Analysis of Parameterized-Chain Networks: The Dependency Graph and its Full, Consistent Subgraphs

by

Mojtaba Moodi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis studies algorithmic aspects of deadlock analysis for parameterized networks of discrete-event systems. A parameterized network consists of a finite, but arbitrarily large, number of interacting finite-state subsystems, each within one of a fixed, finite number of isomorphism classes. While deadlock analysis of such systems is generally undecidable, decidable subproblems have recently been identified. The decision procedure rests on the construction of a finite *dependency graph* for the network, and the computation of its *full, consistent subgraphs*. We present a software tool for such computations, and apply it to a train network example that extends beyond the current theoretical framework. The results suggest ways in which the framework could usefully be extended.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Prof. John Thistle for his invaluable support throughout this project. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work. I am also thankful to him for encouraging the use of correct grammar and consistent notation in my writings and for carefully reading and commenting on revisions of this manuscript. I would like to acknowledge my friend Hadi Zibaeenejad for useful discussions.

Last but not least, I express my heartfelt gratitude to my parents, my siblings and spouse for their love has always been with me and has given me the energy to reach my goals. I am proud of you and you have a big share in what I have achieved.

Dedication

To my parents for all their love and support and putting me through the best education possible. I appreciate their sacrifices and I wouldn't have been able to get to this stage without them.

To my wife and son, for their endless love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Parameterized Networks

A fundamental impediment to analysis and design of complex control systems is combinatorial explosion: the state size of a collection of N interacting subsystems grows exponentially with N. One potential means of coping with state explosion is to devise methods that are independent of N. Vector discrete-event systems [25] and Petri nets support such methods to some extent for certain classes of systems.

In this thesis, we employ the model of *parameterized networks*.

A *parameterized system* is generally a family of finite-state models, indexed by some parameter or parameters. In a relatively simple case, a parameterized *network* might be a model of N isomorphic, interacting, finite-state subsystems, where the value of the parameter N is finite but arbitrarily large; this indeed yields an infinite set of finite-state network models, indexed by the values of the integer parameter N.

Parameterized systems have received considerable attention in the model-checking literature [1, 24, 6]. Indeed, if the underlying logic of a hardware or software system is in essence independent of specific parameter values, one might expect to be able to establish correctness without focusing on the individual finite-state models that arise when parameters are instantiated. Similarly, in control, it is to be expected that many instances of control problems give rise to control logic that is in essence independent of the values of specific parameters. Ideally, one might be able to adapt prototypical synthesis methods for finite-automaton models in such a way as to extract from the controller design the essential parameter-independent underlying logic.

The problems of analyzing blocking and deadlock-freedom in parameterized networks have been studied by [22] and by [31]. In general, such problems are undecidable [22], but nontrivial decidable subproblems exist: Zibaeenejad and Thistle identified a class of parameterized networks of ring topology for which deadlock-freedom is decidable, and more recently extended the result to a more general topology - that of "parameterized-chain networks" [30].

Specifically, the more recent results concern networks comprised of a fixed set of finite-state *distinguished* subprocesses, linked by parameterized networks of linear topology, which we call *parameterized chains*. Such networks can model, for example, transportation networks and manufacturing systems. The notion of *weak invariant simulation* is used to impose a direction of control flow on the network. A consequence of this directionality is that portions of the network can be deadlocked only if there exists a "generalized circular wait" - a strongly connected subgraph of a network instance in which each subprocess is waiting for another to execute an action. Moreover, the structural assumptions of Zibaeenejad and Thistle ensure that the existence of generalized circular waits is decidable.

The present thesis discusses algorithmic aspects of the approach. In summary, a finite *dependency graph* is constructed, which shows how the possibility of execution of an event within a given subprocess of an instance of the network may depend on the occurrence of events within another subprocess. A suitably generalized notion of a "circular wait" is captured by a subgraph of the dependency graph that has the properties of *consistency* and *fullness*. Indeed, such a subgraph represents in general an infinite set of generalized circular waits that can arise in network instances of various sizes; in effect, it represents a regular language, each element of which encodes a generalized circular wait in a particular size instance of the network.

Here we discuss the construction of the dependency graph and the computation of consistent, full subgraphs of the dependency graph using a *supergraph*. For technical reasons related primarily to analysis of the reachability of generalized circular-wait states, the theoretical analysis of [30, 32] is restricted to a special case in which the network has a unique, distinguished "input subprocess", a single node which has in-degree greater than one. This restriction is ignored in the present thesis, and our software tool is used to perform experimental research that extends beyond the current theoretical framework and will support extension of the theoretical work. In particular, we analyze a more complex version of the train example of [30] and discuss some of its eight full, consistent dependency subgraphs. These experimental results suggest that the present theoretical results can be usefully extended.

1.2 Literature Review

When a system reaches a state from which there are no possible transitions, it is said to be in a total deadlock. If a subsystem reaches a state from which no further transitions are possible, regardless of any events that may occur within the larger system, the overall system is said to be in a partial deadlock. One method of detecting deadlocks is an exhaustive search over the global system state set [4]. But this method is not suitable when the parameter values in a parameterized network are arbitrary.

Other approaches exploit symmetry of the system to detect deadlocks. For example in [17], all of the states with similar paths to possible deadlock are merged to form a 'virtual' state. This method will reduce the complexity of the deadlock detection, but it is not effective to detect deadlocks in the parameterized networks when the number of processes is arbitrary. There are other abstraction techniques [14, 15], for networks verification with large number of processes; however, it is not obvious how to achieve a suitable abstracted model. To find the abstracted model an iterative procedure is used and it is not guaranteed that a suitable abstraction will be formed.

Model checking *cut-offs* sometimes apply in deadlock analysis of the parameterized networks with finite number of classes of isomorphic subprocesses [7, 9, 8]. Relatively small bounds, cut-offs, can, under some conditions, be established, such that if the property of interest is verified successfully in all instances limited by the cut-offs, the satisfaction of the property in a parameterized network is guaranteed. Unfortunately, results of this nature are based on restrictive models of subsystem interaction, such as the unidirectional passing of tokens that carry no data, around a ring network [7, 9, 8, 20, 19, 10].

In contrast, the approach of Zibaeenejad and Thistle uses the notion of weak invariant simulation to impose directionality of control flow. This allows the modelling of systems such as traffic networks and manufacturing systems that cannot be modelled under the assumptions on which cut-offs are based [30].

1.3 Organization of the Thesis

In the second chapter, we will introduce the preliminary notations of automata, formal languages and graph theory.

In the third chapter, we described the Zibaeenejad-Thistle approach to extending deadlock-analysis methods for parameterized-chain discrete event systems. We introduce

weak invariant simulation and describe the deadlock analysis scheme. In particular we discuss the forward dependency property, and the dependency graph and its full, consistent subgraphs.

The algorithm and software are discussed in the fourth chapter; and a complex traffic network is given as a case study of deadlock analysis in the fifth chapter.

The last chapter offers concluding remarks and suggestions for future work.

Chapter 2

Preliminaries

2.1 Graphs

In this thesis, we are dealing with both directed and undirected graphs. A directed graph (or digraph) $\mathscr{D}$ is a set of nodes connected by directed edges. Formally,

$$G = (V, A) \tag{2.1}$$

where V is a set, representing vertices or nodes, and A is a set of ordered pairs of vertices, called edges. The difference with an undirected graph is that its edges are made of unordered pairs of nodes. We can define an edge by a pair of nodes (u_1, u_2), in which u_1 is a direct predecessor of u_2, and u_2 is a direct successor of u_1. A walk for two nodes u_0 and $u_k \in V$, is defined by $u_0 - u_k$, which is an alternating sequence $u_0, a_1, u_1, a_2, ..., a_k, u_k$ of nodes and edges in which $a_i = (u_{i-1}, u_i)$, $1 \leq i \leq k$. Direct successors of a node are reachable by a walk containing just one edge. The smallest walk is defined by a single edge. A closed walk is a special case, in which the first and last nodes are the same. In-degree is defined as the number of incoming arcs to a node, and the number of outgoing arcs from a node is called out-degree of that node. A directed graph is strongly connected if it contains a walk from a to b and a walk from b to a for all pair of nodes. The maximal strongly connected subgraphs of a graph are called strong components of that graph.

To compute all of the strong components of a dependency graph, we find all of the simple cycles in the dependency graph, and assign a node number to each cycle; we call such a simple cycle a supernode. There is an edge between a pair of supernodes if their corresponding cycles have common nodes. The edges are undirected, and with the supernodes they form an undirected graph called a supergraph.

For more information on graph theory see [5].

2.2 Automata and Formal Languages

Consider an alphabet Σ as a finite nonempty set of distinct symbols (events or letters) $\sigma, \tau, \ldots$. A word or string is a sequence of events. Consider Σ^+ as a set of all (nonempty) finite strings of the form $\sigma_1\sigma_2\ldots\sigma_k$, where $k \geq 1$ and the $\sigma_i \in \Sigma$. The empty string (string with no symbols), is represented by ϵ, where $\epsilon \notin \Sigma$. We then write

$$\Sigma^* := \{\epsilon\} \cup \Sigma^+ \tag{2.2}$$

All of the strings over the alphabet Σ are elements of Σ^*. For a special case of an empty alphabet set ($\Sigma = \varnothing$), we have an empty set of finite strings ($\Sigma^+ = \varnothing$) and $\Sigma^* = \{\epsilon\}$, which means the empty string is its only member. The catenation, *cat*, is defined as a product operation on the strings:

$$cat := \Sigma^* \times \Sigma^* \to \Sigma^* \tag{2.3}$$

The unit element of the catenation operation is ϵ, and catenation is associative:

$$cat(\epsilon, a) = cat(a, \epsilon) = a \qquad a \in \Sigma^* \tag{2.4}$$

$$cat(cat(a, b), c) = cat(a, cat(b, c)) \qquad a, b, c \in \Sigma^+ \tag{2.5}$$

Any subset of Σ^* (an element of the power set $Pwr(\Sigma^*)$) is defined as a language over the alphabet Σ. According to the definition, a language can contain both Σ^* and the empty language $\varnothing$. We denote the set of all languages over the alphabet Σ by $\mathcal{L}(\Sigma)$.

