamma')) \rightarrow R_\gamma(f_{pv}(^\circ\gamma)))$ **then**
        return $\langle \gamma, 0 \rangle$; // extend γ
    end if
    Case 2 ($|^\circ\gamma'| - |^\circ\gamma| \geq 1$):
    if $((R_\gamma(f_{pv}(^\circ\gamma)) \equiv R_{\gamma'}(f_{pv}(^\circ\gamma')))$ and $(r_\gamma(f_{pv}(^\circ\gamma)) = r_{\gamma'}(f_{pv}(^\circ\gamma')))$ **then**
        return $\langle \gamma, 4 \rangle$; //equivalent path pair where $(|^\circ\gamma'| - |^\circ\gamma| \geq 1)$
    else
        return $\langle \gamma, 0 \rangle$; // extend γ
    end if
    Case 3 ($|^\circ\gamma| - |^\circ\gamma'| \geq 1$):
    if $((R_\gamma(f_{pv}(^\circ\gamma)) \equiv R_{\gamma'}(f_{pv}(^\circ\gamma')))$ and $(r_\gamma(f_{pv}(^\circ\gamma)) = r_{\gamma'}(f_{pv}(^\circ\gamma')))$ **then**
        return $\langle \gamma, 5 \rangle$; //equivalent path pair and $(|^\circ\gamma| - |^\circ\gamma'| \geq 1)$
    else
        return $\langle \gamma', 1 \rangle$; // extend γ'
    end if
end for
// Control here if $((R_{\gamma'}(f_{pv}(^\circ\gamma')) \not\rightarrow R_\gamma(f_{pv}(^\circ\gamma)))$ and $(((R_\gamma(f_{pv}(^\circ\gamma)) \not\rightarrow R_{\gamma'}(f_{pv}(^\circ\gamma')))) \forall \gamma' \in \Gamma'$
return $\langle \emptyset, 2 \rangle$; // no extension possible

Algorithm 9 SETOFPATHS **findPostPaths** ($flag, \gamma, \Pi'$)

Inputs: If flag = 0, γ is a path of N_0; $\Pi^{'}$ is a path cover of N_0.
If flag = 1, γ is a path of N_1; $\Pi^{'}$ is a path cover of N_1,
$\Pi^{'}$ has paths whose equivalence are still to be found.
Outputs: Set of all paths $\Gamma_E^+ \subseteq \Pi^{'}$ which follow the path γ
$-$ hence γ can be extended through these paths.

SETOFPATHS $\Gamma_E^+ = \emptyset$;
for each $\gamma^{'} \in \Pi^{'}$ **do**
  if ($\gamma^{\circ} \in {}^{\circ}\gamma^{'}$) **then**
    $\Gamma_E^+ \leftarrow \Gamma_E^+ \cup \{\gamma^{'}\}$;
  end if
end for
return Γ_E^+;

Algorithm 11 STRUCT6TUPLE **trimPrePaths** ($flag, \gamma, \Gamma_P, \Pi^{'}, \Pi_n, \eta_t, \Pi, E$)

Inputs: The first parameter is a flag. The second parameter γ: a path whose extension is sought. If flag = 0, it belongs to N_0, if flag = 1, it belongs to N_1. The third parameter Γ_P: a set of pre-paths of the path through which an extension is sought. If flag = 0, it belongs to N_0, if flag = 1, it belongs to N_1. The fourth parameter $\Pi^{'}$: a set of paths remaining from the original path cover. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The sixth parameter Π_n: a set of non-equivalent paths. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The seventh parameter η_t: the set of corresponding transition pairs. The eighth parameter Π: a set of paths in the final path cover. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The ninth parameter E : pair of paths of N_0 and N_1.

Outputs: The output of this function is a six tuple structure. The elements of this structure are as follows: 1. The set Γ_P of trimmed pre-paths of N_0 or N_1 whose equivalent paths are found in stand-alone basis. 2.E, 3.η_t, 4. Π, 5. $\Pi^{'}$ and 6. Π_n.
/* Invoked only after ensuring that extension is possible */

for each $\gamma^{'} \in \Gamma_P - \{\gamma\}$ **do**
  $\langle \gamma, \lambda \rangle \Leftarrow$ **findEqvDCP** ($flag, \gamma^{'}, \eta_t, \Pi^{'}$);
  /* $\lambda = 0, 1$– suggests extension – can be ignored here already being considered for extension */
  if ($\lambda = 3$) **then**
    $\eta_t = \eta_t \bigcup \{ \langle$ last(γ), last($\gamma^{'}$) $\rangle \}$; $E \leftarrow E \bigcup \{\langle \gamma, \gamma^{'} \rangle\}$; $\Pi \leftarrow \Pi \bigcup \{\gamma^{'}\}$; $\Pi^{'} \leftarrow \Pi^{'} - \{\gamma^{'}\}$; $\Gamma_P = \Gamma_P - \{\gamma^{'}\}$; $\eta_p = \eta_p \cup \{\gamma^{\circ}, (\gamma^{'})^{\circ}\}$;
  end if
  if ($\lambda = 4$) **then**
    $\eta_t = \eta_t \bigcup \{ \langle$ last(γ), last($\gamma^{'}$) $\rangle \}$; $E \leftarrow E \bigcup \{\langle \gamma, \gamma^{'} \rangle\}$; $\Pi \leftarrow \Pi \bigcup \{\gamma^{'}\}$; $\Pi^{'} \leftarrow \Pi^{'} - \{\gamma^{'}\}$; $\eta_p = \eta_p \cup \{\gamma^{\circ}, (\gamma^{'})^{\circ}\}$; $\Gamma_P = \Gamma_P - \{\gamma^{'}\}$;
  end if
  if ($\lambda = 5$) **then**
    $\eta_t = \eta_t \bigcup \{ \langle$ last(γ), last($\gamma^{'}$) $\rangle \}$;
    $E \leftarrow E \bigcup \{\langle \gamma, \gamma^{'} \rangle\}$;
    $\Pi \leftarrow \Pi \bigcup \{\gamma^{'}\}$;
    $\Pi^{'} \leftarrow \Pi^{'} - \{\gamma^{'}\}$;
    $\eta_p = \eta_p \cup \{\gamma^{\circ}, (\gamma^{'})^{\circ}\}$;
    $\Gamma_P = \Gamma_P - \{\gamma^{'}\}$;
  end if
  if ($\lambda = 2$) **then**
    $\Pi_n = \Pi_n \cup \{\gamma^{'}\}$;
    $\Gamma_P = \emptyset$;
  end if
end for
return $\langle \Gamma_P, E, \eta_t, \Pi_n, \Pi^{'}, \Pi_n \rangle$;

Algorithm 10 SETofSETSofPATHS **findSetOfSetsOfPrePaths** ($flag, \gamma, \gamma^{'}, \Pi^{'}$)

Inputs: If flag = 0, γ: path of N_0 which triggers extension; $\gamma^{'}$: path of N_0 through which extension of γ is sought; $\Pi^{'}$: path cover of N_0 whose equivalence are still to be found. If flag = 1, γ: path of N_1 which triggers extension; $\gamma^{'}$: path of γ of N_1 through which extension is sought; $\Pi^{'}$: path cover of N_1 whose equivalence are still to be found.
Outputs: Set of all possible subsets of pre-paths of $\gamma^{'}$.
/* Note: Paths leading to $\gamma^{'}$ whose equivalent has been found without extension do not figure in $\Pi^{'}$. */

SETOFPATHS $\Gamma_P = \emptyset$; SETofSETSofPATHS $\chi_P = \emptyset$;
/* Obtain in Γ_P all pre-paths of $\gamma^{'}$ whose post-place is not same as γ° */
for each $\gamma^{''} \in \Pi^{'}$ **do**
  if $((\gamma^{''})^{\circ} \in {}^{\circ}\gamma^{'} \wedge \gamma^{\circ} \neq (\gamma^{''})^{\circ})$ **then**
    $\Gamma_P \leftarrow \Gamma_P \cup \{\gamma^{''}\}$;
  end if
end for
$\Gamma_P = \Gamma_P \cup \{\gamma\}$;
/* Check if Γ_P contains a subset of paths having identical post-place $-$ construct all such subsets for $p \in \Gamma_P^{\circ}$ call it Ψ_p $-$ some members of Ψ_p can be unit sets. */
$\Gamma_P^{\circ} = \Gamma_P^{\circ} - \{p \mid p \in \Gamma_p^{\circ} \wedge \Gamma_P^{\circ} \notin^{\circ} \gamma^{'}\}$;
for each $p \in \Gamma_P^{\circ}$ **do**
  if $(|{}^{\circ}p| > 1)$ **then**
    $\Psi_p = \{ \gamma^{''} \in \Gamma_P | (\gamma^{''})^{\circ} = p \}$;
    $\Gamma_P^{\circ} = \Gamma_P^{\circ} - ({}^{\circ}p)^{\circ}$;
  else
    $\Psi_p = \{ \gamma^{''} \in \Gamma_P | (\gamma^{''})^{\circ} = p \}$;
  end if
end for
/* Construct the Cartesian product of the members of Ψ_p, $p \in \Gamma_P^{\circ}$ $-$ call it χ_P $-$ return χ_P */
$\chi_P = \times_{p \in \Gamma_P^{\circ}}(\Psi_p)$;
return χ_P;

Algorithm 12 Path **extend** $(flag, \gamma^{'}, \Gamma_P)$

Inputs: The first parameter is a flag. The second parameter $\gamma^{'}$: a path through which extension of all the paths in Γ_P is sought. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The third parameter Γ_P: a set of pre-paths of $\gamma^{'}$ which together are extended through $\gamma^{'}$. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1.
Outputs: The extended path $\gamma_e = \Gamma_P.\gamma^{'}$

/* Construct the extended path γ_e */
${}^{\circ}\gamma_e = \emptyset$;
/* Obtain pre-places ${}^{\circ}\gamma^{'}$ which are to figure as the parameter of ${}^{\circ}\gamma_e$ */
${}^{\circ}\gamma^{'} = {}^{\circ}\gamma^{'} - \Gamma_P^{\circ}$;
/* It contains those pre-places of $\gamma^{'}$ paths leading to which have been found to have equivalent paths and hence have not been included in Γ_P. */
${}^{\circ}\gamma_e = {}^{\circ}\Gamma_P \cup {}^{\circ}\gamma^{'}$ /* indicates the pre-places of the paths being extended through $\gamma^{'}$.*/
/* Obtain post-places */
$\gamma_e^{\circ} = (\gamma^{'})^{\circ}$
/* Obtain last transitions */
${}^{\circ}(\gamma_e^{\circ}) = {}^{\circ}((\gamma^{'})^{\circ})$;
/* Obtain R_{γ_e} */
$\overline{v} = \langle r_{\Gamma_P}(f_{pv}({}^{\circ}\Gamma_P)), r_{\gamma^{'}}(f_{pv}({}^{\circ}\gamma^{'}))\rangle$;
$R_{\gamma_e} = \bigwedge_{\gamma_p \in \Gamma_P} R_{\gamma_p}(f_{pv}({}^{\circ}\gamma_p)) \wedge R_{\gamma^{'}}(f_{pv}({}^{\circ}\gamma^{'}))\{\overline{v}/f_{pv}({}^{\circ}\gamma^{'})\}$
/* method of substitution */
$r_{\gamma_e} = r_{\gamma^{'}}(f_{pv}({}^{\circ}\gamma^{'}))\{\overline{v}/f_{pv}({}^{\circ}\gamma^{'})\}$;
/* Obtain r_{γ_e} */
/* Create tree */
return γ_e;

Algorithm 13 SETOFPATHS **prepareForExtension** $(flag, \gamma, \Pi^{'}, \Pi^{''}, \Pi_n, \eta_t, \Pi, E)$

Inputs: The first parameter is a flag. The second parameter γ: a path whose extension is sought. If flag = 0, it belongs to N_0, if flag = 1, it belongs to N_1. The third parameter $\Pi^{'}$: a set of paths remaining from the original path cover. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The fourth parameter $\Pi^{''}$: a set of paths remaining from the original path cover. If flag = 0, it belongs to N_1; if flag = 1, it belongs to N_0. The fifth parameter Π_n: a set of non-equivalent paths. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The sixth parameter η_t: the set of corresponding transitions pairs. The seventh parameter Π: a set of paths in the final path cover. If flag = 0, it belongs to N_0; if flag = 1, it belongs to N_1. The eighth parameter E : pair of equivalent paths of N_0 and N_1.

Outputs: The set $\Pi^{'}$ of paths remaining from the original path cover.

$\Pi^{'} = \Pi^{'} - \{\gamma\}$; /* γ has to be extended */
$\Gamma_E^{+} =$ **findPostPaths** $(flag, \gamma, \Pi^{'})$;
/* The function computes the post-paths of γ through which γ can be extended. Such paths include those which emanate from the post-place γ° (under different guards) or those which emanate from the post-places of the last transition of γ.*/
for each $\gamma^{'} \in \Gamma_E^{+}$ **do**
    $\chi_\gamma =$**findSetOfSetsOfPrePaths** $(flag, \gamma, \gamma^{'}, \Pi^{'})$;
    /*The function computes all the pre-paths of $\gamma^{'}$ other than γ. Some of these may not execute in parallel (with γ). For example, let $\{\gamma, \gamma_1, \gamma_2, \gamma_3\}$ be three such pre-paths of $\gamma^{'}$; let γ_2 and γ_3 have an identical post-place. Hence, γ_2 and γ_3 cannot execute in parallel because the models are one-safe. So the pre-paths (including γ) are decomposed into two subsets $\{\gamma, \gamma_1, \gamma_2\}$ and $\{\gamma, \gamma_1, \gamma_3\}$.*/
    $\Pi^{'} = \Pi^{'} - \{\gamma^{'}\}$;
    for each $\Gamma_P \in \chi_\gamma$ **do**
        $\Gamma_P =$ **trimPrePaths** $(flag, \gamma, \Gamma_P, \Pi^{''}, \Pi^{'}, \Pi_n, \eta_t, \Pi, E)$;
        /* Each of the subsets of pre-paths is trimmed of the members (other than γ) which are found to have equivalence (without any extension) with some path in $N_0(N_1)$. If it is detected that such a path may have to be extended before its equivalence is found, then no action is initiated because they are already under consideration for extension. However, if it is found that the path does not merit any further consideration (such as extension), it is put in the set $\Pi_{n,0}(\Pi_{n,1})$ of paths of $N_0(N_1)$ which may have no equivalent path in $N_0(N_1)$. The set is not used for extension. */
        if $(\Gamma_P \neq \emptyset)$ **then**
            $\gamma_e =$ **extend** $(flag, \gamma, \Gamma_P, \gamma^{'})$;
            /* The function constructs an extended path γ_e of the form $\Gamma_P.\gamma^{'}$. The function computes those pre-places of $\gamma^{'}$ paths leading to which have been found to have equivalent paths and hence do not occur in Γ_P. Then the function computes the pre-places of γ_e. In the next two steps, the function obtains the post-places of γ_e as those of $\gamma^{'}$ and the last transition of γ_e as that of $\gamma^{'}$. After that, by method of substitution the function computes the condition R_{γ_e} of execution and the data transformation r_{γ_e} along the extended path γ_e. Finally, it returns the extended path γ_e *
            $\Pi^{'} = \Pi' - \Gamma_P \cup \{\gamma_e\}$;
        end if
    end for
end for
return $\Pi^{'}$;

Algorithm 14 STRUCT6TUPLE **checkEqDCP**(N_0, N_1)

Inputs: The PRES+ models N_0 and N_1.
Outputs: The output of this function is a six tuple structure. The elements of this structure are as follows: 1. Π_0: the final path cover of N_0, 2. Π_1: the final path cover of N_1 corresponding to Π_0, 3. E : a set of ordered pairs $\langle \delta_0, \delta_1 \rangle$ of concatenation of parallel paths of Π_0 and Π_1 respectively, such that $\delta_0 \simeq \delta_1$. 4.η_t: the set of corresponding transition pairs; 5. $\Pi_{n,0}$: the set of paths of N_0 for which no equivalent is found in N_1 even with extension. 6. $\Pi_{n,1}$: the set of paths of N_1 for which no equivalent is found in N_0 even with extension.

Let $\eta_p = \{\langle p, p' \rangle \mid p \in inP_0 \wedge p' \in inP_1 \wedge p' = f_{in}(p)\}$;
  Let η_t, the set of pairs of corresponding transitions, be $\emptyset$;
  $\Pi_0' =$ **constAllPathsDCP**(N_0); $\Pi_1' =$ **constAllPathsDCP**(N_1);
  Let Π_0, Π_1,$\Pi_{n,0}$, $\Pi_{n,1}$ and E be empty;
for each $\alpha \in \Pi_0'$ **do**
  $\langle \beta, \lambda \rangle \Leftarrow$ **findEqvDCP** $(0, \alpha, \eta_t, \Pi_1', f_{in})$;
  if $(\lambda = 3)$ **then**
    $\eta_t = \eta_t \bigcup \{ \langle \text{last}(\alpha), \text{last}(\beta) \rangle \}$; $E \leftarrow E \bigcup \{\langle \alpha, \beta \rangle\}$; $\Pi_0 \leftarrow \Pi_0 \bigcup \{\alpha\}$; $\Pi_0' \leftarrow \Pi_0' - \{\alpha\}$; $\Pi_1 \leftarrow \Pi_1 \cup \{\beta\}$; $\Pi_1' \leftarrow \Pi_1' - \{\beta\}$; $\eta_p = \eta_p \cup \{{}^\circ\alpha, {}^\circ\beta\}$; /* $\beta \simeq \alpha$ */
  else
    if $(\lambda = 0)$ **then**
      /* extend α */
      $\Pi_0' =$ **prepareForExtension** $(0, \alpha, \Pi_0', \Pi_1', \Pi_{n,0}, \eta_t, \Pi_{n_\alpha}, E)$; // The extended path got inserted and their constituent paths got deleted from Π_0' by the above function
    end if
  else
    if $(\lambda = 4)$ **then**
      $\eta_t = \eta_t \bigcup \{ \langle \text{last}(\alpha), \text{last}(\beta) \rangle \}$; $E \leftarrow E \bigcup \{\langle \alpha, \beta \rangle\}$; $\Pi_0 \leftarrow \Pi_0 \bigcup \{\alpha\}$; $\Pi_0' \leftarrow \Pi_0' - \{\alpha\}$; $\eta_p = \eta_p \cup \{{}^\circ\alpha, {}^\circ\beta\}$; /* $\alpha \simeq \beta$ and $(|{}^\circ\alpha| - |{}^\circ\beta| \geq 1)$ */
    end if
  else
    if $(\lambda = 5)$ **then**
      $\eta_t = \eta_t \bigcup \{ \langle \text{last}(\alpha), \text{last}(\beta) \rangle \}$; $E \leftarrow E \bigcup \{\langle \alpha, \beta \rangle\}$; $\Pi_1 \leftarrow \Pi_1 \bigcup \{\beta\}$; $\Pi_1' \leftarrow \Pi_1' - \{\beta\}$; $\eta_p = \eta_p \cup \{{}^\circ\alpha, {}^\circ\beta\}$; /* $\alpha \simeq \beta$ and $(|{}^\circ\beta| - |{}^\circ\alpha| \geq 1)$ */
    end if
  else
    if $(\lambda = 1)$ **then**
      $\Pi_1' =$ **prepareForExtension** $(1, \beta, \Pi_1', \Pi_0', \Pi_{n,0}, \eta_t, \Pi_{n_\beta}, E)$; /* extend β */
      // The extended path got inserted and their constituent paths got deleted from Π_1' by the above function
    end if
  else
    if $(\lambda = 2)$ **then**
      $\Pi_{n,0} = \Pi_{n,0} \cup \{\alpha\}$; /* no scope for extension - α may have no equivalent paths */ $\Pi_0' = \Pi_0' - \{\alpha\}$;
    end if
  end if
end for /* $\forall \alpha \in \Pi_0'$ */
$\Pi_{n,1} = \Pi_1' - \Pi_1$;
Case 1 $((\Pi_{n,0} = \emptyset)$ and $(\Pi_{n,1} = \emptyset))$:
    Report "N_0 and N_1 are the equivalent models."
    break;
Case 2$((\Pi_{n,0} = \emptyset)$ and $(\Pi_{n,1} \neq \emptyset))$:
    Report "$N_0 \sqsubseteq N_1$ and $N_1 \not\sqsubseteq N_0$."
    break;
Case 3$((\Pi_{n,0} \neq \emptyset)$ and $(\Pi_{n,1} = \emptyset))$:
    Report "$N_1 \sqsubseteq N_0$ and $N_0 \not\sqsubseteq N_1$."
    break;
Case 4 $((\Pi_{n,0} \neq \emptyset)$ and $(\Pi_{n,1} \neq \emptyset))$:
    Reports "two models may not be equivalent."
return $\langle \Pi_0, \Pi_1, E, \eta_t, \Pi_{n,0}, \Pi_{n,1} \rangle$;

Complexity of Algorithm 11 (`trimPrePaths`): Step 2 calls `findEqvDCP` function which takes $O((max((|T|log|T|),(|P|log|P|))+|F|+|P|).|T|.|P|)$ time as explained above. Depending on the returned flag value, step 3 or 6 or 9 or 12 is executed. Step 4 or 7 or 10 updates η_t, η_p, E and Π by union operation which take $O(1)$ time and updates $\Pi^{'}$ and Γ_P by deletion which takes $O(|T|.|P|)$ time. Step 13 updates Π_n which takes $O(|T|.|P|)$ time. Step 1 involves a loop which iterates $|T|.|P|$ times. Hence, the overall complexity is $O\left(\{max(|T|log|T|, |P|log|P|)\} + |F| + |T|.|P|.|T|^2.|P|^2\right)$.

Functional Specification of Algorithm 12 (`extend`): The function constructs an extended path γ_e of the form $\Gamma_P.\gamma^{'}$. The function computes those pre-places of $\gamma^{'}$ paths leading to which have been found to have equivalent paths and hence do not occur in Γ_P. Then the function computes the pre-places of γ_e. In the next two steps, the function obtains the post-places of γ_e as those of $\gamma^{'}$ and the last transition of γ_e as that of $\gamma^{'}$. After that, by method of substitution the function computes the condition R_{γ_e} of execution and the data transformation r_{γ_e} along the extended path γ_e. Finally, it returns the extended path γ_e.

Complexity of Algorithm 12 (`extend`): Step 1 initializes $^{\circ}\gamma_e$ to empty in $O(1)$ time. Step 2 computes the pre-places of γ_e and it takes $O(|P|)$ time. Similarly, steps 3 and 4 compute the post-places and last transition of γ_e in $O(|P|)$ and $O(1)$ time, respectively. Step 5 computes R_{γ_e} and r_{γ_e} and it takes $O\left(2^{(|F|)}\right)$ time where, $|F|$ be the length of the normalized formula. Hence, the overall complexity of this function is $O\left(2^{(|F|)}\right)$ which dominates the complexity of all other steps.

Functional Specification of Algorithm 13 (`prepareForExtension`): The function extends a path γ in all possible ways and updates the original path cover $\Pi^{'}$ following extensions of a path by deleting all the pre-paths and the post-path which participated in the extension and adding all the extended paths. It first deletes the path γ which is to be extended from $\Pi^{'}$. It then calls `findPostPaths` to obtain the set of all post-paths of γ in Γ_E^{+}. For each post-path $\gamma^{'}$ of Γ_E^{+}, it first deletes $\gamma^{'}$ from $\Pi^{'}$ and then obtains the mutually exclusive sets of pre-paths using the function `findSetOfSetsOfPrePaths`. For each of the mutually exclusive subsets Γ_P of pre-paths, the function then calls `trimPrePaths` to remove those members which have equivalent paths in the other model. In the next step, it calls `extend` function to obtain an extended path of the form $(\Gamma_P).\gamma^{'}$. It then updates the path cover $\Pi^{'}$ by deleting the pre-paths of Γ_P from $\Pi^{'}$ and adding the extended path γ_e. Finally,the function `prepareForExtension` returns Π'.

Complexity of Algorithm 13 (`prepareForExtension`): Step 1 takes $O(|T|.|P|)$ time. Step 2 calls the function `findPostPaths` which takes $O\left(|P|^2.|T|\right)$ time as explained above. For each of the post-paths in Γ_E^+, Step 4 calls `findSetOfSetsOfPrePaths` function which also takes $O\left(2^{\left(\frac{|T|}{2}\right)}\right)$ time as explained above. Step 5 takes $O(|T|.|P|)$ time. For each set of pre-paths, Step 7 invokes `trimPrePaths` function and it takes $O\left(\{max(|T|log|T|, |P|log|P|) + |F| + |T|.|P|\}.|T|^2.|P|^2\right)$ as explained above. Step 8 checks a condition in $O(1)$ time. If the condition is true, step 9 calls the `extend` function which takes $O\left(2^{|F|}\right)$ time. Then, in step 10, the deletion operation takes $O\left(|T|^2.|P|^2\right)$ time and addition of γ_e takes $O(1)$ time. The loop in step 6 iterates $O(|T|.|P|)$ time. Hence, the overall complexity of the inner loop (steps 6–12) is $O\left(2^{(|F|)}.|T|.|P|\right)$. The loop involved in step 3 also iterates $O(|T|.|P|)$ time. Therefore, the overall complexity is $O\left(\left(\left((2^{\frac{|T|}{2}} + 2^{|F|}\right).|T|.|P|\right).|T|.|P|\right)$.

Functional Specification of Algorithm 14 (`checkEqDCP`): The functional specification of this module is given in section 5.2.

Complexity of Algorithm 14 (`checkEqDCP`): In step 1, construction of η_p takes $O(|P|)$ time. In the same step the function constructs all the paths for the two PRES+ models in $O((\frac{|T|}{|P|})^{|P|}.(|T|^2))$ as given in Chapter 4. Step 3 uses `findEqvDCP` function and takes $O((\{max(|T|log|T|, |P|log|P|)\} + |F|).|T|.|P|)$. time as explained before. The complexity of each iteration of the loop of step 2 is as follows. Checking of proper condition on the flag λ (steps $4, 7, 11, 15, 19, 23$) is $O(1)$. Statements $5, 12, 16, 24$ involves union operation and deletion from sets Π_0', Π_1' which take $O(1)$ and $O(|T|.|P|)$ times, respectively. Hence for cases $\lambda = 2, 3, 4, 5$, time taken is $O(|T|.|P|)$. Cases $\lambda = 0$ and 1, the function `prepareForExtension` takes:
$O\left(\left(2^{(\frac{|T|}{2})} + 2^{|F|)}.|T|.|P|\right)|T|.|P|\right)$.
Hence the body of the loop of step 2 takes:
$O\left(\left(2^{(\frac{|T|}{2})} + 2^{|F|)}.|T|.|P|\right)|T|.|P|\right)$ time. The loop iterates $O(|T|.|P|)$ time. Hence the complexity is $O\ ((2^{(\frac{|T|}{2})} + 2^{|F|)}.|T|.|P|).|T|^2.|P|^2)$. Step 28 takes $O(|T|^2.|P|^2)$ time. From step 29, there are four cases and each case takes $O(1)$ time. Hence, the overall complexity of the `checkEqDCP` function is $O\ ((2^{(\frac{|T|}{2})} + 2^{|F|)}.|T|.|P|).|T|^2.|P|^2) + O((\frac{|T|}{|P|})^{|P|}.(|T|^2))$.

5.2.3 Soundness of the equivalence checking algorithm

The soundness proof hinges upon the following two lemmas.

Lemma 5. *Let C be a parallel combination of concatenated paths which is of the form $\gamma_1||\gamma_2||\ldots||\gamma_t$ such that $^\circ(\gamma_i) \cap {}^\circ(\gamma_j) = \emptyset$, $1 \le i \ne j \le t$. For any i, $1 \le i \le t$, let the concatenated path γ_i be of the form $C'_i.\gamma'_i$, where γ'_i is also a concatenated path contained in γ_i and $C'_i = \{\gamma'_{1,i}||\gamma'_{2,i}||\ldots||\gamma'_{n_i,i}\}$ is a set of parallel concatenated paths such that $(C')^\circ = {}^\circ\gamma'_i$. Then $(C - \{\gamma'_i\}).\gamma'_i \equiv C, 1 \le i \le t$.*

Proof. $C - \{\gamma'_i\} = \gamma_1||\gamma_2||\ldots||\gamma_{i-1}||(\gamma'_{1,i}||\gamma'_{2,i}||\ldots||\gamma'_{n,i})||\gamma_{i+1}||\ldots||\gamma_t$
$= (\gamma_1||\gamma_2||\ldots||\gamma_{i-1}||\gamma_{i+1}||\ldots||\gamma_t)||(\gamma'_{1,i}||\gamma'_{2,i}||\ldots||\gamma'_{n,i})$ (by commutativity of parallel paths)
Hence,
$(C - \{\gamma'_i\}).\gamma'_i = \{(\gamma_1||\gamma_2||\ldots||\gamma_{i-1}||\gamma_{i+1}||\ldots||\gamma_t)||(\gamma'_{1,i}||\gamma'_{2,i}||\ldots||\gamma'_{n,i})\}.\gamma'_i$
$= (\gamma_1||\gamma_2||\ldots||\gamma_{i-1}||\gamma_{i+1}||\ldots||\gamma_t)||\{(\gamma'_{1,i}||\gamma'_{2,i}||\ldots||\gamma'_{n,i}).\gamma'_i\}$
$= (\gamma_1||\gamma_2||\ldots||\gamma_{i-1}||\gamma_{i+1}||\ldots||\gamma_t||\gamma_i) = C$ (by commutativity pf parallel paths). □

Lemma 6. *If Π'_0 (Π'_1) is a path cover of N_0 (N_1) and the function `checkEqDCP` (Algorithm 14) reaches step 29, then so is Π_0 (Π_1).*

Proof. The lemma is proved for Π_0; proof for Π_1 follows identically. Consider any computation $\mu_{0,p}$ of an out-port p of N_0. If $\mu_{0,p}$ can be expressed as a concatenation of parallelizable paths taken from Π_0, then we are done. Since Π'_0 is a path cover of N_0, $\mu_{0,p}$ can be expressed as a concatenation, C'_0 say, of parallelisable paths taken from Π'_0. The function `constructConcatenatedParallelizablePath` (Algorithm 15) constructs the (desired) concatenation C_0 of parallelisable paths of Π_0 from C'_0. The inputs to this function are C'_0 and the set E of pairs of equivalent paths of N_0 and N_1 obtained from the equivalence checking phase. The output of the function is C_0.

The function starts by initializing the concatenation C_0 to empty. Let $last(C'_0)$ ($last(C_0)$) represent the last set of parallelisable paths of C'_0 (C_0). To start with, last(C'_0) is a singleton. Dynamically, for each path γ' in last(C'_0), the function checks whether γ' occurs as the first member of some pair in E. If so, the function updates C'_0 by deleting γ' from C'_0 (i.e., from last(C'_0)) and concatenating γ' with C_0 (i.e., ahead of C_0). Otherwise, the function searches for an extended path γ_e containing γ', occurring as the first member of some pair in E. The fact that such a path surely exists follows from the fact that the function `checkEqDCP` (Algorithm 14) reaches step 29 and hence the loop (steps 2 to 27) of the function terminates, i.e., $\Pi_0^{'}$ is rendered empty; it is given that $\Pi_{n,0}$ is $\emptyset$; hence, every path of $\Pi_0^{'}$, either individually or in extended form, has been put as a first member in E. On finding γ_e, the function updates $C_0^{'}$ by deleting all the paths of $\Pi_0^{'}$ occurring in γ_e from $C_0^{'}$; it then updates C_0 by concatenating γ_e with C_0.

Algorithm 15 PATH **constructConcatenatedParallelizablePath** $(C_O^{'}, E)$

Inputs: The first parameter is a concatenated parallelisable paths of $\Pi_0^{'}$ such that $C_0^{'} \equiv \mu_{0,p}$.
The second parameter E : pair of paths of N_0 and N_1 obtained from equivalence checking phases.
Outputs: The output of this function is a concatenation of parallelisable paths of Π_0 such that $C_0 \equiv C_0^{'}$.

1: $C_0 = \emptyset$;
2: **for** each $\gamma^{'} \in$ last($C_0^{'}$) **do**
3:     **if** ($\gamma^{'}$ = first member of some pair in E) **then**
4:         $C_0^{'} = C_0^{'} - \{\gamma^{'}\}$;
5:         $C_0 = \gamma^{'}.C_0$;
6:     **else**
7:         Let γ_e be the first member of E that contains $\gamma^{'}$;
        Let γ_e be $(\gamma_{e,1} \parallel \gamma_{e,2} \parallel \ldots \parallel \gamma_{e,l}).\gamma^{'}$;
        Let $\{\gamma_e\}$ be $\gamma_{e,1}||\gamma_{e,2}||\ldots||\gamma_{e,l}$;
8:         **for** each $\gamma^{''} \in \{\gamma_e\}$ **do**
9:             $C_0^{'} = C_0^{'} - \{\gamma^{''}\}$;
10:        **end for**
11:        $C_0 = \gamma_e.C_0$;
12:    **end if**
13: **end for**
14: **return** C_0;

The fact that C_0 comprises only the first members of the pairs in E is clear from the conditions associated with the if statement in step 3 and the assignment in steps 5 and 11 of (Algorithm 15) which are the only steps where addition to C_0 takes place; so, for any paths of $\Pi_0^{'}$, the function has found an equivalent path, either for itself or after extending it. The first members of E belong to Π_0. Hence, C_0 contains only paths of Π_0.

Now, it is required to show that $C_0 \equiv C_0^{'}$. Let $C_{0,i}^{'}$ ($C_{0,i}$) be the value of $C_0^{'}$ (C_0) after the i^{th} iteration of the loop (steps 2 – 13); note that $C_{0,0}^{'} = C_0^{'}$ and $C_{0,0} = \emptyset$ are the initial values of $C_0^{'}$ and C_0, respectively. Let the loop (steps 2 – 13) iterate $f+1$ times. We show that $C_0^{'} \equiv C_{0,i}^{'}.C_{0,i}, 0 \leq i \leq f$. In the loop, $C_0^{'}$ shrinks in size (in steps 4 and 9), by losing its paths from its last set $last(C_0^{'})$. So when the loop terminates, $last(C_0^{'})$ must be empty and hence $C_{0,f}^{'} = \emptyset$. Hence, once the above equivalence is proved, for $i = f$, $C_0^{'} \equiv C_{0,f}^{'}.C_{0,f} \equiv C_{0,f} \equiv C_0$ and we have the desired C_0. We now show the equivalence $C_0^{'} \equiv C_{0,i}^{'}.C_{0,i}, 0 \leq i \leq f$, by induction on i.

Basis ($i = 0$): $C'_{0,0}.C_{0,0} \equiv C'_{0,0}.\emptyset \equiv C'_{0,0} \equiv C'_0$.

Induction hypothesis: Let $\forall i, 0 \leq i \leq k$, the statement $C'_0 \equiv C'_{0,i}.C_{0,i}$ be true, for any $k < f$.

Induction step:

Case 1: Let the $(k+1)^{th}$ iteration find the condition associated with step 3 to hold, i.e., γ' occurs as the first member of some pair in E whereupon it goes through steps 4 and 5. From step 4, $C'_{0,k+1} = C'_{0,k} - \{\gamma'\}$ and from step 5, $C_{0,k+1} = \gamma'.C_{0,k}$, where γ' has an equivalence with some path in N_1 (by construction of E). From the definition of set of parallelisable paths (Definition 18) and Lemma 5,
$C'_{0,k+1}.C_{0,k+1} \equiv (C'_{0,k} - \{\gamma'\}).(\gamma'.C_{0,k})$ (by steps 4 and 5 of Algorithm 15)
$\equiv ((Prefix(C'_{0,k}).last(C'_{0,k})) - \{\gamma'\}).(\gamma'.C_{0,k})$
(where $Prefix(C'_0)$ is the concatenation C'_0 minus $last(C'_0)$)
$\equiv (Prefix(C'_{0,k}).(last(C'_{0,k}) - \{\gamma'\})).(\gamma'.C_{0,k})$
(since $\gamma' \in last(C'_{0,k})$, deleting γ' from $C'_{0,k}$ only affects $last(C'_{0,k})$)
$\equiv (Prefix(C'_{0,k}).((last(C'_{0,k}) - \{\gamma'\}).\gamma')).C_{0,k}$
(by associativity of concatenation)
$\equiv (Prefix(C'_{0,k}).(last(C'_{0,k}))).C_{0,k}$
(by Lemma 5 applied on last($C'_{0,k}$) which is a
set of parallel paths containing a single path γ')
$\equiv C'_{0,k}.C_{0,k}$
$\equiv C'_0$ (by induction hypothesis).

Case 2: Let the $(k+1)^{th}$ iteration find the negation of the condition associated with step 3 to hold. Let the concatenated path γ_e containing γ' found in step 7 be of the form $(\gamma_{e,1}||\gamma_{e,2}||\ldots||\gamma_{e,l}).\gamma')$.

After the loop (steps 8 – 10),

$$C'_{0,k+1} = C'_{0,k} - \{\gamma_e\}. \tag{5.1}$$

From lemma 5, applied repeatedly for each execution of step 9 in the loop, we have,

$$C'_{0,k} \equiv (C'_{0,k} - \{\gamma_e\}).\gamma_e \equiv C'_{0,k+1}.\gamma_e.(from\ (5.1)) \tag{5.2}$$

After execution of step 11,

$$C_{0,k+1} \equiv \gamma_e.C_{0,k}. \tag{5.3}$$

Hence, $C'_{0,k+1}.C_{0,k+1} \equiv (C'_{0,k} - \{\gamma_e\}).(\gamma_e.C_{0,k})$ (from 5.1 and 5.3)
$\equiv \{(C'_{0,k} - \{\gamma_e\}).\gamma_e\}.C_{0,k}$ (associativity of concatenation)
$\equiv C'_{0,k}.C_{0,k}$ (from 5.2)
$\equiv C'_0$ (by induction hypothesis)

□

Theorem 8. *If the function* `checkEqDCP` *(Algorithm 14) reaches step 30 and* (a) *returns* $\Pi_{n,0} = \emptyset$*, then* $N_0 \sqsubseteq N_1$ *and* (b) *if it returns* $\Pi_{n,1} = \emptyset$*, then* $N_1 \sqsubseteq N_0$*.*

Proof. We give the proof of part (a) below; that of part (b) follows identically. From Lemma 6, we can conclude that Π_0 gives a path cover of N_0. Hence, for any out-port p of N_0, any computation $\mu_{0,p}$ can be represented as a concatenation, $Q_{0,0}.Q_{0,1}.\cdots.Q_{0,l}$ say, of sets of parallel paths, such that $p \in Q^{\circ}_{0,l}$, ${}^{\circ}Q_{0,0} \subseteq inP_0$, $Q_{0,l}$ is a singleton $\{\alpha_l\}$, say, and $Q_{0,i}$ contains only paths from Π_0, $0 \leq i \leq l$. Whenever a path α is introduced in Π_0 (only in steps 5 and 12 of Algorithm 14) an entry $\langle \alpha, \beta \rangle$ is introduced in E (with $\alpha \simeq \beta$). Hence, for any path $\alpha \in Q_{0,i}$ for some i, there exists a path β of N_1 such that $\langle \alpha, \beta \rangle \in E$. Hence, we can construct a concatenation, $C_{1,p'}$ say, of parallel paths of N_1 such that $C_{1,p'} = Q_{1,0}.Q_{1,1}.\cdots.Q_{1,l}$, where $Q_{1,i} = \{\beta \mid \langle \alpha, \beta \rangle \in E \text{ and } \alpha \in Q_{0,i}\}$. We show that (1) $C_{1,p'}$ is a computation of $f_{out}(p)$ in N_1 and (2) $C_{1,p'} \simeq \mu_{0,p}$.

Proof of (1)**:** $C_{1,p'}$ is alternatively rewritten as a sequence of sets of places, namely, $\langle {}^{\circ}Q_{1,0}, {}^{\circ}Q_{1,1}, \ldots, {}^{\circ}Q_{1,i}, {}^{\circ}Q_{1,i+1}, \ldots, {}^{\circ}Q_{1,l}, Q^{\circ}_{1,l} \rangle$. It is required to prove $(a) {}^{\circ}Q_{1,0} \subseteq inP_1$, $(b) f_{out}(p) \in Q^{\circ}_{1,l}$, $(c) Q^{\circ}_{1,i+1} = (Q^{\circ}_{1,i})^+, 0 \leq i \leq l$.

Proof of (a)**:** A pair $\langle \alpha, \beta \rangle$ of paths is put in E by the function `checkEqDCP` (Algorithm 14) (steps 5,12 and 16) if they are ascertained to be equivalent by the function `findEqvDCP` (Algorithm 8); the latter will examine their equivalence only if they are found to satisfy the property ${}^{\circ}\alpha \subseteq inP_0 \Rightarrow {}^{\circ}\beta \subseteq inP_1$ and ${}^{\circ}\beta = f_{in}({}^{\circ}\alpha)$ by the function `findCandidate` (Algorithm 7 – step 7). Hence, ${}^{\circ}Q_{0,0} \subseteq inP_0 \Rightarrow {}^{\circ}Q_{1,0} \subseteq inP_1$ and ${}^{\circ}Q_{1,0} = f_{in}({}^{\circ}Q_{0,0})$.

Proof of (b)**:** By construction of $C_{1,p'}$ from $\mu_{0,p}$, $Q_{1,l} = \{\beta_l\}$ is a singleton and if $Q_{0,l} = \{\alpha_l\}$, then $\langle \alpha_l, \beta_l \rangle \in E$ (by construction of $C_{1,p'}$). Again, by a similar reasoning as in proof of (a), we find that `findCandidate` (Algorithm 7) ensures in steps 26-30, that $\alpha_l^\circ \in outP_0 \Rightarrow \beta_l^\circ (= Q_{1,l}^\circ) \in f_{out}(\alpha_l^\circ) = f_{out}(p)$.

Proof of (c): Given $C_{1,p'} = Q_{1,0}.Q_{1,1}.\ldots.Q_{1,l}$. We construct the corresponding sequence of subsets of marking $\rho = \langle M_0, M_1, \ldots, M_l, M_{l+1} \rangle$ such that

$$P_{M_0} = {}^\circ C_{1,p'} \tag{5.4}$$

$$\begin{aligned} \forall i, 0 \le i \le l, P_{M_{i+1}} &= \{p \mid p \in Q_{1,i}^\circ \cap {}^\circ Q_{1,i+1}\} \ldots (a) && (5.5)\\ &\cup \{p \mid p \in Q_{1,i}^\circ - {}^\circ Q_{1,i+1}\} \ldots (b) && (5.6)\\ &\cup \{p \mid p \in P_{M_i} - {}^\circ Q_{1,i}\} \ldots (c) && (5.7) \end{aligned}$$

$$P_{M_l} = Q_{1,l}^\circ \tag{5.8}$$

Now, $C_{1,p'}$ is a computation of N_1 if ρ is a computation of p' (by alternate definition of computation, section 3.2); hence it is required to prove $(I) P_{M_0} \subseteq inP_1$, $(II) P_{M_{l+1}} = \{p'\}$, $(III) P_{M_{i+1}} = P_{M_i^+}, 0 \le i < l$.

Proof of (I) and (II): (I) and (II) are already proved in part (a) and part (b).

Proof of (III): If $p \in P_{M_{i+1}}$ by clause 5.5(a) or 5.5(b), then $p \in Q_{1,i}^\circ$, where $Q_{1,i} = T_{M_i}$, the set of enabled transitions from marking M_i. If $p \in P_{M_{i+1}}$ by clause 5.5(c), then $p \in P_{M_i}$ and $p \notin {}^\circ Q_{1,l} \Rightarrow p \in P_{M_i}$ and $p \notin T_{M_i}$. Thus, the set $P_{M_{i+1}}$ of places satisfies the first clause of definition (Definition 1) of place successor marking. Hence $P_{M_{i+1}} = P_{M_i^+}$.

Proof of (2): In $C_{1,p'}$, $\forall p_1 \in Q_{1,i}^\circ$, $0 \le i \le l$, there exists a concatenated path γ_{p_1} of the form $Q_{1,0}^{(p_1)}.Q_{1,1}^{(p_1)}.\ldots.Q_{1,i}^{(p_1)}$ such that $(Q_{1,i}^{p_1})^\circ = \{p_1\}$ (by Definition 19 of concatenated paths in subsection 4.1.1). The path γ_{p_1} has a condition of execution $R_{\gamma_{p_1}}^{Q_{1,0}}(f_{pv}({}^\circ\gamma_{p_1}))$ and the data transformation $r_{\gamma_{p_1}}^{Q_{1,0}}(f_{pv}({}^\circ\gamma_{p_1}))$ (as explained in subsection 4.1.1). Similarly, in $\mu_{0,p}$, $\forall p_0 \in Q_{0,i}^\circ$, $0 \le i \le l$, we can have a path γ_{p_0}. So, to prove that $C_{1,p'} \simeq \mu_{0,p}$, we have to show that $\forall i, 0 \le i \le l$, $\forall p_1 \in Q_{1,i}^\circ$, $\exists p_0 \in Q_{0,i}^\circ$ such that $\langle p_0, p_1 \rangle \in \eta_p$, $R_{\gamma_{p_1}}^{Q_{1,0}}(f_{pv}({}^\circ\gamma_{p_1})) \equiv R_{\gamma_{p_0}}^{Q_{0,0}}(f_{pv}({}^\circ\gamma_{p_0}))$ and $r_{\gamma_{p_1}}^{Q_{1,0}}(f_{pv}({}^\circ\gamma_{p_1})) = r_{\gamma_{p_0}}^{Q_{0,0}}(f_{pv}({}^\circ\gamma_{p_0}))$ by induction on i.

Basis ($i = 0$): $\forall p_1 \in Q^{\circ}_{1,0}, \exists \beta \in Q_{1,0}$, such that $\beta^{\circ} = \{p_1\}$. So, $\gamma_{p_1} = \beta$. By construction of $C_{1,p'}$ from $\mu_{0,p}$, $\exists \alpha, \langle \alpha, \beta \rangle \in E$ and $\alpha \in Q_{0,0}$. Let $\{p_0\}$ be α°; so $\alpha = \gamma_{p_0}$. As $\langle \alpha, \beta \rangle \in E$, $R_{\alpha}(f_{pv}(^{\circ}\alpha) \equiv R_{\beta}(f_{pv}(^{\circ}\beta)$ and $r_{\alpha}(f_{pv}(^{\circ}\alpha) = r_{\beta}(f_{pv}(^{\circ}\beta)$ (ensured by the function `findEqvDCP` (Algorithm 8)). Therefore, $R^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1})) \equiv R^{Q_{0,0}}_{\gamma_{p_0}}(f_{pv}(^{\circ}\gamma_{p_0}))$ and $r^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1})) = r^{Q_{0,0}}_{\gamma_{p_0}}(f_{pv}(^{\circ}\gamma_{p_0}))$; also, $\langle p_0, p_1 \rangle \in \eta_p$ as ensured by *checkEqDCP*.

Induction Hypothesis: Let $\forall i, 0 \le i \le k < l, \forall p_1 \in Q^{\circ}_{1,i}, \exists p_0 \in Q^{\circ}_{0,i}$ such that the properties $R^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1})) \equiv R^{Q_{0,0}}_{\gamma_{p_0}}(f_{pv}(^{\circ}\gamma_{p_0}))$,
$r^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1})) = r^{Q_{0,0}}_{\gamma_{p_0}}(f_{pv}(^{\circ}\gamma_{p_0}))$ and $\langle p_0, p_1 \rangle \in \eta_p$ hold.

Induction Step: Required to prove that $\forall p_1 \in Q^{\circ}_{1,k+1}, \exists p_0 \in Q^{\circ}_{0,k+1}$ such that $R^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1})) \equiv R^{Q_{0,0}}_{\gamma_{p_0}}(f_{pv}(^{\circ}\gamma_{p_0}))$, $r^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1})) =$
$r^{Q_{0,0}}_{\gamma_{p_0}}(f_{pv}(^{\circ}\gamma_{p_0}))$ and $\langle p_0, p_1 \rangle \in \eta_p$. Let $\gamma_{p_1} = C'_1.\beta$, where $\beta^{\circ} = \{p_1\}$ and C'_1 is a set of parallelisable paths such that $(C'_1)^{\circ} = {}^{\circ}\beta$. Let $C'_1 = \beta_1 || \beta_2 || \ldots || \beta_{t_1}$. Now, $\exists \alpha, \langle \alpha, \beta \rangle \in E$ (by construction of $C_{1,p'}$ from $\mu_{0,p}$). Therefore, as ensured by the function `findEqvDCP` (Algorithm 8) $R_{\alpha}(f_{pv}(^{\circ}\alpha)) \equiv R_{\beta}(f_{pv}(^{\circ}\beta))$, $r_{\alpha}(f_{pv}(^{\circ}\alpha)) = r_{\beta}(f_{pv}(^{\circ}\beta))$ and $\langle \alpha^{\circ}, \beta^{\circ} \rangle$. Let p_0 be α°. $\gamma_{p_0} = C'_0.\alpha$, where C'_0 is a set of parallelisable paths of the form $\alpha_1 || \alpha_2 || \ldots || \alpha_{t_1}$ such that $\langle \alpha_i, \beta_i \rangle \in E, 1 \le i \le t_1$. Therefore,

$R^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1}))$
$\equiv \bigwedge_{i=1}^{t_1} R_{\beta_i}(f_{pv}(^{\circ}\beta_i)) \wedge R_{\beta}(f_{pv}(^{\circ}\beta))\{\overline{v_1}/f_{pv}(^{\circ}\beta)\}$
where, $\overline{v_1} = \langle r_{\beta_1}(f_{pv}(^{\circ}\beta_1)), r_{\beta_2}(f_{pv}(^{\circ}\beta_2)), \ldots, r_{\beta_{t_1}}(f_{pv}(^{\circ}\beta_{t_1})) \rangle)$
$\equiv \bigwedge_{i=1}^{t_1} R_{\alpha_i}(f_{pv}(^{\circ}\alpha_i)) \wedge R_{\alpha}(f_{pv}(^{\circ}\alpha))\{\overline{v_0}/f_{pv}(^{\circ}\alpha)\}$
where, $\overline{v_0} = \langle r_{\alpha_1}(f_{pv}(^{\circ}\alpha_1)), r_{\beta_2}(f_{pv}(^{\circ}\alpha_2)), \ldots, r_{\alpha_{t_1}}(f_{pv}(^{\circ}\alpha_{t_1})) \rangle$;
$\equiv R^{Q_{0,0}}_{C_{p_0}}(f_{pv}(^{\circ}C_{p_1}))$.
Since by induction hypothesis $r_{\alpha_j}(f_{pv}(^{\circ}\alpha_j)) = r_{\beta_j}(f_{pv}(^{\circ}\beta_j))$,
$1 \le j \le t_1$, and $R_{\alpha_i}(f_{pv}(^{\circ}\alpha_i)) \equiv R_{\beta_i}(f_{pv}(^{\circ}\beta_i))$,
since $R_{\beta}(f_{pv}(^{\circ}\beta)) \equiv R_{\alpha}(f_{pv}(^{\circ}\alpha)))$
Similarly, $r^{Q_{1,0}}_{\gamma_{p_1}}(f_{pv}(^{\circ}\gamma_{p_1}))$
$= r_{\beta}(f_{pv}(^{\circ}\beta))\{\overline{v_1}/f_{pv}(^{\circ}\beta)\}$
$= r_{\alpha}(f_{pv}(^{\circ}\alpha))\{\overline{v_0}/f_{pv}(^{\circ}\alpha)\}$
since, $\langle \alpha, \beta \rangle \in E$ and $\overline{v_1} = \overline{v_0}$ by induction hypothesis
$= r^{Q_{0,0}}_{C_{p_0}}(f_{pv}(^{\circ}C_{p_0}))$.

$\square$

5.3 Experimental Results

The implementation of the techniques described in this chapter is referred to in the sequel as the `DCPEQX` module. The experimentation in this chapter is in continuation of the experimentation taken up in the previous chapter where paths in a path cover of PRES+ models were constructed as a precursor to applying the equivalence checking by invoking `DCPEQX`. Accordingly, experimentation has been carried out along two courses — one using hand constructed models and the other using models constructed by the same automated model constructor. Preparation of the example suite remains the same as that mentioned in Chapter 4. For checking equivalence between two paths, we have used the normalizer reported in [121].

5.3.1 Experimentation using hand constructed models

We have tested our `DCPEQX` module on the ten sequential examples as reported in Section 4.3 (Table 4.1) which are transformed using the SPARK compiler. The compiler takes as input a sequential program and generates its optimized version. The preparation of the examples and the set of transformations applied on each of them are already discussed in Chapter 4.

A typical output of the `DCPEQX` module for the `MODN` example is given in Figure 5.4. (The details of the `MODN` examples and the corresponding models are given in Figures 4.12, 4.11 and 4.13 of Chapter 4.) The output depicts the condition of execution and the data transformation for each path in normalized form. It is also to be noted that for the `MODN` example, path extension occurs twice in model 1 $-$ once for path 10 and next time for path 14. Lines $20-32$ of the output from `DCPEQX` indicate the following. For path 10 of model 1, the condition of execution $(0-1*n+1*s<0)$ is reported (as output line 21); the corresponding path in model 2 is also designated as path 10; its condition of execution is $(0-1*n+1*b>=0) \wedge (0-1*n+1*s<0)$ (output line 25). Therefore, it is identified that path extension of path 10 of model 1 is needed (output line 23). The condition of execution of the concatenation of path 10 and path 11 of model 1 matches with the condition of execution of path 10 of model 2 reported in output lines 27 -28; however, the data transformation is not matched as reported (in output line 28) because the number of pre-places for the concatenated path for model 1 is three corresponding to the variables n, s and a; in contrast, the number of pre-places for path 10 of model 2 is two corresponding to the variables n and s. The path extension is reported to be needed (in output line 29) . Concatenation of path 10 and path 11 of model 2 results in a concatenated path after path extension and this concatenated path is equivalent to the concatenation of paths 10 and 11 of model 1 both having identical data transformation $a := 0 + 1 * a + 1 * 0 - 1 * n + 1 * s$ (output lines 31-32). The first path extension output is depicted in lines 20-32 in Figure 5.4. Extension is carried out similarly for path 14.

Table 5.1 depicts our observations made through this line of experimentation vis-a-vis the performance of `FSMDEQX (PE)` module [14]. Both `FSMDEQX (PE)` module and `DCPEQX` module could establish equivalence for all the examples listed in the table except for the `MINANDMAX-S` example for which the transformed version is obtained by applying the loop swapping transformation which cannot be handled by the equivalence checker `FSMDEQX (PE)`. The column designated Extension (`DCPEQX`) indicates that in our method, for five examples, the costly path extension is needed. In comparison, `FSMDEQX (PE)` needs path extension in three more cases (column Extension (`FSMDEQX (PE)`)). The reason is that the PRES+ model being value based captures data independence more vividly incorporating parallelism in the model structure overriding the control flow of the input program wherever possible whereas the FSMD model retains the control dependence of the input program. The columns "`FSMDEQX (PE)` Time" and "`DCPEQX` Total Time" record the times taken by the FSMD equivalence checking method and by the `DCPEQX` module, respectively. They include the path construction times also. The FSMD equivalence checking is found to be slightly faster than our PRES+ equivalence checking. An interesting observation in this regard is that for FSMD models, the path construction overhead is negligible because unlike PRES+ models, they do not have any thread level parallelism. More specifically, path construction for FSMD models involves only identification of the cut-points, which are essentially the control flow bifurcation points. In contrast, for PRES+ models, the path construction process involves not only identification of the back edges but also keeping track of the sequence of maximally parallelisable transitions through a (forward) token tracking execution and identification of the dynamic cut-points which are used to construct the path using a backward traversal of the sequence. The column "`DCPEQX` Path Const Time" reproduces the observations recorded in Table 4.2; the entries in the column "`DCPEQX` EqChk Time" are obtained by subtracting the sum of two columns under "DCPEQX Path Const Time" from those in the column "DCPEQX Total Time". By comparing the figures in the column "DCPEQX EqChk Time" with those in "FSMDEQX (PE) Time", we notice that the DCPEQX module actually needs less time than the FSMDEQX (PE) module for the equivalence checking phase in almost all the cases; for the examples namely, MODN, GCD, PERFECT, DCT, LCM, LRU, PERFECT and PRIMEFAC, the benefit is about two times; however, for the example TLC, the performance gain is as high as 18 times.

We have also tested our DCPEQX module on five sequential examples which are transformed using two thread level parallelizing compilers namely, PLuTo and Par4All. These compilers take a sequential program as an input and generate its parallel counterpart. The preparation of the examples as well as the set of transformations applied are already discussed in Chapter 4, Section 4.3.

Example	Paths		Extension (FSMDEQX (PE))	Extension (DCPEQX)	FSMDEQX (PE) Time (μs)	DCPEQX			
						Path Const Time (μs)		EqChk	Total
	Orig	Transf				Orig	Transf	Time (μs)	Time (μs)
MODN	17	17	YES	YES	16001	5532	4834	8506	18872
SUMOFDIGITS	9	9	YES	YES	8000	1051	1168	6288	8507
PERFECT	13	9	YES	YES	8456	2929	1679	5077	9685
GCD	16	15	YES	NO	12567	6561	3240	3957	13758
TLC	28	23	YES	YES	16121	7355	8532	862	16749
DCT	1	1	NO	NO	2102	796	785	2054	3635
LCM	16	15	YES	NO	16231	6693	3825	6224	16742
LRU	18	18	YES	NO	20001	6345	6783	11435	24563
PRIMEFAC	10	10	YES	YES	6352	1065	1217	5505	7787
MINANDMAX-S	21	21	×	NO	×	6234	6225	5936	18395

Table 5.1: DCP induced equivalence checking times for hand constructed models of sequential examples

Example	Paths-Orig	Path-Transf		DCPEQX Time (μs)	
		PLuTo	Par4All	PLuTo	Par4All
BCM	3	3	3	4659	4659
MINANDMAX-P	21	21	21	24335	24335
LUP	35	34	34	33633	31235
DEKKER	17	17	17	45428	44952
PATTERSON	12	12	12	23231	23231

Table 5.2: DCP induced equivalence checking times for hand constructed models of parallel examples

```
##################### PATH  EQUIVALENCE #########################################
1 For Path 1 MODEL 1 ...THE CONDITION IS -- THE TRANSFORMATION IS s :=  0
2 PATH 1 OF MODEL 2 IS MATCHED WITH PATH 1 OF MODEL1
3 For Path 2 MODEL 1 ...THE CONDITION IS -- THE TRANSFORMATION IS i :=  0
4 PATH 2 OF MODEL 2 IS MATCHED WITH PATH 2 OF MODEL1
5 For Path 3 MODEL 1 ...THE CONDITION IS -- THE TRANSFORMATION IS a :=  0 + 1 * a
6 PATH 3 OF MODEL 2 IS MATCHED WITH PATH 3 OF MODEL1
7 For Path 4 MODEL 1 ...THE CONDITION IS -- THE TRANSFORMATION IS b :=  0 + 1 * b
8 PATH 4 OF MODEL 2 IS MATCHED WITH PATH 4 OF MODEL1
9 For Path 5 MODEL 1 ...THE CONDITION IS -- THE TRANSFORMATION IS n :=  0 + 1 * n
10 PATH 5 OF MODEL 2 IS MATCHED WITH PATH 5 OF MODEL1
.            .                 .                      .             .
20 For Path 10 MODEL 1...
21 THE CONDITION IS (  0 - 1 * n + 1 * s < 0 )
22 THE TRANSFORMATION IS  k : = 0 + 1  * a + 1 * 0 - 1 * n + 1 * s
23 PATH EXTENSION......
24 For Path 10 in MODEL 2...
25 THE CONDITION IS (  0 - 1 * n + 1 * s < 0 )AND(0 - 1 * n + 1 * b >= 0 )
26 THE TRANSFORMATION IS l :=  0 - 1 * n + 1 * s
23 For Path 11 MODEL 1...
24 THE CONDITION IS (0 - 1 * n + 1 * b >= 0 )
25 THE TRANSFORMATION IS a := 0 + 1 * a  + 1 * l
26 PATH 10 EXTEND THROUGH PATH 11 FOR MODEL 1
27 THE CONDITION IS (  0 - 1 * n + 1 * s < 0 )AND(0 - 1 * n + 1 * b >= 0 )
28 MATCHED WITH PATH 10 OF MODEL 2 THE TRANSFORMATION MISMATCH..
29 PATH EXTENSION .....
30 PATH 10 OF MODEL 2 WITH PATH 11 OF MODEL 2 THE TRANSFORMATION IS
31 a :=  0 + 1 * a + 1 * 0 + 1 * n + 1 * s
32 PATH 10 AND PATH 11 OF MODEL 2 IS MATCHED WITH PATH 10 AND PATH 11 OF MODEL1
.              .             .                  .                         .
46 PATH 16 OF MODEL 2 IS MATCHED WITH PATH 16 OF MODEL1
47 For Path 17 ...THE CONDITION IS -- THE TRANSFORMATION IS i :=  1 + 1 * i
48 PATH 17 OF MODEL 2 IS MATCHED WITH PATH 17 OF MODEL1
49 <<<<<<<<<<<<<<<<< THE TWO MODEL ARE EQUIVALENT >>>>>>>>>>>>>>>>>
500 ###################### Verification Report ##############################
51 Exec time is 0 sec and 18872 microsecs
52 #############################################################################
```

Figure 5.4: Output of `DCPEQX` module for the `MODN` example

The last two columns of Table 5.2 show the equivalence checking times for the parallelizing compilers PLuTo and Par4All. It is to be noted that the costly path extension procedure is not needed for the above parallel examples. The FSMD equivalence checking method fails to validate these transformations because thread level parallelism is not supported by FSMD models.

5.3.2 Experimentation using the automated model constructor

As mentioned in Chapter 4, the experimentation using automated model constructor has been considered only for such scenarios where both original and transformed versions of the programs are sequential in nature. The experimental set up is exactly similar to what we have already discussed in Chapter 4. Table 5.3 records the corresponding observations; for all examples listed in the table the source and the transformed programs were successfully declared to be equivalent by the `DCPEQX` module. It may be noted that under the column `FSMDEQX Time`, there are two sub-columns `PE` and `VP`; the former corresponds to the runtimes recorded for the path extension `FSMDEQX` module [74] and the latter to those recorded for the value propagation based `FSMDEQX` module [20]. For the `MINANDMAX-S` example, both `FSMDEQX (PE)` and `FSMDEQX (VP)` fail because the loop swapping transformation is involved. The last five rows involve code motion across loops (and indicated by the `CM`) suffix.

From Table 5.1 and the first ten rows of Table 5.3, until `MINANDMAX-S`, we observe that path extension is needed for automatically constructed models exactly in those cases where it is needed for the manually constructed ones. By comparing the entries in the column "`DCPEQX` Total Time" of Table 5.3 with those in the corresponding column in Table 5.1 we notice that the total time needed for equivalence checking time is proportional to the model size. Unlike the observations recorded with manually constructed models, for the automated models the times taken by the equivalence checking phase of the `DCPEQX` module are found to be comparable with those recorded for the `FSMDEQX` model for most of the examples; for the remaining ones, however, no definitive conclusions can be drawn in favor of one module over the other.

Example	Paths		Extension	Extension	FSMDEQX Time (μs)		DCPEQX			
			(FSMDEQX)	(DCPEQX)	PE	VP	Path Const Time (μs)		EqChk	Total
	Orig	Transf					Orig	Transf	Time (μs)	Time (μs)
MODN	43	42	YES	YES	16001	15892	11345	10863	15581	37789
SUMOFDIGITS	28	9	YES	YES	8000	8000	6341	5834	13302	25477
PERFECT	100	27	YES	YES	8456	8372	33432	10943	9299	53674
GCD	52	49	YES	NO	12567	12563	15534	13426	12472	41432
TLC	103	52	YES	YES	16121	14230	195938	86723	5795	288671
DCT	14	14	NO	NO	2102	1902	18913	16724	6717	42354
LCM	52	49	YES	NO	16231	16174	16534	14426	12285	43245
LRU	178	178	YES	NO	20001	19872	447174	387155	21456	855785
PRIMEFAC	49	26	YES	YES	6352	6149	11116	10730	5568	27414
MINANDMAX-S	56	51	×	NO	×	×	12544	12230	15989	40763
DIFFEQ	44	34	YES	NO	42500	42389	16342	11652	36195	64189
DHRC	121	107	YES	YES	188300	186729	4494567	4092345	185674	8772586
PRAWN	782	782	YES	NO	293400	291676	7508172	7023523	293876	78037279
IEEE 754	430	415	YES	YES	195741	186824	2976048	2975124	195330	6146482
BARCODE	884	1024	YES	YES	125189	125189	3019502	6174098	123175	9316779
QRS	178	156	YES	NO	20001	19346	447174	387155	21456	855785
EWF	540	525	YES	YES	34368	33413	2046828	1261312	36524	3344664
LCM-CM	52	49	–	NO	×	16035	16534	14426	12285	43245
IEEE 754-CM	430	415	–	YES	×	176572	2976048	2975124	195330	6146482
PERFECT-CM	100	27	–	YES	×	7278	33432	10943	9299	53674
LRU-CM	178	178	–	NO	×	18549	447174	387155	21456	855785
QRS-CM	178	156	–	NO	×	19234	447174	387155	21456	855785

Table 5.3: DCP induced equivalence checking times for sequential examples using automated model constructor

5.3.3 Experimental results after introducing errors

Finally, we take the original behaviours for some examples taken from the sequential and parallel example suites and manually inject some errors in the code level. The objective of this line of experimentation is to check the efficacy of the equivalence checker in detecting incorrect code motions. We have introduced the following types of (both instruction level and thread level) erroneous code transformations:

Type 1: non-uniform boosting up code motions from one branch of an if-then-else block to the block preceding it which introduce false-data dependencies in the other branch of the if-then-else block; this has been injected in the `GCD` and `MODN` examples.

Type 2: non-uniform duplicating down code motions from the basic block preceding an if-then-else block to one branch of the if-then-else block which remove data dependency in the other branch; this has been injected in the `TLC` example.

Type 3: mix of some correct code motions and incorrect code motions in `LCM` and

LRU examples.

Type 4: data-locality transformations which introduce false data-locality in the body of the loop in MINANDMAX-P and PATTERSON examples.

For each of these examples, the PRES+ models are constructed both manually and by the automated model constructor using the same procedures as elaborated in previous chapter. All the erroneous programs are given in Appendix A. In the following example, we discuss some important observations regarding our experimentation with the example MODN with Type 1 error.

Example 16. *Figure 4.12(a) in Chapter 4 depicts the source program of MODN which is transformed using SPARK compiler; the trimmed version of the optimized code is reproduced in Figure 5.5(a). Figure 5.5(b) depicts the erroneous code. During optimization using SPARK compiler, the statement $t = (l - n)$ is uniformly moved from the segment preceding `if-else` basic block to both `if` block and `else` block. However, in the erroneous code in Figure 5.5(b), the statement $t = (l - n)$ is non-uniformly moved only into the `if` block. For the test input $n = 7, a = 5, b = 6$, the program in Figure 5.5(a) yields the result 2 while the program in Figure 5.5(b) yields 4. Now, we feed the source program of Figure 4.12(a) and the PRES+ model of the erroneous program of Figure 5.5(b) to our equivalence checker; the checker successfully determines that the two programs are non-equivalent. The typical tool output for this example is given in Figure 5.7; specifically, path 9 of (model 1) has been identified to have no equivalent path in model 2 (output-lines 17-18).* ■

Errors	Example	no. of opns moved	FSMDEQX(PE) Non-EqChk Time (μs)	FSMDEQX(VP) Non-EqChk Time (μs)	DCPEQX (hand const.) Non-EqChk Time (μs)	DCPEQX (automated model const.) Non-EqChk Time (μs)
Type 1	MODN	1	15456	13471	17255	34048
	GCD	1	10435	10142	12523	42872
Type 2	TLC	2	14592	13780	16434	123414
Type 3	LRU	2	19278	16143	23143	733452
	LCM	1	11412	10619	12134	51231
Type 4	MINANDMAX-P	2	×	×	24347	×
	PATTERSON	4	×	×	10913	×

Table 5.4: Non-equivalence checking times for faulty translations

Table 5.4 depicts the descriptions of the errors introduced in the examples, the number of operations moved and the execution times taken by the FSMDEQX module and by the DCPEQX module; (in each cases, the non-equivalence has been detected by the modules successfully;) the performance of the latter is assessed on the hand constructed as well as automatically constructed PRES+ models for each example. The last three columns of the Table 5.4 record these respective times (including path construction times). It is to be noted that in all cases, the non-equivalence detection time is comparable with the equivalence checking time. It is worth mentioning that in course of this experiment our equivalence checker has identified a bug of the PLuTo compiler (possibly due to faulty usage of an existing variable namely, $t1$, holding intermediate results in the source program as the loop control variable $t1$ in the transformed program) around the program given in Figure 5.6. The error was successfully detected by our equivalence checker.

5.4 Conclusion

This chapter deals primarily with an equivalence checking method based on paths induced by dynamic cut-points. It has been formally established first that any path based equivalence checking approach, where the paths are defined using dynamic cut-points introduced in the previous chapter, would be a valid one. A specific method belonging to this class has been described in detail and illustrated. A sophisticated path extension mechanism has been devised to handle some code motion scenarios. The complexity and correctness issues have been treated comprehensively. Experiments on some sequential programs under code motion transformations and parallelizing transformations have been carried out and the results analyzed and found to be compatible with the expected behaviour predicted theoretically. Errors have been manually injected and non-equivalence checking capability of the implementation has been studied. Devising an efficient path based equivalence checking method where the costly path extension is not needed is our next goal.

```
int main(void) {
    int s = 0, i = 0, n, b,
        sout, a, k, l, t;
    do {
        if (i <= 15) {
            i = (i + 1);
            k = (b % 2);
            l = (a * 2);
            b = (b / 2);
            if (k == 1) {
                s = (s + a);
                t = (l - n);
                a = l;
            } else {
               t = (l - n);a = l;
            }
           /* t=(l-n) is removed
           in erroneous code */
            if (s >= n) {
               s = (s - n);
            }
            if (l >= n) {
          a = t;
            }
        } else
            break;
    } while (1);
    sout = s;
    printf("%d \n", sout);
}
            (a)
```

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

Figure 5.5: (a) Transformed program using SPARK compiler; (b) the erroneous version

```
int i=0, n, a=6, b=7, t1;
 # pragma scop
  while (i < n){
         t1 = a*b;  i++;
  }
# pragma endscop
            (a)
```

```
int i=0, n, a=6, b=7, t1;
# CLooG code
while (t1 < n){
      t1 = a*b; t1++;
 }
# CLooG code
     (b)
```

Figure 5.6: PLuTo Bug: (a) Source program – (b) transformed program

```
#################### PATH  EQUIVALENCE #############################
For Path 1 ...THE CONDITION IS -- THE TRANSFORMATION IS s :=  0
PATH 1 OF MODEL 2 IS MATCHED WITH PATH 1 OF MODEL1
For Path 2 ...THE CONDITION IS -- THE TRANSFORMATION IS i :=  0
PATH 2 OF MODEL 2 IS MATCHED WITH PATH 2 OF MODEL1
For Path 3 ...THE CONDITION IS -- THE TRANSFORMATION IS a :=  0 + 1 * id
PATH 3 OF MODEL 2 IS MATCHED WITH PATH 3 OF MODEL1
For Path 4 ...THE CONDITION IS -- THE TRANSFORMATION IS b :=  0 + 1 * id
PATH 4 OF MODEL 2 IS MATCHED WITH PATH 4 OF MODEL1
For Path 5 ...THE CONDITION IS -- THE TRANSFORMATION IS n :=  0 + 1 * id
PATH 5 OF MODEL 2 IS MATCHED WITH PATH 5 OF MODEL1
For Path 6 ...THE CONDITION IS (  -15 + 1 * i <= 0 ) THE TRANSFORMATION
IS s :=  0 + 1 * s PATH 6 OF MODEL 2 IS MATCHED WITH PATH 6 OF MODEL1
For Path 7 ...THE CONDITION IS  (  -15 + 1 * i > 0 ) THE TRANSFORMATION IS
s :=  0 + 1 * s PATH 7 OF MODEL 2 IS MATCHED WITH PATH 7 OF MODEL1
For Path 8 ...THE CONDITION IS -- THE TRANSFORMATION IS k := 0 + 1 * b
PATH 8 OF MODEL 2 IS MATCHED WITH PATH 8 OF MODEL1
For Path 9 ...THE CONDITION IS (  -1 + 1 * k == 0 ) THE TRANSFORMATION IS
l :=  0 + 1 * a + 1 *  0 + 1 * n CANDIDATE PATH:PATH 9 MISMATCH NO EXTENSION
For Path 10  ... THE CONDITION IS (  0 - 1 * n < 0 )
THE TRANSFORMATION IS  k : = 0 +  1 * n + 1 * s
PATH 10 OF MODEL 2 IS MATCHED WITH PATH 10 OF MODEL1
              .             .                 .                         .
<<<<<<<<<<<<<<<<< THE TWO MODEL ARE NOT EQUIVALENT >>>>>>>>>>>>>>>>>
##################### Verification Report #############################
Exec time is 0 sec and 17255 microsecs
############################################################################
```

Figure 5.7: Output of error detection of `DCPEQX` module for the `MODN` example

Chapter 6

Static Cut-point Induced Path Based Equivalence Checking Method

In the previous chapter, we have discussed a DCP based equivalence checking procedure, referred to as `DCPEQX` procedure, where we have shown that by introducing extra cut-points, any computation can be captured *syntactically* as a concatenation of parallel paths. In the present chapter, we show that a computation can also be captured *semantically* in terms of paths defined using static cut-points only without introducing dynamic cut-points. Hence sound equivalence checking procedures using such paths can also be devised. Towards this objective, we first modify the definition of paths; we then establish the validity of such static cut-point induced path based equivalence checking methods; we next discuss how the DCP based path construction procedure, described in chapter 4, is modified to obtain an SCP based path construction procedure, which we referred to as the `SCPEQX` method. Throughout this chapter by "cut-points" we would mean SCPs and DCPs will be explicitly mentioned to be so.

6.1 Model paths using static cut-points only

Before defining the static cut-point induced paths formally, we demonstrate through following two examples how static cut-point induced paths capture computations, albeit semantically, while the DCP based paths capture them syntactically. Recall that

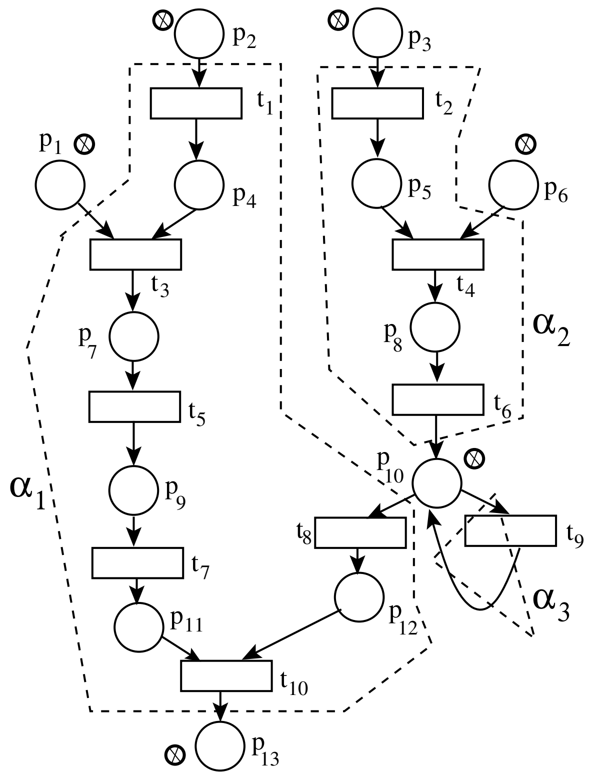

Figure 6.1: SCP Induced Paths of a PRES+ Model.

in Chapter 4 we motivated the need for dynamic cut-points using Example 6. We first reproduce this example here to show how the SCPs result in paths which can capture the computation. The second example will reveal certain intricacies which will finally lead to the definition of the SCP based paths.

Example 17. *Let us consider the example of Figure 6.1. By the static cut-point definition (Definition 12, Chapter 4), the set C of cut-points is* $\{p_1, p_2, p_3, p_6, p_{10}, p_{13}\}$ *and the paths will be* $\alpha_1 = \langle\{t_1\},\{t_3\},\{t_5\},\{t_7,t_8\},\{t_{10}\}\rangle$, $\alpha_2 = \langle\{t_2\},\{t_4\},\{t_6\}\rangle$ *and* $\alpha_3 = \langle\{t_9\}\rangle$ *respectively. Let us now try to express a computation* $\mu_{p_{13}}$ *of the out-port* p_{13} *in terms of paths, where* $\mu_{p_{13}} = \langle T_1 = \{t_1,t_2\}, T_2 = \{t_3,t_4\}, T_3 = \{t_5,t_6\}, T_4 = \{t_7,t_9\}, T_5 = \{t_9\}, T_6 = \{t_8\}, T_7 = \{t_{10}\}\rangle$. *Note that in this sequence, the members of any maximally parallelisable set can be arbitrarily ordered among themselves. While reordering the sequence* $\mu_{p_{13}}$ *the only constraint that is to be followed is that if a transition* t_1 *has some pre-place which is a post-place of some transition* t_2*, i.e.,* $t_2^{\circ} \cap {}^{\circ}t_1 \neq \emptyset$*, then* t_2 *must precede* t_1*. Using this constraint, the computation* $\mu_{p_{13}}$ *can be rewritten as* $\langle\{t_2\},\{t_4\},\{t_6\},\{t_9\},\{t_9\},\{t_1\},\{t_3\},\{t_5\},\{t_7,t_8\},\{t_{10}\}\rangle$. *Therefore, the computation* $\mu_{p_{13}}$ *can be represented as* $\langle\alpha_2.\alpha_3.\alpha_3.\alpha_1\rangle$. ■

In the following example, we describe a special case where a path cannot be formed due to presence of the parallel threads with at least one thread containing a loop.