We can represent discrete event systems as state machines, or *generators*. The following 5-tuple, with Σ as a finite nonempty set of alphabet, X a nonempty set of states, $x_0 \in X$ as its initial state and X_m as a marked state set is an automaton (or generator) over the alphabet Σ:

$$A = (X, \Sigma, \eta, x_0, X_m) \tag{2.6}$$

η is the state transition function over Σ, and can be extended to Σ^*:

$$\eta : X \times \Sigma^* \to X \tag{2.7}$$

In this case η is a deterministic function. The nondeterministic version of this transition function can be defined as:

$$\eta : X \times \Sigma^* \to 2^X \tag{2.8}$$

where 2^X is the power set of X. If $\eta(x, \sigma) = \varnothing$, it means the transition $\eta(x, \sigma)$ is not defined. Let $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, where these alphabets could have some common symbols ($\Sigma_1 \cap \Sigma_2 \neq \varnothing$). Let $\Sigma = \Sigma_1 \cup \Sigma_2$. We define the *natural projection operation* from Σ^* to Σ_i^* as the following map:

$$P_i : \Sigma^* \to \Sigma_i^* \qquad (i = 1, 2) \tag{2.9}$$

This operation has the following properties:

$$P_i(\epsilon) = \epsilon \tag{2.10}$$

$$P_i(\sigma) = \begin{cases} \epsilon & \text{if } \sigma \notin \Sigma_i \\ \sigma & \text{if } \sigma \in \Sigma_i \end{cases} \tag{2.11}$$

$$P_i(s\sigma) = P_i(s)P_i(\sigma) \qquad s \in \Sigma^*, \sigma \in \Sigma \tag{2.12}$$

The action of P_i is to erase all of the occurrences of $\sigma \notin \Sigma_i$ from a string s. As shown by 2.12 the natural projection is catenative. We define the inverse of the natural projection as following:

$$P_i^{-1} : Pwr(\Sigma_i^*) \to Pwr(\Sigma^*) \tag{2.13}$$

The inverse of the natural projection has the following action on a language $L \subseteq \Sigma_i^*$,

$$P_i^{-1}(L) := \{s \in \Sigma^* | P_i(s) \in L\} \tag{2.14}$$

We define the *synchronous product* of $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ as $L_1||L_2 \subseteq \Sigma^*$ according to:

$$L_1||L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2) \tag{2.15}$$

The synchronous product of L_1 and L_2 represents the evolution of L_1 and L_2 at the same time, with respect to the common events [27].

A relation between two sets, A and B, is any subset of their product ($R \subseteq A \times B$). The relation R is a binary relation on A, in the case of $A = B$. The converse of a binary relation R is denoted by $R^c := \{(b, a) \in B \times A : (a, b) \in R\}$ [29].

Chapter 3

Parameterized-Chain Discrete-Event-System Networks

Parameterized Discrete Event Systems (PDES) are useful for modeling systems comprising arbitrarily large numbers of isomorphic subprocesses, but key properties such as deadlock-freedom and nonblocking are undecidable even for simple topologies such as that of a ring (each subprocess only interacts with its previous and next 'neighbours' in this topology) [21]. We can study more general network topologies, networks with several parameterized segments, by introducing Parameterized-Chain Discrete-Event-System Networks (PCDN). A PCDN is made of fixed number of 'distinguished' subprocesses and 'linear parameterized' sections. Although distinguished subprocesses may be different from each other in terms of their structures, each parameterized section contains an arbitrary number of isomorphic subprocesses which are obtained by relabeling of a template. Interactions among these subsystems are represented by the execution of the shared events. (By assumption, within a parameterized segment, each shared event is shared only between two neighbour subprocesses.) By restricting the interactions between the subprocesses, using 'weak invariant simulation', we can achieve decidability in deadlock analysis.

An instance of a PCDN (obtained by assigning lengths to each of the linear parameterized sections) is a directed graph which may contain several cycles. We isolate each cycle by disabling the events that are shared with subprocesses outside of the cycle. The result is a ring network, and we calculate the state pairs with the *forward dependency property*. After doing this calculation for all of the rings in the PCDN, we create the dependency graph. We can characterize the partial and total reachable deadlocks of the PCDN by finding *full, consistent* subgraphs of the dependency graph. As an example we use this method to analyse the deadlocks of a complex traffic network [30].

3.1 Parameterized Chain Networks

A parameterized discrete event system, PDES $\boldsymbol{\mathcal{P}}$, can be represented by a set of any synchronous products of M isomorphic finite-state subprocesses, where M is a natural number and greater than two. Formally,

$$\boldsymbol{\mathcal{P}} = \{\|_{i=1}^{M} P_i : \ M \in \mathbb{N}, \ M > 2\}, \tag{3.1}$$

where

$$P_i = (X_i, \Sigma_i, \xi_i, x_i^0, X_{mi}), \tag{3.2}$$

with $X_1 = X_2 = ... = X_M$, and M is the unknown parameter representing the number of subprocesses in the instance of a PDES. Subprocesses P_i, $1 \leq i \leq M$, are formed by appropriate relabelling of a template subprocess P_1, and are therefore isomorphic to each other. The subalphabet Σ_i is made of two parts, shared event symbols (Σ_{S_i}) and local or unshared event symbols (Σ_{L_i}). To instantiate the PDES $\boldsymbol{\mathcal{P}}$, we can define a fixed value for M. As an instantiation of the PDES $\boldsymbol{\mathcal{P}}$, consider $P^M = \|_{i=1}^{M} P_i = (X, \Sigma, \xi, x^0, X_m)$. The global alphabet of $\boldsymbol{\mathcal{P}}$ is $\Sigma = \cup_{i=1}^{M} \Sigma_i$, in which the shared event subset is $\Sigma_S = \cup_{i=1}^{M} \Sigma_{S_i}$ and the local event subset is represented by $\Sigma_L = \Sigma \setminus \Sigma_S$. We suppose that a shared event is only shared between the two consecutive subprocesses, so it can only occur in both of them at the same time. x_i^0 and x_i in 3.2 represent the initial state of the i^{th} subprocess and the state of the i^{th} subprocess P_i, respectively. We can represent the states of the instance of the PDES, P^M by $x = (x_1, x_2, ..., x_M)$, which is an M-tuple. X is representing the set of all global states, which are the M-tuples.

If an event is shared between the two subprocesses P_{i-1} and P_i, $1 \leq i \leq M$, and P_i is in a state from which it can execute that event, which is not executable by P_{i-1}, then subprocess P_i is prevented from executing the event.

Definition. Companion states, $\chi_j(\sigma_i)$, of a shared event σ_i (for the i^{th} subprocess) in the j^{th} subprocess are states x_j in P_j, with $\xi_j(x_j, \sigma_i) \neq \varnothing$.

As we have stated before, the nonblocking and deadlock-freedom properties for a general network with undefined number of interacting isomorphic finite-state subprocesses are undecidable; and we would like to formulate a decidable subproblem to check the deadlock-freedom property [21].

We define the PDES $\boldsymbol{\mathcal{P}}$, with *linear* topology, if for any member $\|_{i=1}^{M} P_i \in \boldsymbol{\mathcal{P}}$, subprocess P_i, $1 < i < M$, has events shared only with both P_{i-1} and P_{i+1}, and P_1 and P_M respectively have events shared only with P_2 and P_{M-1}. We assume all subprocesses have the same

state set and that each event is shared between at most two subprocesses. We suppose that the local events, Σ_L do not have any indices, and symbols in Σ_{S_i} either have index $i-1$ or index i: symbols in $\Sigma_{i-1} \cap \Sigma_i$ (shared events between generators P_{i-1} and P_i) have the lower index, $i-1$, while the symbols in $\Sigma_i \cap \Sigma_{i+1}$ (shared events between P_i and P_{i+1}) have index i.

To instantiate the linear PDES, we define the bijection κ and set the first subprocess, P_1, as the template subprocess. κ is a bijection relation of order N, such that, $\kappa : \Sigma \to \Sigma$ and $\forall \alpha \in \Sigma_i$, $\kappa(\alpha) \in \Sigma_{i+1}$, and it has the following properties:

$$(\forall \sigma \in \Sigma_{L_i})(\kappa(\sigma) = \sigma) \tag{3.3}$$

$$(\forall \sigma_i \in \Sigma_i \cap \Sigma_{i+1})(\kappa(\sigma_i) = \sigma_{i+1}) \tag{3.4}$$

$$(\forall \alpha \in \Sigma_i)(\forall x \in X)(\xi_i(x, \alpha) = \xi_{i+1}(x, \kappa(\alpha))) \tag{3.5}$$

where the common state set of all subprocesses in the automaton is represented by the variable X. Relation 3.3 shows how κ acts on the local events, while relation 3.4 specifies how κ acts on the shared events. Relation 3.5 is the representation of the isomorphism, which shows parameterized subprocesses are identical up to the bijection of the indices of the shared events [29].

As an example of a linear PDES, the reader may refer to the traffic network example of figure 5.1 of chapter 5. In that example each route has an arbitrary length and can be modeled by the linear PDES. By using the bijection κ we can obtain the RA_{i+1} model from the RA_i model as depicted in figure 5.3.