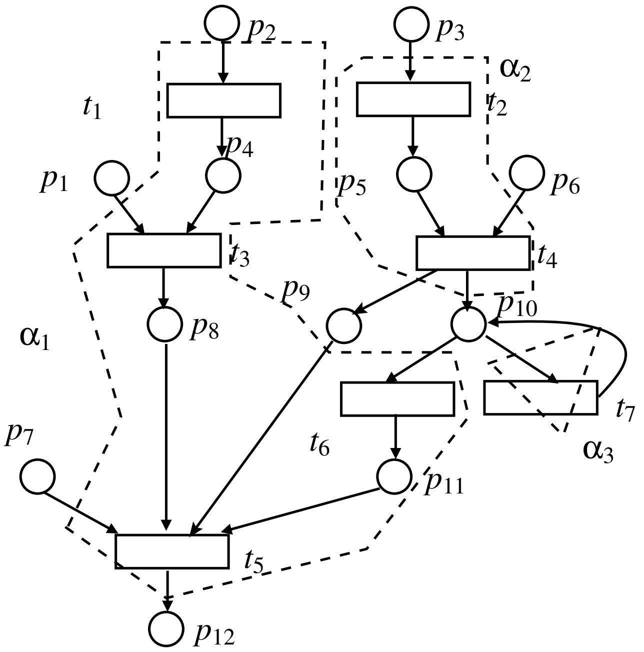

Figure 6.2: Modified Path for SCP Method.

Example 18. *In Figure 6.2, according to the definition of static cut-points (Definition 12, Chapter 4), the places $p_1, p_2, p_3, p_6, p_7, p_{10}$ and p_{12} are static cut-points. Using these cut-points, paths cannot be constructed so that they extend from a set of cut-points to a cut-point and also permit any computation to be captured as their sequence. For example, for Figure 6.2 we may capture one path as $\alpha_1^{'} = \langle\{t_1, t_2\}, \{t_3, t_4\}, \{t_6\}, \{t_5\}\rangle$ and the other as $\alpha_2^{'} = \langle\{t_7\}\rangle$, but no computation that goes through the loop at least once can be captured through any of their sequences. Instead, if we permit the path α_1 to end at p_9 although it is not a cut-point, then we have a set of paths as $\{\alpha_1, \alpha_2, \alpha_3\}$ as shown in Figure 6.2 and any computation can be represented in terms of these paths. For example, $\mu_{p_{12}} = \langle T_1 = \{t_1, t_2\}, T_2 = \{t_3, t_4\}, T_3 = \{t_7\}, T_3 = \{t_7\}, T_4 = \{t_6\}, T_5 = \{t_5\}\rangle$; using constraint mentioned in Example 17, the computation $\mu_{p_{12}}$ is rewritten as $\langle\{t_2\}, \{t_4\}, \{t_7\}, \{t_7\}, \{t_1\}, \{t_3\}, \{t_6\}, \{t_5\}\rangle$ where upon it can be represented as $\langle\alpha_2.\alpha_3.\alpha_3.\alpha_1\rangle$.* ■

Hence, we need to change the definition of path of a PRES+ model. The definition of path is therefore, formally defined as follows.

Definition 23 (SCP induced path in a PRES+ model)**.** *A finite path* α *in a PRES+ model from a set* T_1 *of transitions to a transition* t_j *is a finite sequence of distinct sets of parallelisable transitions of the form* $\langle T_1 = \{t_1, t_2, \ldots, t_k\}, T_2 = \{t_{k+1}, t_{k+2}, \ldots, t_{k+l}\}, \ldots, T_n = \{t_j\}\rangle$ *satisfying the following properties:*

(i) $^\circ T_1$ contains at least one cut-point or one co-place of a cut-point.

(ii) T_n° contains at least one cut-point.

(iii) There is no cut-point in T_m°, $1 \leq m < n$.

(iv) $\forall i, 1 < i \leq n, \forall p \in {}^\circ T_i$, if p is neither a cut-point nor a co-place of a cut-point, then $\exists k, 1 \leq k \leq i-1, p \in T_{i-k}^\circ$; thus, any pre-place of a transition set in the path which is neither a cut-point nor a co-place of a cut-point must be a post-place of some preceding transition set in the path.

(v) There do not exist two transitions t_i and t_l in α such that $^\circ t_i \cap {}^\circ t_l \neq \emptyset$.

(vi) $\forall i, 1 \leq i \leq n$, T_i is maximally parallelisable within the path, i.e., $\forall l \neq i, \forall t \in T_l$ in the path, $T_i \cup \{t\}$ is not parallelisable.

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

6.2 Capturing any computation in terms of Paths

In this section, we formally establish that any computation can be captured by a set of SCP-paths. To develop an intuitive perception of the formal reasoning used to establish the result, we illustrate the mechanism by the following example.

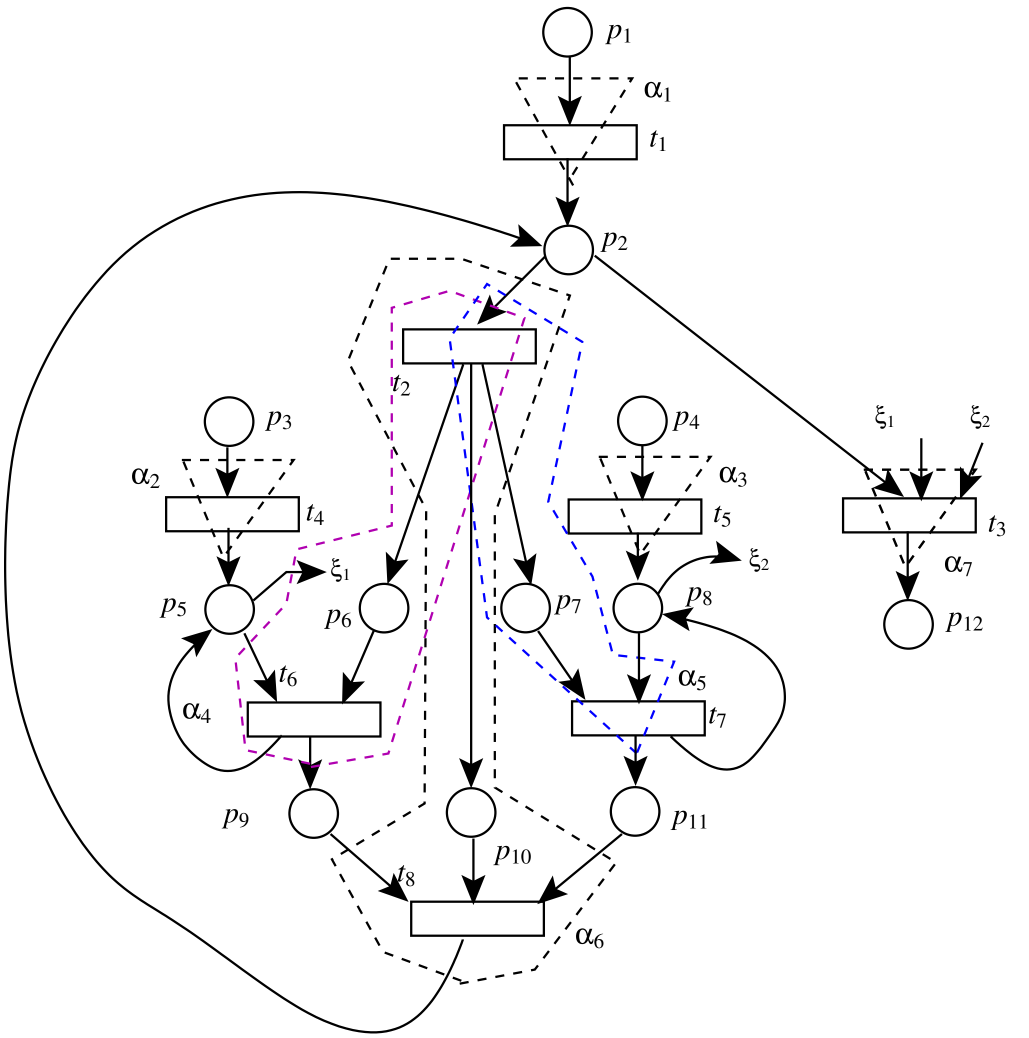

Figure 6.3: SCP Induced Paths of a PRES+ model.

Example 19. *Consider the model given in Figure 6.3. By Definition 12, the set* C *of static cut-points is* $\{p_1, p_2, p_3, p_4, p_5, p_8, p_{12}\}$*; the corresponding path set* Π *is* $\{\alpha_1 = \langle\{t_1\}\rangle, \alpha_2 = \langle\{t_4\}\rangle, \alpha_3 = \langle\{t_5\}\rangle, \alpha_4 = \langle\{t_2\},\{t_6\}\rangle,\ \alpha_5 = \langle\{t_2\},\{t_7\}\rangle\},\ \alpha_6 = \langle\{t_2\},\{t_8\}\rangle\}$*, and* $\alpha_7 = \langle\{t_3\}\rangle\}$*. Let us consider the computation* $\mu = \langle\{t_1,t_4,t_5\},\{t_2\}, \{t_6,t_7\},\ \{t_8\},\{t_2\},\ \{t_6,t_7\},\ \{t_8\},\{t_3\}\rangle$ *of the out-port* p_{12}*. We can reorder the transition sets of* μ *using the fact that any maximally parallelisable set of transitions can be partitioned arbitrarily and the members of the partition executed in any arbitrary order so that the sequence corresponding to a path occurs as a whole without having member transitions of other paths interspersed within the sequence. This arrangement permits us to view the reordered* μ*, referred to as* μ^r*, as a sequence of paths. The steps are as follows.*

The last member in the computation μ is identified as the unit set $\{t_3\}$. From Π, we notice that t_3 occurs as the last transition in the path α_7; so α_7 must be the last member in μ^r; hence the reordered *sequence in the first step becomes $\mu^{r(1)} = \langle \alpha_7 \rangle$. Deleting the member sets of transitions of α_7 from μ, the latter becomes $\mu^{(1)} = \langle \{t_1,t_4,t_5\},\{t_2\},\{t_6,t_7\}, \{t_8\},\{t_2\},\{t_6,t_7\},\{t_8\}\rangle$.*

Now, the last transition set in $\mu^{(1)}$ is the unit set $\{t_8\}$; it is found to occur as the last transition in the path $\alpha_6 = \langle \{t_2\},\{t_8\}\rangle$; the transition $\{t_8\}$ is deleted from μ; the other transition t_2 occurs in the path α_4 and α_5 as well; hence, t_2 is not deleted from μ. the path α_6 is placed before the path α_7 in μ^r thereby, $\mu^{r(1)}$ becomes $\mu^{r(2)} = \langle \alpha_6.\alpha_7\rangle$ and the computation $\mu^{(1)}$ becomes $\mu^{(2)} = \langle \{t_1,t_4,t_5\},\{t_2\},\{t_6,t_7\}, \{t_8\},\{t_2\},\{t_6,t_7\}\}\rangle$.

Now, the last member in $\mu^{(2)}$ is $\{t_6,t_7\}$ which is not a unit set. The transition t_6 is the last member of the path α_4; the transition t_7 is the last member of the path α_5. As the transitions t_6 and t_7 are parallelisable, the paths α_4 and α_5 are also parallelisable — a fact we prove subsequently; hence, they can be chosen to be placed in any order before α_6 in $\mu^{r(3)}$. Let us decide to place α_5 and then α_4; so $\mu^{r(4)} = \langle \alpha_4.\alpha_5.\alpha_6.\alpha_7\rangle$. Using the same reason, as explained above, the transition t_2 is not deleted from μ and $\mu^{(4)}$ becomes $\langle \{t_1,t_4,t_5\},\{t_2\},\{t_6,t_7\}, \{t_8\},\{t_2\}\rangle$. Now, in $\mu^{(4)}$, the last member comprising $\{t_2\}$ is not the last transition of any paths. Hence, t_2 is deleted from $\mu^{(4)}$. Therefore, the new $\mu^{(4)}$ becomes $\mu^{(5)}$ is $\langle \{t_1,t_4,t_5\},\{t_2\},\{t_6,t_7\}, \{t_8\}\rangle$.

It may now be noted that the last three members in $\mu^{(1)}$ is the same as those of $\mu^{(5)}$; so the process by which $\mu^{(1)}$ got transformed to $\mu^{(5)}$ and $\mu^{r(1)}$ got transformed to $\mu^{r(4)}$, as described in the above paragraphs, will be repeated resulting in $\mu^{r(8)} = \langle \alpha_4.\alpha_5.\alpha_6.\alpha_4.\alpha_5.\alpha_6.\alpha_7\rangle$ and $\mu^{(8)} = \langle \{t_1,t_4,t_5\}\rangle$.

Now the last (and the only member) of $\mu^{(8)}$ is $\{t_1,t_4,t_5\}$. They are respectively the last (and only) transitions of the parallelisable paths α_1,α_2 and α_3. Hence these paths can be placed in arbitrary order in $\mu^{r(8)}$. Repeating the steps, described above, thrice for the three paths, we get the final reordered sequence $\mu^r = \mu^{r(11)} = \langle \alpha_1.\alpha_2.\alpha_3.\alpha_4.\alpha_5.\alpha_6.\alpha_4.\alpha_5.\alpha_6.\alpha_7\rangle$ and $\mu^{(11)}$ becomes empty whereupon the process terminates. We shall formally establish that $\mu^r \simeq \mu$. ■

We formalize the above discussion using the following two theorems.

Theorem 9. *Let $\alpha_1 = \langle T_{1,1},T_{2,1},\ldots,T_{m,1}\rangle$ and $\alpha_2 = \langle T_{1,2},T_{2,2},\ldots,T_{n,2}\rangle$ be two paths* *such that their last transitions $T_{m,1}$ and $T_{n,2}$ are parallelisable. Then, α_1 and α_2 are parallelisable.*

Proof. Let it not be so. From Definition 17 of parallelisable pairs of paths, we have the following cases:

Case 1: $\alpha_1 \succ \alpha_2$. From Definition 16, there exist at least one set of paths $\alpha_{k_1}, \alpha_{k_2}, \ldots,$ α_{k_n} and a set of places $p_1 \in {}^\circ\alpha_1$ and $p_{k_m} \in {}^\circ\alpha_{k_m}, 1 \leq m \leq n$, such that $\langle$ last$(\alpha_2), p_{k_1}\rangle$, $\langle$ last$(\alpha_{k_1}), p_{k_2}\rangle, \ldots, \langle$ last$(\alpha_{k_n}), p_1\rangle \in O \subseteq T \times P$, $n \geq 0$, and none of them is a back edge. Therefore, using the fact that last(α) $\succ$ first(α), for any path α, and reading the above sequence of edges backward, we have last(α_1)=$T_{m,1} \succ$ first$(\alpha_1) \succ$ last$(\alpha_{k_n}) \succ$ first$(\alpha_{k_n}) \succ$ last$(\alpha_{k_{n-1}}) \succ \ldots \succ$ first$(\alpha_{k_2}) \succ$ last$(\alpha_{k_1}) \succ$ first$(\alpha_{k_1}) \succ$ last$(\alpha_2) = T_{n,2}$. Hence, $T_{m,1} \succ T_{n,2} \Rightarrow T_{n,2} \not\asymp T_{m,1}$ (Contradiction).

Case 2: $\alpha_2 \succ \alpha_1$. Following the same argument, as for Case 1, by symmetry with α_1 and α_2 interchanged, we again obtain the refutation of the hypothesis $T_{m,1} \asymp T_{n,2}$.

Case 3: $\exists \alpha_k, \alpha_l, (\alpha_k \neq \alpha_l \wedge \alpha_1 \succeq \alpha_k \wedge \alpha_2 \succeq \alpha_l \wedge {}^\circ\alpha_k \cap {}^\circ\alpha_l \neq \emptyset)$. ${}^\circ\alpha_k \cap {}^\circ\alpha_l \neq \emptyset \Rightarrow \exists t_{i,k} \in \alpha_k, \exists t_{j,l} \in \alpha_l$ such that ${}^\circ t_{i,k} \cap {}^\circ t_{j,l} \neq \emptyset$. Let the last transitions of the paths α_k and α_l be $t_{r,k}$ and $t_{s,l}$, respectively. Since $\alpha_1 \succeq \alpha_k$, $T_{m,1} \succeq t_{r,k} \succeq t_{i,k}$; (recall that $T_{m,1} = \{t_{m,1}\}$). Similarly, since $\alpha_2 \succeq \alpha_l$, $T_{n,2} \succeq t_{s,l} \succeq t_{j,l}$. Thus, $T_{m,1} \succeq t_{i,k}$, $T_{n,2} \succeq t_{j,l}$ and ${}^\circ t_{i,k} \cap {}^\circ t_{j,l} \neq \emptyset$. Therefore, $T_{m,1}$ is not parallelisable with $T_{n,2}$ (from Definition 4).

□

Theorem 10. *Let Π be the set of all paths of a PRES+ model obtained from a set of static cut-points. For any computation μ_p of an out-port p of the model, there exists a reorganized sequence μ_p^r of paths of Π such that $\mu_p \simeq \mu_p^r$.*

Construction of a sequence μ_p^r of (concatenation of) paths from μ_p: Algorithm 16 (*constructPathSequence*) describes a recursive function for constructing from a given computation μ_p and a set Π of paths the desired reorganized sequence μ_p^r of paths of Π such that $\mu_p^r \simeq \mu_p$. If μ_p is not empty, then a path α is selected from Π such that last(α)$\cap$ last$(\mu_p) \neq \emptyset$; if all its transitions are found to occur in μ_p, then it is put as the last member in the reorganized sequence; the member transitions of α are deleted from μ_p examining the latter backward; the transitions in the last member of α are always deleted from μ_p; each of the other transitions of α is deleted from μ_p only if it does not occur in any other path in Π. If $|\text{last}(\mu_P)| > 1$, then each member transition of last(μ_p) will result in one path which has to be processed separately through above steps. Once all these paths are processed, the last(μ_p) will get deleted from μ_p. The resulting μ_p is then reordered recursively; the process terminates when the input μ_p becomes empty.

Proof. ($\mu_p^r \simeq \mu_p$): We first prove that Algorithm 16 terminates; this is accomplished in two steps; first, it is shown that each invocation comprising four loops terminate; we next show that there are only finitely many recursive invocations.

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

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

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

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

Basis $m = 0$: $\mu_p^{(n)} = \langle\rangle = \mu_p^{r(n)}$ (by line 2 of the n^{th} invocation)

Induction Hypothesis: Let for $m = k-1$, $\mu_p^{(n-k+1)} \simeq \mu_p^{r(n-k+1)}$

Induction step: Let m be k. Let us assume that

$\mu_p^{(n-k)} \simeq \mu_p^{(n-k+1)}.\mu_l^{r(n-k)}$ (Lemma 7 – proved subsequently)

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

$\simeq \mu_p^{r(n-k)}$ (by line 40 (return statement) for the $(n-k)^{th}$ invocation) □

Lemma 7. $\mu_p^{(n-k)} \simeq \mu_p^{(n-k+1)}.\mu_l^{r(n-k)}$

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

$$\mu_p^{(k)} \simeq \mu_p^{(k+1)}.\mu_l^{r(k)} \simeq \mu_p^{(k+1)}.\langle\alpha_{1,k},\alpha_{2,k},\ldots,\alpha_{s,k}\rangle, 1 \leq k \leq n, \tag{6.1}$$

assuming that $\langle\alpha_{1,k},\alpha_{2,k},\ldots,\alpha_{s,k}\rangle$ is what is extracted as $\mu_l^{r(k)}$ from $\mu_p^{(k)}$ in line 40 of the kth iteration. Now, by step 6, the last transition of all the paths in the sequence $\mu_l^{r(k)}$ are parallelisable; hence, from Theorem 9, the paths of $\mu_l^{r(k)}$ are parallelisable. We prove that their transitions can be suitably placed in the member sets of $\mu_p^{(k+1)}$ (as larger sets of parallelisable transitions) to get back $\mu_p^{(k)}$.

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

Let $\mu_l^r = \langle\alpha_1,\alpha_2,\ldots,\alpha_s\rangle$ before line 37 just after the end of the for-loop comprising lines $10-35$, where s is the cardinality $|\Pi_{last}(\mu_p)|$ before entry to the loop (because any path has only one unit set of transitions as its last member). Thus, the for-loop comprising lines $10-35$ executes s times visiting step 33; let $\mu_p^{-(i)}$, $\mu_p^{+(i)}$ respectively denote the values of μ_p before and after the i^{th} iteration of the loop. Let $\mu_l^{r(i)}$ be the value of μ_l^r after the i^{th} execution of the loop. We have the following boundary conditions: $\mu_p^- = \mu_p^{-(1)}, \mu_p^+ = \mu_p^{+(s)}, \mu_l^{r(1)} = \langle\alpha_1\rangle$ and $\mu_l^r = \mu_l^{r(s)} = \langle\alpha_1,\alpha_2,\ldots,\alpha_s\rangle$. The i^{th} iteration of the for-loop comprising lines $10-35$ starts with $\mu_l^{r(i-1)}$ and obtains

$\mu_l^{r(i)}$, $1 \leq i \leq l$; so let $\mu_l^{r(0)} = \langle\rangle$ be the value of μ_l^r with which the first execution of the loop takes place.

We prove that $\mu_p^{-(i)} \simeq \mu_p^{+(i)}.\alpha_i, 1 \leq i \leq s$. If this relation indeed holds, then specifically for $i = 1$, $\mu_p^{-(1)} \simeq \mu_p^{+(1)}.\alpha_1$; for $i = 2$, $\mu_p^{-(2)} (= \mu_p^{+(1)}) \simeq \mu_p^{+(2)}.\alpha_2$. Combining these two, therefore,

$$\mu_p^- = \mu_p^{-(1)} \simeq \mu_p^{+(1)}.\alpha_1 \simeq (\mu_p^{+(2)}.\alpha_2).\alpha_1 \simeq \mu_p^{+(2)}.(\alpha_2.\alpha_1) \simeq \mu_p^{+(2)}.(\alpha_1.\alpha_2) \simeq \mu_p^{+(2)}.\mu_l^{r(2)}.$$

Proceeding this way, we have $\mu_p^- = \mu_p^{-(1)} \simeq \ldots \simeq \mu_p^{+(l)}.\mu_l^{r(l)} = \mu_p^+.\mu_l^r$.

Now, let $\mu_p^{-(i)} = \langle T_{1,i}, T_{2,i}, \ldots, T_{k_i,i}\rangle$, $\alpha_i = \langle T'_{1,i}, T'_{2,i}, \ldots, T'_{l_i,i}\rangle$ and $\mu_p^{+(i)} = \langle T^+_{1,i}, T^+_{2,i}, \ldots,$ $T^+_{n,i}\rangle$. Note that $\{\alpha_i \mid 1 \leq i \leq s\} \subseteq \Pi_{last(\mu_p)}$ and unless all the paths are extracted out, $T_{k_i,i}$ does not become empty and hence $\mu_p^{-(i)}, 1 \leq i \leq s$, do not change in length. For each transition set $T'_{j,i}$ of α_i, $1 \leq j \leq n$, there exists some transition set $T_{k,i}$ of $\mu_p^{-(i)}$, $1 \leq k \leq k_i$, such that $T'_{j,i} \subseteq T_{k,i}$. Specifically, for $j = l_i$, $T'_{l_i,i} \subseteq T_{k_i,i}$, since $\alpha_i \in \Pi_{last(\mu_p^-)}$ as ensured in step 6. For other values of $j, 1 \leq j < l_i$, the while-loop in steps 16-18, identifies proper $T_{k,i}$ in $\mu_p^{-(i)}$ such that $T'_{j,i} \subseteq T_{k,i}$; note that since α_i has figured in μ_l^r, step 32 is surely executed for α_i; so α' has been rendered empty ($\langle\rangle$) through execution of step 27 and hence the while-loop in steps 16-18 does not exit with $i = 0$. Now, step 13 and the for-loop in steps 22-26 ensure that $T^+_{k,i} \cup T_{j,i} = T_{k,i}$.

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

$$\begin{aligned}
\mu_p^+(i).\alpha_i &= \langle T^+_{1,i}, T^+_{2,i}, \ldots, T^+_{n,i}\rangle.\langle T'_{1,i}, T'_{2,i}, \ldots, T'_{l_i,i}\rangle \\
&= \langle T_{1,i}, \ldots, (T^+_{n_1,i} \parallel T'_{1,i}), \ldots (T^+_{n_2,i} \parallel T'_{2,i}), \ldots, (T^+_{n_{l_i-1},i} \parallel T'_{l_i,i}), \ldots, (T^+_{n_i,i} \parallel T'_{l_i,i})\rangle \\
&\qquad \text{(by commutativity of independent transitions)} \\
&= \langle T_{1,i}, \ldots, T_{n_1,i}, \ldots, T_{n_2,i}, \ldots, T_{n_{l_i-1},i}, \ldots T_{n,i}\rangle \\
&= \mu_p^{-(i)}
\end{aligned}$$

□

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

Definition 24 (SCP induced path cover)**.** *A finite set of paths $\Pi = \{\alpha_0, \alpha_1, \ldots, \alpha_k\}$ is said to be a path cover of a PRES+ model N if any computation μ of an out-port of N can be represented as a concatenations of paths from Π.*

From Theorem 10, it follows that a set of paths obtained from a given set of static cut-points is a path cover of the model.

Algorithm 16 SEQUENCE **constructPathSequence** (μ_p, Π)

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

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

6.2.1 Validity of Static cut-point induced path based equivalence checking method

Before describing the validity of static cut-point induced path based equivalence checking method, it is to be noted that definitions of path equivalence, corresponding transitions and definition of corresponding places (Definition 21) remain the same for static cut-point based equivalence checking method.

Theorem 11. *A PRES+ model N_0 is contained in another PRES+ model N_1, denoted as $N_0 \sqsubseteq N_1$, if there exists a finite path cover $\Pi_0 = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ of N_0 for which there exists a set $\Psi_1 = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_m\}$ of sets of paths of N_1 such that for all $i, 1 \leq i \leq m$, (i) $\alpha_i \simeq \beta$, for all $\beta \in \Gamma_i$. (ii) For each $\alpha_i, 1 \leq i \leq m$, each pre-place of α_i has a place-correspondence with some pre-place of β, where $\beta \in \Gamma_i$, (iii) all the post-places of α_i have correspondence with all the post-places of $\beta \in \Gamma_i$.*

Proof. Consider any computation $\mu_{0,p}$ of an out-port p of N_0. From Theorem 10, corresponding to $\mu_{0,p}$, there exists a reorganized sequence $\mu^r_{0,p} = \langle \alpha^p_1, \alpha^p_2, \ldots, \alpha^p_n \rangle$, say, of not necessarily distinct paths of N_0 such that (i) $\alpha^p_j \in \Pi_0$, $1 \leq j \leq n$, (ii) for each occurrence of a transition t in $\mu_{0,p}$, there exists exactly one path in $\mu^r_{0,p}$ containing that occurrence, (iii) $p \in (\alpha^p_n)^\circ$ and (iv) $\mu_{0,p} \simeq \mu^r_{0,p}$.

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

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

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

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

Algorithm 17 Sequence **parallelizeSeqSetsOfPaths** (μ^r_p)

Inputs: μ^r_p : a sequence of sets of paths
Outputs: $\mu_{||}$: a sequence of maximally parallelisable sets of transitions of all the paths in μ^r_p.

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

Algorithm 17 describes the construction method of $\mu_{1,p'}$ from $\mu^r_{1,p'}$ (and hence will be invoked with its input μ^r_p instantiated with $\mu^r_{1,p'}$). The parallelized version of the input μ^r_p is computed in $\mu_{||}$ which is to be assigned to $\mu_{1,p'}$ on return. In the initialization step (step 1), a working set Γ of paths is initialized to the first member of μ^r_p and the latter is removed from μ^r_p. In step 2, some path β is taken from Γ and put into $\mu_{||}$. In the outermost while-loop (steps 3-26), member sets of Γ are taken one by one (in steps 4-6) from μ^r_p; for each chosen set, its member paths are taken in the loop comprising steps 7-25; for each chosen path β, its member sets (of maximally parallelisable transitions) are examined one after another and checked against the members of $\mu_{||}$ from the beginning for fusion with them to construct larger sets of parallelisable transitions (steps $9-24$). For each chosen set T_p of transitions of β, one of the following two situations may arise:

Case 1 : The member T_p of the chosen path β of μ_p^r is found to succeed all the members in $\mu_{||}$, i.e., T_p is not parallelisable with any member of $\mu_{||}$. In this case, all the remaining members (including T_p) of β is concatenated at the end of $\mu_{||}$ [Steps 18-20] .

Case 2 : The member T_p of β is found not to succeed the c^{th} member $\mu_{||,c}$ of $\mu_{||}$, i.e., T_p is parallelisable with $\mu_{||,c}$, as argued later. In this case, T_p is combined (through union) with $\mu_{||,c}$; the successor transition sets of β need to be compared with only the subsequent members of $\mu_{||}$, i.e., with $\mu_{||,c+1}$ onwards [Step 22] .

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

The outer loop (steps 3-26) terminates because μ_p^r is finite to start with; (step 1 outside the loop reduces its length by one;) step 5 inside the loop reduces its length by one on every iteration of the loop. The for-loop (steps 7-25) terminates because the set Γ contains a finite number of paths and loses the chosen path in each iteration as per the semantics of the for-construct. The loop comprising steps 9-24 terminates because every path β in μ_p^r contains a finite number of sets of transitions and step 12 reduces the length by one in every iteration of the loop; if, however, any of these iterations do visit steps 19-20, then in step 20, β becomes empty and hence, this will be the last iteration of the while loop comprising steps 9 to 24. The loop comprising steps 13-17 terminates because at any stage, and hence on entry to the loop, $\mu_{||}$ has only a finite number of sets of transitions and in every iteration c increases by one; so finally, the second condition $c \leq$ length($\mu_{||}$) is bound to become false if the first condition does not become false by then.

Proof of $\mu_{1,p'} \simeq \mu_{1,p'}^r$: Let the initial value of μ_p^r (with which the function in Algorithm 17 is invoked), denoted as $\mu_p^r(-1)$, be of the form $\mu_p^r(-1) = \langle \Gamma_1, \Gamma_2, \ldots, \Gamma_n \rangle$, where, for all $i, 1 \leq i \leq n, \Gamma_i = \{\beta_{1,i}, \beta_{2,i}, \ldots, \beta_{t_i,i}\}$. So the outermost while-loop (steps 3-26) executes n times; for the $i-th$ execution of this loop, the inner for-loop executes t_i times; together, there are $t_1 \times t_2 \times \ldots \times t_n = t$, say, iterations in each of which a path

$\beta_{j,i}$ is accounted for. The algorithm treats these paths identically without making any distinction among paths from the same set or different sets. Hence we can treat the members of μ_p^r as a flat sequence of paths of the form $\langle\beta_1,\beta_2,\ldots\beta_t\rangle$. Let $\mu_p^r(i)$ and $\mu_{||}(i)$ respectively indicate the values of μ_p^r and $\mu_{||}$ at step 8 after the $i-th$ path β_i in the above sequence has been treated. So, the first time step 8 is executed, the value of $\mu_{||}$ is $\mu_{||}(0)$ = the first member β_1 of $\mu_p^r(-1)$ and $\mu_p^r(0)$ contains all the remaining members $\beta_2,\ldots,\beta_t$ of $\mu_p^r(-1)$. The final value returned by the algorithm (step 27) is $\mu_{||}(t)$ and $\mu_p^r(t)=\emptyset$ (by negation of the condition of the outermost while loop (steps 3-26)). We have to prove that $\mu_{1,p'}=\mu_{||}(t)\simeq\mu_p^r(-1)=\mu_{1,p'}^r\simeq\mu_{0,p}^r\simeq\mu_{0,p}$. We prove the invariant

$$\mu_{||}(i).\mu_p^r(i)\simeq\mu_p^r(-1),\forall i,0\le i\le t\ldots\ldots Inv(1) \tag{6.2}$$

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

$\mu_{||}(t).\mu_p^r(t)\simeq\mu_p^r(-1)\Rightarrow\mu_{1,p'}.\emptyset\simeq\mu_p^r\Rightarrow\mu_{1,p'}\simeq\mu_{1,p'}^r$, which would accomplish the proof as $\mu_{1,p'}^r\simeq\mu_{0,p}^r$ holds because the former has been obtained by *equivalence substitution of each member* in the latter and $\mu_{0,p}^r\simeq\mu_{0,p}$ by Theorem 10.

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

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

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

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

$$\mu_{||}(m-1).\beta_m \simeq \mu_{||}(m-1)(j).\beta_m(j), \forall j, 0 \leq j \leq n_m \ldots\ldots Inv(2) \tag{6.3}$$

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

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

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

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

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

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

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

Case 1: $T_{k,m}$ is found to succeed all the members of $\mu_{||}(m-1)(k-1)$ from T_c onwards – Hence, $T_{k,m}$ is parallelisable with no members of $\mu_{||}(m-1)(k)$. In this case, step 20 is executed resulting in concatenation of all the transition sets of $\beta_m(k-1)$ with $\mu_{||}(m-1)(k-1)$ and $\beta_m(k)$ becomes empty. So, $\mu_{||}(m-1)(k) = \mu_{||}(m-1)(k-1).\beta_m(k-1)$;
hence, $\mu_{||}(m-1).\beta_m \simeq \mu_{||}(m-1)(k-1).\beta_m(k-1)$ [by Induction hypothesis]

$= \mu_{||}(m-1)(k).\beta_m(k)$ (since $\beta_m(k) = \emptyset$)

Case 2: $T_{k,m} \not\succ T_c$ – This implies $T_{k,m} \asymp T_c$, as argued below. Note that between the two transition sets $T_{k,m}$ and T_c, there can be three mutually exclusive relations possible namely, $T_{k,m} \succ T_c, T_c \succ T_{k,m}$ and $T_{k,m} \asymp T_c$. It is given that $T_{k,m} \not\succ T_c$; now, let $T_c \succ T_{k,m}$. The transition set T_c in $\mu_{||}(m-1)$ is contributed to by paths which precede the path β_m in μ^r_p. Hence T_c does not succeed $T_{k,m}$. Therefore, $T_{k,m} \asymp T_c$. Let $\mu_{||}(m-1) = \langle T_1, T_2, \ldots, T_c, T_{c+1}, \ldots, T_k, \ldots T_{k_{m-1}} \rangle$. For all $s, 1 \leq s \leq k_{m-1} - c$, $T_c \not\succ T_{c+s}$. By an identical reasoning, $T_{k,m}$ does not also succeed T_{c+s} because otherwise T_{c+s} would have preceded in the path β_m. Therefore, $T_{c+s}.T_{k,m} \simeq T_{k,m}.T_{c+s}$. So, the concatenation $\langle T_{c+1}, \ldots, T_{k_{m-1}} \rangle . \langle T_{k,m}, T_{k+1,m}, \ldots, T_{l_m,m} \rangle$ is computationally equivalent to
$\langle T_{k,m}, T_{c+1}, \ldots, T_{k_{m-1}} \rangle . \langle T_{k+1,m}, \ldots, T_{l_m,m} \rangle$. Now,
$\mu_{||}(m-1).\beta_m \simeq \mu_{||}(m-1)(k-1).\beta_m(k-1)$ [by induction hypothesis]
$= \langle T_1, T_2, \ldots T_c, T_{c+1}, \ldots T_{k_m-1} \rangle . \langle T_{k,m}, T_{k+1,m}, \ldots, T_{l_m,m} \rangle$
$\simeq \langle T_1, T_2, \ldots T_c, T_{k,m}, T_{c+1}, \ldots T_{k_m-1} \rangle . \langle T_{k+1,m}, \ldots, T_{l_m,m} \rangle$
$\simeq \langle T_1, T_2, \ldots T_c \cup T_{k,m}, T_{c+1}, \ldots T_{k_m-1} \rangle . \beta_m(k)$

[since, by step 12, $\beta_m(k) = \langle T_{k+1,m}, \ldots, T_{l_m,m} \rangle$]
$= \mu_{||}(m-1)(k).\beta_m(k)$ [by step 22].
Note that since $T_{k,m} \succ T_{c-1}$ in $\mu_{||}(m-1)(k-1)$, as identified in the loop steps 13-17, T_c cannot be combined with T_{c-1} through union.

Proof of $\mu_{1,p'}$ being a computation: Recall that $\mu_{1,p'}$ is obtained from the sequence $\mu^r_{1,p'} = \langle \Gamma^{p'}_1, \Gamma^{p'}_2, \ldots, \Gamma^{p'}_n \rangle = \langle \{\beta_{1,1}, \beta_{2,1}, \ldots, \beta_{l_1,1}\}\{\beta_{1,2}, \beta_{2,2}, \ldots, \beta_{l_2,2}\}, \ldots, \{\beta_n\} \rangle$ of sets of paths of N_1, which, in turn, was constructed from the sequence $\mu^r_{0,p} = \langle \alpha^p_1, \alpha^p_2, \ldots, \alpha^p_n \rangle$ of paths of Π_0 satisfying the following properties: (i) $p' = \beta^\circ_n$, (ii) for all $j, 1 \leq j \leq n$, for all $k, 1 \leq k \leq l_j, \beta_{k,j} \simeq \alpha^p_j$, (iii) each of the places in ${}^\circ\Gamma^{p'}_j$ has correspondence with some place in ${}^\circ\alpha_j$ and (iv) all the places in Γ°_j have correspondence with with all the places in α°_j.

Let $\mu_{1,p'}$ be $\langle T_1, T_2, \ldots, T_l \rangle$, where T_1 is the first member of $\beta_{1,1}$ (by step 1 and first time execution of steps 7 and 11 of Algorithm 17). By property (iii) above, the places in ${}^\circ\beta_{1,1} \supseteq {}^\circ T_1$ have correspondence with those in ${}^\circ\alpha^p_1 \subseteq inP_0$. Since only the input places of N_1 have correspondence with the input places of N_0, ${}^\circ T_1 \subseteq {}^\circ\beta_{1,1} \subseteq inP_1$. It has already been proved that $p' = f_{out}(p) \in outP_1$. Since Algorithm 17 introduces the transition sets of the paths strictly in order from $\Gamma^{p'}_1$ to $\Gamma^{p'}_n$, T_l is a unit set containing the last transition of β_n and hence, $p' \in T_l^{\circ}$. Now, consider any $T_i \in \mu_{1,p'}$, $1 \leq i < l$; $T_{i+1} \succ T_i$ as ensured by the condition $T_p \succ T_c$ associated with the while loop of steps 13-17. For any $i, 1 \leq i < l$, let $T_{i+1}^{\circ} \subseteq P_{M_{i+1}}$ and $T_i^{\circ} \subseteq P_{M_i}$. It is required to prove that $M_{i+1} = M_i^+$, where $P_{M_i^+} = \{p \mid p \in t^{\circ} \wedge t \in T_m\} \cup \{p \mid p \in P_M \wedge p \notin^{\circ} T_m\}$, by first clause of Definition 1 of successor marking. We have the following two cases:

Case 1: $p_1 \in T_{i+1}^{\circ} \subseteq P_{M_{i+1}}$ – $p_1 \in T_{i+1}^{\circ} \Rightarrow \exists t_1 \in T_{i+1}$ such that $p \in t_1^{\circ}$. Now, $T_{i+1} = T_{M_i}$, the set of enabled transitions for the marking M_i. So, $p_1 \in t_1^{\circ}$ and $t_1 \in T_{i+1} = T_{M_i} \Rightarrow p_1 \in P_{M_i^+}$ by virtue of its being in the first subset of $P_{M_i^+}$.