We define the structure of a PCDN with a PCDN graph. This graph consists of both distinguished nodes (typically denoted by squares) and parameterized nodes (denoted by circles). While distinguished nodes represent the corresponding distinguished subprocesses, parameterized nodes represent linear PDES. Parameterized nodes have in- and out-degrees of one. An instance of a PCDN can be obtained by choosing parameter values – namely, the lengths of instances of each of the linear PDES – and expanding the parameterized nodes into linear subgraphs of the appropriate lengths. In an instance of a PCDN, subprocess interact through shared events only with their neighbours. The directed arcs indicate the directions of weak invariant simulation of one neighbour by another, as explained below.

A PCDN graph is strongly connected, and contains at most one input subprocess – a distinguished node with in-degree greater than one. It may contain many output nodes – distinguished nodes with out-degree greater than one. As an example of a PCDN graph, reader may refer to the traffic network example of figure 5.1 of chapter 5.

In order to ensure decidability properties, Zibaeenejad and Thistle impose some assumptions on PCDNs in terms of the property of "weak invariant simulation" [30].

3.2 Weak Invariant Simulation

Simulation relations are used to define the semantics of process algebra in the form of the Calculus of Communicating Systems (CCS) to describe similarities between different processes [18]. In this section we will introduce the notion of *weak invariant simulation* for analysis of the synchronous products. Before that, we introduce simulation and invariant simulation relations [18].

Suppose $G_i = (X_i, \Sigma_i, \xi_i, x_{0i}, X_{mi}), i = 1, 2$, represent two generators whose alphabets Σ_1 and Σ_2 do not intersect.

Definition. Consider the binary relation $\mathcal{S} \subseteq X_1 \times X_2$, in which X_i are the states of the two generators represented by G_i, $i = 1, 2$. The relation $\mathcal{S}$ is a simulation relation if, for every $(x_1, x_2) \in \mathcal{S}$ and every $\alpha \in \Sigma_2$ and $\hat{x}_2 \in \xi_2(x_2, \alpha)$, $\alpha \in \Sigma_1$ and there exists $\hat{x}_1 \in \xi_1(x_1, \alpha)$ such that $(\hat{x}_1, \hat{x}_2) \in \mathcal{S}$. Such a relation guarantees that any sequence of events that is executed by G_2 from x_2 can also be executed by G_1 from x_1.

Now, we define invariant simulation as follows:

Definition. Consider a simulation relation $\mathcal{IS} \subseteq X_1 \times X_2$ states of two generators G_1 and G_2. The relation $\mathcal{IS}$ is invariant if, for any $(x_1, x_2) \in \mathcal{IS}$, and any $\alpha \in \Sigma_1 \cap \Sigma_2$, the following holds:

$$(\forall \hat{x}_1 \in \xi_1(x_1, \alpha))(\forall \hat{x}_2 \in \xi_2(x_2, \alpha))[(\hat{x}_1, \hat{x}_2) \in \mathcal{IS}]. \tag{3.6}$$

The 'invariance' property guarantees that the simulation relation will be preserved, as long as the two generators execute the same sequence of events. The difference between the definitions of invariance and simulation is in the the second universal quantifier of 3.6, which is an existential quantifier in the definition of simulation.

Consider $\mathcal{S}$ as a simulation relation. Then $G_1 \mathcal{S} G_2$ represents simulation of the generator G_2 by G_1: if there exists a simulation $\mathcal{S}$, such that $(x_{0_1}, x_{0_2}) \in \mathcal{S}$, then generator G_1 simulates generator G_2.

Let $P_{\hat{\Sigma}} : \Sigma^* \to \hat{\Sigma}^*$ be a natural projection with $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\hat{\Sigma} \subseteq \Sigma$. Now we define the weak version of the simulation relation as follows:

Definition. The binary relation $\mathcal{WS} \subseteq X_1 \times X_2$ is a weak simulation of G_2 by G_1 with respect to $\hat{\Sigma}$, if for every $(x_1, x_2) \in \mathcal{WS}$ and every $l_2 \in \Sigma_2^*$, if $\hat{x}_2 \in \xi_2(x_2, l_2) \neq \varnothing$, there exists $l_1 \in \Sigma_1^*$ and $\hat{x}_1 \in \xi_1(x_1, l_1)$ such that the following hold:

1. $P_{\hat{\Sigma}}(l_2) = P_{\hat{\Sigma}}(l_1)$; and

2. $(\hat{x}_1, \hat{x}_2) \in \mathcal{WS}$.

Definition. Let $\mathcal{I}$ be a weak simulation of G_2 by G_1 with respect to $\hat{\Sigma}$. $\mathcal{I}$ is a weak invariant simulation w.r.t $\hat{\Sigma}$ if any pair $(x_1, x_2) \in \mathcal{I}$ and for all $l_1 \in \Sigma_1^*$, $l_2 \in \Sigma_2^*$ and all $\hat{x}_1 \in \xi_1(x_1, l_1)$ and $\hat{x}_2 \in \xi_2(x_2, l_2)$,

$$P_{\hat{\Sigma}}(l_1) = P_{\hat{\Sigma}}(l_2) \Rightarrow (\hat{x}_1, \hat{x}_2) \in \mathcal{I} \tag{3.7}$$

When we have $\Sigma_1 \cap \Sigma_2 \subseteq \hat{\Sigma}$, weak invariant simulation with respect to $\hat{\Sigma}$ indicates that G_1 will not prevent the execution of shared events between G_1 and G_2. On the other hand, when $\Sigma_1 \cap \Sigma_2 \nsubseteq \hat{\Sigma}$, which means not all of the shared events are included in $\hat{\Sigma}$, the simulation relation does not give us enough information about possible prevention of shared events by G_1.

Suppose that we have $(x_i, x_j) \in \mathcal{I}_i$, in which $\mathcal{I}_i$ is a weak invariant simulation by G_i of another subprocess G_j with respect to all of their shared events. The existence of such a relation means that for any event $\sigma_i \in \Sigma_i \cap \Sigma_j$, if $\xi_j(x_j, \sigma_i) \neq \varnothing$, then G_i can reach companion states of the shared event, $\chi_i(\sigma_i)$ by execution of a string s of events that are not shared with G_j; the string $s\sigma_2$ is executable from x_i. By definition of weak invariant simulation we can say that, when G_i and G_j start in the respective states x_i and x_j, whatever shared event G_j may be able to execute, G_i is also in a position to execute that event. It the initial state of G_i is in a weak invariant simulation relation with initial state of G_{i+1} with respect to some alphabet $\hat{\Sigma}$, G_i weakly invariantly simulates G_{i+1} via that relation.

Zibaeenejad-Thistle state assumptions in terms of weak invariance simulations to ensure decidability of existence of 'generalized circular waits' for the special case of a single input subprocess. In this thesis, we implement their algorithms, ignoring the restriction to a single input subprocess. The algorithm is based on the notion of a 'forward dependency property', which is used to define a 'dependency graph'. The reachable generalized circular waits are then represented by the 'full, consistent subgraphs' of the dependency graph.

There are three assumptions on all subprocesses of the PCDN:

1. Each subprocess in the PCDN must have strongly connected transition graph, so there would not be any terminal state as the subprocess evolves.

2. Each shared event can have at most one companion state. In other words, interactions between two subsequent subprocesses via a shared event can occur, if the first subprocess is in a specific state.

3. If G_i weakly invariantly simulates the direct successor G_{i+1} with respect to $\Sigma_i \cap \Sigma_{i+1}$, and their interaction with other parts of the network is ignored, G_i can eventually execute events shared with G_{i+1}.

There are other assumptions on the structure of the input and output subprocesses, as indicated below:

1. For any state of an input subprocess which has an event shared with its successor subprocess, there can not be any event shared with the direct predecessor of the input subprocess from that state.
2. Any input subprocess can always provide resources requested by its successor.
3. Any output subprocess G_i, can reach companion states of shared events between the output subprocess and its successor subprocess, G_{i+1}, by a string that does not contain event shared with its other successor subprocesses (other than G_{i+1}) [30].

The third of these assumptions means that if interactions with other part of the network are ignored, G_i can eventually execute any shared event that G_{i+1} is in a position to execute.

3.3 Deadlock Analysis Scheme

To find the circular waits in an instance of a PCDN graph, first we find all of the individual cycles in the PCDN graph. Then we will create the corresponding isolated cycles (with ring topology) by disabling any transitions of distinguished subprocesses that are shared with subprocesses outside of the cycle.

3.3.1 Forward Dependency Property

The *forward dependency property* is defined using the synchronous products of subprocesses and their neighbours in an isolated cycle of the network graph. We can analyse the occurrence of a circular wait using this property. After restricting the subprocesses by weak invariant simulation, all of the events of a subprocess which are shared with their neighbours with 'lower' index in an isolated cycle will eventually get executed, while those which are shared with neighbours with 'larger' index may be blocked [29]. We define a state

pair in a synchronous production of two consecutive subprocesses as forward dependent if it has only events shared with neighbour with larger index.

Definition. Consider cycle $G^N = \|_{i=1}^{N} G_i = (X, \Sigma, \xi, x^0, X_m)$ and let $\hat{G}^N = (\hat{X}, \Sigma, \hat{\xi}, x^0, X_m)$ be its isolated version. We define the state set of the synchronous product $\hat{R}_i$ by $\hat{G}_{i-1} \| \hat{G}_i = (\hat{R}_i, \Sigma_{i-1} \cup \Sigma_i, \delta_i, x_{i-1}^0 \times x_i^0, X_{m_{i-1}} \times X_{m_i})$, $1 \leq i \leq N$. We define the following property for a state pair $(x_{d_{i-1}}, x_{d_i})$:

$$(\forall \sigma_i \in \Sigma_{i-1} \cup \Sigma_i)[(\delta_i((x_{d_{i-1}}, x_{d_i}), \sigma_i) \neq \varnothing) \Rightarrow (\chi_{i+1}(\sigma_i) \neq \varnothing)]. \tag{3.8}$$

If $(x_{d_{i-1}}, x_{d_i}) \in \hat{R}_i$ satisfies 3.8, for all i, the state $x_d \in X_1 \times X_2 \times \ldots \times X_N$ is forward dependent, which means in the synchronous product $\hat{G}_{i-1} \| \hat{G}_i$, the only events executable in this pair are just shared with the larger-index neighbour in the isolated cycle $\hat{G}_{i+1}$. For a reachable state x in $\hat{G}^N$ the satisfaction of the property 3.8 for all i means all of the subprocesses in the isolated cycle $\hat{G}^N$ will be executed after the execution of the event shared with their very next larger-index neighbours. The state x could have other shared events with subprocesses outside of the cycle $\hat{G}^N$, whose execution may break the existing circular wait. That is why the existence of a circular wait in a network does not guarantee a deadlock in the network. We can find the *generalized circular waits* by constructing the dependency graph.

3.3.2 The Dependency Graph

The dependency graph is constructed by finding dependencies in every isolated cycle of instances of the PCDN graph. Because the definition of forward dependency involves only three subprocesses, for the construction of the dependency graph it suffices to consider instances of size three of every linear, parameterized PDES in the PCDN. To create an isolated cycle of each cycle, we have to disable the shared events with subprocesses that do not belong to that cycle. For each isolated cycle, we have to find the state pairs using the forward dependency property 3.8. We calculate the synchronous product of each pair of the nodes in that isolated cycle and find all of its shared edges with the next node in that cycle. The node pair in the product that have just shared edges as its outgoing edges is the node pair that is participating in the creation of the dependency graph. The states n_{A_1} and n_{A_2} in the node pair (n_{A_1}, n_{A_2}) are vertices of the dependency graph and all pairs (n_{A_1}, n_{A_2}) are the arcs of the dependency graph. As an example of a dependency graph of a traffic network graph 5.1, the reader may refer to the figure 5.6.

3.3.3 Full and Consistent Subgraphs of the Dependency Graph

When there is a path from any node of a graph to any other, it is called a 'strongly connected' graph. A strongly connected subgraph of the dependency graph is *consistent* if 1) the subgraph contains at least one state of the input node, and 2) it contains no more than one state of any distinguished subprocess [29]. The former condition eliminates those subgraphs that only include nodes associated with parameterized sections of the network. The latter condition ensures that a consistent subgraph represents a set of states of a part of the network. Each member of this set represents a state of a subsystem of an instance in the PCDN graph: each node in the subgraph could represent a state of a distinguished or parameterized subprocess. The strongly connected property is needed to analyse the circular waits, but we have to generalize the notion of the circular waits to account for branching in our graph topologies. For this purpose, we have to introduce the fullness property.

Definition. Consider $\bar{\mathscr{D}}$ as a subgraph of the dependency graph $\mathscr{D}$. In $\bar{\mathscr{D}}$, a state of an output subprocess G_j in the network is represented by x_j. For every event σ defined from x_j in G_j that is shared with a direct successory of G_j in some instance of the PCDN, let Y^σ be the state set of that direct successor. Subgraph $\bar{\mathscr{D}}$ is *full* if for every such x_j and σ, and some $y^\sigma \in Y^\sigma$, $\bar{\mathscr{D}}$ contains an arc (x_j, y^σ) [29].

A full consistent subgraph is a graph in which, for every state of an output subprocess and any direct successor, if there is a shared event between them that can be executed when the output subprocess is in its given state, then the subgraph must contain an edge from the state of the output subprocess to some state of the direct successor subprocess. Thus each output subprocess is "dependent" on the set of its direct successors that could possibly execute a shared event with it [30]. As an example of the full, consistent subgraphs of the dependency graph, see the graphs in figure 5.7. The strongly connected, full, consistent subgraphs of the the dependency graph represent all of the reachable generalized circular waits in the PCDN graph [30].

Chapter 4

The Algorithm and the Software

4.1 The Algorithm

Algorithm 1 shows the needed steps for instantiation of the PCDN graph. Given the paths to the PCDN xml file and the directory containing the automaton models, it finds the parameterized and distinguished nodes of the PCDN graph, on the basis of their indegree and outdegree. The user should provide the names of the parameterized nodes with indegree and outdegree of one, because those are otherwise not recognizable as parameterized nodes by the software. For each parameterized node, the software will generate two additional nodes in the PCDN graph using the κ mapping.

Algorithm 1 An algorithm to instantiate a PCDN Graph

```
Take the paths of the PCDN XML file and the directory containing automaton models.
Take the name of the distinguished nodes with indegree and outdegree of one.

for all (nodes in the PCDN graph) do
    if (indegree + outdegree) > 2 then
        The node is a distinguished node.
    else
        The node is a parameterized node.
    end if
end for
for each (parameterized node (P_i)) do
    Generate two additional nodes, P_{i+1} and P_{i+2} using cyclic bijection κ
end for
```

After instantiation of the PCDN graph the software finds all of the simple cycles in the PCDN graph instance, using a depth-first search algorithm. Algorithm 2 shows the procedure of finding all of the distinct cycles in the PCDN graph instance. In depth-first search the idea is to traverse as deeply as possible from node to successor node before backtracking. Specifically, the search proceeds from node to successor node until it reaches a successor that has already been visited. It then backtracks [23].

Algorithm 2 An algorithm to find all of the distinct cycles of a graph using depth first search

```
1: path := new List <Node>;        ▷ The path contains nodes that form one simple cycle
2: cycles := new Set <cycle>;      ▷ The cycles contains all of the cycles from the node n

3: function DFS(Node n)
4:     Mark n as visited;
5:     for all (children of node n) do
6:         if (child is unvisited) then
7:             path ← child;                          ▷ Add the child node to the path
8:             DFS(child);
9:         else if (child has been visited AND child == n) then
10:            path ← child;
11:            cycles ← path;                         ▷ Add the path to the set of cycles
12:            Remove child node from the path;
13:        else if (n has just one child) then
14:            Remove node n from the path;
15:        else
16:            Continue;
17:        end if
18:    end for
19:    Mark n as unvisited;
20:    Remove node n from the path;
21:    return cycles;
22: end function

23: for all (nodes of the PCDN graph) do
24:    cycles ← DFS(node);        ▷ Add all of the cycles to the current set of the cycles
25: end for

26: distinctCycles ← cycles without any duplicate cycle;
```

To find all of the distinct cycles of the PCDN graph instance, we start by finding all of the cycles starting from each node, and then we eliminate the duplicate ones. After finding all of the distinct cycles in the PCDN graph instance, we have to isolate each cycle before calculating the synchronous product of each node with its next neighbour in the direction of the arcs in each cycle. To form an isolated cycle we have to disable some transitions

of the distinguished nodes in that cycle in the PCDN graph instance because the only subprocesses that could have shared events with subprocesses outside of that cycle are the input and output subprocesses. Algorithm 3 represents the procedure of isolating each cycle in the PCDN graph instance.

Algorithm 3 An algorithm to create an isolated cycle

```
1: function ISOLATE(List<Node> cycle)
2:     isolatedCycle := new List<Node>;
3:     for each (node in the cycle) do
4:         if (node is a distinguished node) then
5:             Remove all of the neighbours and their associated edges that do not belong
   to the cycle;
6:             Update the corresponding automaton, by removing those edges and any node
   that does not have any edges in that automaton;
7:             Add this node to the isolatedCycle;
8:         else
9:             Add this node to the isolatedCycle;
10:        end if
11:    end for

12:    return isolatedCycle;
13: end function
```

We calculate the synchronous product of the consecutive nodes in each isolated cycle of the PCDN graph instance and then compare the result with the next node in that cycle. All of the nodes in the synchronous product that have shared events only with the next node in the isolated cycle of the PCDN graph instance will contribute in creation of the dependency graph. The synchronous product's nodes are in the form of pairs of the PCDN graph's nodes. Algorithm 4 shows the computation of the synchronous product. The 'SYNC' function will take two isolated automata as its input and will return their synchronous product. First we find all of the shared edges between the two automata. The initial node of the synchronous product is made of the initial nodes of the two automata. To keep track of the synchronous nodes, we define visited and unvisited stack variables. When we visit these nodes, we will remove them from the unvisited stack variable and add them to the visited stack variable. We add the initial node to the unvisited stack variable at first, and until this stack variable is not empty we follow the following procedure.

We remove the last node inserted in the unvisited stack variable and put it in the

currentPair variable. If we have visited this node before, we will continue to the next node in the unvisited stack; otherwise we will add this node to the visited variable. (We have to visit each node just once.) We check all of the outgoing edges of the first node in the node pair of the currentPair. (We will do the same for the second node in the pair, for edges that are not shared.) If an outgoing edge is not a shared edge, then we will find just the neighbour of the first node and calculate the new node pair using the neighbour of the first node and the second node itself. If that outgoing edge is a shared edge, we will find the neighbours of both nodes in that node pair, and calculate the new node pair using both their neighbours. Now we have to check whether we have visited or calculated this node before. If we have not visited or calculated this node before, we will create it and add it to the unvisited variable. (We have to update the list of the children, parents and edges of the current node and the newly created node.) If we have calculated this node before, we just have to update its properties as before.

We will continue this process, until we have visited all of the nodes of the synchronous product graph. The result will be the synchronous product of the two automata. We do the same for each consecutive pair of the automata in each isolated cycle in the PCDN graph instance.