Case 2: $p_1 \notin T_{i+1}^{\circ}$ but $\in P_{M_{i+1}}$ – So, $p_1 \notin T_{i+1}^{\circ} = T_{M_i}$. Hence, $p_1 \in P_{M_i}$ because $p_1 \in T_{i-k}^{\circ}$ for some $k \geq 1$. So $p_1 \in P_{M_i^+}$ by virtue of its being in the second subset of $P_{M_i^+}$. Therefore, $M_{i+1} = M_i^+$.

□

6.3 Path construction algorithm

The SCP path construction procedure is slightly different from the DCP path construction procedure, reported in Chapter 4. We describe the SCP procedure identifying on course the corresponding modules of the SCP and the DCP methods with the differences underlined. In Figures 6.4(a) and (b), we place the call graphs for both the methods for easy referencing.

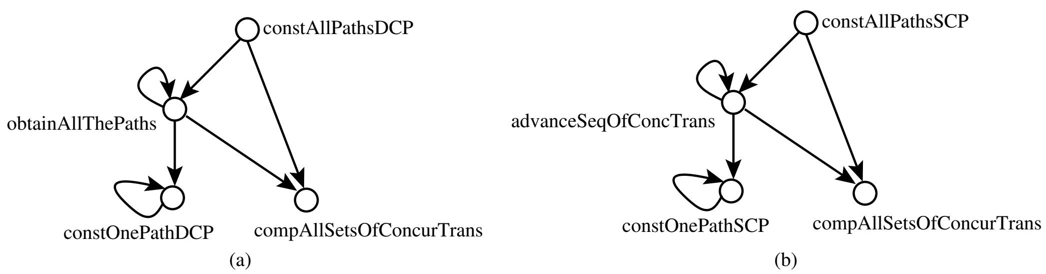

Figure 6.4: Call graphs for path construction method (a) for dynamic cut-points, and for (b) static cut-points

The SCP construction procedure starts with invocation of `constAllPathsSCP` (Algorithm 18) module which is similar to `constAllPathsDCP` (Algorithm 3) with certain differences indicated below. In the initialization part of both the modules, the marking at hand, M_h, is initialized to the set *inP* of in-ports, the set Q of all paths is initialized to empty and the sequence of sets of transitions at hand, T_{sh}, is initialized to the empty sequence. The function module `compAllSetsOfConcurTrans` (Algorithm 4, Chapter 4) is then called to compute the set of all the concurrent transitions possible from M_h. Each of these sets is a possible set of enabled transitions from M_h. For each of these sets, the function `advanceSeqOfConcTrans` (Algorithm 19) is called to compute all the paths that contain this set. Note that the module `constAllPathsDCP` instead invokes the function `obtainAllThePaths` (Algorithm 5) which is different from the function `advanceSeqOfConcTrans`. There is no other difference between these two top level modules. The function `compAllSetsOfConcutTrans` takes as input a marking M_h at hand and returns all the mutually exclusive sets of enabled transitions possible from M_h. The mechanism is identical for the two methods.

The function `advanceSeqOfConcTrans` takes as inputs a marking at hand, M_h, a set T_e of enabled concurrent transitions for M_h and the sequence T_{sh} of sets of enabled transitions obtained prior to M_h. It computes recursively the set of all the paths that involve T_e as one of their members. In each recursive invocation, it appends T_e at the end of T_{sh} and then advances the token from the pre-places $^\circ T_e \subseteq M_h$ to the post-places T_e°; if the new marking M_{new} contains a cut-point, then for each such cut-point p_c, say, the module `constOnePathSCP` (Algorithm 20) is invoked to obtain a path which has p_c as its post-place. If p_c is also an out-port, then it is deleted from M_{new}. The marking at hand M_h is updated to include M_{new} in place of $^\circ T_e$. The function `compAllSetsOfConcutTrans` is then invoked to obtain the set of all possible sets of concurrent transitions. For each set, assigned as T_e, the function `advanceSeqOfConcTrans` is recursively invoked. The recursion terminates if $T_e = \emptyset$. Note that this module is different from the function `obtainAllThePaths` in the following aspects: First, after computing M_{new}, if it is found to contain a cut-point, then the function `obtainAllThePaths` designates all the other places in M_{new} as dynamic cut-points and constructs a path from each of them. Secondly, if any place in M_h is found to contain a back edge, then a degenerate case designation is initiated and suitably terminated. These two steps are not necessary for the SCP method and are accordingly deleted from `obtainAllThePaths` to have the function module `advanceSeqOfConcTrans`.

Finally, the function `constOnePathSCP` is initially invoked from the function `advanceSeqOfConcTrans` with some cut-point p_c. The function `constOnePathSCP` constructs through a series of recursive invocations a path having p_c as its post-place using a backward cone of foci from p_c selecting the members of the path from T_{sh}. The only difference of this module vis-a-vis the corresponding module `constOnePathDCP` lies in the termination condition of the recursive invocation; specifically, for the present SCP method, the termination takes place when the backward progress encounters a set of places which are all cut-points or co-places of some cut-points; in the case of the DCP based method, the path construction proceeds till all the places are count to be cut-point. We illustrate the SCP based path construction method through Example 20 given below.

Example 20. *In Figure 6.2, the static cut-points are p_1, p_2, p_3, p_6, p_7 (in-ports), p_{10} (with a back edge incident on itself) and p_{12} (out-port); `constAllPathsSCP` which is the main module initializes M_h to its in-ports, i.e., $\{p_1, p_2, p_3, p_6, p_7\}$ and the sequence T_{sh} of enabled transitions to the empty set. When M_h is $\{p_1, p_2, p_3, p_6, p_7\}$, the set $\mathcal{T}$ of sets of concurrent transitions is a unit set, i.e., $\{\{t_1,t_2\}\}$. The function `advanceSeqOfConcTrans` is invoked with the parameter T_e as the only member $\{t_1,t_2\} \in \mathcal{T}$. The function appends T_e to T_{sh}. Hence, T_{sh} becomes $\langle\{t_1,t_2\}\rangle$. The two cut-points $p_1, p_6 \in InP$ and hence are not subjected to construct any path from them. In M_h, the places ${}^\circ T_e = \{p_2, p_3\}$ are replaced by $T_e^\circ = \{p_4, p_5\}$ to obtain $\{p_1, p_4, p_5, p_6, p_7\}$ as the new M_h. Now, the new enabled set of transitions for this M_h is $T_e = \{t_3,t_4\}$; a recursive invocation of `advance SeqOfConcTrans` with these new values of M_h and T_e updates T_{sh} as $\langle\{t_1,t_2\},\{t_3,t_4\}\rangle$ by adding T_e at its end. After firing of T_e, their post-places $\{p_8, p_9, p_{10}\}$, designated as M_{new}, say, acquire tokens. As the place p_{10} is a back edge induced cut-point, the path $\alpha_2 = \langle\{t_2\},\{t_4\}\rangle$ is constructed by the function `constOnePathSCP` using backward cone of foci method along T_{sh}. Next, the function `compAllSetsOfConcurTrans` computes the new marking M_h as $\{p_7, p_8, p_9, p_{10}\}$ by replacing ${}^\circ T_e = \{p_1, p_4, p_5, p_6\}$ with $T_e^\circ = \{p_8, p_9, p_{10}\}$ from $M_h = \{p_7, p_8, p_9, p_{10}\}$, the set $\mathcal{T}$ of possible concurrent transitions as $\{\{t_6\},\{t_7\}\}$. Each of these members is assigned to the set T_e of enabled transitions one by one for recursive invocation of the function `advanceSeqOfConcTrans` (in the for-loop starting with line 25). Let `advanceSeqOfConcTrans` be next invoked with $T_e = \{t_7\}$, $T_{sh} = \langle\{t_1,t_2\},\{t_3,t_4\}\rangle$ and $M_h = \{p_7, p_8, p_9, p_{10}\}$. The new T_{sh} becomes $\langle\{t_1,t_2\},\{t_3,t_4\},\{t_7\}\rangle$. The new marking M_{new} becomes $\{p_{10}\}$ whereupon, p_{10} being a cut-point, the path $\alpha_3 = \langle\{t_7\}\rangle$ is constructed using the function `constOnePathSCP`. Also, p_{10} is removed from M_{new} rendering it empty. So*

$M_h = \{p_7, p_8, p_9\}$; the function `compAllSetsOfConcurTrans` *returns $\mathcal{T} = \emptyset$ for this value of M_h; so the path α_3 is put in Q. Now the transition set $\{t_6\}$ is chosen as T_e in the next recursive invocation of* `advanceSeqOfConcTrans` *with $M_h = \{p_7, p_8, p_9\}$ and $T_{sh} = \langle\{t_1,t_2\},\{t_3,t_4\}\rangle$. The set T_e is put in T_{sh} making it $\langle\{t_1,t_2\},\{t_3,t_4\},\{t_6\}\rangle$. The set M_{new} becomes $\{p_{11}\}$ which is not a cut-point; so no path can be constructed and the for-loop consisting of lines 8-18 is skipped. M_h becomes $\{p_7, p_8, p_9, p_{11}\}$ in line 19. For this value of M_h, the set $\mathcal{T}$ of sets of concurrent transitions is computed as the unit set $\{\{t_5\}\}$ by* `compAllSetsOfConcurTrans`*. So the loop comprising lines 25-28 executes* `advanceSeqOfConcTrans` *only once invoking the function with $T_e = \{t_5\}, M_h = \{p_7, p_8, p_9, p_{11}\}$. For this invocation, T_{sh} becomes $\langle\{t_1,t_2\},\{t_3,t_4\}, \{t_6\},\{t_5\}\rangle$, M_{new} becomes $T_e^{\circ} = \{p_{12}\}$ which being an out-port is a cut-point. Hence the loop 8-18 is executed invoking* `constOnePathSCP` *with the above values of T_{sh} and $P = \{p_{12}\}$. This function proceeds as follows. It first extracts $\{t_5\} = last(T_{sh}) \cap {}^{\circ}P$ as T, modifies T_{sh} to $\langle\{t_1,t_2\},\{t_3,t_4\},\{t_6\}\rangle$ by removing its last member, obtains P' (in line 3) as $\{p_7, p_8, p_9, p_{11}\}$ (i.e. the pre-places of $T = \{t_5\}$); it then deletes from P', the cut-point (in-port) p_7 and the co-place p_9 of the cut-point p_{10} from these pre-places; So P' becomes $\{p_8, p_{11}\}$. It then recursively invokes itself with these new values of $P = P'$ and $T_{sh} = \langle\{t_1,t_2\},\{t_3,t_4\},\{t_6\}\rangle$ and puts $T = \{t_5\}$ at the end of the returned sequence of the path being constructed by this recursive invocation. The new recursive invocation obtains $T = \{t_6\} = last(T_{sh}) \cap {}^{\circ}P$, modifies T_{sh} to $\langle\{t_1,t_2\},\{t_3,t_4\}\rangle$ and P' as $\{p_8, p_{10}\}$ (in line 3). Since p_{10} is a cut-point, it is removed from P' in line 4 making $P' = \{p_8\}$. $T = \{t_6\}$ is put ahead of $\{t_5\}$ in the partially constructed path sequence making it $\langle\{t_6\},\{t_5\}\rangle$. Two more recursive invocations are similarly carried out to obtain the path $\alpha_1 = \langle\{t_1\},\{t_3\},\{t_6\},\{t_5\}\rangle$. On return, the function* `advqanceSeqOfConcTrans` *puts this path in Q and renders M_{new} empty in steps 14-16. The loop comprising steps 8-18 is exited. Step 19 computes M_h as $\{p_7, p_8, p_9, p_{11}\}$. Invocation of* `compAllSetsOfConcurTrans` *yields an empty $\mathcal{T}$ whereupon the function* `advanceSeqOfConcTrans` *returns control with $Q = \{\alpha_1\}$ through step 23. Now only returns from the previous invocations take place putting α_3 and finally, α_1 in Q. The function finally returns control to the function* `constAllPathsSCP` *with $Q = \{\alpha_3, \alpha_2, \alpha_1\}$.* ■

The above algorithm is now analysed for termination, complexity, soundness and completeness in the following subsections.

6.3.1 Termination and complexity analysis of the path construction algorithm

The termination proofs of all the functions involved in the SCP based method are identical with the corresponding modules used for the DCP based method. As discussed in the previous section, the complexity of the algorithm is dominated by the complexity of the function `compAllSetsOfConcurTrans` which is the same for the DCP method. Following an identical reasoning as put forward for the DCP method (Chapter 4), the complexity of the algorithm is $O(|T|)^2$. In the following, we treat the soundness and completeness of the method separately.

Algorithm 18 SETOFPATHS **constAllPathsSCP** (PRES+ N)

Inputs: A PRES+ model N
Outputs: Set of all paths Q

1: $M_h \Leftarrow inP$; /* Place − marking at hand − initialized to in-ports*/
$Q \Leftarrow \emptyset$; /* set of all paths − initially empty */
$T_{sh} \Leftarrow \langle\rangle$; /* Transition sequence at hand − initially empty */
2: $\mathcal{T}$ = **compAllSetsOfConcutTrans** (M_h, N);
// it takes M_h and forms all possible sets of concurrent transitions that are bound to M_h
3: $\forall T \in \mathcal{T}$
$Q \Leftarrow Q \bigcup$ **advanceSeqOfConcTrans** (T_{sh}, M_h, T, N);
/* Invokes **advanceSeqOfConcTrans** to obtain the set Q of all paths */
4: **return** Q;

6.3.2 Soundness of the path construction algorithm

Theorem 12. *Any member of the set Q returned by the function* `constAllPathsSCP` *satisfies the properties of the paths (as given in Definition 23).*

Proof. Let there be a path $\alpha = \langle T_1, T_2, \cdots, T_n \rangle$ in the set Q returned by the function `constAllPathsSCP` which does not satisfy all the properties of a path as listed in Definition 23. (The fact that any member of Q has such a form (as that of α) is obvious from step 10 of `advanceSEQofConcTrans` function and steps 1, 6 and 8 of the function `constOnePathSCP` which ensure that the path α obtained comprises only a sequence of sets of parallel transitions.) The definition of an SCP based path (Definition 23) differs from Definition 13of a DCP based path in clauses 1, 2 and 4. Hence, here we prove the only the three modified cases; proofs of all the other cases are similar to the corresponding cases of the soundness proof of DCP based methods (Theorem 4).

Algorithm 19 SETOFPATHS **advanceSeqOfConcTrans** (T_{sh}, M_h, T_e, N)

Inputs: The first parameter is the sequence T_{sh} of sets of concurrent transitions. The second parameter is the marking at hand M_h. The third parameter is a set T_e of enabled maximally parallelisable transitions. The fourth parameter is the PRES+ model N.
Outputs: The function returns the set of paths corresponding to the set of cut-points in the model N.

1: SETOFPATHS $Q = \emptyset$;
2: **if** $T_e == \emptyset$ **then**
3:    **return** Q;
4: **end if**
5: $\forall t \in T_e$, mark t;
6: $T_{sh} \Leftarrow$ append(T_{sh}, T_e); /* modify T_{sh} by appending T_e */
7: $M_{new} \Leftarrow T_e^{\circ}$; /* post-places of T_e acquire tokens */
8: **for** each $p_c \in M_{new}$ **do**
9:    **if** p_c is a cutpoint **then**
10:       $\alpha =$ **constOnePathSCP** $(\{p_c\}, T_{sh}, N)$;
      /* Traverse backward from $\{p_c\}$ along T_{sh} to construct a path up to some cutpoints */
11:       $Q \Leftarrow Q \cup \{\alpha\}$; /* Update Q */
      /* if any other out-place p of ${}^{\circ}p_c$ is also a cutpoint, then α is a path to that cutpoint also $-$ so delete the place p to avoid repetition of effort (Steps 12, 13) */
12:       Let $S = \{p \mid {}^{\circ}p = {}^{\circ}p_c$ and p is a cutpoint $\}$;
13:       $M_{new} = M_{new} - S$;
14:       **if** $(|p_c^{\circ}| = 0) \vee$ (all transitions of p_c° are marked) /* p_c is an out-port */ **then**
15:          $M_{new} \Leftarrow M_{new} - \{p_c\}$;
         /* p_c° have already occurred in some path $-$ this step prevents them from appearing in the subsequent set of enabled transitions */
16:       **end if**
17:    **end if**
18: **end for**
19: $M_h \Leftarrow (M_h - {}^{\circ}T_e) \cup M_{new}$; /* modify M_h by deleting the pre-set places of the transitions enabled */
20: $\mathcal{T} =$ **compAllSetsOfConcutTrans** (M_h, N);
21: **if** $(\mathcal{T} = \emptyset)$ and $(M_h \neq \emptyset)$ **then**
22:    "Report as invalid PRES+ Model"; return Q;
23: **else**
24:    **for** each $T_e \in \mathcal{T}$ **do**
25:       $T_e \Leftarrow T_e - \{$marked transitions of $T_e\}$
      $Q \Leftarrow Q \cup$ **advanceSeqOfConcTrans** (T_{sh}, M_h, T_e, N) //call itself recursively;
26:       **return** Q;
27:    **end for**
28: **end if**

Case 1: *None of the members in* ${}^{\circ}T_1$ *is either a cut-points or co-place of a cut-point.*

The path α has been constructed through n invocations of `constOnePathSCP` function; the first $n-1$ invocations have put the transition sets $T_n, T_{n-1}, \ldots, T_2$ in step 8; the n^{th} invocation returns a path comprising a sequence $\langle T_1 \rangle$ of length 1 in step 6. So, in this invocation, P' is found to be empty in step 5, i.e., after step 4. After step 3 of the k^{th} invocation, P' contains all the pre-places of ${}^{\circ}T_1$: Prior to step 4, P' must have been $P_c \cup \{p \mid p$ is a co-place of a cut-point$\}$. Therefore, ${}^{\circ}T_1 = P_c \cup \{p \mid p$ is a co-place of a cut-point$\}$.

Algorithm 20 PATH **constOnePathSCP** (P, T_{sh}, N)

Inputs: The first parameter is the set P of places. The second parameter is sequence T_{sh} of sets of concurrent transitions. The third parameter is the PRES+ model N.
Outputs: The function returns a path α.

1: $T = \text{last}(T_{sh}) \cap {}^\circ P$; /*$T$ is earmarked. The remaining ones in last(T_{sh}), if any, do not fall in the cone of influence of P */
2: $T_{sh}^{'} = T_{sh} - \text{last}(T_{sh})$; /* Ignore last($T_{sh}$) altogether in further backward traversal */
3: $P' = (P - T^\circ) \cup {}^\circ T$;
4: $P' = P' - P_c$; $P' = P' - \{p \mid p$ is a co-places of a cut-point$\}$;
/* Delete from P' all the cut-points and co-places of these (since paths do not move backward beyond them in the construction) */
5: **if** $P' = \emptyset$ **then**
6:     **return** (PATH) $\langle T \rangle$;
7: **else**
8:     **return** append(**constOnePathSCP**$(P', T_{sh}^{'}, N), T)$;
/* append T at the end of the sequence obtained by continuing backward */
9: **end if**

Case 2: *None of the members T_n° is a cut-point.* T_n has been placed in the path by the function `constOnePathSCP` in its first invocation from the function `advanceSeqOfConcTrans` in step 9 where it is ensured that the first parameter P is a unit set containing a cut-point. Also, step 1 of `constOnePathSCP` ensures that T_n° contains this cut-point.

Case 4: *The condition $\forall i,\ 1 < i \leq n, \forall p \in {}^\circ T_i$, if p is neither a cut-point nor a co-place of a cut-point, then $\exists l,\ 1 \leq l \leq i-1,\ p \in T_{i-l}^\circ$ does not hold. In other words, there exists a set T_i of concurrent transitions in the path which has a pre-place p which is neither a cut-point nor a co-place of a cut-point but is not included as a post place of any of the preceding transitions T_1 to T_{i-1}. Let T_i be the last such transition in the path with such a pre-place p.* Now `constOnePathSCP` is invoked first time from step 10 of `advanceSeqOfConcTrans` with the first parameter $P = \{p_c\}$, i.e., P containing a single cut-point. There is a recursive invocation subsequently when T_i has been included in the path with the first parameter P' containing $p \in {}^\circ T_i$ (due to step 3 – the union term). The subsequent recursive invocations of `constOnePathSCP` always has its first parameter P whose members satisfy the following two properties:

1. They are not cut-points (due to step 4 of the previous invocation), and
2. They are not in T° (due to step 3), where $T = \text{last}(T_{sh}) \cap {}^\circ P$.

Since p satisfies both (1) and (2) for all the recursive invocations (because, as per the premise, $p \notin T_1^\circ \cup T_2^\circ \cup \cdots \cup T_{i-1}^\circ$), P' (in step 8) will always contain

p. Thus, P' never becomes $\emptyset$ and hence `constOnePathSCP` never terminates (contradiction to Lemma 1, Chapter 4).

□

6.3.3 Completeness of the path construction algorithm

Theorem 13. *The set of paths returned by the function* `constAllPathSCP` *is a path cover of the model.*

Proof. Let $\mu_p = \langle T_1, T_2, \ldots, T_k \rangle$ be a computation not covered by the paths computed by the algorithm. From Theorem 10, there is a reorganized sequence of paths corresponding to μ_p, namely, $\mu_p^r = \langle \alpha_1, \alpha_2, \ldots, \alpha_l \rangle = \langle \langle T_{1,1}, T_{1,2}, \ldots, T_{1,n_1} \rangle . \langle T_{2,1}, T_{2,2}, \ldots, T_{2,n_2} \rangle . \ldots \langle T_{i,1}, T_{i,2}, \ldots, T_{i,n_i} \rangle . \ldots . \langle T_{l,1}, T_{l,2}, \ldots, T_{l,n_l} \rangle \rangle$ such that $\mu_p \simeq \mu_p^r$. Let α_i be the first path of the above sequence which is not constructed by the algorithm. Now, the set ${}^\circ T_{i,n_i}$ must be a subset of some reachable marking M_j of the model. The function `advanceSeqOfConcTrans` goes through all the reachable markings. So the module must have been invoked at some point with the parameters $M_h = M_j$ and $T_e \supseteq T_{i,l_i}$. This invocation computes a value for M_{new} in step 7 which contains T_{i,l_i}°. Since T_{i,l_i} is the last transition of the path α_i, T_{i,l_i}°, and hence M_{new}, would contain a cut-point. So, the invocation will execute the loop comprising steps 8-18. Specifically, in step 10, it will invoke `constAllPathSCP` with the cut-point in T_{i,l_i}°. From the soundness of the function `constAllPathSCP`, the path α_i will be constructed. [Contradiction]

□

6.4 Static equivalence checking

In this section, we describe a procedure for checking equivalence between two i/o-compatible PRES+ models using static cut-point induced paths; in the sequel, we refer to this method as `SCPEQX` method.

6.4.1 Equivalence Checking Algorithm

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

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

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

The modulewise functional description of the equivalence checking mechanism is captured through the Algorithms 21 and 22.

Algorithm 21 STRUCT4TUPLE **chkEqvSCP**(N_0, N_1)

Inputs: The PRES+ models N_0 and N_1.

Outputs: A six tuple structure comprising

1. E : a set of ordered pairs of the form $\langle \alpha_i, \beta_j \rangle$ of paths of Π_0 and Π_1 respectively, such that $\alpha_i \simeq \beta_j$.
2. η_t: the set of corresponding transition pairs;
3. $\Pi_{n,0}$: the set of paths of N_0 for which no equivalent is found in N_1 even with extension.
4. $\Pi_{n,1}$: the set of paths of N_1 for which no equivalent is found in N_0.

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

Algorithm 22 SETOFPATHS **findEqvSCP** (α, η_p, Π_1)

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

Outputs: Set Γ of equivalent paths.

$\Gamma = \emptyset$;
$\Gamma' = \{\beta \mid \beta \in \Pi'_1 \wedge (\forall p \in {}^\circ\alpha, \exists p' \in {}^\circ\beta$ s.t. $\langle p, p' \rangle \in \eta_p \vee \forall p' \in {}^\circ\beta, \exists p \in {}^\circ\alpha$ s.t $\langle p, p' \rangle \in \eta_p) \wedge$
$\forall p \in \alpha^\circ$ if $p \in outP_0$, then $\exists p' \in \beta_0$ s.t. $p' = f_{out}(p) \in outP_1\}$
/* for candidate path selection */
for each $\beta \in \Gamma'$ **do**
  if $R_\beta(f_{pv}({}^\circ\beta)) \equiv R_\alpha(f_{pv}({}^\circ\alpha))$ **then**
    if $r_\beta(f_{pv}({}^\circ\beta)) = r_\alpha(f_{pv}({}^\circ\alpha))$ **then**
      $\Gamma' = \Gamma' \cup \{\beta\}$
    else
      report "β not equivalent to α in spite of having pre-place correspondence and equivalent condition of execution";
      $\Gamma' = \emptyset$; // all not equivalent
    end if
  end if
end for
return Γ;

```
int i=1,x=10;
while (i<=10)
      i++;
output x;
```
(a)

```
int i=1,x;
while (i<=10)
      i++;
x=10;
output x;
```
(b)

Figure 6.5: Code motion across loop transformation.

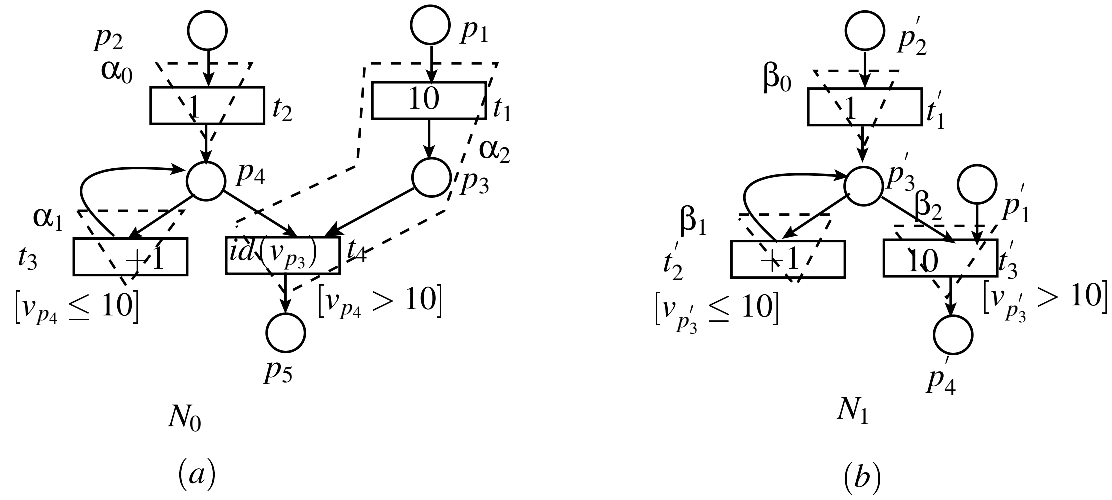

Figure 6.6: Illustrative example for the equivalence checking algorithm.

We illustrate the equivalence checking mechanism by the following example which involves a type of transformation namely, code motion across a loop. The verification of such transformations using CDFG based models such as FSMDs is a non-trivial task [20].

Example 21. *Figure 6.5(b) gives the program obtained from the program of Figure 6.5(a) by moving the instruction* $x = 10$ *preceding the loop to the segment following the loop. Figure 6.6(a) depicts the PRES+ model* N_0 *corresponding to the Figure 6.5(a) and Figure 6.6(b) represents the PRES+ model* N_1 *corresponds to Figure 6.5(b). The set of variables (for both* N_0, N_1*) is* $V = \{i, x\}$*. The place to variable associations are* $f^0_{pv} = \{p_4 \mapsto i, \{p_3, p_5\} \mapsto x, \{p_1, p_2\} \mapsto \delta\}$ *and* $f^1_{pv} = \{p'_3 \mapsto i, \{p'_1, p'_4\} \mapsto x, p'_2 \mapsto \delta\}$*. The bijection* f_{in} *is* $\{p_1 \mapsto p^{'}_1, p_2 \mapsto p^{'}_2\}$ *and the bijection* $f_{out} : p_5 \mapsto p'_4$*. In Figure 6.6(a), the cut-points are* p_1, p_2, p_4 *and* p_5*; the paths are* $\alpha_0 = \langle\{t_2\}\rangle$*,* $\alpha_1 = \langle\{t_3\}\rangle$ *and* $\alpha_2 = \langle\{t_1\}, \{t_4\}\rangle$*. Hence, the path cover* Π_0 *of* N_0 *is* $\{\alpha_0, \alpha_1, \alpha_2\}$*. In Figure 6.6 (b), the cut-points are* p'_1, p'_2, p'_3, p'_4 *and the paths are* $\beta_0 = \langle\{t^{'}_1\}\rangle$*,* $\beta_1 = \langle\{t^{'}_2\}\rangle$ *and* $\beta_2 = \langle\{t^{'}_3\}\rangle$*. Hence, the path cover* Π_1 *of* N_1 *is* $\{\beta_0, \beta_1, \beta_2\}$*. The equivalence checking method progresses through the following steps.*

The set η_p *of corresponding places is initialized to* f_{in}*. The sets* $\eta_t, E, \Pi_{n,0}$*,* $\Pi_{n,1}$ *are initialized to* $\emptyset$*. For* α_0*, the method identifies the path* β_0 *as the candidate for examining equivalence with* α_0 *because their respective pre-places are related by the relation* f_{in} *and the method identifies that* $R_{\alpha_0}(f^0_{pv}(^\circ\alpha)) \equiv R_{\beta_0}(f^1_{pv}(^\circ\beta)) \equiv \top$ *and* $r_{\alpha_0}(f^0_{pv}(^\circ\alpha)) = r_{\beta_0}(f^1_{pv}(^\circ\beta))$ *unary constant function* 1*. Hence, it infers* $\alpha_0 \simeq \beta_0$*. Consequently, the following update operations take place: (i)* $\eta_t = \{\langle last(\alpha_0) = t_2, last(\beta_0) = t'_1\rangle\}$*, (ii)* $\eta_p = \{\langle p_4, p'_3\rangle\}$ *and (iii)* $E = \{\langle\alpha_0, \beta_0\rangle\}$*. Similarly,* α_1 *and* α_2 *are found to have equivalence with* β_1 *and* β_2*, respectively and the sets* η_t, η_p *and* E *are updated. At this stage, the following entities are as follows:*
$\eta_t = \{\langle ^\circ(\alpha^\circ_0),\ ^\circ(\beta^\circ_0)\rangle,\ \langle\ ^\circ(\alpha^\circ_1),\ ^\circ(\beta^\circ_1)\ \rangle,\ \langle\ ^\circ(\alpha^\circ_2),\ ^\circ(\beta^\circ_2)\ \rangle\}$, $\eta_p = \{\langle\ \alpha^\circ_0,\ \beta^\circ_0\ \rangle,\ \langle\ \alpha^\circ_1,\ \beta^\circ_1\ \rangle,\ \langle\ \alpha^\circ_2,\ \beta^\circ_2\ \rangle\}$, *and* $E = \{\langle\alpha_0, \beta_0\rangle, \langle\alpha_1, \beta_1\rangle, \langle\alpha_2, \beta_2\rangle\}$*. At last, the method identifies that* $\Pi_{n,0}, \Pi_{n,1} = \emptyset$ *and accordingly declares that the two models* N_0 *and* N_1*.* ■

In the following example, we describe the validation steps for a thread level parallelizing transformation.

Example 22. *Figure 6.8(a) depicts a PRES+ model* N_0 *which can be obtained from the simple program* P_s *given in Figure 6.7(a). Figure 6.8(b) depicts the PRES+ model* N_1 *corresponding to the program* P_t *given in Figure 6.7(b) which is obtained by loop spitting followed by thread level parallelizing transformation of* P_s*.*

```
int i=0,k,m,n;
while (i<=10){
       m=m+10;
       n=n+10;
       i++;
}
k=m+n;
```

(a)

```
int i=j=0,k,m,n;
while (i<=10){
       m=m+10;
       i++;
}
||
while (j<=10){
       n=n+10;
       j++;
}
k=m+n;
```

(b)

Figure 6.7: A thread level parallelizing transformation–(a) P_s: source program and (b) P_t: transformed program.

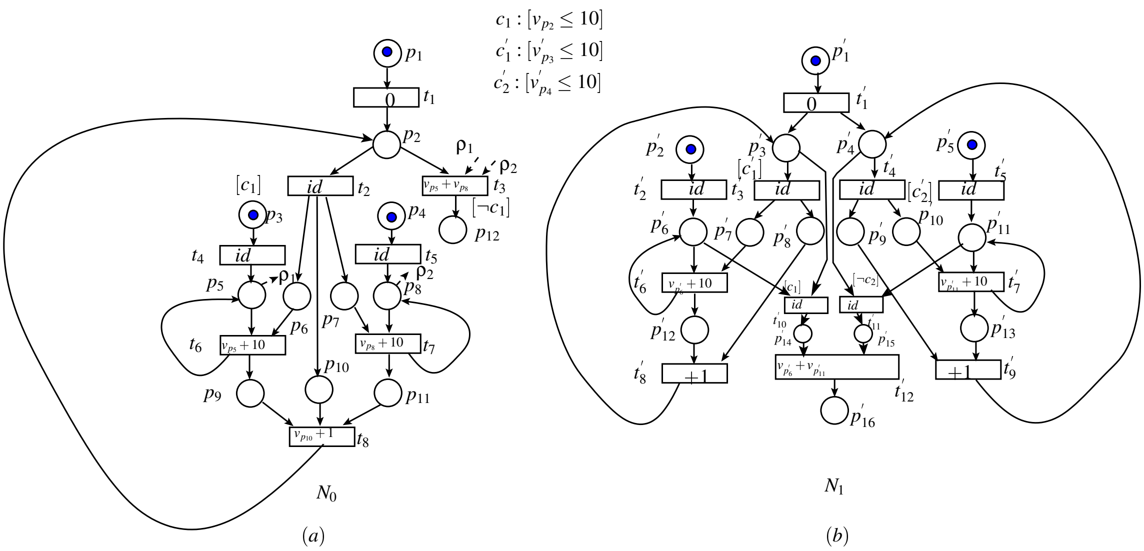

Figure 6.8: Illustrative example for validation of a parallelizing transformation.

Recall that the program P_s and its corresponding net N_0 have also been given as Example 3; the construction of the model N_0 has been explained in detail in that example; hence we skip the details here. Recall that the set of variables $V = \{i, m, n, k\}$ and the place to variable mapping is $f_{pv}^0 : \{\{p_1, p_6, p_7, p_9, p_{11}\} \mapsto \delta$ for dummy in-ports and synchronizing places. $\{p_2, p_{10}\} \mapsto i, \{p_3, p_5\} \mapsto m, \{p_4, p_8\} \mapsto n, p_{12} \mapsto k\}$.

Now, consider the program P_t. In this transformed program, the single while-loop of P_s is split into two different parallel loops corresponding to two different independent statements "$m = m + 10$" and "$n = n + 10$" which constitute the bodies of the respective while-loops that are parallellized. The loop control variables corresponding to these two loops are i and j which start with an identical initial value 0*. In the* *corresponding PRES+ model N_1 of Figure 6.8(b), the places $p_3^{'}$ and $p_4^{'}$ represent these loop control variables i and j; the transition $t_1^{'}$ not only initializes the places $p_3^{'}$ and $p_4^{'}$ but also creates two parallel threads corresponding to the* `parbegin` *statement. The subnet corresponding to the while-loops are obtained by the same reasoning used to obtain the subnet in N_0 of the while-loop of P_s. In the present case, however, the loop exit transitions $t_{10}^{\prime}$ and $t_{11}^{\prime}$ associated with $\neg c_1$ and $\neg c_2$ respectively, only achieve exits from the loops; more specifically, unlike the exit transition t_3 of N_0 in Figure 6.8(a), it cannot accomplish the task of the assignment statement "$k = m+n$" because that happens only after merging of the two parallel threads. The transition $t_{12}^{\prime}$ serves two purposes — it accomplishes the merging of the two parallel threads corresponding to the* `parend` *statement and accordingly have $p_{14}^{\prime}$ and $p_{15}^{\prime}$ as its pre-places; secondly, it captures the computation corresponding to the assignment statement "$k = m+n$" producing the output token corresponding to the output variable k at the out-port $p_{16}^{\prime}$. So, for N_1, the set of variables is $V = \{i, j, m, n, k\}$ and the place to variable mapping $f_{pv}^1 = \{\{p_1^{\prime}, p_7^{\prime}, p_{12}^{\prime}, p_{10}^{\prime}, p_{13}^{\prime}\} \mapsto \delta, \{p_2^{\prime}, p_6^{\prime}\} \mapsto m, \{p_3^{\prime}, p_8^{\prime}\} \mapsto i, \{p_4^{\prime}, p_9^{\prime}\} \mapsto j, \{p_5^{\prime}, p_{11}^{\prime} p_{15}^{\prime}\} \mapsto n, p_{16}^{\prime} \mapsto k\}$.*

Using the path construction algorithm, the sets of paths obtained from Figures 6.8(a) and 6.8(b) are $\Pi_0 = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7\}$, $\Pi_1 = \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7, \beta_8\}$, where $\alpha_1 = \langle\{t_1\}\rangle, \alpha_2 = \langle\{t_4\}\rangle, \alpha_3 = \langle\{t_5\}\rangle, \alpha_4 = \langle\{t_2\},\{t_6\}\rangle$, $\alpha_5 = \langle\{t_2\},\{t_7\}\rangle$, $\alpha_6 = \langle\{t_2\},\{t_8\}\rangle$, $\alpha_7 = \langle\{t_3\}\rangle$ and $\beta_1 = \langle\{t_1^{'}\}\rangle, \beta_2 = \langle\{t_2^{'}\}\rangle, \beta_3 = \langle\{t_5^{'}\}\rangle, \beta_4 = \langle\{t_3^{'}\},\{t_6^{'}\}\rangle$, $\beta_5 = \langle\{t_4^{'}\},\{t_7^{'}\}\rangle, \beta_6 = \langle\{t_3^{'}\},\{t_8^{'}\}\rangle$, $\beta_7 = \langle\{t_4^{'}\},\{t_9^{'}\}\rangle$ and $\beta_8 = \langle\{t_{10}^{'}, t_{11}^{'}\},\{t_{12}^{'}\}\rangle$.