Algorithm 4 An algorithm to calculate the synchronous product of the two automata

```
1: function SYNC(Automaton A1, Automaton A2)
2:     Automaton result = new Automaton();
3:     sharedEdges := new List <String>;
4:     for all (edges in A1) do
5:         for all (edges in A2) do
6:             if (Edges were the same) then
7:                 add that edge to the sharedEdges;
8:             end if
9:         end for
10:    end for

11:    visited := new Stack <NodePair>; ▷ visited contains the list of the result's nodes that have been visited.
12:    unVisited := new Stack <NodePair>; ▷ unVisited contains the list of the result's nodes that are not visited.

13:    nodePair_init := new NodePair(N_1init, N_2init);      ▷ Initial node of the synchronous product is made of the two initial nodes of the automata A1 and A2
14:    unVisited.push(nodePair_init);
15:    while (unVisited is not empty) do
16:        currentPair = unVisited.pop();
17:        if (!visited.contains(currentPair)) then
18:            visited.push(currentPair);
19:            for all (out-edges of the first node in the currentPair) do
20:                if (!sharedEdges.contains(outEdge)) then
21:                    Find the child node of the firs node using this outedge;
22:                    VISIT(result, currentPair, childNodeOfFirstNodeInPair, secondNodeInPair, visited, unVisited, outEdge);
23:                else
24:                    Find the child nodes of the first and second node using this out-edge;
25:                    VISIT(result, currentPair, childNodeOfFirstNodeInPair, childNodeOfSecondNodeInPair, visited, unVisited, outEdge);
26:                end if
27:            end for
```

```
28:             for all (out-edges of the second node in the currentPair) do
29:                 if (!sharedEdges.contains(outEdge)) then
30:                     Find the child node of the second node using this out-edge;
31:                     Visit(result, currentPair, FirstNodeInPair,
    childNodeOfSecondNodeInPair, visited, unVisited, outEdge);
32:                 else
33:                     Continue;
34:                 end if
35:             end for
36:         else
37:             continue;
38:         end if
39:     end while

40:     return result;
41: end function

42: procedure Visit(Automaton result, NodePair currentPair, Node
    firstNodeInPair, Node secondNodeInPair, Stack visited, Stack unVisited,
    String edge)
43:     newNodePair = newNodePair(firstNodeInPair, secondNodeInPair);
44:     if (!visited.contains(newNodePair) AND !unVisited.contains(newNodePair))
    then
45:         Create newNodePair and add it to the result graph;
46:         unVisited.push(newNodePair);
47:         Update(result, currentPair, newNodePair, edge)
48:     else if (!visited.contains(newNodePair) AND unVisited.contains(newNodePair))
    then
49:         Find the previously calculated newNodePair.
50:         Update(result, currentPair, newNodePair, edge)
51:     else
52:         Find the previously visited newNodePair.
53:         Update(result, currentPair, newNodePair, edge)
54:     end if
55: end procedure
```

```
56: procedure UPDATE(Automaton result, Nodepair np1, NodePair np2, String edge)
57:     create the edge between the two node pairs in the synchronous graph.
58:     add the first node pair as the parent of the second node pair.
59:     add the second node pair as the child of the first node pair.
60:     add the created edge as out-edge of the first node pair.
61:     add the created edge as in-edge of the second node pair.
62: end procedure
```

Algorithm 5 This algorithm finds the nodes of the synchronous product that just have shared events with the next automaton

```
1: function GETDEPTRANSITIONS(Automaton A1, Automaton A2, Automaton A3)
2:     result := new List <String>;  ▷ result contains the transitions of the dependency graph
3:     Automaton product = SYNC(A1, A2);
4:     sharedEdges = The names of the all of the shared edges between product and A3;
5:     for each (node of the product) do
6:         outEdgesNames = new List <String>;
7:         for each (edge of the node) do
8:             Add edge name to the outEdgesNames
9:         end for
10:        if (sharedEdges contains all of the outEdgesNames) then
11:            Add the node name to the result;
12:        end if
13:    end for
14:    return result;
15: end function
```

To find which node pairs of the synchronous product have a role in the creation of the dependency graph, we have to compare the synchronous product with the next automaton in that isolated cycle. Those nodes in the synchronous product that have shared events only with the next automaton in that isolated cycle make a contribution to the creation of the dependency graph. Algorithm 5 shows this procedure in detail. GetDepTransitions will take three consecutive automata in an isolated cycle, and calculate the synchronous product of the first two. It finds the name of the shared edges between the synchronous product and the third automaton. For each node of the synchronous product, if all of its outgoing edges are shared edges with the third automaton, that node will be returned as

part of the result.

To create the dependency graph, we connect the first node in such a node pair of the synchronous product to the second node by a directed "no-name" edge. (These synchronous product nodes are thus the list of the transitions of the dependency graph.) At the end, we have to eliminate the duplicate edges in the dependency graph. Algorithm 6 represents this procedure in detail. In this algorithm, to calculate the dependency graph transitions we have to check the position of the automaton in the isolated cycle. If the first automaton is the last node in the isolated cycle, the second and the third ones are the first and second nodes in the isolated cycle, and so on.

Algorithm 6 An algorithm to create the dependency graph

```
1: Instantiate the PCDN graph using algorithm 1;
2: Find all of the distinct isolatedCycles using algorithms 2 and 3;
3: depTransitions := new List <String>;          ▷ depTransitions contains all of the
   transitions of the dependency graph.
4: for each (cycle of the isolatedCycles) do
5:     for each (node of the cycle) do
6:         if (The node is the last but one in the cycle) then
7:             Add all of the transitions calculated with GetDepTransitions(node,
   lastNode, firstNode) to the depTransitions;
8:         else if (The node is the last node in the cycle) then
9:             Add all of the transitions calculated with GetDepTransitions(node,
   firstNode, secondNode) to the depTransitions;
10:        else
11:            Add all of the transitions calculated with GetDepTransitions(node,
   nextNode, secondNextNode) to the depTransitions;
12:        end if
13:    end for
14: end for

15: Remove the duplicate transitions from depTransitions;
```

After creation of the dependency graph, we find all of the full, consistent subgraphs of the dependency graph. A consistent subgraph is a strongly connected subgraph of the dependency graph that contains at least one input node and does not contain more than one state of any distinguished subprocess. The following statement defines the fullness condition:

If there is an output node in a maximal connected component of a dependency graph, we have to find all of the children of that output node in the PCDN graph. Then, we have to check the state of that output node in that component, and if the automaton of that output node in that specific state and the automaton of the output node's children have any shared event, there should be exactly one edge between that output node and any state of that child in the maximal connected component of the dependency graph to satisfy the fullness condition.

Each cycle in the dependency graph is a strongly connected component, so we will find all of the distinct cycles in the dependency graph using algorithm 2. We assign an ID to each cycle and call each one of them a 'supernode'. Each created supernode is connected with just one edge to another supernode, if they have any common node in their cycles in the dependency graph. These supernodes and their edges will form an undirected graph, called the 'supergraph'. Algorithm 7 shows the procedure of the creation of a valid supergraph. By valid we mean there is no supernode that has more than one state of a given distinguished node, which would violate one of the consistency conditions.

Algorithm 7 An algorithm to create a valid supergraph.

```
1: Create the dependency graph using algorithm 6;
2: Find distinct cycles in the dependency graph using algorithm 2;
3: for each (distinct cycle in the dependency graph) do
4:     Assign an Id to form a supernode
5:     if (All nodes of each supernode are distinct states of distinguished nodes.) then
6:         This supernode is a valid supernode;
7:     else
8:         Remove this supernode from the list of supernodes;
9:     end if
10: end for
11: for (i = 0; i < numberOfSuperNodes − 1; i++) do
12:     for (j = i + 1; j < numberOfSuperNodes; j++) do
13:         if (i ! = j AND superNodes have shared nodes) then
14:             Create an undirected edge between them
15:         end if
16:     end for
17: end for
```

Next, we have to merge the cycles represented by valid subgraphs to form maximal consistent subgraphs of the dependency graph. However, for consistency, we must ensure that

Distinguished Node	States
AA1	3
AA2	3
AB1	3
AB2	3
AC1	3
AC2	3
IA1	2, 3
IA2	2, 3, 4, 6
IB1	2, 3
IB2	2, 4, 6
IC1	2, 3
IC2	2, 3, 4, 6
A1	3

Table 4.1: List of the distinguished nodes and their states

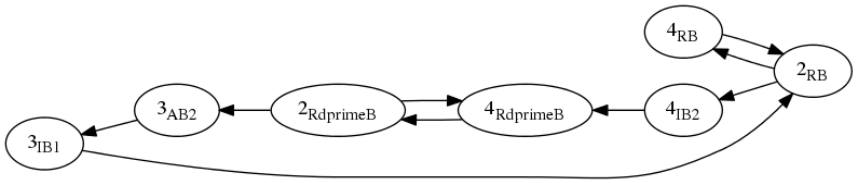
we do not merge supernodes featuring different states of the same distinguished subprocess.

This is a version of forbidden-pair problem [12, 13, 28], which is NP-complete. We solve it by first choosing an assignment of states to each of the distinguished subprocesses. We then remove from the supergraph all supernodes that are inconsistent with that assignment, and then find the connected components of the resulting subgraph. The merger of all of the supernodes in any such subgraph is a maximal consistent subgraph of the dependency graph. All such mergers are computed, for every possible assignment of states to the distinguished processes.