Let $f_{in} : inP_0 \leftrightarrow inP_1$ be $\{p_1 \mapsto p_1^{\prime}, p_3 \mapsto p_2^{\prime}\rangle, p_4 \mapsto p_5^{\prime}\}$; let $f_{out} : outP_0 \leftrightarrow outP_1$ be $p_{12} \mapsto p_{16}^{\prime}$. Therefore, the place-correspondence relation η_p is initialized as $\{\langle p_1, p_1^{\prime}\rangle, \langle p_3, p_2^{\prime}\rangle, \langle p_4, p_5^{\prime}\rangle, \langle p_{12}, p_{16}^{\prime}\rangle\}$. For each path of Figure 6.8(a), the equivalent path of Figure 6.8(b) is obtained by the following steps:

For the path α_1: The function `chkEqvSCP` *calls the function* `findEqvSCP` *which first identifies some candidate paths from Π_1, whose pre-places are in place-correspondence with the pre-place of the path α_1. The path β_1 is identified as the only candidate path for α_1 because $\langle {}^{\circ}\alpha_1 = p_1, {}^{\circ}\beta_1 = p_1^{\prime}\rangle \in \eta_p$. Since the conditions of execution of the paths α_1 and β_1 are equivalent and their data transformations are same, the function* `findEqvSCP` *returns the set $\Gamma = \{\beta_1\}$ as the set of paths equivalent to α_1. On return, the caller function* `chkEqvSCP` *updates the following sets: E becomes $\{\langle\alpha_1, \beta_1\rangle\}$, $\alpha_1^{\circ} = \{p_2\}$ should be associated with $\beta_1^{\circ} = \{p_3^{\prime}, p_4^{\prime}\}$; since*

$f^0_{pv}(p_2) = f^1_{pv}(p'_3) = i$, so, $\langle p_2, p'_3\rangle$ is put in η_p. However, $f^1_{pv}(p'_4) = j \neq f^0_{pv}(p_2)$; but j is an uncommon variable and at this point, $j = i$ since $p'_3, p'_4 \in \beta^\circ_1$. So we can proceed with the association $j \mapsto i$ and $\langle p_2, p'_4\rangle$ is also put in η_p; η_t becomes $\{\langle$ last(α_1),last(β_1)$\}$. Similarly, it is found that $\alpha_2 = \langle\{t_4\}\rangle \simeq \beta_2 = \langle\{t'_2\}\rangle \Rightarrow \langle\alpha^\circ_2, \beta^\circ_2\rangle = \langle p_5, p'_6\rangle \in \eta_p$ and $\alpha_3 = \langle\{t_5\} \simeq \beta_3 = \langle\{t'_5\}\rangle \Rightarrow \langle\alpha^\circ_3, \beta^\circ_3\rangle = \langle p_8, p'_{11}\rangle \in \eta_p$.

For the path α_4: The candidate path is chosen as β_4 by `findEqvSCP` using the following steps. First, it is found that $^\circ\alpha_4 = \{p_2, p_5\}$, $^\circ\beta_4 = \{p'_3, p'_6\}$ and $\langle p_2, p'_3\rangle, \langle p_5, p'_6\rangle \in \eta_p$; it is next identified that the conditions of execution $R_{\alpha_4}(v_{p_2} \leq 10)$ and $R_{\beta_4}(v_{p'_3} \leq 10)$ are same because $\langle p_2, p'_3\rangle \in \eta_p$ and hence the values $v_{p_2} = v_{p'_3}$ always holds. Similarly, from the place correspondence of p_5, p'_6, their data transformations $r_{\alpha_4} = v_{p_5} + 10$ and $r_{\beta_4} = v_{p'_6} + 10$ are identified to be identical. The fact that $\alpha_5 \simeq \beta_5$ will be inferred identically this time using the correspondence $p_2 \in {}^\circ\alpha_5$ also with $p'_4 \in {}^\circ\beta_5$. In the process, $\langle p_9, p'_{12}\rangle, \langle p_{11}, p'_{13}\rangle$ are included in η_p, the former due to $\alpha_4 \simeq \beta_4$ and the latter due to $\alpha_5 \simeq \beta_5$.

For the path α_6: While choosing the candidate path, the function `findEqvSCP` identifies that $^\circ\alpha_6 = \{p_2, p_9, p_{11}\}$ and the pre-places of both paths $^\circ\beta_6 = \{p'_3, p'_{12}\}$, $^\circ\beta_7 = \{p'_4, p'_{13}\}$ are in η_p with the pre-places of α_6; also, $R_{\alpha_6} \equiv R_{\beta_6}$ as well as $R_{\alpha_6} \equiv R_{\beta_7}$; similarly, $r_{\alpha_6} = r_{\beta_6}$ and $r_{\alpha_6} = r_{\beta_7}$. So it returns $\Gamma = \{\beta_6, \beta_7\}$. The caller function registers both these paths to be equivalent to α_6 and suitably updates E, η_t and η_p which happens to remain unchanged.

For the path α_7: The pre-places $^\circ\alpha_7 = \{p_2, p_5, p_8\}$ are used identically by `findEqvSCP` to identify β_8 as the only candidate path since $^\circ\beta_8 = \{p'_3, p'_6, p'_{11}, p'_4\}$ and $\langle p_2, p'_3\rangle, \langle p_5, p'_6\rangle, \langle p_8, p'_{11}\rangle, \langle p_2, p'_4\rangle \in \eta_p$. It also identifies the equivalence of their conditions of execution and equality of data transformations; so it returns β_8 as the equivalent of α_7. On return, the caller function finds that all the paths of N_0 have equivalent paths in N_1 with proper correspondence of their pre-places; also all the paths of N_1 are found to have equivalence with some path in N_0; accordingly, it declares the models (and hence the programs P_s, P_t) to be equivalent.

■

Figure 6.9 describes a situation, where the code C_3 is moved and executed in parallel with C_0 and C_1. Figures 6.10(a) and (b) give the PRES+ model corresponding

```
#parbegin
   C0, C1;
#parend

#parbegin
   C2, C3;
#parend
    C4;
```

(a)

```
#parbegin
   C0, C1, C 3;
#parend
   C'4;
```

(b)

Figure 6.9: Code motion transformation for parallel programs.

the code schema of Figures 6.9(a) and (b). The path construction procedure as described in section 6.3, constructs only one path α as shown in Figure 6.10(a) using the backward cone of foci method. Similarly, using the same procedure, a single path β is constructed as shown in Figure 6.10(b). The path α is equivalent with the path β as their pre-places have correspondence, their conditions of execution are both 'true' and their data transformations identical.

The above algorithm is now analysed for termination, complexity and soundness in the following subsections.

6.4.2 Termination of the equivalence checking algorithm

The path construction algorithm terminates reported in [17]. Therefore, the respective path covers Π_0 and Π_1 of N_0 and N_1 produced by this algorithm are finite and the equivalence checking phase starts with finite Π_0 and Π_1.

Theorem 14. *`chkEqvSCP function` (Algorithm 21) always terminates.*

Proof. The function `findEqvSCP` terminates because step 2 has to examine only a finite (number of paths of) Π_1 to construct Γ'; thus, the set Γ' is finite and hence the loop comprising steps 3-7 executes finite number of times. So this function terminates. The function `chkEqvSCP` has an inner loop comprising steps 5-7 which is executed a finite number of times since Γ is finite. The outer loop comprising steps 2-11 executes

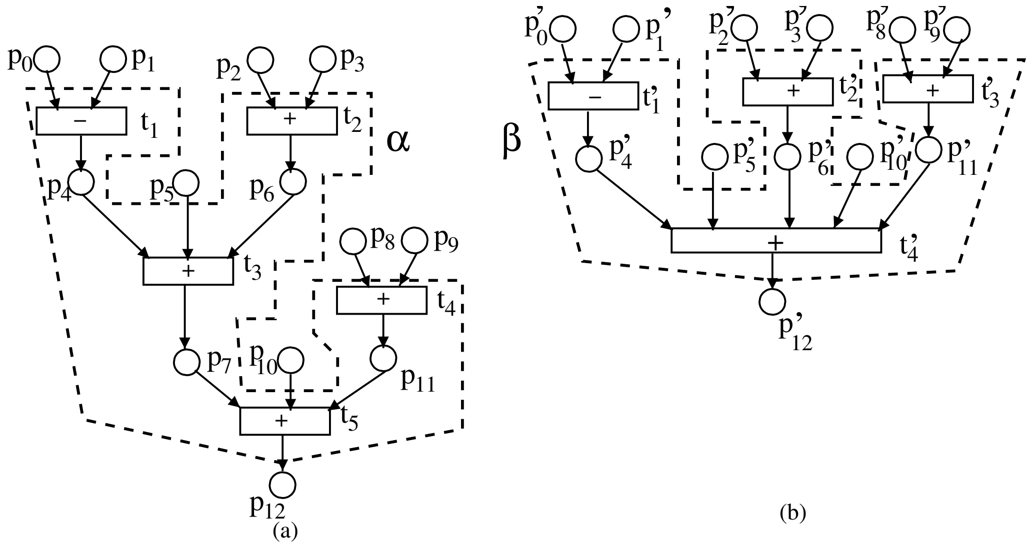

Figure 6.10: Initial and transformed behaviour of PRES+ models.

also executes finite number of times since Π_0 is finite. So both the loops terminate and hence so does the function.

□

6.4.3 Complexity analysis of the equivalence checking algorithm

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

Complexity of Algorithm 22 (`findEqvSCP`) : Step 1 is an initializing step which takes in $O(1)$ time. Step 2 takes $O(|\Pi_1|) = O(|P|^3)$ time, which is the complexity of path construction as explained in section 4.2.2. Step 4 compares the condition of execution and the data transformation for each path. Hence the complexity for each of this comparison is $O(|F|)$, where $|F|$ is the maximum of the lengths of the formulae representing the data transformations and conditions of execution of paths of N_0, N_1. Computation of such formulae is exponential in the number of variables which, in turn, is upper-bound by the number of places, i.e., $O(|F|)$ is $O\left(2^{|P|}\right)$. Step 5 is a union operation needing just $O(1)$ time with β being blindly put at the end of Γ. The loop iterates as many times as $O(|\Pi_1|) = O\left(|P|^3\right)$. Hence, the overall complexity is $= O\left(|P|^3 + 2^{|P|}.\,|P|^3\right) = O\left(2^{|P|}.|P|^3\right)$.

Complexity of Algorithm 21 (`chkEqvSCP`): In step 1, construction of η_p takes $O(|P|)$ time. In the same step, the function constructs all the paths for the two PRES+ models in $O\left(|P|^3\right)$ time as given in section 4.2.2. The complexity of each iteration of the loop of step 2 is as follows. Step 3 uses `findEqvSCP` function and takes $O\left(2^{|P|}.|P|^3\right)$ time as explained above. Π_1 is updated by the set minus operation in $O(|P|^3)$ time. So step 3 takes $O\left(2^{|P|}.|P|^3\right)$ time. Checking of the condition $\Gamma \neq \emptyset$ in step 4 takes $O(1)$ time. The complexity of the inner loop starting at step 5 is as follows. The update operations of η_t and E in step 6 take $O(1)$ time whereas that of η_p takes $O\left(|P|^2\right)$ time. So the body of the loop (step 6) takes $O\left(|P|^2\right)$ time; the loop executes $O(|P|^3)$ time; hence it takes $O(|P|^5)$ time. Step 9 takes $O(1)$ time. So, the `if`-statement comprising 4-10 takes $O(|P|^5)$ time. Thus, the body of the outer loop (steps 2-11) takes $O\left(2^{|P|}.|P|^3\right)$ (step 3) + $O\left(|P|^5\right) = O\left(2|P|.|P|^3\right)$ time. The loop executes $O\left(|P|^3\right)$ time. So the complexity of the loop is $O\left(\left(2^{|P|}.|P|^3\right).|P|^3\right) = O\left(2^{|P|}.|P|^6\right)$. Step 12 takes $O\left(|P|^3\right)$ time and step 13 takes $O(1)$ time. So the overall complexity of this module is $O\left(2^{|P|}.|P|^6\right)$.

6.4.4 Soundness of the equivalence checking algorithm

Theorem 15. *If the function* `chkEqvSCP` *(Algorithm 21) reaches step 14 and (a) returns $\Pi_{n,0} = \emptyset$, then $N_0 \sqsubseteq N_1$ and (b) if it returns $\Pi_{n,1} = \emptyset$, then $N_1 \sqsubseteq N_0$.*

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

Consider any computation $\mu_{0,p}$ of N_0. Step 2 of the function `chkEqvSCP` calls the function `constAllPathsSCP` and yields the set Π_0 of paths of N_0 from the set of cut-points. From Theorem 10, there exists a reorganized sequence of $\mu^r_{0,p}$ of paths of Π_0 such that $\mu^r_{0,p} \simeq \mu_{0,p}$. Hence, Π_0 is a path cover of N_0. So, from Theorem 11, it is required to prove that for every member α in Π_0, there is a path β of N_1 such that (i) $\alpha \simeq \beta$, (ii) the pre-places of α have correspondence with the pre-places of β and (iii) the post-places of α have correspondence with those of β. It may be noted that the algorithm `chkEqvSCP` finds a path β of Π_1 by calling the function `findEqvSCP` such that conditions (i) and (ii) are satisfied (as ensured by steps 2 and the loop comprising steps 3-7). Condition (iii) is satisfied by step 6 in `chkEqvSCP`.

□

6.5 Experimental Results

The static cut-point induced path based equivalence checking method is implemented in *C* and tested on some sequential as well as parallel examples on a 2.0 GHz Intel(R) Core(TM)2 Duo CPU machine (using only a single core). We refer to this implementation as the `SCPEQX` module. Similar to the experimentation with the dynamic cut-point based path based equivalence checking modules described in Chapter 5, the entire experimentation with the static cut-point induced path based equivalence checking module has also been carried out along two courses − one using hand constructed models and the other using models constructed by the same automated model constructor. Preparation of the example suite remains the same as that mentioned in Chapter 4. For checking equivalence between two paths, we have used the normalizer reported in [121]. The entire module is available in [14].

6.5.1 Experimentation using hand constructed models

Before discussing the observations regarding the performance of the `SCPEQX` module vis-a-vis those of the `FSMDEQX (PE)` module [14] and the `DCPEQX` module, it is worthwhile to examine the progress of the `SCPEQX` module through its output produced for the `MODN` example and compare it with the corresponding output of the `DCPEQX` module. For the `MODN` example, Figure 6.11 depicts the output produced by the path construction module and Figure 6.12 depicts the output of the equivalence checking module of the SCPEQX method. (The details of the `MODN` examples and the corresponding models are given in Figures 4.11, 4.12 and 4.13 of Chapter 4.) The output of the equivalence checking module depicts the condition of execution and the data transformation for each path in normalized form. In Figure 6.11, it is to be noted that the number of paths and the number of cut-points in `MODN` original are 11 and 16, respectively; however, the number of paths and cut-points in dynamic cut-point induced path construction method are 17 and 25 which is depicted in Figure 4.14. Although SCPEQX has no scope for path extension (as cutpoints are present only at loop entry points apart from the in-ports and the out-ports) and paths cannot extend beyond the loop entry points, the corresponding provision was retained for verification. In Figure 6.12, it may be noted that the path extension is indeed not needed for this example. This is also found to hold for all the examples experimented with, as well.

```
********************* Finding all paths of model N0 ****************************
Finding Cut-points type=0: Out-ports type=1 : In-ports, type=2: Backedge
********************************************************************************
The cutpoint list is:-
p1(type=1) p2(type=1) p3(type=1) p4(type=1) p5(type=1) p6(type=2)
p7(type=2) p8(type=2) p9(type=2) p10(type=2) p11(type=2) p12(type=2)
p13(type=2) p14(type=2) p15(type=2) p18(type=0) p22(type=2)
********************************************************************************
path 0 : <{t1}>  path 1 : <{t2}> path 2 : <{t3}> path 3 : <{t4}>
path 4 : <{t5}>  path 5 : <{t7}> path 6 : <{t6}{t8}{t9}{t11}{t13}>
path 7 : <{t6 }{t8} {t9} {t11} {t13} {t14}>
path 8 : <{t6} {t8}{ t9 }{t11}{t13}{ t15}{t16}{t18}>
path 9 : <{t6} {t8} {t9} {t11}{t13}{t15} {t16} {t18}{t19}>
path 10 : <{t6}{t8}{t9}{t11}{t13}{t15}{t16} {t18}{t19}{t20}>
##################### Path construction time #####################################
 No of places in N0: 28  No. of transitions in N0: 21
 No of paths in initial path cover of N0: 11  Exec time is 0 sec and 4208 microsecs
##################################################################################
```

Figure 6.11: Output of Static Cut-point Induced Path Constructor

Table 6.1 replicates the sizes of the original and the transformed PRES+ models in terms of numbers of their places and transitions (trans) for ready references; the number of static cut-points (SCP) and the paths have been recorded which are much less than those for the DCPEQX module as expected (see Table 4.2). The last two columns depict the path construction times for both original and transformed PRES+ models. It is to be noted that since the number of paths is less for the SCPEQX module, path construction times in Table 6.1 are smaller than the corresponding path construction times in Table 4.2.

We have also tested our SCPEQX method on the same set of five sequential examples and their parallelized versions obtained using PLuTo and Par4All. Table 6.2 summaries the path construction times for the SCPEQX method for these examples;

Example	Original PRES+				Transformed PRES+				Path Const. Time (μs)	
	Place	Trans	SCP	Paths	Place	Trans	SCP	Path	Original	Transformed
MODN	28	21	17	11	27	20	16	11	4208	3762
SUMOFDIGITS	11	9	8	5	10	9	6	5	752	697
PERFECT	19	14	12	10	14	10	11	8	2157	1987
GCD	31	27	14	10	19	17	12	10	5832	3120
TLC	30	28	16	16	40	39	16	16	6278	5672
DCT	25	18	6	1	20	13	6	1	796	782
LCM	34	28	14	10	22	18	12	10	5617	3316
LRU	39	37	18	12	45	42	20	12	5476	5987
PRIMEFAC	12	10	7	5	12	10	8	5	956	924
MINANDMAX-S	28	21	11	14	28	21	11	14	4213	4123

Table 6.1: SCP induced path construction times for hand constructed models of sequential examples

Example	Original PRES+				Transformed PRES+								Path Construction Time (μs)		
					PLuTo				Par4All				Org	PLuTo	Par4All
	place	trans	SCP	path	place	trans	SCP	path	place	trans	SCP	path			
BCM	10	6	6	1	11	7	6	1	11	7	6	1	455	434	434
MINANDMAX-P	28	21	11	14	28	21	11	14	28	21	11	14	4189	4189	4189
LUP	55	53	30	18	52	50	29	18	52	50	29	18	9873	9113	8978
DEKKER	34	32	20	12	30	29	18	12	30	29	18	12	3902	3123	3123
PATTERSON	32	30	15	10	30	28	14	10	30	28	14	10	4812	4624	4642

Table 6.2: SCP induced path construction times for hand constructed models of parallel examples

the numbers of their places and transitions (trans) are included for ready reference. The number of static cut-points (SCP) and paths are observed. The last three columns depict the path construction times which are again found to be smaller than the corresponding path construction times recorded in Table 4.5 because the number of static cut-point induced paths is less than that of the DCP induced paths (as reported in Table 4.5).

Table 6.3 depicts our observations on the performance of `SCPEQX` module made through this line of experimentation vis-a-vis the performance of `FSMDEQX (PE)` [14] and `DCPEQX` modules. In all the cases, the costly path extension is not needed for the `SCPEQX` module. We have put the two columns Extension (`FSMDEQX (PE)`) and Extension (`DCPEQX`) for ready references. The columns `FSMDEQX (PE)` Time and `SCPEQX` Total Time record the equivalence checking times taken by the FSMD equivalence checking module and the `SCPEQX` module, respectively. These figures include the path

Example	Paths		Extension	Extension	FSMDEQX (PE)	DCPEQX Time (μs)		SCPEQX			
			(FSMDEQX (PE))	(DCPEQX)	Time (μs)	EqChk	Total	Path Const Time (μs)		EqChk	Total
	Orig	Transf						Orig	Transf	Time (μs)	Time (μs)
MODN	11	11	YES	YES	16001	8506	18872	4208	3762	8345	16324
SUMOFDIGITS	5	5	YES	YES	8000	6288	8507	752	697	5660	7109
PERFECT	10	10	YES	YES	8456	5077	9685	2157	1987	4197	8341
GCD	10	10	YES	NO	12567	3957	13758	5832	3120	3226	12178
TLC	16	16	YES	YES	16121	862	16749	6278	5672	395	12345
DCT	1	1	NO	NO	2102	2054	3635	796	782	1545	3123
LCM	10	10	YES	NO	16231	6224	16742	5617	3316	3258	12461
LRU	12	12	YES	NO	20001	11435	24563	5476	5987	10878	22341
PRIMEFAC	5	5	YES	YES	6352	5505	7787	956	924	5152	7062
MINANDMAX-S	14	14	×	NO	×	5936	18395	4213	4123	4388	12724

Table 6.3: Equivalence checking results for several sequential examples using hand constructed models

construction times also. The columns DCPEQX EqChk Time and SCPEQX EqChk Time depicts only the equivalence checking time for both the methods. Calculation procedure of SCPEQX EqChk Time is similar to the procedure used for DCPEQX EqChk Time which has already been discussed in Chapter 5. By comparing the three columns namely, FSMDEQX (PE) Time, DCPEQX EqChk Time and SCPEQX EqChk Time, we observe that the performance of SCPEQX module is marginally better than the DCPEQX module and significantly better than the FSMDEQX (PE) module. In terms of total time, SCPEQX module has shown slightly better performance than the DCPEQX module but has been found to be worse than the FSMDEQX (PE) module in quite a few cases.

Example	DCPEQX Time (μs)		SCPEQX Time (μs)	
	PLuTo	Par4All	PLuTo	Par4All
BCM	4659	4659	3561	3561
MINANDMAX-P	24335	24335	18341	18341
LUP	33633	31235	29845	31012
DEKKER	13428	14352	11231	10234
PATTERSON	11231	11231	7456	8423

Table 6.4: Equivalence checking results for several parallel examples using hand constructed models

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

Figure 6.12: Output of SCPEQX module for the MODN example

The last two columns of Table 6.4 show the corresponding performance of the SCPEQX module and the DCPEQX module for validating parallelizing transformations. It is again observed that the costly path extension is not needed for the above parallel examples and SCPEQX times are slightly smaller than the corresponding DCPEQX times due to lesser number of paths.

6.5.2 Experimentation using the automated model constructor

For reasons mentioned in Chapter 4, for the experimentation using the automated model constructor we have only considered programs whose both original and transformed versions are sequential in nature. The experimental set up is exactly similar to what we have already discussed in Chapter 4.

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

Table 6.5: Equivalence checking results for several sequential examples using automated model constructor

During experimentation with automatically constructed models, we have observed that the costly path extension is again not needed for any of the cases. From Table 6.5, we notice that the time taken by the equivalence checking phase of the SCPEQX module is slightly better than the corresponding time taken by DCPEQX module (except for PRIMEFAC example). However, no distinct improvement can be identified for the SCPEQX module compared to the DCPEQX module. The Total Time for SCPEQX module is much higher than the FSMDEQX modules. However, the total time for SCPEQX module is slightly better than the total time for DCPEQX module.

6.5.3 Experimental results after introducing errors

The experimental set up for assessing the performance of the SCPEQX module for erroneous programs is exactly identical to that of the DCPEQX module comprising four types of errors as mentioned in Chapter 5.

Errors	Example	FSMDEQX (PE) Non-EqChk Time (μs)	FSMDEQX (VP) Non-EqChk Time (μs)	DCPEQX (hand const.) Non-EqChk Time (μs)	DCPEQX (automated) Non-EqChk Time (μs)	SCPEQX (hand const.) Non-EqChk Time (μs)	SCPEQX (automated) Non-EqChk Time (μs)
Type 1	MODN	15456	13471	17255	34048	12835	27132
	GCD	10435	10142	12523	42872	12123	35413
Type 2	TLC	14592	13780	16434	123414	11345	42174
Type 3	LRU	19278	16143	23143	733452	21872	685412
	LCM	11412	10619	12834	51231	11123	45987
Type 4	MINANDMAX-P	×	×	24347	×	15463	×
	PATTERSON	×	×	10913	×	6534	×

Table 6.6: Non-Equivalence checking times for faulty translations

The last five columns of Table 6.6 depict the non-equivalence detection times for the three equivalence checkers to identify the set of non-equivalent paths in each cases.

6.6 Conclusion

This chapter deals primarily with an efficient path construction algorithm and an equivalence checking method based on paths induced by static cut-points. It has been formally established first that any path based approach of checking equivalence between two PRES+ models, where the paths are defined using only static cut-points, is valid. It is also argued that the costly path extension mechanism which has been found to be necessary in the previous chapter for dynamic cut-point induced paths to handle some code motion scenarios is not needed for the static cut-point induced paths. A specific method belonging to this class has been described in detail and illustrated. The complexity and correctness issues have been treated comprehensively. Experiments on some sequential programs under code motion transformations and parallelizing transformations have been treated. On a comparative basis, the SCPEQX module has been found to be somewhat better than `DCPEQX` method; however, both take more time than `FSMDEQX` methods primarily because the path construction time for PRES+ model is significantly higher than the corresponding time for FSMD model. Some of the limitations of the present work are its inability to handle loop-shifting, software pipelining and other loop transformations for array handling programs.

Chapter 7

Conclusion

Compilation of any software program usually involves application of some compiler transformation techniques so that an optimized intermediate code is produced. For validation of all such transformations, one may try to verify the optimizing compiler which is, in general, not even partially decidable [95]. An effective alternative is to establish the computational equivalence between the original and the transformed programs, whereby we can claim that the transformations applied *for the specific instance* is correct. (Although this problem (of checking equivalence between two programs) too is not semi-decidable [95], devising a compiler verifier in the spirit of a general program verifier is held to be a much more difficult task [110].) Developing such an equivalence checker for translation validation has been the main objective of the present work.

For program analysis, it is necessary to translate any program to its equivalent formal model representation. As the main target of this work is to validate code optimizing and several parallelizing transformations, a parallel model of computation (MoC) had to be chosen. In this work, the PRES+ model, whose underlying structure is a one-safe Petri net model where tokens occupying the places are permitted to hold values, has been selected as the parallel MoC. PRES+ models have been constructed by hand from both the original and the transformed programs. The present work concentrates on devising an equivalence checker which takes as inputs two PRES+ models, N_0 and N_1, say, where N_0 corresponds to the source program and N_1 to the transformed program and returns either "yes" or "no" as its output. If the equivalence checker gives a "yes" response, then the two programs are equivalent, i.e., the transformations which are carried out by the compiler on the source program are correct; however, if it gives a "no" response, then the two programs are not necessarily inequivalent. Given the fact that the equivalence checking problem is not even semi-decidable, this behaviour of an equivalence checking algorithm is only to be expected. In other words, an equivalence checking method, even as a partial decision procedure, can only be sound but not complete and it may give *false negative results*. In other words, the "no" answer is synonymous to "may be non-equivalent".

The basic steps of the equivalence checking procedure devised in this work are as follows: (1) In the first step, a PRES+ model is partitioned into several fragments which are called *paths*; the paths are obtained by cutting a loop in at least one cut-point so that any computation of the model becomes representable as a concatenation of these paths [50]. (2) It is then checked whether for all paths in N_0, there exists a path in N_1 such that the two paths are equivalent, i.e., their data computations and conditions of execution are identical and their input and output places have correspondence. (3) Finally, steps 1 and 2 are repeated with N_0 and N_1 interchanged. We first summarize the contributions of this thesis. We then discuss some directions of future research.

7.1 Contributions

The major contributions of this work are as follows:

PRES+ model and its computational equivalence: Keeping in view the main target of equivalence checking of two models, we have defined computations of any out-port of a PRES+ model formally as a sequence of sets of transitions; each member set in the sequence comprises transitions which can execute independent of each other and are accordingly called parallelisable transitions. The first set in the sequence contains transitions which have only some in-ports as their pre-places; the last set has a single transition with the out-port in question as its post-place. Two entities are involved in any computation, namely, the condition of its execution and the data transformation it produces on the input token values to obtain an output token value. The notion of equivalence between a computation of an out-port of N_0 and that of the corresponding out-port of N_1 has then been defined. Finally, the computational containment and computational equivalence of N_0 and N_1 have been formally captured.

Dynamic cut-point induced path based equivalence checking: We have proposed two equivalence checking techniques. In the first one, by suitable placement of cut-points, the path boundaries are so ascertained that any computation can be captured as a sequence of sets of parallelisable paths; this form mimics the representation of the computation more syntactically in the sense that each path is defined as a sequence of sets of parallelisable transitions where each set is a subset of some set of transitions occurring in the computation. In a path, only the pre-places of the first set of parallelisable transitions and the post-places of the last unit set of transition are cut-points; the former is designated as the *pre-place set of a path* and the latter as its *post-place set*. Dynamic cut-points are introduced, hand-in-hand with construction of paths, using a token tracking execution. It has been formally established that for a given set of static and dynamic cut-points, the path set obtained is unique and provides *a path cover of the model* in the sense that any computation can be captured by a sequence of parallelisable paths. We refer to such paths as *Dynamic cut-point based paths or DCP paths, in short*. The DCP path construction procedure has been described in detail; its complexity analysis and correctness treatment involving termination, soundness and completeness have been carried out formally.

A path has then been characterized by a predicate depicting the condition that the token values at its pre-places must satisfy for execution of the path and a functional expression (*data transformation of the path*) over these token values depicting the value assumed by all the post-places of the path after its execution. Two paths α of the model N_0 and β of N_1 are said to be equivalent, symbolically denoted as $\alpha \simeq \beta$, if they have identical condition of execution and data transformation. Assuming that there is a correspondence relation from the set of in-ports of N_0 to the set of in-ports of N_1, if two equivalent paths are found to have correspondence among their pre-places, then their post places are made to have correspondence.

We have then formally established the validity of an equivalence checking method based on DCP paths; more specifically it is shown that, if for each DCP path α in a path cover of a model N_0, there exists a DCP path β in N_1 such that $\alpha \simeq \beta$ and their pre-places have correspondence and the correspondence of the post-places of the paths conforms to a given bijective relation among the out-ports of the models, then N_0 is indeed contained in N_1, symbolically $N_0 \sqsubseteq N_1$. Introduction of dynamic cut-points is found to shorten the paths to the extent that code motions across basic block boundaries fall in different path segments in the two models; this necessitated a special step called *path extensions* whereby path segments are needed to be concatenated to sets of parallel paths. The algorithmic modules of such an equivalence checking procedure, called `DCPEQX`, have been described in detail; complexity analysis of the procedure has been carried out and correctness issues such as termination and soundness have been treated formally.

Static cut-point induced path based equivalence checking: It has next been underlined that paths obtained from only the static cut-points which cut the loops in the model is also able to capture computations, albeit semantically. More specifically, in this case, a computation has been shown to be equivalent to a sequence of paths utilizing the property that subsets of parallelisable transitions can be executed in any arbitrary order. Validity of equivalence checking procedures which use such static cut-point induced paths have been formally established. The algorithmic modules of an equivalence checking procedure, referred to as `SCPEQX` method, has been described, the complexity analysis carried out and correctness issues treated formally.

Both the equivalence checking procedures have been implemented in *C* and experimentation carried out along two courses. The first course has used hand constructed models and the second one has used models generated by an automated model constructor. We have satisfactorily tested on several sequential and parallel examples. The translation is carried out by SPARK [56] HLS (high level synthesis) compiler and two thread level parallellizing compilers PLuTo and Par4All. For checking equivalence between two paths, we have used a normalizer which is reported in [20, 121]. For sequential benchmarks, we have compared both the methods with the two FSMD equivalence checking method reported in [20, 74]. For parallel examples, `FSMDEQX` modules could not be used because those modules cannot handle them.

In course of comparing the performance of `FSMDEQX` modules with `DCPEQX` and `SCPEQX` modules, we have observed that the path construction overhead for FSMD models is negligible compared to the PRES+ models because FSMD models do not support any thread level parallelism. Hence, the module performances have been compared in terms of both total times taken by the modules and the times taken only during the equivalence checking phase; for PRES+ models, the equivalence checking times have been obtained by subtracting the path construction times from the total times; for the FSMD models, the path construction times have been taken to be zero.

For the manually constructed models (for the sequential examples), comparison has been done only with `FSMDEQX (PE)`. The times needed by the equivalence checking phases have been found to be much smaller than those needed by the `FSMDEQX (PE)` module; also, in this regard, the `SCPEQX` module fares slightly better than the `DCPEQX` module. However, in terms of total time, while `SCPEQX` modules shows similar improvement over the `DCPEQX` module, both of them have been found to be worse than the `FSMDEQX` module for many examples.

For the models (of sequential examples) generated by the automated model constructor, however, no distinct improvement could be observed for the `SCPEQX` module over the `FSMDEQX` modules even for the equivalence checking phase. In terms of total time, `FSMDEQX` modules by far outperform the `SCPEQX` module. In terms of both these times – equivalence checking times and total times – the `SCPEQX` module performs slightly better than `DCPEQX` module.

For detecting non-equivalence between source programs and their transformed versions with manually injected errors, again `SCPEQX` module performs only a bit slower than the `FSMDEQX` modules for the manual models but is markedly slower for models generated automatically; for both types of models, `SCPEQX` module performs somewhat better than `DCPEQX` module.

For parallel examples, comparison was possible only between `SCPEQX` module and `DCPEQX` module because FSMD models cannot capture thread level parallelism and accordingly, `FSMDEQX` modules cannot handle parallel programs. A possible reason for markedly inferior performance of `SCPEQX` (and `DCPEQX`) module is the significantly larger size of automatically constructed models compared to the manually constructed models with no optimization provided in the automatically constructed models. Moreover, compared to various intricacies involved in path based PRES+ equivalence checking, both the prototype checkers, `DCPEQX` and `SCPEQX`, are themselves not optimized at all.

7.2 Comparison to related work

Translation validation was introduced by Pnueli et al. in [109] and were demonstrated by both Necula et al. [106] and Rinard et al. [118]. It is to be noted that all the above techniques are basically bisimulation based methods. A major limitation of these above methods [106, 109, 118] is that they can verify only structure preserving transformations. If the code is moved beyond the basic block boundaries [45, 56, 57, 72, 73, 114], those methods cannot validate. However, the above mentioned bisimulation approach is further enhanced by Kundu et al. [84] where they verified the high-level synthesis tool named, SPARK. This method handles loop shifting and software pipelined based transformations. The major limitation of this work is that it cannot handle code motion across loops as well as loop swapping transformations. To alleviate this shortcoming, a path based equivalence checker for the FSMD model is proposed for sophisticated uniform and non-uniform code motions and code motions across loops [20, 74]. They, however, presently cannot handle loop swapping transformations as well as several thread-level parallelizing transformations because being a sequential model of computation (MoC), FSMD models cannot capture parallel behaviours straightway; modeling concurrent behaviours using CDFGs is significantly more complex due to all possible interleavings of the parallel operations. The above mentioned pieces of work use variable based models as their modelling paradigm. In this work, we have used a value based model so the data dependence is captured more vividly. More specifically, in this work, we have focused on validation of several structure preserving, non-structure preserving and thread level parallelizing transformations using Petri net based models of programs with provisions for capturing token values over domains depending on token types. Such models have been presented in [37, 38] and used for propositional and temporal property verification of programs. Accordingly, all these methods can work with finite abstractions of the models ignoring the data values of the tokens. No work has been reported in the literature on validation of optimizing and parallelizing transformations using this modelling paradigm; mechanisms targeted to such analyses need to deal with the token values resulting in infinite state systems. On course to building such mechanism(s) as a first time effort, the present work imposes certain restrictions and uses certain features as given below (all of which may not be indispensable).

First of all, we have a place to variable association which results from the programs being modelled in a natural way. This association has made the task of establishing the path to path equivalence of the two models easier; however, such an association is not a must; as path level equivalence is identified starting from the in-ports of the models having a bijective correspondence, place correspondence can be made independent of the variable correspondence. The models are assumed to be deterministic and completely specified. Lack of non-determinism has permitted us to handle only read-only shared variables. Computations in a model use the feature that enabled transitions are simultaneously fired which lies at the core of our path structure; in the absence of writable shared variables among parallel threads, simultaneous firing of the enabled transitions capture all other schedules of these transitions. Path structures use this feature. Hence incorporating non-determinism would be a nontrivial task.

The two equivalence checking mechanisms devised in this work for the PRES+ models are path based ones. In this regard, the present work is akin to the path based equivalence checking mechanisms of the FSMD models reported in [20, 74]. However, identification of paths in a PRES+ model has more intricacies because of presence of thread level parallelism in the PRES+ models. Accordingly, during our experimentation, the `FSMDEQX` modules have scored better than the `SCPEQX` module (which, in turn, is better than the `DCPEQX` module) primarily due to the path construction overhead of the PRES+ models.

Another aspect is the model construction overhead; FSMD model construction is much easier because it does not seek to capture the scope of parallelism among data independent (sequential) segments of the code through the model structure; in contrast, PRES+ models, being value based, has the potential for capturing such parallelism through the model structure. In fact, if a PRES+ model constructor does not ensure this feature in the constructed models, then, in essence, this modelling paradigm loses its worth for validating various optimizing and parallelizing transformations. The observation that the `DCPEQX` and the `SCPEQX` modules perform better than the `FSMDEQX (PE)` module at the equivalence checking phase for the hand constructed models is due to the fact that the models are lean and have parallelism captured in their structures. (That this observation does not hold for the automatically constructed models is due to significant increase in the model size because the automated model constructor was not adequately optimized [14].) To automatically build such models thorough data flow analysis becomes needed as reported in [122].

FSMD based equivalence checking currently does not handle loop swapping but it is easily handled through PRES+ based equivalence checking. Code motion across loops is not handled by `FSMDEQX (PE)` but is handled by `FSMDEQX (VP)` and also by both `DCPEQX` and the `SCPEQX`. Parallelising transformations are currently not handled by FSMD based equivalence methods but these are handled by PRES+ based equivalence checking. All of these benefits may be attributed to the value based nature of the PRES+ model. Accordingly, these transformations are handled during the model construction phase of PRES+ based equivalence checking. This is an important qualitative difference with the FSMD based equivalence checking methods.

7.3 Scope for future work

The methods developed in this work can be enhanced to overcome their limitations. Also, there is scope of enhancing the other developed methods in other verification problems. In the following, we discuss both aspects of future works.