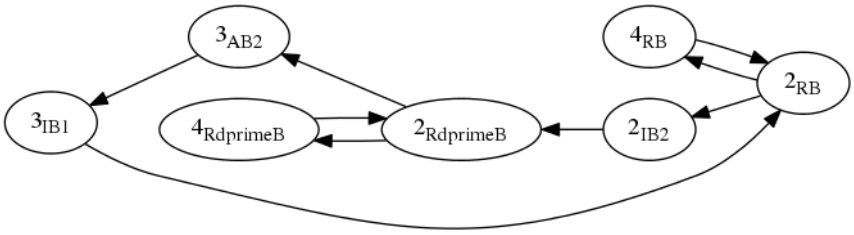
For example, the distinguished nodes of the dependency graph in figure 5.6 are given in table 4.1. As depicted in figure 4.1, we can either have state 4 of IB_2, or state 2 of IB_2.

The total number of legal combinations that do not violate the consistency condition is 384. After finding all of the maximal consistent subgraphs in the supergraph, we will remove those subgraphs that do not satisfy the fullness condition.

Algorithm 8 finds the allowed combinations of distinguished nodes.



(a) Subgraph containing state 4 of IB_2



(b) Subgraph containing state 2 of IB_2

Figure 4.1: Two maximal connected full, consistent subgraphs of the dependency graph of figure 5.6

Algorithm 8 This algorithm will generate a list of different combinations of states of distinguished subprocesses.

```
1:  function getDistCombinations( )
2:      checkedNames := new HashSet <String>; ▷ checkedNames are the names of the
        distinguished nodes that have been entered to the distinguishedNodes map.
3:      distinguishedNodes := new HashMap < String, Set < DependencyNode > >;  ▷
        distinguishedNodes keys are the names of the distinguished node names, and the
        corresponding values are the dependency node names with different states.
4:      for all (Nodes in the dependency graph) do
5:          if (Node is a distinguished node) then
6:              if (checkedNames contains the node name) then
7:                  distinguishedNodes.get(node name).add((DependencyNode) node);
8:              else
9:                  checkedNames.add(node name);
10:                 Add the new key and its corresponding value to the distinguishedNodes;
11:             end if
12:         end if
13:     end for
```

14: *combinations* := Sets.cartesianProduct(values of the *distinguishedNodes*);
15: return *combinations*;
16: **end function**

The following algorithm traverses the graph for each combination, from every supernode that has an input node in its corresponding cycle that does not violate the combination condition. Those paths that do not satisfy the combination will be omitted. Some of the supernodes with input node in their corresponding cycles, 'input-supernodes', violate the combination, so we have to check which input-supernode could be the initial node of traversing. The result of algorithm 9 contains all of the maximal consistent subgraphs that have the same combination of the distinguished nodes in their path. Algorithm 10 traverses the supergraph for each one of the combinations of the distinguished nodes, and for each combination, tries every input-supernode that satisfies that combination. As a result it finds all of the maximal consistent subgraphs of the supergraph.

Algorithm 9 This function starts traversing the supergraph from each input-supernode that satisfies the combination of the distinguished nodes.

1: **function** TRAVERSE(List <DependencyNode> combination, Set<SuperNode> input-SuperNodes)
2: $startNodes$:= new HashSet<SuperNode>;
3: **for all** (input-supernodes) **do**
4: **if** (combination contains all of the distinguished nodes of the input-supernode) **then** Add input-supernode to the $startNodes$;
5: **end if**
6: **end for**
7: $result$ = new HashSet<Set<SuperNode> >; ▷ $result$ contains all of the distinct subgraphs which are maximal connected, consistent subgraphs.
8: **for** ($startNode$: $startNodes$) **do**
9: $visitInfo$ = new HashMap<SuperNode, Boolean>; ▷ $visitInfo$ have some information about the visitation of each supernode. values of each supernode could be: false means "not visited" or true means"visited"
10: **for all** (supernodes in the supergraph) **do**
11: $visitInfo$.put(supernode, false);
12: **end for**
13: $subGraph$ = new HashSet<SuperNode>; ▷ $subGraph$ contains all of the supernodes which are allowed to be connected to form one of the fully connected components.
14: $allDistinguishedNodes$ = new HashSet<Node>; ▷ $allDistinguishedNodes$ is the list of all of the distinguished nodes that have been visited so far.
15: $unVisited$ = new Stack<SuperNode>(); ▷ This list contains supernodes that are going to be visited.
$unVisited$.push($startNode$); ▷ Add the start node to the unVisited list
16: **while** $unVisited$ is not empty **do**
17: $currentNode$ = $unVisited$.pop(); ▷ // Remove the first supernode from the unvisited list.
18: **if** (We have not visited $currentNode$ before) **then**
19: Visit $currentNode$ and update $visitInfo$
20: Collect all of the distinguished nodes of the $currentNode$ in $allDistinguishedNodes$
21: **if** $combination$ contains all of the $allDistinguishedNodes$ **then**
22: Add $currentNode$ to the $subGraph$

```
                    for all (Children of the currentNode) do
                        push each child to unVisited
                    end for
                else
                    Adding currentNode violates the combination condition, So we have to remove those distinguished nodes added to the allDistinguishedNodes
                end if
            else
                Continue;
            end if
        end while
        result.add(subGraph);
    end for
     return result;
end function
```

Algorithm 10 This function will traverse the supergraph, and generate all of the possible maximal consistent subgraphs.

```
1: function CONSISTENT( )
2:     result := new HashSet<Set<SuperNode> >;
3:     combinations = GETDISTCOMBINATIONS( );
4:     inputSuperNodes := new HashSet<SuperNode>;    ▷ inputSuperNodes contains those supernodes that have at least one input node in their associated cycles.
5:     for all (Supernodes in the supergraph) do
6:         if (Supernode has one or more input nodes) then
7:             Add the supernode to inputSuperNodes;
8:         end if
9:     end for
10:    for all (Combinations in combinations) do
11:        Calculate TRAVERSE(combination, inputSuperNodes);
12:        Add it to the result;
13:    end for
14:    return result;
15: end function
```

Now, to get the maximal connected full, consistent subgraphs, we have to omit those

subgraphs which violate the fullness condition. Note that the label of a node in the dependency graph specifies a subprocess - be it a distinguished subprocess or an instance of the template for a parameterized section - and also a state of that subprocess.

Now, if a node in the subgraph corresponds to a given state of a given output subprocess, then for any successor subprocess such that the output subprocess can execute a shared event with that successor subprocess from the given state of the output subprocess, fullness requires that the subgraph contain exactly one edge linking the given node to a node corresponding to the successor subprocess. For a given output-subprocess node, algorithm 11 returns the list of all such successor subprocesses.

Algorithm 11 This function will return list of the neighbors' names for each dependency-graph output node.

```
1: function findOutputNodeNeighbours( )
2:     outputSuperNodes := new HashSet<SuperNode>;  ▷ List of the supernodes that have at least one output node
3:     for all (supernodes in the supergraph) do
4:         if (supernode has an output node) then
5:             Add the supernode to outputSuperNodes
6:         end if
7:     end for
8:     outputNeighbours := new HashMap<Node, Set<String> >;
9:     for each (output − supernode in outputSuperNodes) do
10:        for each (outputNode in output − supernode) do
11:            outputNodePCDN = corresponding automaton of the output node;
12:            outputNodeEdgesInAutomaton = List of the out-edges of the output node in a specific state;
13:            outputNodeChildren = list of the children of the output node in automaton in a specific state;
14:            for each (child in outputNodeChildren) do
15:                childEdges = List of the edges in the child's automaton;
16:                if (outputNodeEdgesInAutomaton and childEdges have shared edges) then
17:                    Add it to outputNeighbours;
18:                else
19:                    Continue;
20:                end if
21:            end for
22:        end for
23:    end for
24:    return outputNeighbours;
25: end function
```

Algorithm 12 will calculate all of the maximal full, consistent subgraphs. For each 'output-node, child' pair that we calculated in algorithm 11 we find a set of supernodes that have the output-node and its child in their cycles. There might be more than one supernode needed to satisfy the fullness condition. The variable 'fullnessConditions' contains those combinations of supernodes that satisfy the fullness condition for a given created sub-

graphs. For each output node in the dependency graph, we create another variable that holds the fullness conditions for that output node, and name it 'fullnessConditionsOfCurrentOutputNode'. 'fullnessConditionsOfCurrentOutputNode' is a list of sets, and each set represents those supernodes that satisfy the fullness condition for just one 'output-node, child' pair. We have to search through all of the supernodes. (There could be more than one supernode with this combination of 'output-node, child', because some supernodes share parts of their cycles with other supernodes.) The list contains all of the sets of the supernodes for different children of the current output node. To satisfy the fullness condition for one output node, there should be an edge from that output node to each of the children in the list computed by algorithm 11. The 'fullnessConditionsOfCurrentOutputNode' has the following structure (suppose the output node in a specific state should have three children connected to it in the subgraph, to satisfy the fullness condition): [all of the supernodes which have pair of 'output-node, child1'], [all of the supernodes which have pair of 'output-node, child2'], [all of the supernodes which have pair of 'output-node, child3']. In this structure, the output node is connected to just one of its children in each bracket. So we have to calculate all of the combinations that make the output node connected to all of its children once. If the output node and child1 appear in three different cycles of the dependency graph, the output node and child2 appear in four different cycles of the dependency graph and the output node and child3 appear in two different cycles of the dependency graph, we have 3*4*2 different combinations of those cycles that could satisfy the fullness condition for this specific output node.