Enhancing the PRES+ equivalence checkers for handling non-determinism: While using PRES+ models for validating code motion transformations of sequential programs so that the transformed programs still remain sequential, we do not need to handle non-determinism. However, for parallelizing transformations, we may need to handle shared variables and non-determinism. As such, PRES+ models essentially provide parallel models of computation. To utilize its full potential it is needed to incorporate shared variables which invariably brings in non-determinism. One possible enhancement of the present work is to incorporate non-determinism and shared variables in their entirety. This would necessitate modification of the notion right from the model level. Specifically, conventional definition considers all the non-deterministic choices among the bound transitions as enabled transitions [38]. Since at any step of computation, only one of the enabled transitions is fired, it is ensured that only one of the non-deterministic choices is exercised (disabling the other choices in its wake). The notion of simultaneous firing of all the enabled transitions is ingrained in the path structure of the present work. Foregoing this feature may need to rebuild the entire edifice from the scratch. If this feature is retained, then the definition of enabled transitions needs to be suitably modified so that non-deterministic choices appear in a mutually exclusive manner in the set of all possible enabled transition sets resulting

from a given set of bound transitions. Essentially, non-determinism is manifested by the following property:
Let T_b be the set of bound transitions. Let $p \in^\circ T_b$ such that the number of post-transitions of p is more than one. These post-transitions of p reflects non-determinism if the conjunction of their guard conditions is satisfiable. Since the mechanism needs to detect presence of such non-determinism symbolically, it has to solve the satisfiability problem over integers; unless the guard conditions are linear, it cannot be achieved.

Once the non-determinism is detected the mechanism of computing all the possible sets of concurrent (enabled) transitions remains the same as the one used in the present work with deterministic models. To what extent the various stages of the present mechanisms would hold beyond this point itself would need independent study; however, this would be necessary because without shared variables, the entire gamut of parallelizing transformations would remain uncovered.

Deriving bisimulation relations from path based PRES+ equivalence checkers: For sequential MoCs, it is possible to derive a bisimulation relation from the output of a path based equivalence checker [83]. It is to be noted that none of the earlier methods that establish equivalence through construction of bisimulation relations has been shown to tackle code motion across loops and several thread level parallelizing transformations. Both `DCPEQX` and `SCPEQX` procedures have the capability of validating such transformations. So, if we evolve a mechanism of deriving a bisimulation relation from the outputs of the above two path based equivalence checking procedures (in the same line as demonstrated in [83]), then it can be shown that bisimulation relations exist under such transformations.

Translation validation using PRES+ models for loop transformations: The two equivalence checking procedures described in this work have been shown to successfully handle some loop transformations such as, loop splitting and merging for scalars (vide Example 22). Since the PRES+ model in its present form does not provide for representing arrays, it cannot be examined to what extent the model is suitable for handling loop transformations which invariably involve arrays. So, one immediate scope of the present work is to extend the model to represent arrays. McCarthy's access and change functions [97] can provide an effective way of representing arrays in the model. Since the PRES+ models have the potential of capturing both control dependence and data dependence, it will be interesting to examine how this feature influence the process of validation of such transformations using PRES+ models.

Other scheme for cut-point introduction: There should exist scope for making the module more efficient and also exploring other alternatives followed by performance comparison through experiments. One immediately apparent alternative is to have additional cut-points at the branching points (i.e., places having more than one mutually exclusive post-transitions). Such an approach would result in *lesser* number of *shorter* paths but would necessitate more frequent path extensions. Another alternative can be to build the entire mechanism on computations based on firing of only one of the enabled transitions at a time; this, however, would possibly necessitate devising newer definitions and algorithms and accordingly merits an independent treatment.

Bisimulation based equivalence checking for PRES+ models: A bisimulation based equivalence checking is reported in [84]. The work uses message passing for communication among the parallel threads. A sophisticated transformation namely, loop shifting, has been shown to be verifiable using this method. There is no method available in the literature for bisimulation based equivalence checking for parallel programs which communicate through shared variables. PRES+ models may provide a uniform modelling paradigm for both kinds of communications among the parallel threads. Bisimulation based equivalence checking approaches resort to iterations to arrive at the bisimulation relation. Although deriving bisimulation relation from the output of path based equivalence checking methods has been shown to be possible, scenarios resulting out of transformations such as, loop shifting, will remain beyond the scope of such mechanisms. Accordingly, devising a bisimulation based equivalence checking method for PRES+ models is an important future direction.

7.4 Conclusion

Many safety critical applications such as, automobiles, avionics, manufacturing processes, nuclear reactors, etc., involve concurrent or parallel subsystems. They are required to be dependable in their performance. Hence, there is a growing concern to develop automated methods for formally verifying concurrent embedded systems. A typical synthesis flow of complex systems like VLSI circuits or embedded systems transforms the input behaviour to optimize time and physical resources using code transformation techniques which change the control flow in the behavioural specification significantly. Accordingly, the challenges in establishing validity of a translation phase by demonstrating equivalence between the original behaviour and the transformed behaviour increases manifold. This thesis has addressed verification of certain behavioural transformations during the code optimization phase of a compiler. We believe integrating these methods with compilers will make the synthesis process more rigorous.

Appendix A

Appendix

A.1 List of examples

```
int main(void) {
    int s=0,i=0,n,a,b,sout,sT0_6,sT1_8,sT2_8,sT3_10,sT4_14,sT5_12,sT6_15;
    do {
        sT0_6 = (i <= 15);/* Statement is trimmed */
        if (sT0_6) {/* In trimmed version if (i<=15) */
            i=(i + 1);sT1_8=(b % 2);sT5_12=(a * 2);sT2_8=(sT1_8== 1);
            sT4_14 =(sT5_12 >= n);b=(b / 2);
            if (sT2_8) { s = (s + a);sT6_15 = (sT5_12 - n);a = sT5_12;
            } else {sT6_15 = (sT5_12 - n);a = sT5_12;
            }/* Replace by a = a*2 */
            sT3_10 = (s >= n);/* Trimmed out this statement */
            if (sT3_10) {s = (s - n);
            }/* end of if-else (sT3_10) */
            if (sT4_14) {a = sT6_15;
            }/* end of if-else (sT4_14) */
        } /* end of loop condition */
        else
            break;
    } while (1);
    sout = s;return 0;
}
```

Figure A.1: SPARK output of MODN

```
main() {
    int y1, y2, res, yout, i, res = 1;
    for (i = 0; y1 != y2; i++) {
        if (y1 % 2 == 0)
           if (y2 % 2 == 0) {
               res = res * 2;
               y1 = y1 / 2;
               y2 = y2 / 2;
           } else
               y1 = y1 / 2;
        else if (y2 % 2 == 0)
            y2 = y2 / 2;
        else if (y1 > y2)
            y1 = y1 - y2;
        else
            y2 = y2 - y1;
    }
    res = res * y1;
    yout = res;
}
```

(a)

```
int main(void){
 int y1; int y2;int res; int yout;
 int i;int sT0_6;int sT1_8; int sT2_8;
 int sT3_9; int sT4_9;int sT5_17;int sT6_17;
 int sT7_19;int returnVar_main;
  res = 1;i = 0;
  do {
        sT0_6 = (y1 != y2);
        if (sT0_6) {
            i = (i + 1);
            sT1_8 = (y1 % 2);
            sT2_8 = (sT1_8 == 0);
        }
        if (sT2_8) {
           sT3_9 = (y2 % 2);
           sT4_9 = (sT3_9 == 0);
           if (sT4_9) {
              res = (res * 2);
              y1 = (y1 / 2);y2 = (y2 / 2);
           }
           else {
              y1 = (y1 / 2);
           }
        }
        else {
             sT5_17 = (y2 % 2);
             sT6_17 = (sT5_17 == 0);
             if (sT6_17) {
                y2 = (y2 / 2);
             }
             else {
                    sT7_19 = (y1 > y2);
                    if (sT7_19) {
                       y1 = (y1 - y2);
                    }
                    else {
                       y2 = (y2 - y1);
                    }
             }
       }
    }
    else
    break;
  } while (1);
   res = (res * y1);yout = res;
}
```

(b)

Figure A.2: Original and transformed program of GCD

```
void main ()
{
 int a0;int i0;int i7;
 int a7;int b1;int i1;
 int i2;int a2;int a3;
 int i3;int i4;int a4;
 int b5;int i5;int i6;
 int a6;int b0;int c4;
 int d2;int b6;int d3;
 int c7;int c5;int d0;
 int d1;int c6;int d5;
 int d7;int tmp0,tmp1;
 int d4;int d6;int o4;
 int o0;int tmp2;int o2;
 int o6;int o1;int o7;
 int o3; int o5;
  q00:a0=i0+i7;a7=i0-i7;b1=i1+i2;
      a2=i1-i2;a3=i3+i4;a4=i3-i4;
      b5=i5+i6;a6=i5-i6;
      goto q02;
  q02:b0=a0+a4;c4=a0-a4;d2=a2+a6;
      b6=a2-a6;d3=a3+a7;c7=a3-a7;
      c5=b5*678;goto q03;
  q03:d0=b0+b1;d1=b0-b1;c6=b6*678;
      d5=c5+c7;d7=c5-c7;tmp0=d2*4;
      tmp1=d3*4;tmp0=d2*5;
      tmp1=d3*5;goto q04;
  q04:d4=c4+c6;d6=c4-c6;o4=d0*678;
      o0=d1*678;tmp2=tmp0+tmp1;
      tmp1=d5*4;tmp1=d5*5;tmp1=d7*4;
      tmp1=d7*5;goto q05;
  q05:o2=tmp2;o6=tmp2;tmp0=d4*5;
      tmp0=d4*4;tmp0=d6*5;
      tmp0=d6*4;goto q06;
  q06:tmp2=tmp0+tmp1;tmp2=tmp0-tmp1;
      tmp2=tmp0+tmp1;
      tmp2=tmp0-tmp1;goto q07;
  q07:o1=tmp2;o7=tmp2;o3=tmp2;o5=tmp2;
      goto q09;
  q09:;
}
```

(a)

```
void main ()
{
 int a0;int i0; int i7; int a7;
 int b1;int i1; int i2; int a2;
 int a3;int i3; int i4; int a4;
 int b5;int i5; int i6; int a6;
 int b0;int c4; int d2; int b6;
 int d3;int c7; int c5; int d0;
 int d1;int c6; int d5; int d7;
 int tmp0;int tmp1;int d4;int d6;
 int o4;int o0; int tmp2;int o2;
 int o6;int o1; int o7; int o3;
 int o5;
  q00:a0=i0+i7;a7=i0-i7;
      b1=i1+i2;a2=i1-i2;
      a3=i3+i4;a4=i3-i4;
      b5=i5+i6;a6=i5-i6;
      goto q02;
  q02:b0=a0+a4;c4=a0-a4;
      d2=a2+a6;b6=a2-a6;
      d3=a3+a7;c7=a3-a7;
      c5=b5*678;goto q03;
  q03:d0=b0+b1;d1=b0-b1;
      c6=b6*678;d5=c5+c7;
      d7=c5-c7;tmp0=d2*4;
      tmp1=d3*4;tmp0=d2*5;
      tmp1=d3*5;goto q04;
  q04:d4=c4+c6;d6=c4-c6;
      o4=d0*678;o0=d1*678;
      tmp2=tmp0+tmp1;tmp1=d5*4;
      tmp1=d5*5;tmp1=d7*4;
      tmp1=d7*5;goto q06;
  q06:o1=tmp2;o7=tmp2;o3=tmp2;
      o5=tmp2;goto q07;
  q07:;

}
```

(b)

Figure A.3: Original and transformed program of DCT

```
main(){
 int current_state,
 newHL,newFL,cars,
 timeOutL,timeOutS,
 newST,FarmL,state,
 HiWay,StartTimer,newstate,;
  if (current_state == 0) {
    newHL = 4;newFL = 6;
    if (cars == 1 && timeOutL == 1) {
       newstate = 4;newST = 1;
    } else {
       newstate = 0;newST = 0;
    }
  }
  if (current_state == 4) {
    newHL = 2;newFL = 6;
    if (timeOutS == 1) {
        newstate = 2;newST = 1;
    } else {
        newstate = 6;newST = 0;
    }
  }
  if (current_state == 2) {
    newHL = 6;newFL = 4;
    if (cars == 0 || timeOutL == 1) {
        newstate = 6;newST = 1;
    } else {
        newstate = 2;newST = 0;
    }
  }
  if (current_state == 6) {
    newHL = 6;newFL = 2;
    if (timeOutS = 1) {
        newstate = 0;newST = 1;
    } else {
        newstate = 6;newST = 0;
    }
  }
  if (current_state == 7) {
    newHL = 0;newFL = 0;
    newstate = 0;newST = 0;
 }
 state = newstate;HiWay = newHL;
 FarmL = newST;StartTimer = newST;
}
```

(a)

```
int main(void){
 int current_state;int newstate, newHL,
 cars; timeOutL, timeOutS,newFL,
 newST, FarmL, state, HiWay,StartTimer,
 sT0_6,sT1_10,sT2_10,sT3_10,sT4_21,sT13_40;
 sT5_25,sT6_36,sT7_40,sT8_40,sT9_40,sT10_51
 sT11_55,sT12_66,sT14_40,
 sT0_6=(current_state == 0);
 sT6_36 = (current_state == 2);
   if (sT0_6) {sT1_10 = (timeOutL == 1);
       sT2_10 = (cars == 1);newHL = 4;
       newFL=6;ST3_10=((sT2_10) && (sT1_10));
     if (sT3_10){newstate = 4;newST = 1;
        sT10_51 = (current_state == 6);
        sT4_21 = (current_state == 4);}
     else{newstate = 0;newST = 0;
         sT10_51 = (current_state == 6);
         sT4_21 = (current_state == 4);
         }
   }
  else{sT10_51 = (current_state == 6);
      sT4_21 = (current_state == 4);}
  if (sT4_21){sT5_25 = (timeOutS == 1);
     newHL = 2;newFL = 6;
    if (sT5_25){newstate = 2;newST = 1;
        sT13_40 = (timeOutL == 1);
        sT14_40 = (cars == 0);
    }
    else{newstate = 6;newST = 0;
         sT13_40 = (timeOutL == 1);
         sT14_40 = (cars == 0);
    }
  }
  else{sT12_66 = (current_state == 7);
  }
  if (sT10_51){newHL= 6;newFL= 2;
      timeOutS= 1;sT11_55= 1;
    if (1){newstate = 0;newST = 1;
    }
    else{newstate = 6;newST = 0;
    }
  }
  if (sT12_66){newHL = 0;newFL= ewstate = 0;
     state =HiWay=FarmL=StartTimer = 0;
  }
  else{state = newstate;HiWay = newHL;
      FarmL = newST;StartTimer = newST;
  }
}
```

(b)

Figure A.4: Original and transformed program of TLC

```
int n, sum;
   if(n > 9){
     sum = 0;
     Loop:
          if(n > 0){
            sum += n%10; n = n/10;
            goto Loop;
          }
          else if(sum > 9){
                 n = sum;sum = 0;
                 goto Loop;
          }
          else{
               n = sum;
          }
   }
```

(a)

```
int n, sum;
 while(n > 9){
       sum = 0;
       while(n > 0){
             sum += n%10;n = n/10;
       }
    n = sum;
 }
```

(b)

Figure A.5: Original and transformed program of SUMOFDIGITS

```
int sum = 1, i = 2, n, out;
  while( i < n ){
    if(n % i == 0)
      sum = sum + i;
      i = i + 1;
  }
  if( sum == n ){
     out = 1;
  }
  else {
     out = 0;
  }
```

(a)

```
int sum = 1, i = 2, n, out;
 if( i < n ){
    Loop:
     if(n % i == 0 && i+1 < n){
        sum = sum + i;i = i + 1;goto Loop;
     }
     if(n % i != 0 && i+1 < n){
       i = i + 1;goto Loop;
     }
     if(n % i == 0 && i+1 >= n){
       sum = sum + i;i = i + 1;
     }
     if(n % i != 0 && i+1 >= n){
       i = i + 1;
     }
 }
 if( sum == n ){
     out = 1;
 }
 else {
     out = 0;
}
```

(b)

Figure A.6: Original and transformed program of PERFECT

```
main() {
    int y1, y2, res, yout,
        yout1, i, res = 1;
    for (i = 0; y1 != y2; i++) {
        if (y1 % 2 == 0)
           if (y2 % 2 == 0) {
               res = res * 2;
               y1 = y1 / 2;
               y2 = y2 / 2;
           } else
               y1 = y1 / 2;
        else if (y2 % 2 == 0)
            y2 = y2 / 2;
        else if (y1 > y2)
            y1 = y1 - y2;
        else
            y2 = y2 - y1;
    }
    res = res * y1;
    yout = res;
    yout1=(y1*y2)/yout;
}
       (a)
```

```
int main(void)
{
 int y1; int y2,res,yout,yout1;
 int i;int sT0_6,sT1_8,sT2_8;
 int sT3_9,sT4_9,sT5_17,sT6_17;
 int sT7_19, returnVar_main;
  res = 1;i = 0;
  do {
        sT0_6 = (y1 != y2);
        if (sT0_6) {
            i = (i + 1);
            sT1_8 = (y1 % 2);
            sT2_8 = (sT1_8 == 0);
         }
        if (sT2_8) {
           sT3_9 = (y2 % 2);
           sT4_9 = (sT3_9 == 0);
           if (sT4_9) {
              res = (res * 2);
              y1 = (y1 / 2);y2 = (y2 / 2);
           } else {
              y1 = (y1 / 2);
             }
        } /* sT2_8       */
        else {
            sT5_17 = (y2 % 2);
            sT6_17 = (sT5_17 == 0);
            if (sT6_17) {
                y2 = (y2 / 2);
            } else {
                sT7_19 = (y1 > y2);
                if (sT7_19) {
                   y1 = (y1 - y2);
                } else {
                  y2 = (y2 - y1);
                }
            }
       }
    }
    else
    break;
    } while (1);
    res = (res * y1);
    yout = res;
    yout1=(y1*y2)/yout;
}
     (b)
```

Figure A.7: Original and transformed program of LCM

```
void main ()
{
 int eop;int breakLoop;int clk;int X; int Y;int reset;
 int found;int newGuy;int mru;int i;int last;int temp;
 int j;int temp2;int temp_list; int list;int pushTo;int temp1;
 int temp3;int temp4;int lru;
  q000: eop=0; breakLoop=0;goto q001;
  q001: if (clk!=1){goto q001;}
        else {goto q002;}
  q002: if (eop==0){X=100;Y=200;goto q003;}
        else {goto q033;}
  q003: if (clk!=1) {goto q003;}
        else {goto q004;}
  q004: if (reset==1){eop=1;breakLoop=1;goto q005;}
        else {eop=0;breakLoop=0;goto q005; }
  q005: if (eop==0){found=0;newGuy=mru;i=0;goto q006;}
        else {goto q002;}
  q006: if (i<last&&found==0&&breakLoop==0){temp=0;j=0;goto q007;}
        else {goto q014;}
  q007: if (j<=i) {temp=temp*256;j=j+1;goto q007;}
        else {temp2=temp+8;goto q009;}
  q009: temp_list=list%temp2;goto q010;
  q010: temp_list=temp_list/temp;goto q011;
  q011: if (temp_list==newGuy) {found=1;goto q012;}
        else{i=i+1;goto q012;}
  q012: if (clk!=1){goto q012;}
        else {eop=0;breakLoop=0;goto q013;}
  q013: if (reset==1){eop=1;breakLoop=1;goto q006;}
        else{goto q006;}
  q014: if (eop==0){eop=0;goto q015;}
        else {goto q002;}
  q015: if (found==1){pushTo=i;goto q018;}
        else {goto q016;}
  q016: if (last<7){pushTo=last+1;goto q017;}
       else{pushTo=last;goto q017;}
  q017:last=pushTo;goto q018;
  q018: if (clk!=1){goto q018;}
        else {goto q019;}
  q019:if (reset==1){eop=1;goto q020;}
       else {goto q020;}
  q020: if (eop==0) {temp=0;j=0;eop=0;goto q021;}
        else {goto q002;}
  q021: if (j<=pushTo){temp=temp*256;j=j+1;goto q021;}
        else {temp_list=list%temp;goto q022;}
  q022: temp_list=temp_list*256;temp1=temp*256;goto q023;
  q023: list=list/temp1;goto q024;
  q024: list=list*temp1;goto q025;
  q025: list=list+temp_list;goto q026;
  q026: if (reset==1){eop=1;goto q027;}
        else{goto q027;}
  q027: if (eop==0){list=list/256;goto q028;}
        else{goto q002;}
  q028: list=list*256;temp3=last+1;goto q029;
  q029: list=list+newGuy;temp3=temp3*256;temp4=last*256;goto q030;
  q030: temp_list=list%temp3;goto q031;
  q031: temp_list=temp_list/temp4;goto q032;
  q032: lru=temp_list; goto q002;
  q033:  ;
 }
```

```
void main ()
{
 int eop;int breakLoop;int clk;int reset;
 int X;int Y;int found;int newGuy;int mru;
 int i;int last;int temp;int j;int temp2;
 int temp_list;int list;int pushTo;int temp1;
 int temp3;int temp4;int lru;



  q000: eop=0;breakLoop=0;goto q001;
  q001:if (clk!=1){goto q001;}
       else{goto q002;}
  q002:if (eop==0){goto q003;}
       else {goto q033;}
  q003:if (clk!=1){goto q003;}
       else if (! (clk!=1) &&reset==1)
        {eop=1;breakLoop=1;X=100;Y=200;goto q005;}
       else {eop=0;breakLoop=0;X=100;Y=200;goto q005;}
  q005:if (eop==0){found=0;newGuy=mru;i=0;goto q006;}
       else{goto q002;}
  q006:if (i<last&&found==0&&breakLoop==0){temp=0;j=0;goto q007;}
       else{goto q014;}
  q007:if (j<=i){temp=temp*256;j=j+1;goto q007;}
       else{temp2=temp+8;goto q009;}
  q009:temp_list=list%temp2;goto q010;
  q010:temp_list=temp_list/temp;goto q011;
  q011:if (temp_list==newGuy){found=1;goto q012;}
       else{i=i+1;goto q012;}
  q012:if (clk!=1){goto q012;}
       else{eop=0;breakLoop=0;goto q013;}
  q013:if (reset==1){eop=1;breakLoop=1;goto q006;}
       else{goto q006;}
  q014:if (eop==0){eop=0;goto q015;}
       else{goto q002;}
  q015:if (found==1){pushTo=i;goto q018;}
       else{goto q016;}
  q016:if (last<7){pushTo=last+1;goto q017;}
       else{pushTo=last;goto q017;}
  q017:last=pushTo;goto q018;
  q018:if (clk!=1){goto q018;}
       else{goto q019;}
  q019:if (reset==1){eop=1;goto q020;}
       else{goto q020;}
  q020:if (eop==0){temp=0;j=0;eop=0;goto q021;}
       else{goto q002;}
  q021:if (j<=pushTo){temp=temp*256;j=j+1;goto q021;}
       else{temp_list=list%temp;goto q022;}
  q022:temp_list=temp_list*256; temp1=temp*256; goto q023;
  q023:list=list/temp1;goto q024;
  q024:list=list*temp1;goto q025;
  q025:list=list+temp_list;goto q026;
  q026:if (reset==1){eop=1;goto q027;}
       else{goto q027;}
  q027:if (eop==0){list=list/256;goto q028;}
       else{goto q002;}
  q028:list=list*256;temp3=last+1;goto q029;
  q029:list=list+newGuy;temp3=temp3*256;temp4=last*256;goto q030;
  q030:temp_list=list%temp3;goto q031;
  q031:temp_list=temp_list/temp4;goto q032;
  q032:lru=temp_list;goto q002;
  q033:;
 }
```

Figure A.8: Source and transformed program of LRU

```
int k,m,x,xout;
 while(x > 1)
 {
   for(k = 2; k <= x; k++){
      m = x % k;
      if(m == 0){
        output (k);
        x = x / k; break;
      }
   }
 }
      (a)
```

```
int k,m;
  while(x > 1){
    for(k = 2; k <= x; k++){
       m = x % k;
       if(m == 0 && x/k > 1){
         output(k);
         x = x / k;
         break;
       }
       if(m == 0 && !(x/k > 1)){
         output(k);
         x = x / k;
         goto breakLoop;
       }
    }
 }
 breakLoop:;
       (b)
```

Figure A.9: Original and transformed program of PRIMEFAC

A.2 List of erroneous program

Type 2 error

```
int main(void)
{
 int current_state;int newstate;int newHL;int newFL;
 int cars;int timeOutL;int timeOutS;
 int newST;int FarmL;int state;int HiWay;
 int StartTimer;int sT0_6;int sT1_10;
 int sT2_10;int sT3_10;int sT4_21;sT13_40;
 int sT5_25;int sT6_36;int sT7_40;
 int sT8_40;int sT9_40;int sT10_51; int sT11_55;
 int sT12_66; int sT14_40;
 sT0_6 = (current_state == 0); sT6_36 = (current_state == 2);
   if (sT0_6)
   {
    sT1_10 = (timeOutL == 1);sT2_10 = (cars == 1);newHL = 4;
    newFL = 6;ST3_10 = ((sT2_10) && (sT1_10));
    if (sT3_10)
     {newstate = 4;newST = 1;
      sT10_51 = (current_state == 6);
      sT4_21 = (current_state == 4);
     }
    else
     {newstate = 0;newST = 0;
      sT10_51 = (current_state == 6);sT4_21 = (current_state == 4);
     }
   }
  else
   {sT10_51 = (current_state == 6);sT4_21 = (current_state == 4);
   }
  if (sT4_21)
   {sT5_25 = (timeOutS == 1);newHL = 2;newFL = 6;
    if (sT5_25)
     {newstate = 2;newST = 1;sT13_40 = (timeOutL == 1);
      sT14_40 = (cars == 0);
     }
    else{newstate = 6;newST = 0;sT13_40 = (timeOutL == 1);
         sT14_40 = (cars == 0);}
   }
  else
   {sT12_66 = (current_state == 7);}
  if (sT10_51)
   {newHL= 6;newFL= 2;timeOutS= 1;sT11_55= 1;
    if (1){newstate = 0;newST = 1;}
    else{newstate = 6;newST = 0;}
   }
  if (sT12_66)
   {newHL = 0;
    newFL = 0;
    newstate= 0;
    newST = 0;
    state = 0;
    HiWay = 0;
    FarmL = 0;
    StartTimer = 0;
   }
  else
   {state = newstate;
    HiWay = newHL;
    FarmL = newST;
    StartTimer = newST;}
}
                (a)
```

```
int main(void)
{
 int current_state;int newstate;int newHL;int newFL;
 int cars;int timeOutL;int timeOutS;
 int newST;int FarmL;int state;int HiWay;
 int StartTimer;int sT0_6;int sT1_10;
 int sT2_10;int sT3_10;int sT4_21;sT13_40;
 int sT5_25;int sT6_36;int sT7_40;
 int sT8_40;int sT9_40;int sT10_51; int sT11_55;
 int sT12_66; int sT14_40;
 sT0_6 = (current_state == 0); sT6_36 = (current_state == 2);
   if (sT0_6)
   {
    sT1_10 = (timeOutL == 1);sT2_10 = (cars == 1);newHL = 4;
    newFL = 6;ST3_10 = ((sT2_10) && (sT1_10));
    newST = 0;/* move from else block */
    if (sT3_10)
     {newstate = 4;newST = 1;
      sT10_51 = (current_state == 6);
      sT4_21 = (current_state == 4);
     }
    else
     {newstate = 0;
      sT10_51 = (current_state == 6);sT4_21 = (current_state == 4);
     }
   }
  else
   {sT10_51 = (current_state == 6);sT4_21 = (current_state == 4);
   }
  if (sT4_21)
   {sT5_25 = (timeOutS == 1);newHL = 2;newFL = 6;
    if (sT5_25)
     {newstate = 2;newST = 1;sT13_40 = (timeOutL == 1);
      sT14_40 = (cars == 0);
     }
    else{newstate = 6;newST = 0;sT13_40 = (timeOutL == 1);
         sT14_40 = (cars == 0);}
   }
  else
   {sT12_66 = (current_state == 7);}
  if (sT10_51)
   {newHL= 6;newFL= 2;timeOutS= 1;sT11_55= 1;
    if (1){newstate = 0;newST = 1;}
    else{newstate = 6;newST = 0;}
   }
  if (sT12_66)
   {newHL = 0;
    newFL = 0;
    newstate = 0;
    newST = 0;
    state = 0;
    HiWay = 0;
    FarmL = 0;
    StartTimer = 0;
   }
  else
   {state = newstate;
    HiWay = newHL;
    FarmL = newST;
    StartTimer = newST;}
}
                (b)
```

Figure A.10: Correct and erroneous program of TLC

Type 3 error

```
void main ()
{
 int eop;int breakLoop;int clk;int reset;
 int X;int Y;int found;int newGuy;int mru;
 int i;int last;int temp;int j;int temp2;
 int temp_list;int list;int pushTo;int temp1;
 int temp3;int temp4;int lru;



  q000: eop=0;breakLoop=0;goto q001;
  q001:if (clk!=1){goto q001;}
       else{goto q002;}
  q002:if (eop==0){goto q003;}
       else {goto q033;}
  q003:if (clk!=1){goto q003;}
       else if (! (clk!=1) &&reset==1)
        {eop=1;breakLoop=1;X=100;Y=200;goto q005;}
       else {eop=0;breakLoop=0;X=100;Y=200;goto q005;}
  q005:if (eop==0){found=0;newGuy=mru;i=0;goto q006;}
       else{goto q002;}
  q006:if (i<last&&found==0&&breakLoop==0){temp=0;j=0;goto q007;}
       else{goto q014;}
  q007:if (j<=i){temp=temp*256;j=j+1;goto q007;}
       else{temp2=temp+8;goto q009;}
  q009:temp_list=list%temp2;goto q010;
  q010:temp_list=temp_list/temp;goto q011;
  q011:if (temp_list==newGuy){found=1;goto q012;}
       else{i=i+1;goto q012;}
  q012:if (clk!=1){goto q012;}
       else{eop=0;breakLoop=0;goto q013;}
  q013:if (reset==1){eop=1;breakLoop=1;goto q006;}
       else{goto q006;}
  q014:if (eop==0){eop=0;goto q015;}
       else{goto q002;}
  q015:if (found==1){pushTo=i;goto q018;}
       else{goto q016;}
  q016:if (last<7){pushTo=last+1;goto q017;}
       else{pushTo=last;goto q017;}
  q017:last=pushTo;goto q018;
  q018:if (clk!=1){goto q018;}
       else{goto q019;}
  q019:if (reset==1){eop=1;goto q020;}
       else{goto q020;}
  q020:if (eop==0){temp=0;j=0;eop=0;goto q021;}
       else{goto q002;}
  q021:if (j<=pushTo){temp=temp*256;j=j+1;goto q021;}
       else{temp_list=list%temp;goto q022;}
  q022:temp_list=temp_list*256; temp1=temp*256; goto q023;
  q023:list=list/temp1;goto q024;
  q024:list=list*temp1;goto q025;
  q025:list=list+temp_list;goto q026;
  q026:if (reset==1){eop=1;goto q027;}
       else{goto q027;}
  q027:if (eop==0){list=list/256;goto q028;}
       else{goto q002;}
  q028:list=list*256;temp3=last+1;goto q029;
  q029:list=list+newGuy;temp3=temp3*256;temp4=last*256;goto q030;
  q030:temp_list=list%temp3;goto q031;
  q031:temp_list=temp_list/temp4;goto q032;
  q032:lru=temp_list;goto q002;
  q033:;
 }
```

```
 void main ()
{
 int eop;int breakLoop;int clk;int reset;
 int X;int Y;int found;int newGuy;int mru;
 int i;int last;int temp;int j;int temp2;
 int temp_list;int list;int pushTo;int temp1;
 int temp3;int temp4;int lru;



  q000: eop=0;breakLoop=0;goto q001;
  q001:if (clk!=1){eop=0; goto q001;}
      /* eop=0 moves from q020 */
       else{goto q002;}
  q002:if (eop==0){goto q003;}
       else {goto q033;}
  q003:if (clk!=1){goto q003;}
       else if (! (clk!=1) &&reset==1)
        {eop=1;breakLoop=1;X=100;Y=200;goto q005;}
       else {eop=0;breakLoop=0;X=100;Y=200;goto q005;}
  q005:if (eop==0){found=0;newGuy=mru;i=0;goto q006;}
       else{goto q002;}
  q006:if (i<last&&found==0&&breakLoop==0){temp=0;j=0;goto q007;}
       else{goto q014;}
  q007:if (j<=i){temp=temp*256;j=j+1;goto q007;}
       else{temp2=temp+8;goto q009;}
  q009:temp_list=list%temp2;goto q010;
  q010:temp_list=temp_list/temp;goto q011;
  q011:if (temp_list==newGuy){found=1;goto q012;}
       else{i=i+1;goto q012;}
  q012:if (clk!=1){goto q012;}
       else{eop=0;breakLoop=0;goto q013;}
  q013:if (reset==1){eop=1;breakLoop=1;goto q006;}
       else{goto q006;}
  q014:if (eop==0){eop=0;goto q015;}
       else{goto q002;}
  q015:if (found==1){pushTo=i;goto q018;}
       else{goto q016;}
  q016:if (last<7){pushTo=last+1;goto q017;}
       else{pushTo=last;goto q017;}
  q017:last=pushTo;goto q018;
  q018:if (clk!=1){goto q018;}
       else{goto q019;}
  q019:if (reset==1){eop=1;goto q020;}
       else{goto q020;}
  q020:if (eop==0){temp=0;j=0;goto q021;}
       else{goto q002;}
  q021:if (j<=pushTo){temp=temp*256;j=j+1;goto q021;}
       else{temp_list=list%temp;goto q022;}
  q022:temp_list=temp_list*256; temp1=temp*256; goto q023;
  q023:list=list/temp1;goto q024;
  q024:list=list*temp1;goto q025;
  q025:list=list+temp_list;goto q026;
  q026:if (reset==1){eop=1;goto q027;}
       else{goto q027;}
  q027:if (eop==0){list=list/256;goto q028;}
       else{goto q002;}
  q028:list=list*256;temp3=last+1;goto q029;
  q029:list=list+newGuy;temp3=temp3*256;temp4=last*256;goto q030;
  q030:temp_list=list%temp3;goto q031;
  q031:temp_list=temp_list/temp4;goto q032;
  q032:lru=temp_list;goto q002;
  q033:;
 }
```

Figure A.11: Correct and erroneous program of LRU

Type 4 error

```
int main()
{
  int num, max, min, i, j, out;
  printf ("Enter seven numbers: ");
  scanf ("%d", &num);
  max = min = num;
  #pragma scop
  for ( i = 0; i < 3; i++)
    {scanf ("%d", &num);
      if (max < num)max = num;
    }
   for ( j = 0; j < 3; j++)
   {scanf ("%d", &num);
    if (min > num)min = num;
   }
  #pragma endscop
 out = min+max;
 printf ("%d ", out);
 return 0;
}
   (a)
```

```
int main()
{ int num, max, min, i,j, out;
 printf ("Enter seven numbers: ");
 scanf ("%d", &num);
 max = min = num;
 # CLooG code
   for ( i = 0; i < 3; i++)
     {scanf ("%d", &num);
      if (max < num)max = num;
     }
 \PAR
   for ( j = 0; i < 3; j++)
    {
      if (min > num)min = num;
     }
 # CLooG code
 out = min+max;
 printf ("%d ", out);
 return 0;
}
    (b)
```

Figure A.12: Correct and erroneous program of MINMAX

A.3 List of PRES+ models

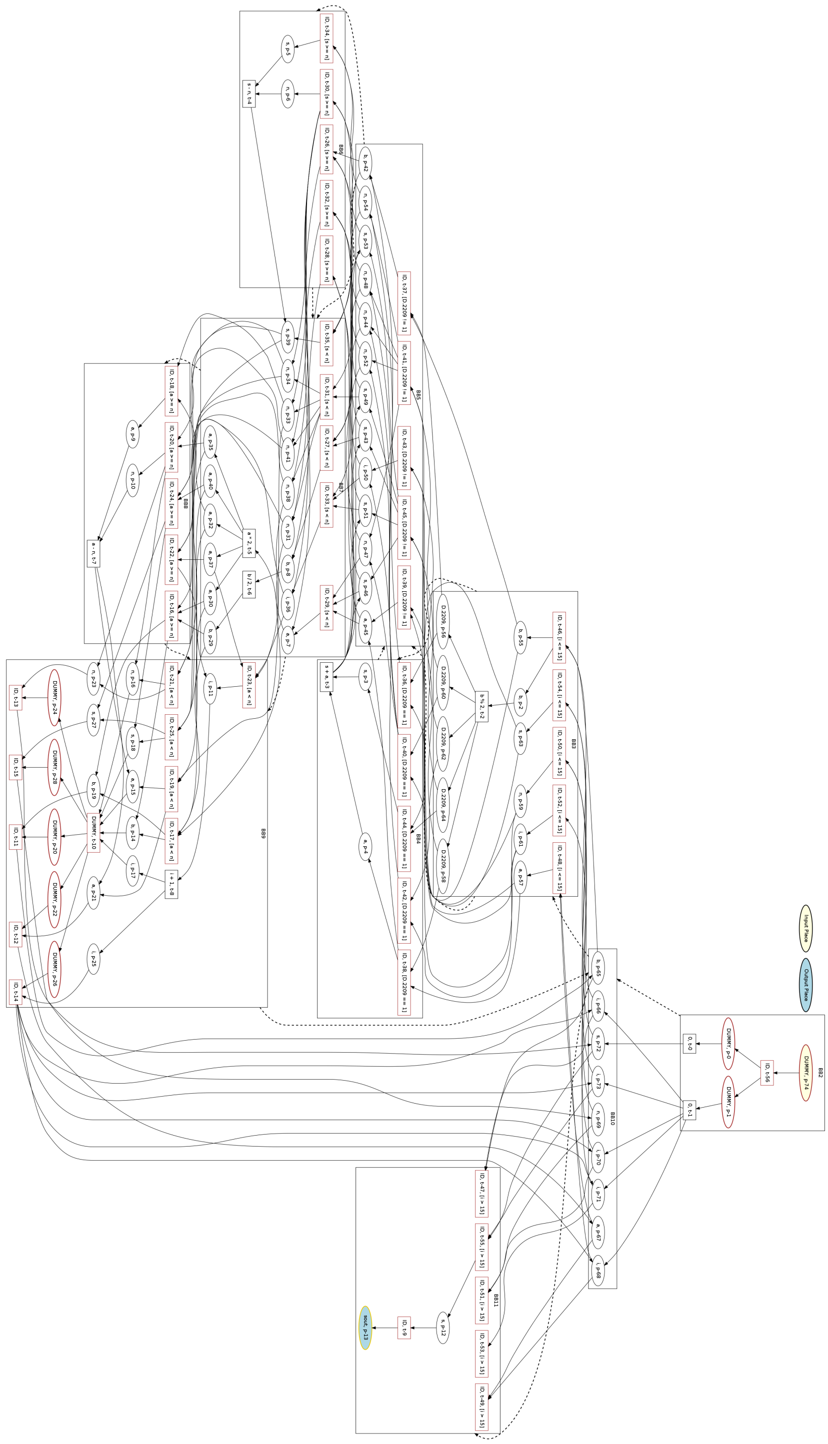

Figure A.13: PRES+ model for MODN original using automated model constructor (to be viewed with adequate magnification in PDF viewer)