Algorithm 12 This function will generate maximal connected full, consistent subgraphs.

```
1: function fullConsistent( )
2:     consistentSubGraphs = consistent( );          ▷ generate all of the consistent subgraphs.
3:     outputNeighbours = findOutputNodeNeighbours( );          ▷ List of the all of the neighbors of each output node that should be connected to that output node in the subgraph, to satisfy the fullness condition.
4:     fullnessConditions := new HashSet<List<SuperNode> >;
5:     for each (output node in key set of outputNeighbours) do
6:         fullnessConditionsOfCurrentOutputNode = new ArrayList<Set<SuperNode> >;
7:         for each (child of the output node) do
8:             fullnessCond = new HashSet<SuperNode>;      ▷ fullnessCond is a set of supernodes that have the pair of 'output-node, child' in their cycles.
9:             for each (supernode in the supergraph) do
10:                if (The supernode contains the output node) then
11:                    if (The output node is the last node in the cycle) then
12:                        if (The first node in the cycle is in the list of the children of the output node to satisfy the fullness condition) then
13:                            Add the supernode to the fullnessCond;
14:                        end if
15:                    else
16:                        if (The next node in the cycle is in the list of the children of the output node to satisfy the fullness condition) then
17:                            Add the supernode to the fullnessCond;
18:                        end if
19:                    end if
20:                end if
21:            end for
22:            Add the fullnessCond to the fullnessConditionsOfCurrentOutputNode;
               ▷ fullnessConditionsOfCurrentOutputNode is for all of the possible combinations for one output node and all of its children.
23:        end for
24:        combinations = Sets.cartesianProduct(fullnessConditionsOfCurrentOutputNode);
25:        fullnessConditions.addAll(combinations);
26:    end for
```

```
27:     for each (consistent subgraph in consistentSubGraphs) do
28:         for each (combination in fullnessConditions) do
29:             if (consistent subgraph has shared supernode with combination) AND (consistent subgraph does not contain all of the supernodes of the combination) then
30:                 Remove the consistent subgraph;
31:                 exit the loop;
32:             end if
33:         end for
34:     end for
35:     return consistentSubGraphs;          ▷ consistentSubGraphs contains all of the consistent, full subgraphs.
36: end function
```

4.2 The Implementation

The software is written in Java and it uses two external open source libraries: 1) *Simple XML SERIALIZATION*[1] and 2) *Guava*[2] to parse the input XML files and do some part of the calculations, respectively. To draw the dependency graph and its full, consistent subgraphs the software utilizes *Graphviz*[3]. To generate the PCDN graph and the automaton of each node in the PCDN, we will use the *Integrated Discrete-Event Systems Software (IDES)*[4]. IDES provides an environment like paper and pen for drawing and exporting the defined models of the automata to various file formats. We export these models as an XML file with .xmd extension. The user should create the PCDN graph and the automaton models of the distinguished nodes and template nodes of each parameterized section using IDES, and save all of the automata models in a separate directory.

Figure 4.2 represents the class diagram of the simple package. The classes in this package are used to parse the XML file that is given by the user. All of the attributes in this diagram have sets of getters and setters that are omitted. Each element in the XML file is represented by a class in this package. At the top we have a 'Model' element in the XML file which have 'Data' and 'Meta' elements. A 'Data' element contains information about nodes and elements and their relationship, while a 'Meta' element contains information about positioning of nodes and edges in the IDES. Although we do not need the information in the 'Meta' element, in order to parse the XML file correctly we have to define the

[1] http://simple.sourceforge.net
[2] https://github.com/google/guava
[3] http://www.graphviz.org/
[4] https://qshare.queensu.ca/Users01/rudie/www/software.html

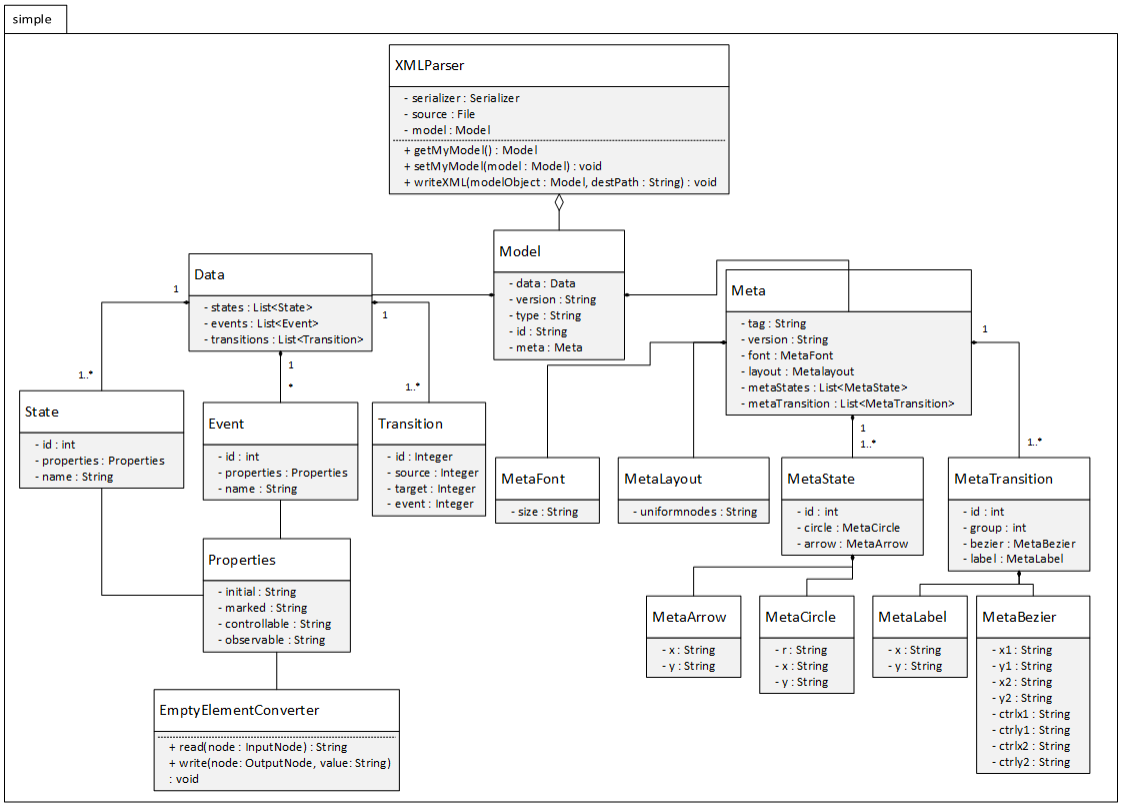


Figure 4.2: UML Class diagram of the simple package

corresponding classes. At the top of the diagram in figure 4.2, 'XMLParser' is the class that is responsible for reading and writing to the XML file. It takes the path of the XML file and returns all of the elements of the XML file in a model variable. The model variable later will be used to get information on the graph. The 'Data' element could have one or many 'States' and 'Transitions' elements. It could have zero or more 'Event' elements. In a PCDN network, edges does not have any name, so there is no 'Event' element in 'Data', but in other automaton models, the 'Data' element has as many 'Event' elements as there are edges in the automaton graph. 'State' and 'Event' elements have the same 'Properties' element as shown in figure 4.2. The 'Properties' element could have an empty initial element, which after parsing the XML file is converted to null. The absence of an initial element is also represented by the null value. To overcome this problem we have created the 'EmptyElementConverter' class. This class converts the null value to the empty string, in case of presence of the empty initial tag.

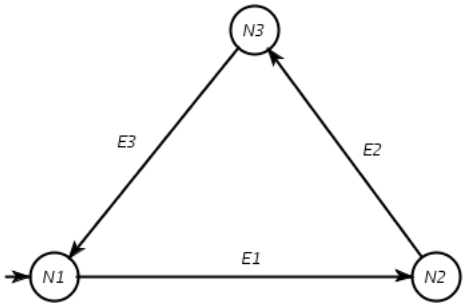


Figure 4.3: Sample directed graph with three nodes and three edges

There are some naming conventions to create these graphs:

1. Node names of the PCDN graph and the automaton file names must be the same.
2. Node names of the PCDN graph must start with letters followed by numbers, and the edges should not have any name.
3. Because the node names of the PCDN graph are the file names of the automaton graphs, they should not contain any special characters including ', ", etc.
4. If the template subprocess has been defined, e.g. as 'B_1', naming other subprocesses using similar names like 'B_2', 'B_3', etc. is prohibited and the software will generate two additional subprocesses for each parameterized node in the PCDN graph, by incrementing parameters of the template node by one and two.
5. In the automaton graph, node names should be just numbers and the edge names could be any appropriately suggestive names.

Figure 4.3 shows a sample directed graph with three nodes and three edges that has been created using IDES, and listing 4.1 shows the XML file exported by the IDES.

The node and edge names are given between the 'name' element of 'state' and 'event' elements. The 'id' attribute of the 'model' element indicates the name of the graph, while the 'id' attribute of 'state', 'event' and 'transition' elements are generated automatically by the IDES and indicate a unique identifier for the nodes, edges and transitions in the graph, respectively. The 'initial' empty element indicates the first node is an initial node and the 'observable' empty element gives more information about the event, which does not have any application in our software. In the 'transition' element, 'source', 'target' and

'event' attributes show the edge id and its source and destination nodes' id. The 'meta' element and all of its contents are used to show the positions of the nodes and edges in the IDES and do not have any application in our software. In the PCDN graph, because the edges do not have name, its XML file does not have 'event' elements or 'event' attributes; and 'transition' elements indicate which nodes are connected to each other.