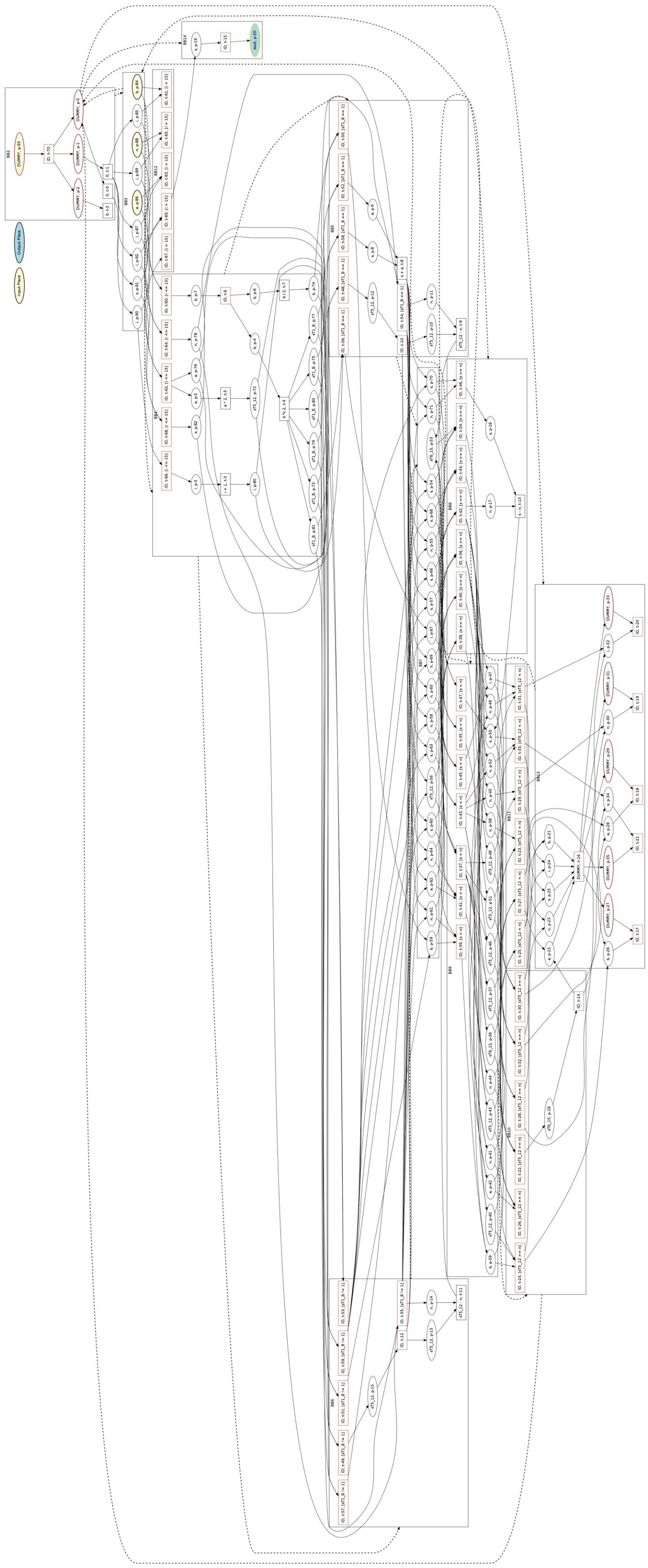

Figure A.14: PRES+ model for MODN transformed using automated model constructor (to be viewed with adequate magnification in PDF viewer)

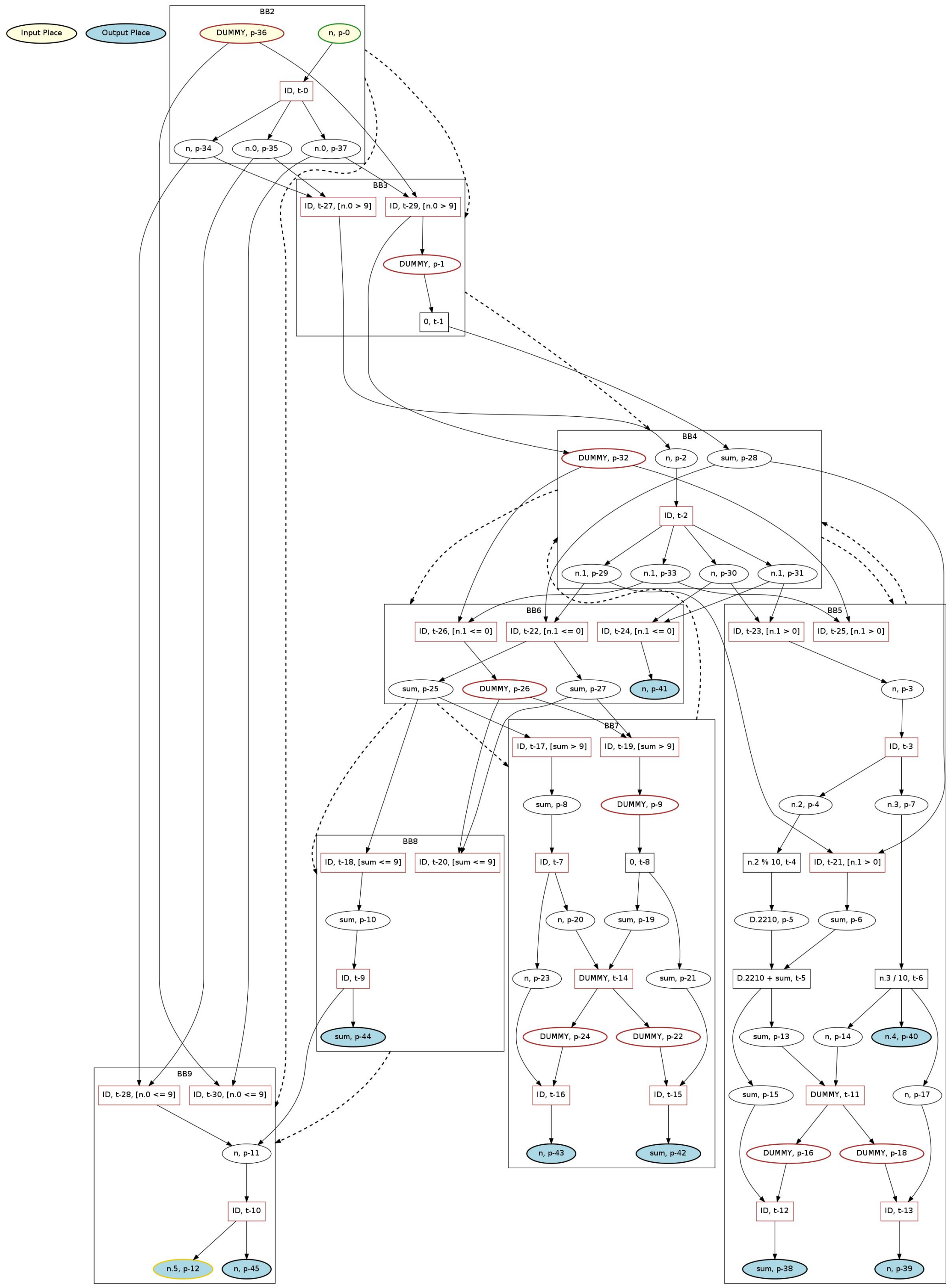

Figure A.15: PRES+ model for sum of the digits (SOD) original

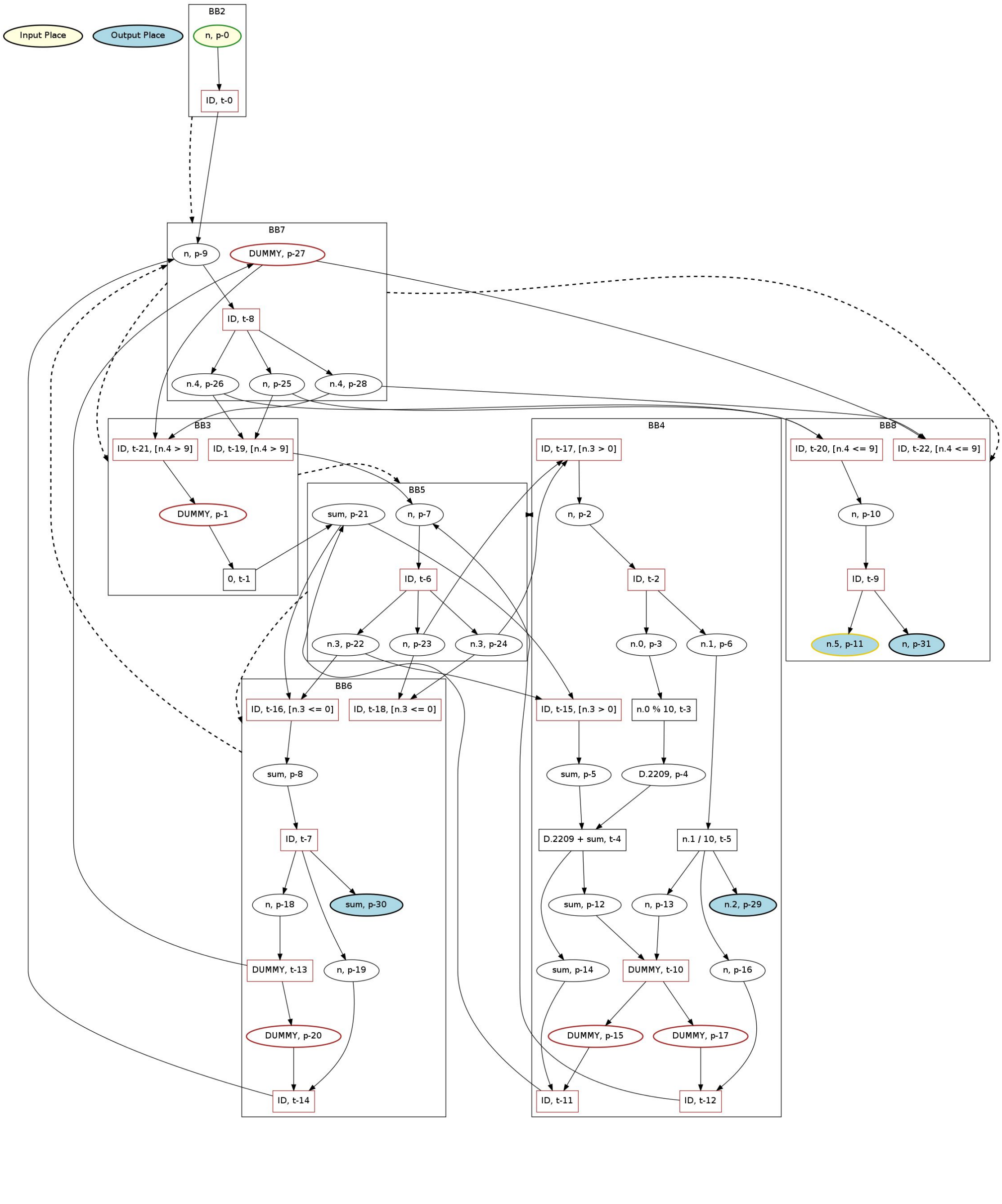

Figure A.16: PRES+ model for sum of the digits (SOD) transformed

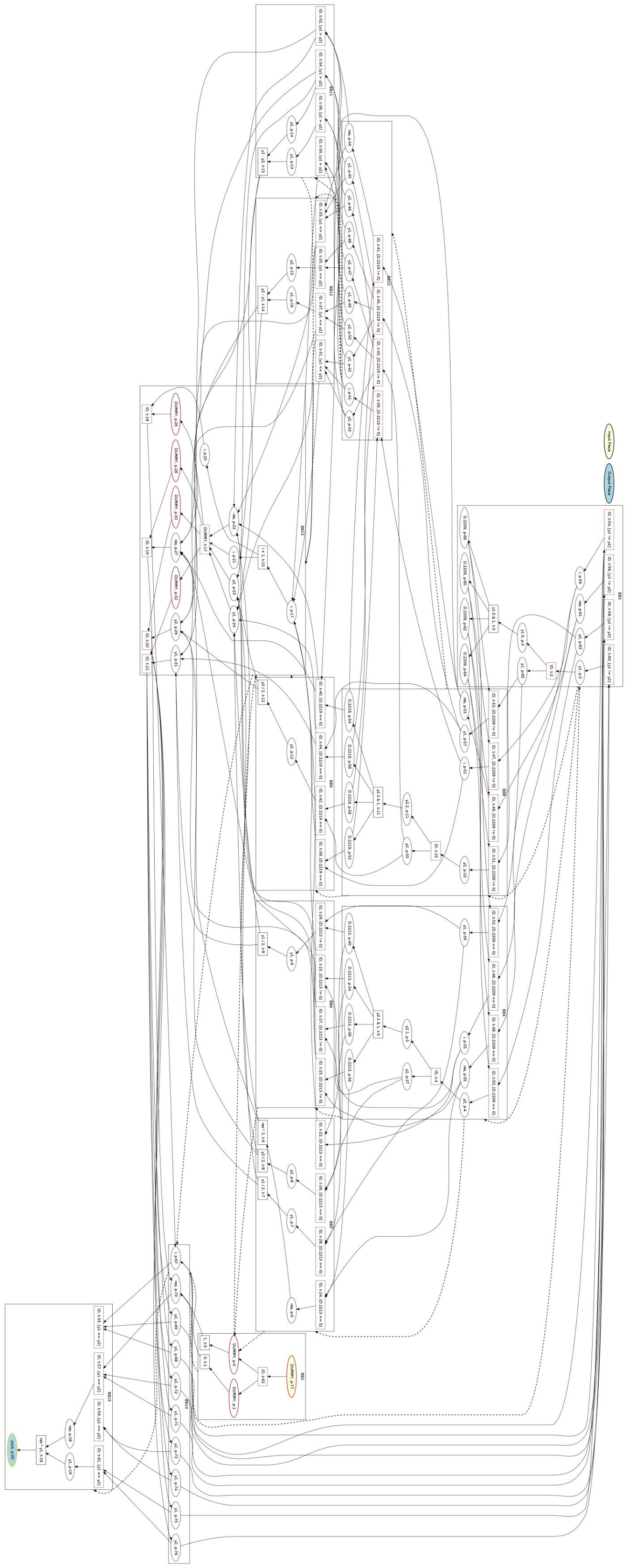

Figure A.17: PRES+ model for GCD original (to be viewed with adequate magnification in PDF viewer)

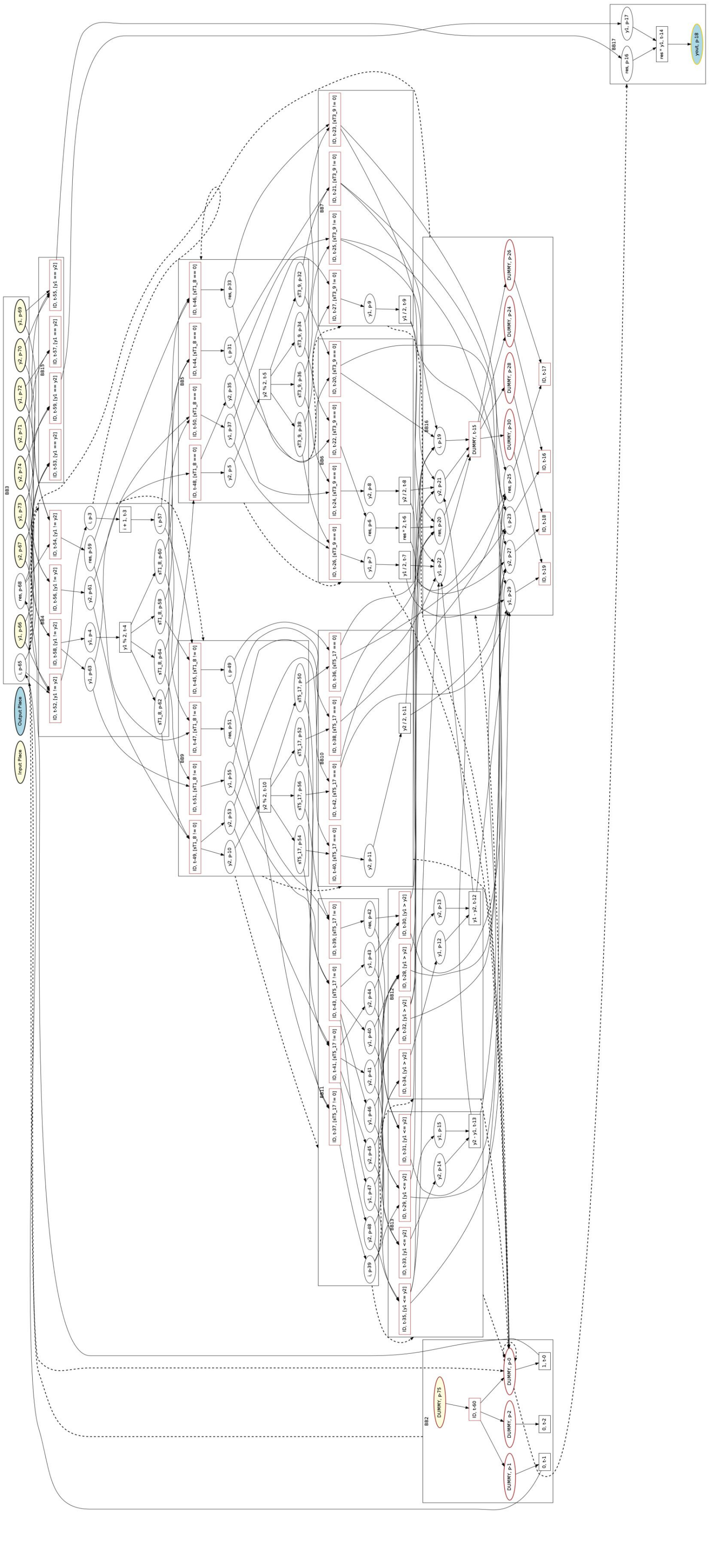

Figure A.18: PRES+ model for GCD transformed (to be viewed with adequate magnification in PDF viewer)

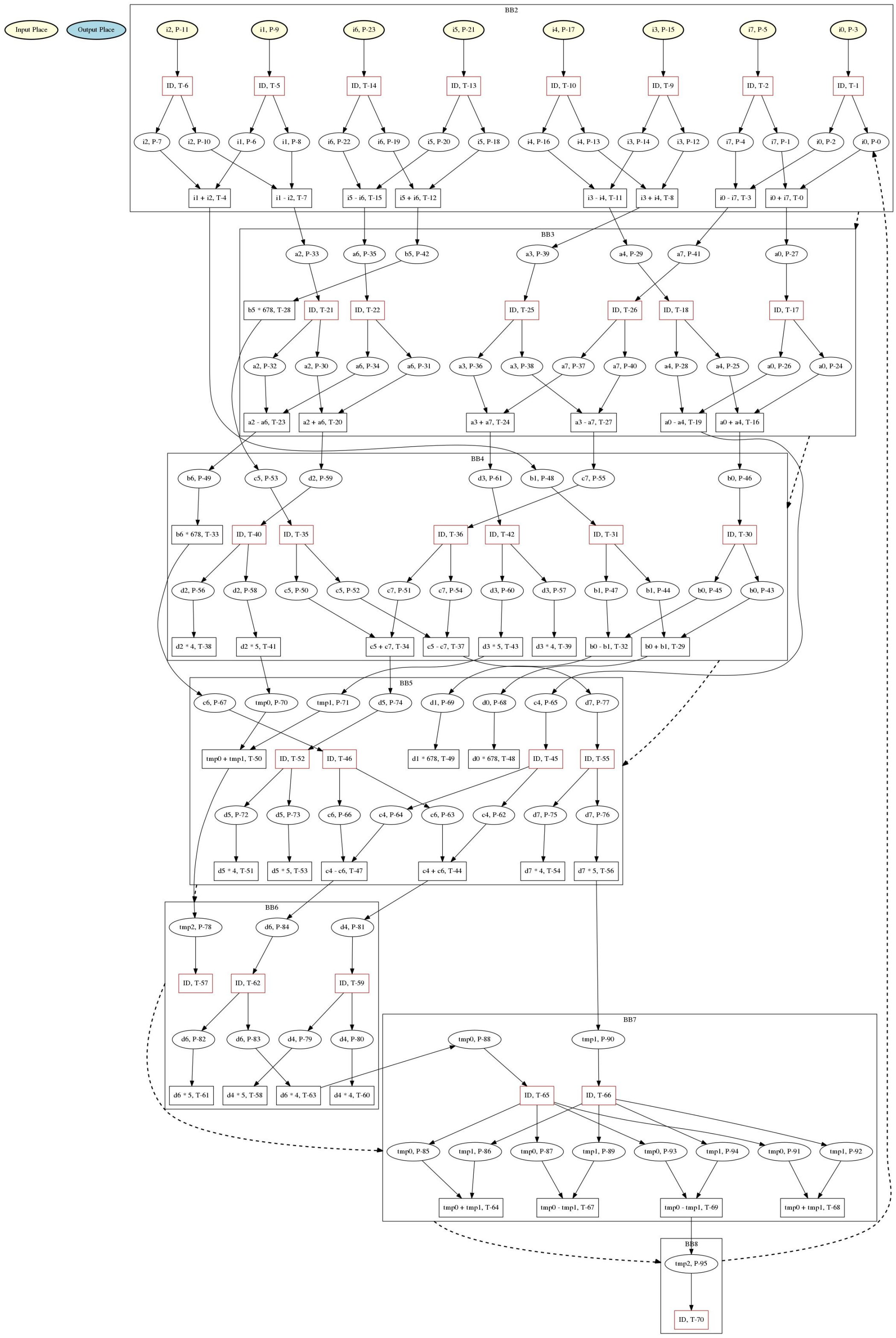

Figure A.19: PRES+ model for DCT original

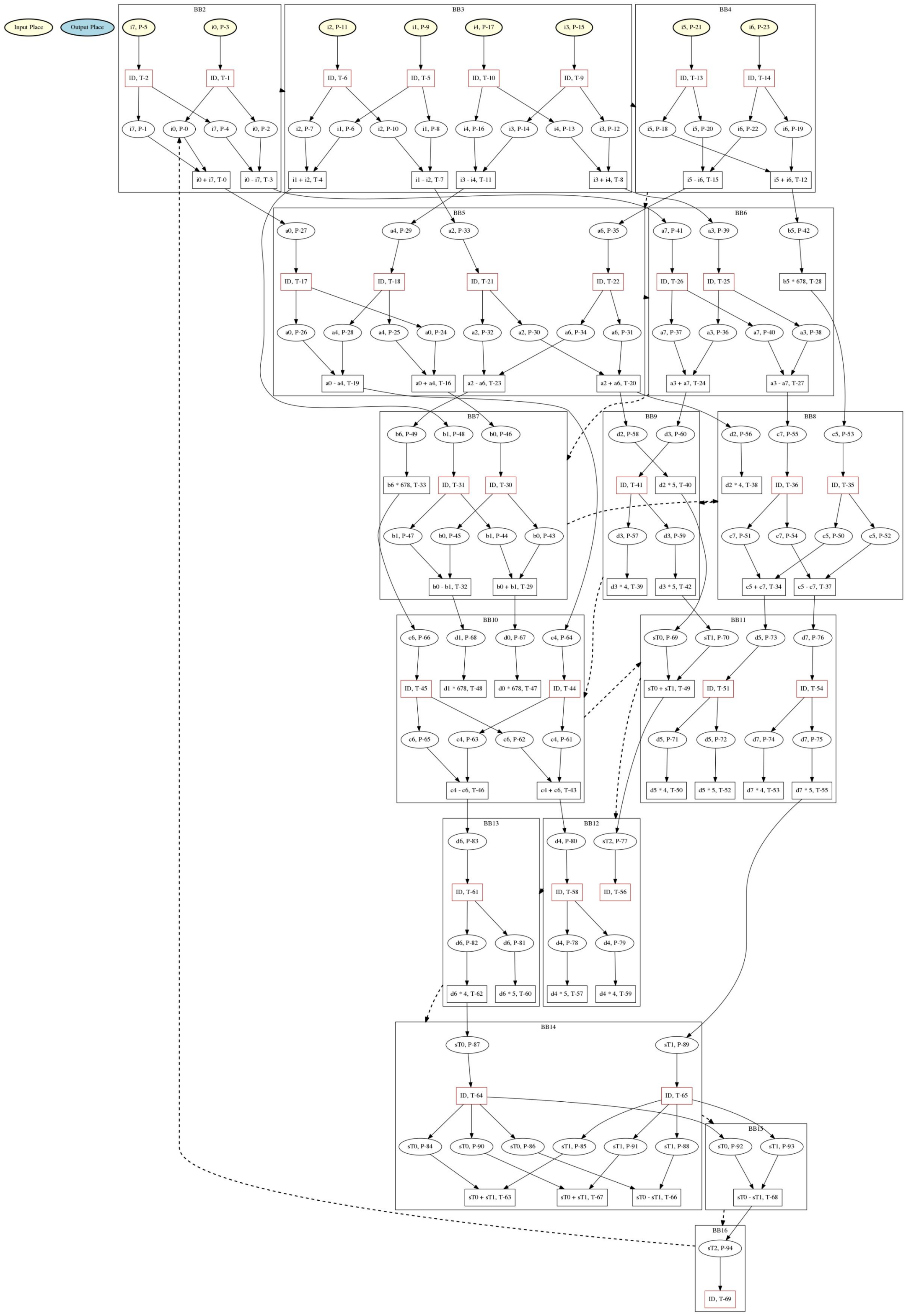

Figure A.20: PRES+ model for DCT transformed

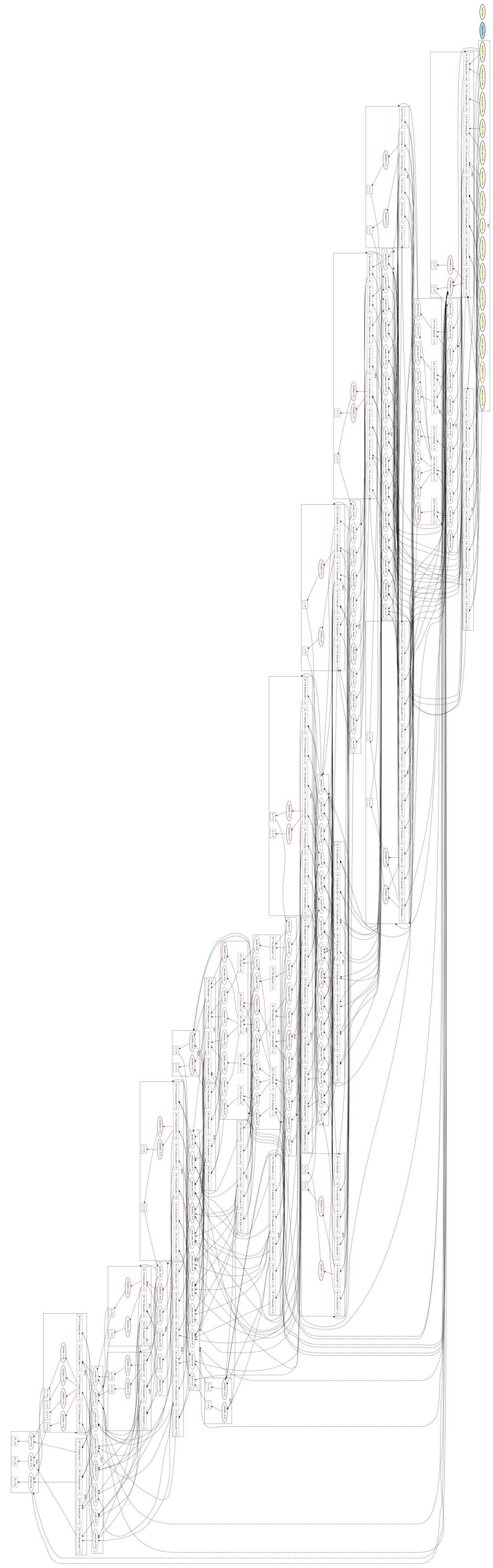

Figure A.21: PRES+ model for TLC original (to be viewed with adequate magnification in PDF viewer)

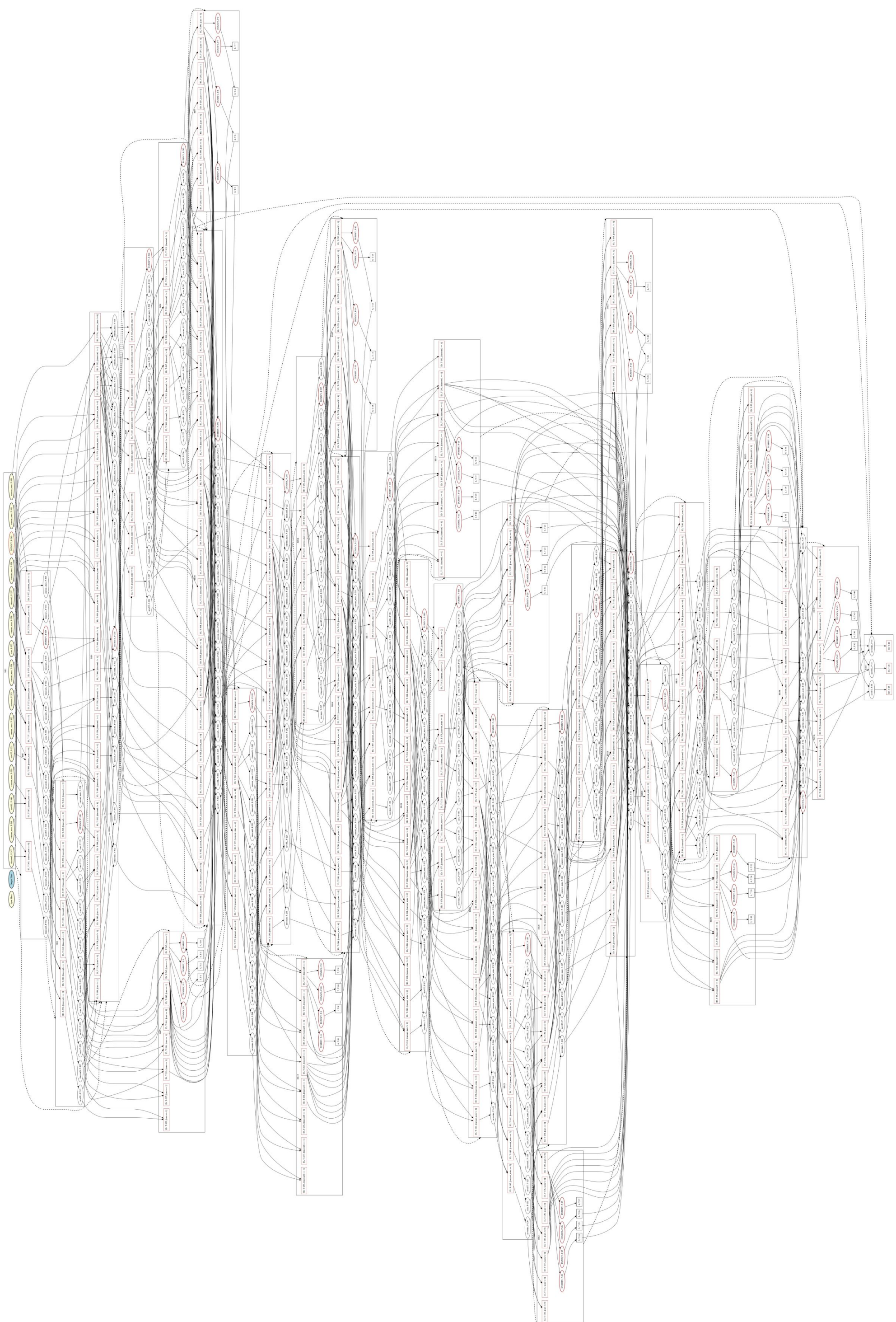

Figure A.22: PRES+ model for TLC transformed (to be viewed with adequate magnification in PDF viewer)

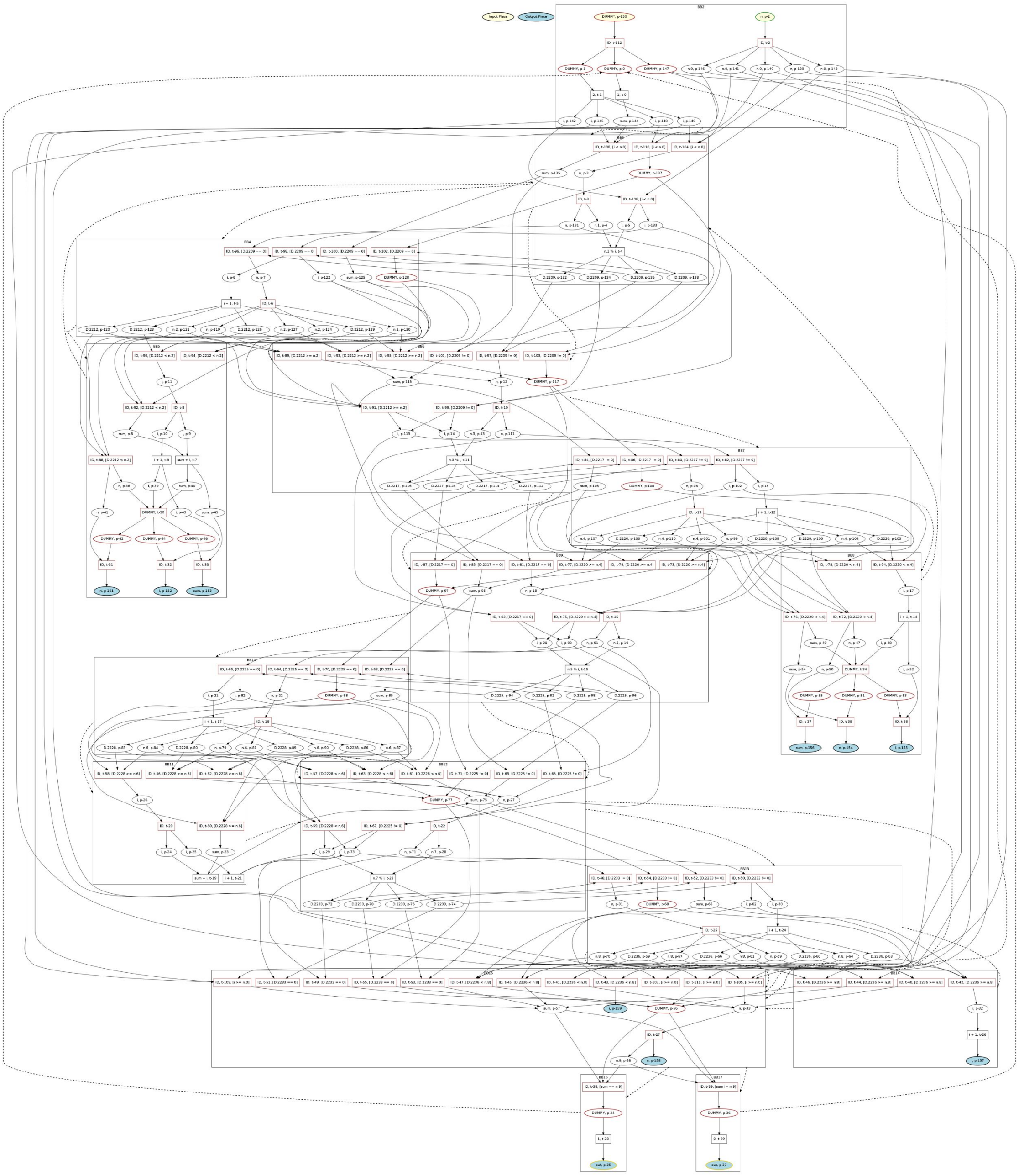

Figure A.23: PRES+ model for PERFECT original (to be viewed with adequate magnification in PDF viewer)

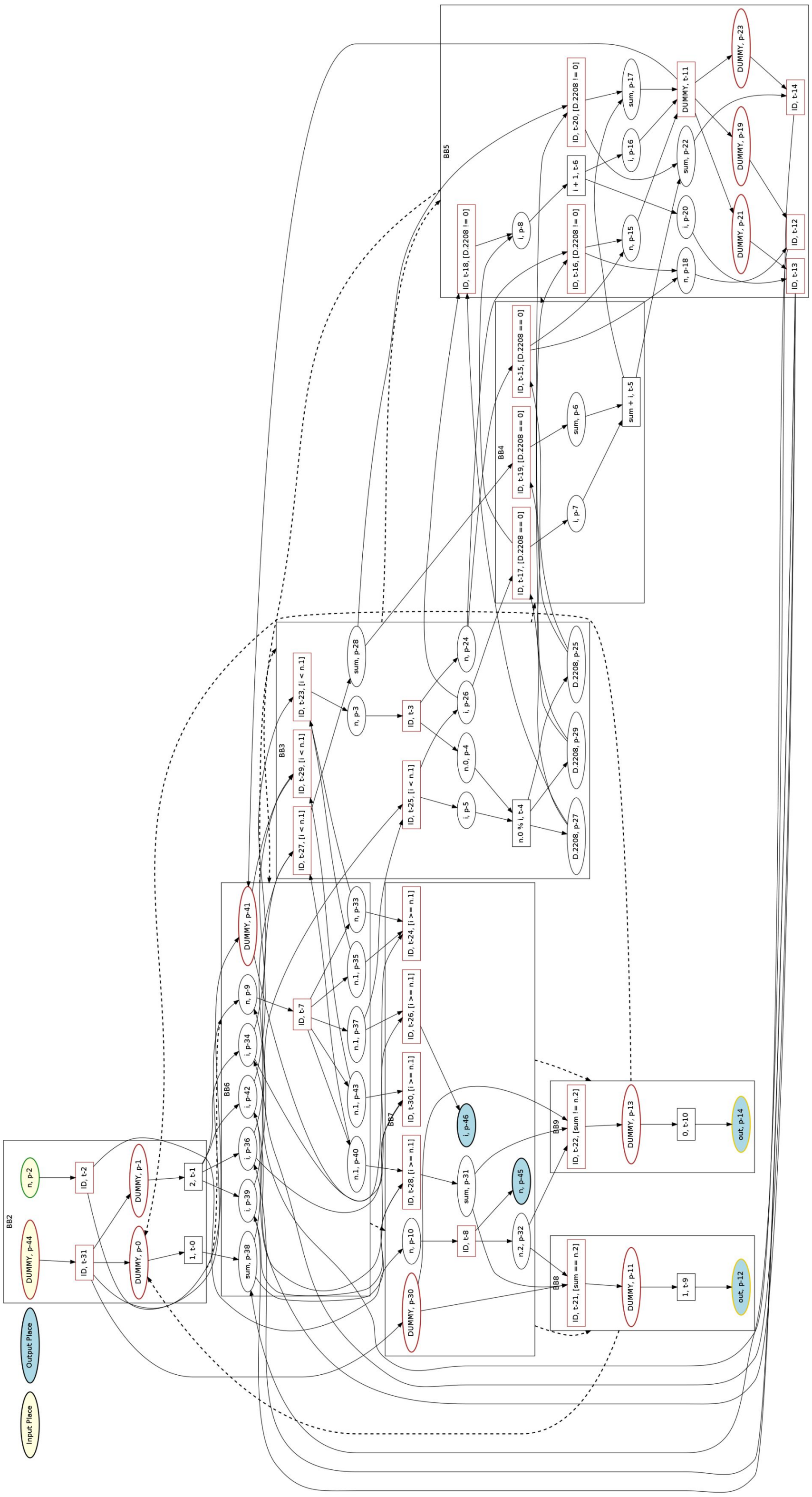

Figure A.24: PRES+ model for PERFECT transformed

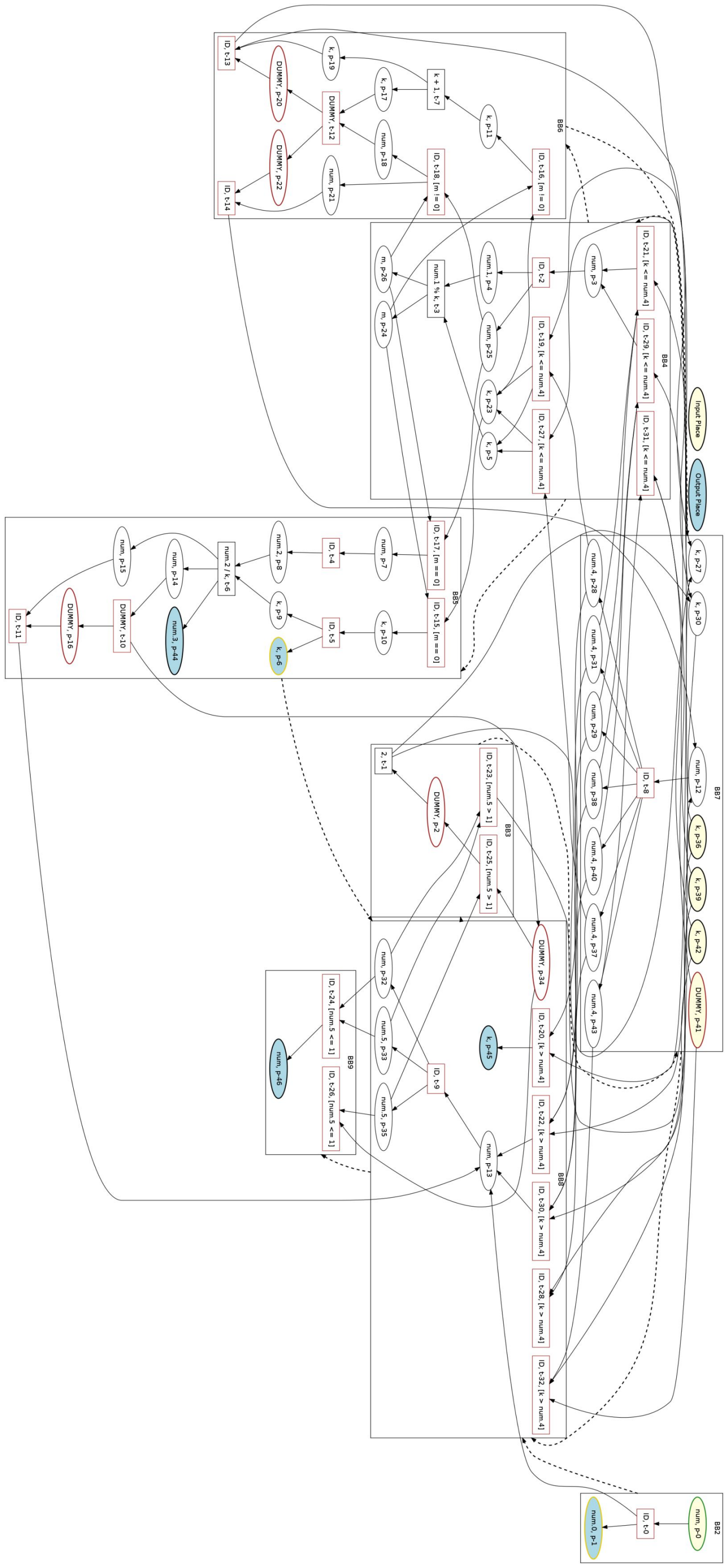

Figure A.25: PRES+ model for PRIMEFAC original

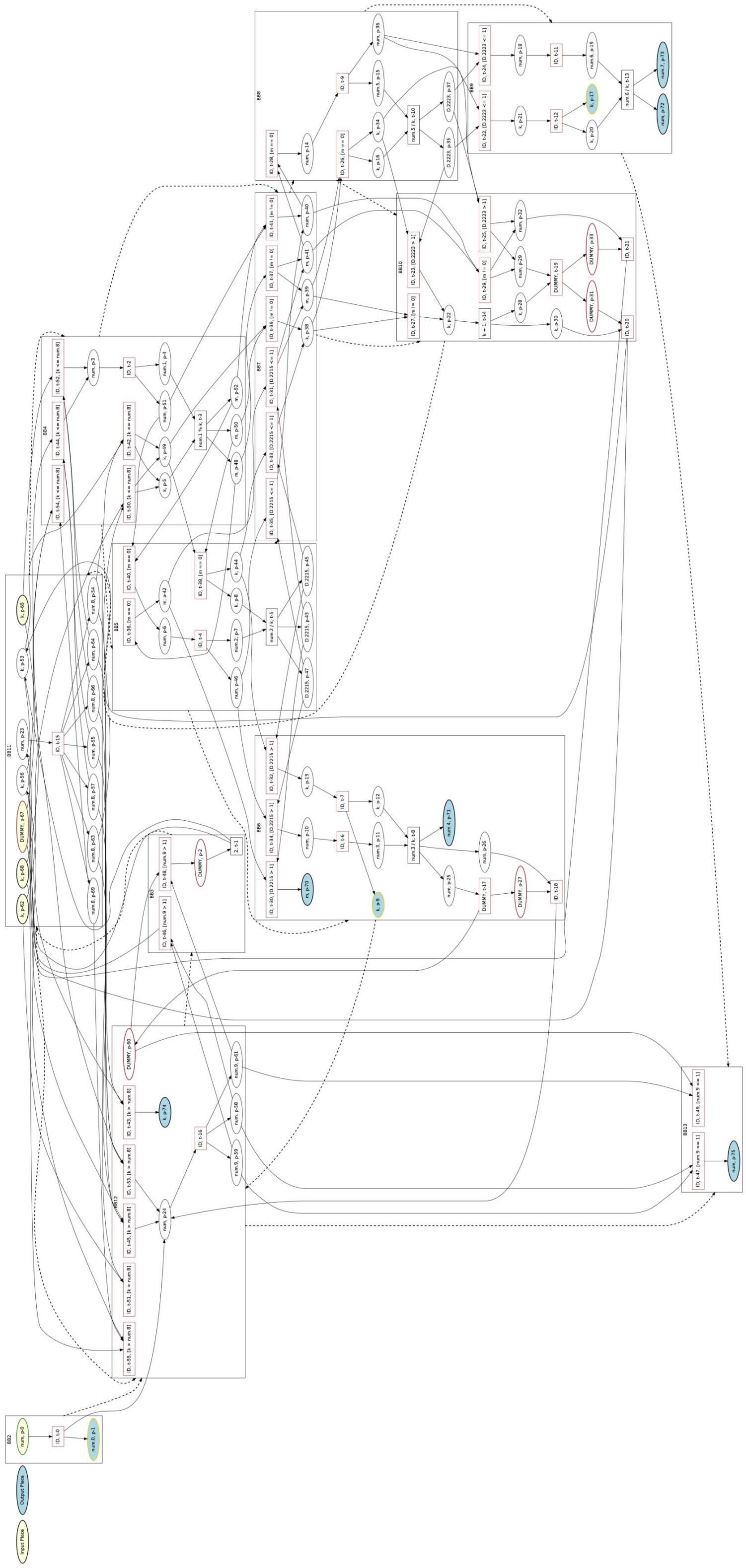

Figure A.26: PRES+ model for PRIMEFAC transformed (to be viewed with adequate magnification in PDF viewer)

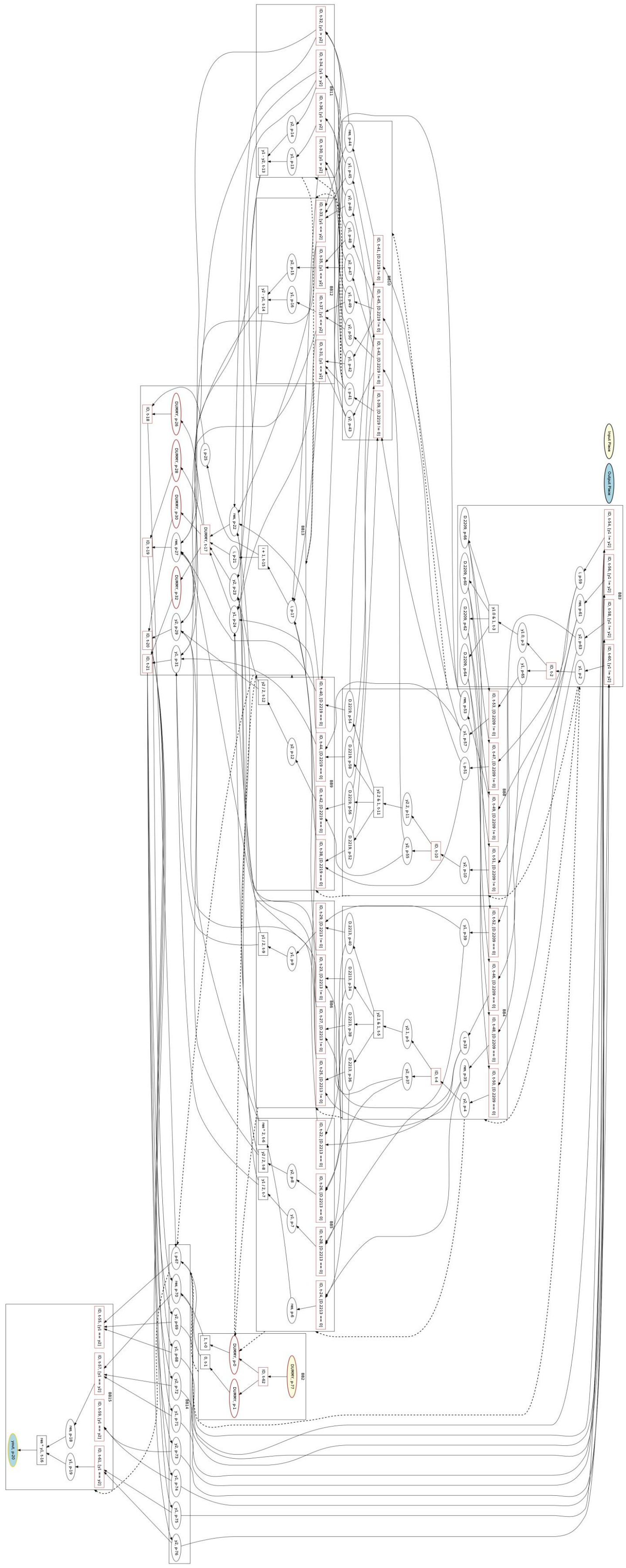

Figure A.27: PRES+ model for LCM original (to be viewed with adequate magnification in PDF viewer)

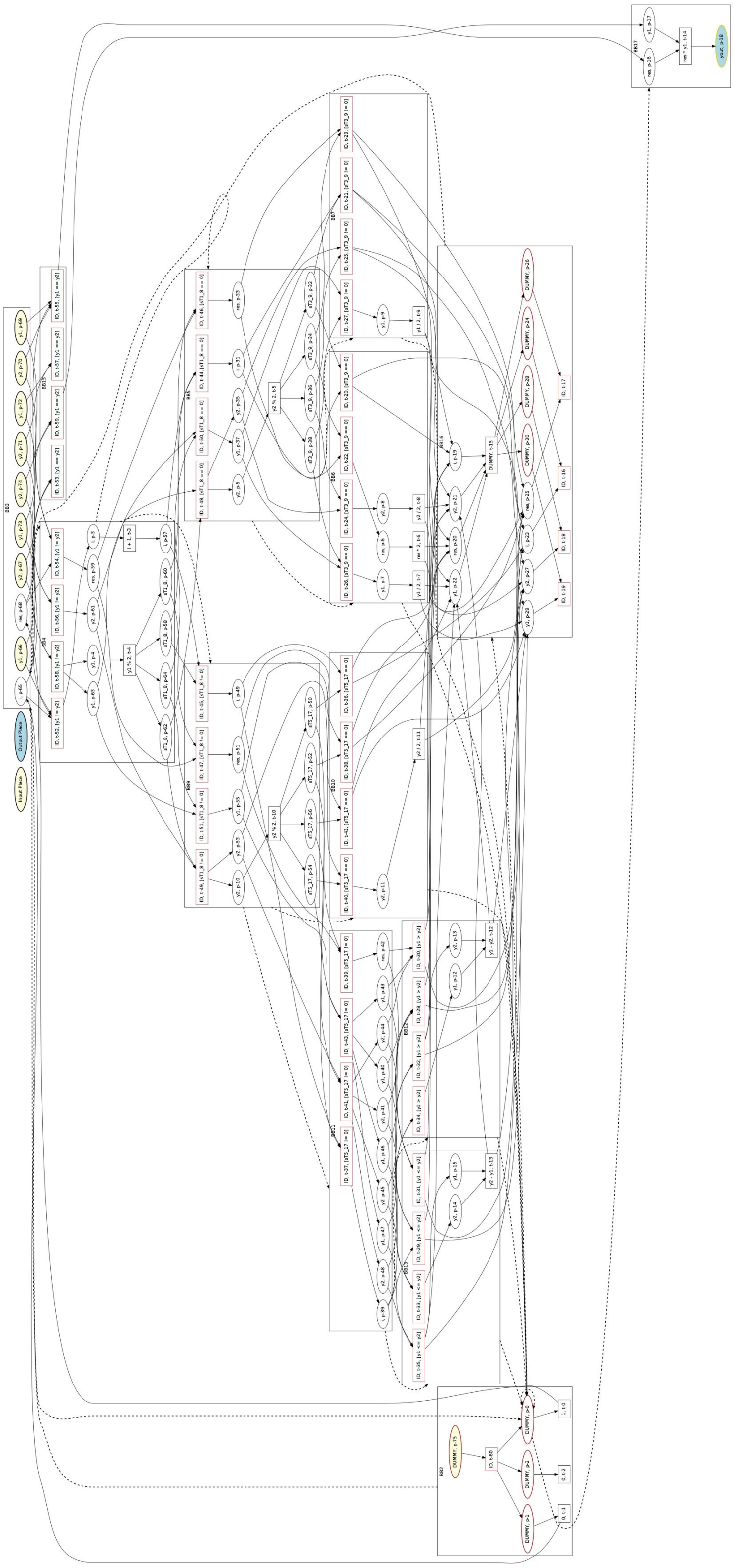

Figure A.28: PRES+ model for LCM transformed (to be viewed with adequate magnification in PDF viewer)

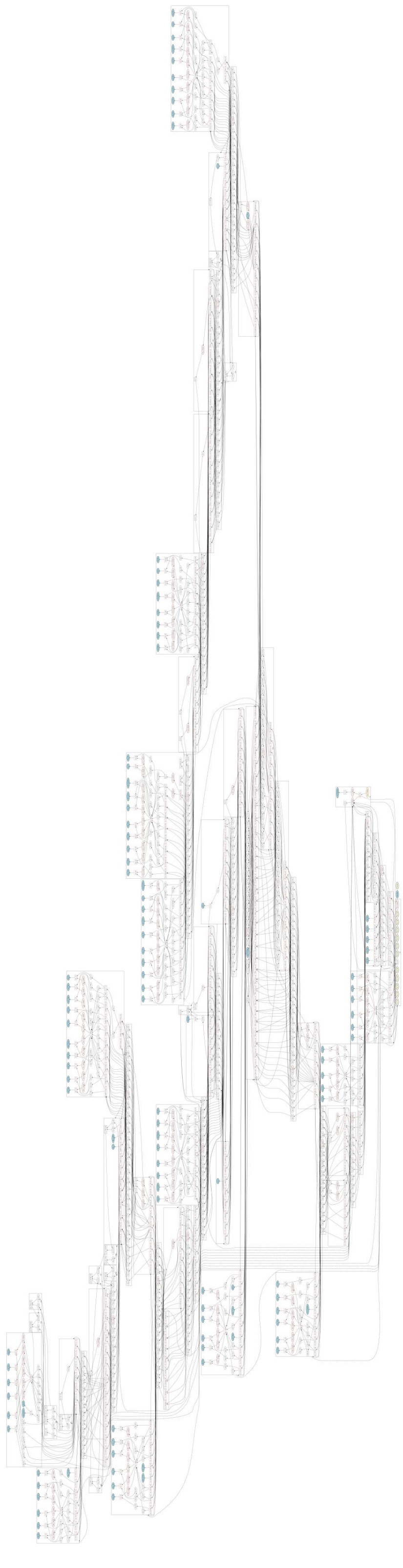

Figure A.29: PRES+ model for LRU original (to be viewed with adequate magnification in PDF viewer)

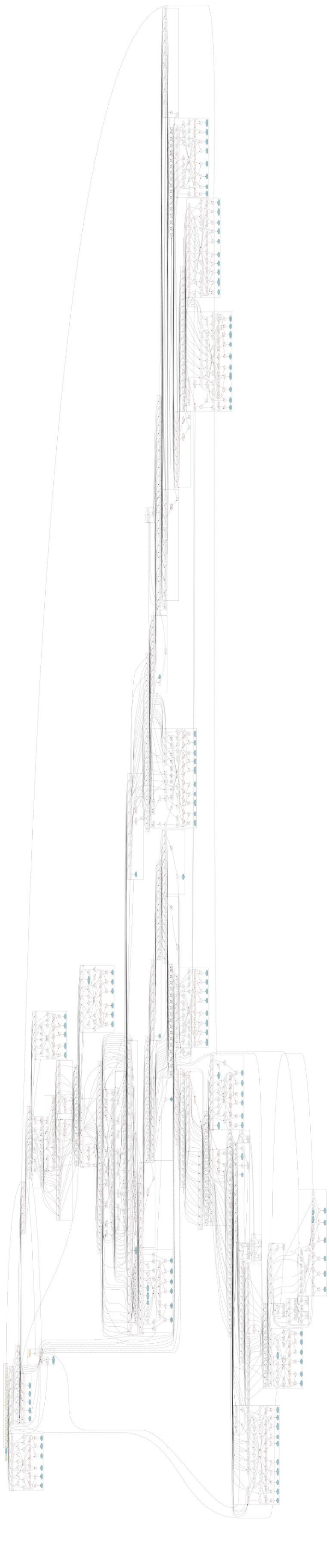

Figure A.30: PRES+ model for LRU transformed (to be viewed with adequate magnification in PDF viewer)

Bibliography

[1] LLVM Compiler. http://llvm.org/.

[2] Par4All. http://www.par4all.org/.

[3] PVS Specification and Verification System. http://pvs.csl.sri.com/.

[4] UPPAAL tool. http:/www.uppaal.org/.

[5] Z3 SMT Solver. http:/www.z3.codeplex.com/.

[6] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Trans. Softw. Eng.*, 14(5):584–594, 1988.

[7] S. G. Akl. *Parallel Computation: Models and Methods*. Prentice-Hall, Inc., 1997.

[8] S. G. Akl. Editorial note. *Parallel Processing Letters*, 25(1), 2015.

[9] F. Arendt and B. Kluhe. Modelling and verification of real-time software using interpreted petri nets. *Annual Review in Automatic Programming*, 15:35 – 40, 1990.

[10] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, December 1994.

[11] C. Baier. Polynomial time algorithms for testing probabilistic bisimulation and simulation. In *Computer Aided Verification*, pages 50–61, 1996.

[12] C. Baier, B. Engelen, and M. E. Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. *J. Comput. Syst. Sci.*, 60(1):187–231, 2000.

[13] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[14] S. Bandyopadhyay. Supporting code and examples for thesis. https://cse.iitkgp.ac.in/~chitta/pubs/rep/thesisBench.zip, https://github.com/soumyadipcsis/Equivalence-checker/blob/master/thesisBench.zip, Aug 2016.

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

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

[17] S. Bandyopadhyay, D. Sarkar, K. Banerjee, C. Mandal, and K. R. Duddu. A path construction algorithm for translation validation using pres+ models. *Parallel processing letter (to appear)*, 2015.

[18] S. Bandyopadhyay, D. Sarkar, and C. Mandal. An efficient equivalence checking method for petri net based models of programs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015 (Poster), Florence, Italy, May 16-24, 2015, Volume 2*, pages 827–828, 2015.

[19] S. Bandyopadhyay, D. Sarkar, and C. Mandal. An efficient path based equivalence checking for petri net based models of programs. In *Proceedings of the 9th India Software Engineering Conference, Goa, India, February 18-20, 2016*, pages 70–79, 2016.

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

[21] G. Barany and A. Krall. Optimal and heuristic global code motion for minimal spilling. In *Compiler Construction*, pages 21–40, 2013.

[22] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In *CAV*, pages 291–295, 2005.

[23] E. Best and R. R. Devillers. Synthesis of persistent systems. In *Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings*, pages 111–129, 2014.

[24] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI 08*, 2008.

[25] A. Bouajjani, A. Muscholl, and T. Touili. Permutation rewriting and algorithmic verification. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 399–408, 2001.

[26] F. Brandner and Q. Colombet. Elimination of parallel copies using code motion on data dependence graphs. *Computer Languages, Systems & Structures*, 39(1):25–47, 2013.

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

[28] L. Chang, X. He, and S. M. Shatz. A methodology for modeling multi-agent systems using nested petri nets. *International Journal of Software Engineering and Knowledge Engineering*, 22(7):891–926, 2012.

[29] B. Charron-Bost, S. Merz, A. Rybalchenko, and J. Widder. Formal verification of distributed algorithms (dagstuhl seminar 13141). *Dagstuhl Reports*, 3(4):1–16, 2013.

[30] X. Chen, H. Hsieh, F. Balarin, and Y. Watanabe. Formal verification for embedded system designs. *Design Automation for Embedded Systems*, 8:139–153, 2003.

[31] T.-H. Chiang and L.-R. Dung. Verification method of dataflow algorithms in high-level synthesis. *J. Syst. Softw.*, 80(8):1256–1270, 2007.

[32] A. Chutinan and B. H. Krogh. Verification of infinite-state dynamic systems using approximate quotient transition systems. *IEEE Trans. Automat. Contr.*, 46(9):1401–1410, 2001.

[33] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[34] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl. Sprint: a tool to generate concurrent transaction-level models from sequential code. *EURASIP J. Appl. Signal Process.*, 2007(1):1–15, 2007.

[35] R. Cordone, F. Ferrandi, M. D. Santambrogio, G. Palermo, and D. Sciuto. Using speculative computation and parallelizing techniques to improve scheduling of control based designs. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 898–904, Piscataway, NJ, USA, 2006. IEEE Press.

[36] A. Corradini, L. Ribeiro, F. L. Dotti, and O. M. Mendizabal. A formal model for the deferred update replication technique. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, pages 235–253, 2013.

[37] L. Cortes, P. Eles, and Z. Peng. Verification of embedded systems using a petri net based representation. In *System Synthesis, 2000. Proceedings. The 13th International Symposium on*, pages 149–155, 2000.

[38] L. A. Cortés, P. Eles, and Z. Peng. Modeling and formal verification of embedded systems based on a petri net representation. *JSA*, 49(12-15):571–598, 2003.

[39] S. A. da Costa and L. Ribeiro. Verification of graph grammars using a logical approach. *Sci. Comput. Program.*, 77(4):480–504, 2012.

[40] A. Darte and G. Huard. Loop shifting for loop compaction. *J. Parallel Programming*, 28(5):499–534, 2000.

[41] P. C. Diniz and J. M. P. Cardoso. Code transformations for embedded reconfigurable computing architectures. In *Generative and Transformational Techniques in Software Engineering III*, pages 322–344, 2009.

[42] L. C. V. Dos Santos, M. J. M. Heijligers, C. A. J. Van Eijk, J. Van Eijnhoven, and J. A. G. Jess. A code-motion pruning technique for global scheduling. *ACM Trans. Des. Autom. Electron. Syst.*, 5(1):1–38, 2000.

[43] L. C. V. Dos. Santos and J. Jress. A reordering technique for efficient code motion. In *Procs. of the 36th ACM/IEEE Design Automation Conference*, DAC '99, pages 296–299, New York, NY, USA, 1999. ACM.

[44] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.*, 311(1-3):221–256, 2004.

[45] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentellni. Design of embedded systems: Formal models, validation and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[46] P. Eles, Z. Peng, and D. Karlsson. Formal verification in a component-based reuse methodology. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS 2002), October 2-4, 2002, Kyoto, Japan*, pages 156–161, 2002.

[47] J. Fernandez and L. Mounier. "on the fly" verification of behavioural equivalences and preorders. In *Computer Aided Verification*, pages 181–191, 1991.

[48] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478 –490, july 1981.

[49] K. Fisler and M. Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Formal Methods in Computer-Aided Design*, pages 115–132, 1998.

[50] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings the 19th Symposium on Applied Mathematics*, pages 19–32, Providence, R.I., 1967. American Mathematical Society. Mathematical Aspects of Computer Science.

[51] B. Freisleben and T. Kielmann. Automated transformation of sequential divide-and-conquer algorithms into parallel programs. *Computers and Artificial Intelligence*, 14:579–596, 1995.

[52] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):166–178, 1992.

[53] R. Gupta and M. Soffa. Region scheduling: an approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421 –431, apr 1990.

[54] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs. In *Proceedings of DATE'03*, pages 270–275, Washington, DC, USA, 2003. IEEE Computer Society.

[55] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits. *IEE Proceedings: Computer and Digital Technique*, 150(5):330–337, September 2003.

[56] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *Proc. of Int. Conf. on VLSI Design*, pages 461–466, Washington, DC, USA, Jan 2003. IEEE Computer Society.

[57] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Transactions on CAD of ICS*, 23(2):302–312, Feb 2004.

[58] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):1–31, October 2004.

[59] S. Gupta, M. Reshadi, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *Proceedings of the 15th international symposium on System Synthesis*, ISSS '02, pages 261–266, New York, NY, USA, 2002. ACM.

[60] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *International Symposium on System Synthesis*, pages 171–176, 2001.

[61] S. Gupta, N. Savoiu, S. Kim, N. Dutt, R. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Proceedings of DAC'01*, pages 269–272, 2001.

[62] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[63] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[64] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[65] J. He and T. Hoare. CSP is a retract of CCS. *Theor. Comput. Sci.*, 411(11-13):1311–1337, 2010.

[66] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[67] Y. Hu, C. Barrett, B. Goldberg, and A. Pnueli. Validating more loop optimizations. *Electronic Notes in Theoretical Computer Science*, 141(2):69–84, 2005. Proceedings of the Fourth International Workshop on Compiler Optimization meets Compiler Verification (COCV 2005).

[68] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993. 10.1007/BF01205185.

[69] R. Jain, A. Majumdar, A. Sharma, and H. Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 686–689, New York, NY, USA, 1991. ACM.

[70] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, May 2007.

[71] N. E. Johnson. *Code Size Optimization for Embedded Processors*. PhD thesis, University of Cambridge, 2004.

[72] M. Kandemir, S. W. Son, and G. Chen. An evaluation of code and data optimizations in the context of disk power reduction. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 209–214, 2005.

[73] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. *IEEE Trans. Very Large Scale Integr. Syst.*, 9:801–804, December 2001.

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

[75] C. Karfa, C. Mandal, D. Sarkar, S. Pentakota, and C. Reade. A formal verification method of scheduling in high-level synthesis. In *7th International Symposium on Quality Electronic Design, 2006.*, pages 71–78, March 2006.

[76] T. Kim and X. Liu. A functional unit and register binding algorithm for interconnect reduction. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29:641–646, April 2010.

[77] Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machines with datapath (fsmd). In *Proceedings of the 5th International Symposium on Quality Electronic Design*, ISQED '04, pages 110–115, Washington, DC, USA, 2004. IEEE Computer Society.

[78] Y. Kim and N. Mansouri. Automated formal verification of scheduling with speculative code motions. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, GLSVLSI '08, pages 95–100, New York, NY, USA, 2008. ACM.

[79] J. C. King. Program correctness: On inductive assertion methods. *IEEE Trans. Software Eng.*, 6(5):465–479, 1980.

[80] K. Klai, S. Haddad, and J. Ilié. Modular verification of petri nets properties: A structure-based approach. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, pages 189–203, 2005.

[81] J. Knoop, O. Ruthing, and B. Steffen. Lazy code motion. In *PLDI*, pages 224–234, 1992.

[82] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. PPoPP '99, pages 13–24, 1999.

[83] C. M. Kunal Banerjee and D. Sarkar. Deriving bisimulation relations from path extension based equivalence checkers. In *WEPL*, pages 1–2, 2015.

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

[85] S. Kundu, S. Lerner, and R. Gupta. Translation validation of high-level synthesis. *IEEE Transactions on CAD of ICS*, 29(4):566–579, 2010.

[86] G. Lakshminarayana, K. Khouri, and N. Jha. Wavesched: A novel scheduling technique for control-flow intensive behavioural descriptions. In *Proc. of ICCAD*, pages 244–250, Nov 1997.

[87] G. Lakshminarayana, A. Raghunathan, and N. Jha. Incorporating speculative execution into scheduling of control-flow-intensive design. *IEEE Transactions on CAD of ICS*, 19(3):308–324, March 2000.

[88] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 46–57, New York, NY, USA, 1992. ACM.

[89] C.-H. Lee, C.-H. Shih, J.-D. Huang, and J.-Y. Jou. Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In *Asia and South Pacific Design Automation Conference*, pages 497–502, 2011.

[90] C.-H. Lee, C.-H. Shih, J.-D. Huang, and J.-Y. Jou. Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In *(ASP-DAC), 2011 16th Asia and South Pacific*, pages 497–502, 2011.

[91] J.-H. Lee, Y.-C. Hsu, and Y.-L. Lin. A new integer linear programming formulation for the scheduling problem in data path synthesis. In *Procs. of the International Conference on Computer-Aided Design*, pages 20 –23, Washington, DC, USA, nov 1989. IEEE Computer Society.

[92] Q. Li, L. Shi, J. Li, C. J. Xue, and Y. He. Code motion for migration minimization in STT-RAM based hybrid cache. In *IEEE Computer Society Annual Symposium on VLSI*, pages 410–415, 2012.

[93] T. Li, Y. Guo, W. Liu, and M. Tang. Translation validation of scheduling in high level synthesis. In *ACM Great Lakes Symposium on VLSI*, pages 101–106, 2013.

[94] D. Lime, O. H. Roux, C. Seidner, and L. Traonouez. Romeo: A parametric model-checker for petri nets with stopwatches. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 54–57, 2009.

[95] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Kogakusha, Tokyo, 1974.

[96] N. Mansouri and R. Vemuri. A methodology for automated verification of synthesized rtl designs and its integration with a high-level synthesis tool. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, FMCAD '98, pages 204–221, London, UK, 1998. Springer-Verlag.

[97] J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[98] V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):776–813, 2003.

[99] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[100] S.-M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and vliw processors. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 55–71, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[101] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[102] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[103] M. Naija, S. B. Ahmed, and J. Bruel. New schedulability analysis for real-time systems based on MDE and petri nets model at early design stages. In *ICSOFT-EA 2015 - Proceedings of the 10th International Conference on Software Engineering and Applications, Colmar, Alsace, France, 20-22 July, 2015.*, pages 330–338, 2015.

[104] K. S. Namjoshi, G. Tagliabue, and L. D. Zuck. A witnessing compiler: A proof of concept. In *Runtime Verification*, pages 340–345, 2013.

[105] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 77–88, New York, NY, USA, 2002. ACM.

[106] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.

[107] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *ICPP 1993. International Conference on Parallel Processing, 1993*, volume 2, pages 120 –124, aug. 1993.

[108] C. A. Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[109] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, pages 151–166, 1998.

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

[111] R. Radhakrishnan, E. Teica, and R. Vemuri. Verification of basic block schedules using rtl transformations. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '01, pages 173–178, London, UK, 2001. Springer-Verlag.

[112] M. Rahmouni and A. A. Jerraya. Formulation and evaluation of scheduling techniques for control flow graphs. In *Proceedings of EuroDAC'95*, pages 386–391, Brighton, 18-22 September 1995.

[113] M. Rakotoarisoa and E. Pastor. BMC encoding for concurrent systems. In *XXVII International Conference of the Chilean Computer Science Society (SCCC 2008), 10-14 November 2008, Punta Arenas, Chile*, pages 127–134, 2008.

[114] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. on Computer*, C-21(2):137–146, Feb. 1972.

[115] T. Raudvere, I. Sander, and A. Jantsch. Application and verification of local nonsemantic-preserving transformations in system design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1091 – 1103, june 2008.

[116] L. Ribeiro, O. M. dos Santos, F. L. Dotti, and L. Foss. Correct transformation: From object-based graph grammars to PROMELA. *Sci. Comput. Program.*, 77(3):214–246, 2012.

[117] M. Rim, Y. Fann, and R. Jain. Global scheduling with code motions for high-level synthesis applications. *IEEE Transactions on VLSI Systems*, 3(3):379–392, Sept. 1995.

[118] M. Rinard and P. Diniz. Credible compilation. Technical Report MIT-LCS-TR-776, MIT, 1999.

[119] C. Rodríguez and S. Schwoon. Verification of petri nets with read arcs. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, pages 471–485, 2012.

[120] O. Ruthing, J. Knoop, and B. Steffen. Sparse code motion. In *IEEE POPL*, pages 170–183, 2000.

[121] D. Sarkar and S. C. De Sarkar. A theorem prover for verifying iterative programs over integers. *IEEE Trans. Software Eng.*, 15(12):1550–1566, 1989.

[122] K. Singh. Construction of Petri net based models for C programs, M.Tech. Dissertation, Dept. of Computer Sc. & Engg., I.I.T., Kharagpur, INDIA. https://cse.iitkgp.ac.in/~chitta/pubs/rep/thesisKulwant.pdf, https://github.

com/soumyadipcsis/Equivalence-checker/blob/master/thesisKulwant.pdf, May 2016.

[123] SPIN.

[124] K. Strehl and L. Thiele. Interval diagrams for efficient symbolic verification of process networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):939 –956, aug 2000.

[125] A. Stump, C. W. Barrett, and D. L. Dill. Cvc: A cooperating validity checker. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 500–504. Springer-Verlag, 2002.

[126] I. Suzuki and T. Murata. A method for stepwise refinement and abstraction of petri nets. *J. Comput. Syst. Sci.*, 27(1):51–76, 1983.

[127] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

[128] J.-B. Tristan and X. Leroy. Verified validation of lazy code motion. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 316–326, New York, NY, USA, 2009. ACM.

[129] A. Turjan. *Compiling Nested Loop Programs to Process Networks*. PhD thesis, Leiden University, 2007.

[130] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, 2004.

[131] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220, 2015.

[132] M. Westergaard. Verifying parallel algorithms and programs using coloured petri nets. *T. Petri Nets and Other Models of Concurrency*, 6:146–168, 2012.

[133] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. Sigref- A symbolic bisimulation tool box. In *Automated Technology for Verification and Analysis*, pages 477–492, 2006.

[134] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[135] Y.-P. You and S.-H. Wang. Energy-aware code motion for gpu shader processors. *ACM Trans. Embed. Comput. Syst.*, 13(3):49:1–49:24, 2013.

[136] S. ZamanZadeh, M. Najibi, and H. Pedram. Pre-synthesis optimization for asynchronous circuits using compiler techniques. In *Advances in Computer Science and Engineering*, volume 6 of *Communications in Computer and Information Science*, pages 951–954. Springer Berlin Heidelberg, 2009.

[137] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.

[138] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Programming Language Design and Implementation*, pages 175–186, 2013.

List of Publications out of this work

Journal/conference papers:

1. **Soumyadip Bandyopadhyay**, Dipankar Sarkar, Chittaranjan Mandal; “An Efficient Equivalence Checking Method for Petri net based Models of Programs;” *International Conference on Software Engineering (ICSE-2015)*, pages: 827 – 828.
2. **Soumyadip Bandyopadhyay**, Dipankar Sarkar, Kunal Banerjee, Chittaranjan Mandal, Krishnam Raju; “A Path Construction Algorithm for Translation Validation using PRES+ Models;” *Parallel Processing Letters* Vol. 26, No. 02, pages: 1 – 25.
3. **Soumyadip Bandyopadhyay**, Dipankar Sarkar, Chittaranjan Mandal; “Validating SPARK: High Level Synthesis compiler;” *IEEE Computer Society Annual Symposium on VLSI (ISVLSI-2015)*, pages: 195 – 198.
4. **Soumyadip Bandyopadhyay**, Dipankar Sarkar, Kunal Banerjee, Chittaranjan Mandal; “A Path-Based Equivalence Checking Method for Petri net based Models of Programs;” *International Conference on Software Engineering and Applications (ICSOFT-EA-2015)*, pages: 319 – 329.
5. **Soumyadip Bandyopadhyay**, Dipankar Sarkar, Chittaranjan Mandal; “ An efficient path based equivalence checking for Petri net based models of programs”, *India Software Engineering Conference, (ISEC 2016)*, pages:70 – 79.

Publications in research fora:

It is to be noted that the following dissemination arising out of this work were not published as part of the proceedings of conference or workshop; these venues rather aimed to provide a appropriate platform for young researchers to discuss their works with experts in their respective research fields; all of these work, however, went through standard peer review process before being accepted.

1. **Soumyadip Bandyopadhyay**; “Behavioural verification using Petri net based models of programs;” *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL): Student Research Competition*, Mumbai, India, 2015.

2. **Soumyadip Bandyopadhyay**, Dipankar Sarkar, Chittaranjan Mandal; "Translation Validation using Path-Based Equivalence Checking of Petri net based Models of Programs;" *IMPECS-POPL Workshop on Emerging Research and Development Trends in Programming Languages (WEPL)*, Mumbai, India, 2015.

3. **Soumyadip Bandyopadhyay**; "Translation Validation using Path-Based Equivalence Checking of Petri net based Models of Programs;" *Inter-Research-Institute Student Seminar in Computer Science (IRISS)*, Goa, India, 2015.

Bio-data

Soumyadip Bandyopadhyay was born in Taki, North 24 PGS, West Bengal on 25^{th} of June, 1986. He received the B.Tech. degree in Computer Science and Engineering from West Bengal University Technology in 2004 He has worked as a Junior Project Assistance (JPA) in the VLSI Consortium project undertaken by the Advanced VLSI Design Laboratory, IIT Kharagpur from July 2008 to September 2012 and his current research interests include formal verification and software verification . He has published ten research papers in different reputed IEEE/ACM/World scientific international journals and conferences. He has received **Tata Consultancy Service Ph.D Fellowship** in 2012.