Listing 4.1: Sample Graph

```
<?xml version="1.0" encoding="UTF-8"?>
<model version="2.1" type="FSA" id="Graph">
<data>
        <state id="1">
                <properties>
                        <initial />
                </properties>
                <name>N1</name>
        </state>
        <state id="2">
                <properties/>
                <name>N2</name>
        </state>
        <state id="3">
                <properties/>
                <name>N3</name>
        </state>
        <event id="1">
                <properties>
                        <observable />
                </properties>
                <name>E1</name>
        </event>
        <event id="2">
                <properties>
                        <observable />
                </properties>
                <name>E2</name>
        </event>
        <event id="4">
                <properties>
```

```
                    <observable />
                </properties>
                <name>E3</name>
        </event>
        <transition id="4" source="1" target="2" event="1">
        </transition>
        <transition id="5" source="2" target="3" event="2">
        </transition>
        <transition id="6" source="3" target="1" event="4">
        </transition>
</data>
<meta tag="layout" version="2.1">
        <font size="12.0"/>
        <layout uniformnodes="false"/>
        <state id="1">
                <circle r="18.0" x="451.0" y="467.0" />
                <arrow x="36.0" y="0.0" />
        </state>
        <state id="2">
                <circle r="18.0" x="737.0" y="467.0" />
                <arrow x="1.0" y="0.0" />
        </state>
        <state id="3">
                <circle r="18.0" x="601.0" y="283.0" />
                <arrow x="1.0" y="0.0" />
        </state>
        <transition id="4">
                <bezier x1="451.0" y1="467.0" x2="737.0" y2="467.0" ctrlx1="
                   546.3333129882812" ctrly1="467.0" ctrlx2="
                   641.6666870117188" ctrly2="467.0" />
                <label x="5.0" y="-14.0" />
        </transition>
        <transition id="5">
                <bezier x1="737.0" y1="467.0" x2="601.0" y2="283.0" ctrlx1="
                   691.6666870117188" ctrly1="405.6666564941406" ctrlx2="
                   646.3333129882812" ctrly2="344.3333435058594" />
                <label x="21.0" y="-3.0" />
        </transition>
```

```
66          <transition id="6">
67                  <bezier x1="601.0" y1="283.0" x2="451.0" y2="467.0" ctrlx1="
                        551.0" ctrly1="344.3333435058594" ctrlx2="501.0" ctrly2=
                        "405.6666564941406" />
68                  <label x="-20.0" y="-9.0" />
69          </transition>
70  </meta>
71  </model>
```

To parse the XML files we create a class for each element of the XML file and place annotations before class declarations or fields to indicate root elements or attributes in the XML file. As an example, to model the 'state' element, we define a class like the following:

```
1  @Root(name="state")
2  public class State {
3     @Attribute
4     private int id;
5     @Element(required = false)
6     private Properties properties;
7        @Element(name ="name")
8     private String name;
9
10    ... setters and getters ...
11 }
```

The '@Attribute' indicates that the following variable is an attribute in the XML file and '@Element(required=false)' indicates that the following element in the XML file is not mandatory. To represent the 'data' element, we have to annotate the class as root and its children as a list by placing '@ElementList(inline=true)' before each field.

Figure 4.4 represents the rest of the classes in our software. The 'Graph' class takes as its input the path to the graph XML file. It uses the simple package and, by creating a model object it generates lists of nodes and edges in separate variables. It has two hashmap variables for the respective assignment of each node and edge to their ids. This class also has methods 'createNode' and 'createEdge'. The createNode method generates a new id and assigns it to the newly created node and then adds that node to the list of nodes. It also updates the hashmap of the nodes and then returns the newly created node as its output. The same procedure applies for the createEdge method. This class has a method to compute all of the cycles and all of the distinct cycles of the graph using a depth-first

Attributes	Explanation
lastId	It is needed for creation of new nodes and edges.
filename	It is the name of the XML file.
name	This is the name of the graph, which is the filename without its extension.
xmlFile	It Contains the XML model object.
states	This is the list of all of the states from the XML file.
events	This is the list of all of the events from the XML file.
transitions	This is the list of all of the transitions from the XML file.
nodes	It is list of all of the nodes in the Graph.
nodeMap	This is a map between a Node object and its ID.
edges	This is list of all of the edges in the Graph
edgeMap	This is a map between an edge object and its ID.
onStack	onStack[v] is true, if v is on the stack.
edgeTo	edgeTo[n] gives previous vertex on path to n.
cycle	one directed cycle (or null if no such cycle).
cycles	all of the directed cycles (or null if no such cycles).

Table 4.2: List of the attributes of the Graph class

search algorithm. The 'getSharedEdges' method will return shared edges between two nodes.

Table 4.2 represents a list of all attributes along with their explanations for the Graph class. The Graph class is the parent of Automaton, Pcdn and Dependency classes.

The 'Node' class is the parent of the 'AutomatonNode', 'PcdnNode' and 'DependencyNode' classes. This class creates lists of children (nodes that are after this node in the directed graph), parents, outgoing and ingoing edges of the node object. It has a copy constructor which creates duplicates of the node using a deep copy method. It has methods to add and remove any child, parent, outgoing edge and ingoing edge from their corresponding lists. Method 'removeNeighbours' takes node 'n' and a cycle as its inputs and then removes those neighbours of the node 'n', that do not belong to the cycle. It removes all of the associated edges of that neighbour from the node 'n' and returns the node as its output. We have overridden the 'equals' and 'hashCode' methods in this class. Objects that are equal must have the same hash code within a running process but unequal objects might also have the same hash code.

The 'Edge' class is the parent of 'AutomatonEdge' and assigns a source and destination

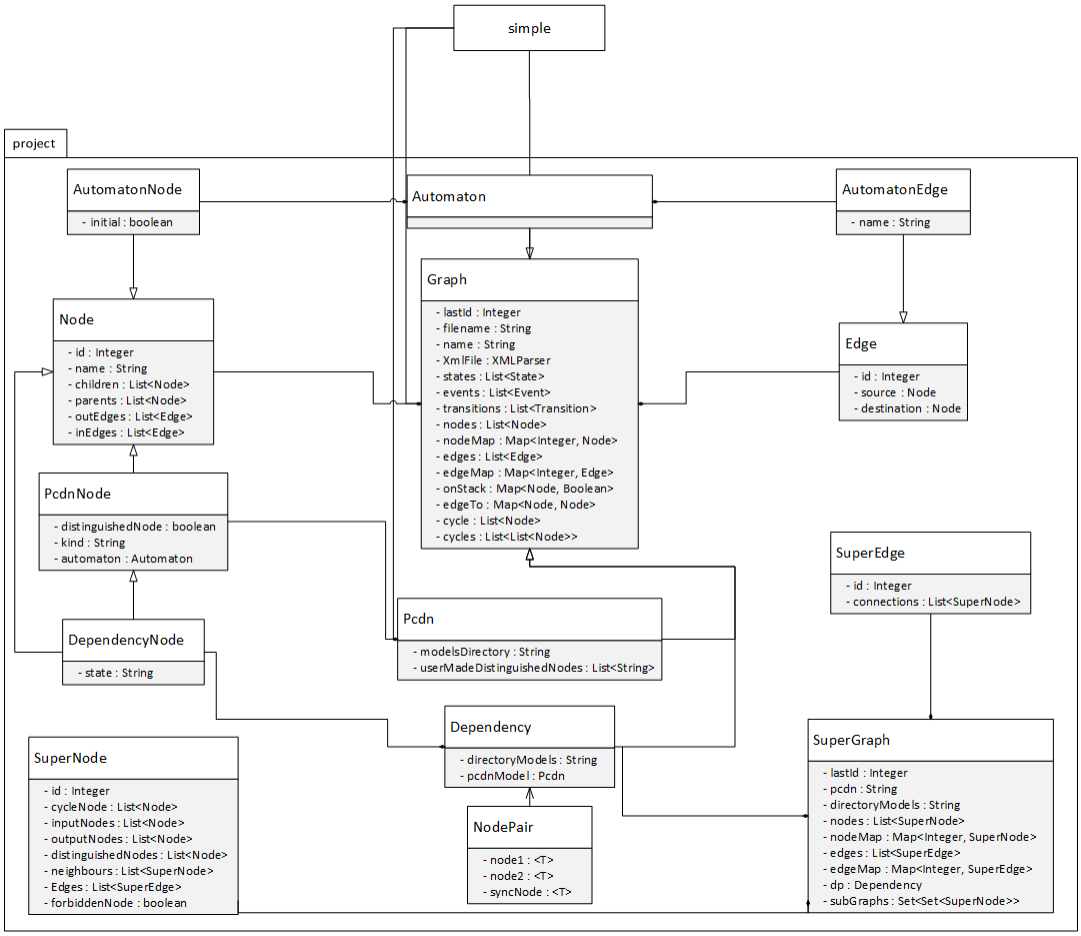


Figure 4.4: UML Class diagram of the project class

node to the edge object. Each edge object has a distinct 'id'. This class also has a copy constructor to duplicate the edge object properly. We have overridden the 'equals' and 'hashCode' methods in this class too.

The 'Pcdn' class will take as its input the paths to the PCDN graph file and the automaton directory, and also asks the user to input the name of the distinguished nodes with indegree and outdegree of one. This class also has a copy constructor to duplicate the Pcdn object. The 'instantiatePcdn' method generates two extra nodes by increasing the node number and all of the parameters in the associated graphs of the parameterized nodes by one and two, in the PCDN graph, respectively. It also creates two XML files for each parameterized section in the automaton directory model. This class has another

method named 'getIsolatedCycles' which returns an isolated version of each distinct cycle in the PCDN graph, by removing shared events of each distinguished node with outside of the cycle. This class also has methods to return lists of parameterized and distinguished nodes in the PCDN graph.

The 'PcdnNode' class extends the 'Node' class and adds the 'kind' attribute to each node. The kind attribute could be 'input node', 'output node' or none, which is calculated in this class. Nodes which have the number of $ingoingedges > outgoingedges$ are considered as input nodes. There is another method that assigns an automaton object to each PcdnNode object. The 'setDistinguishedNode' method finds all of the distinguished and parameterized nodes based on their indegree and outdegree. All of the nodes with $(indegree + outdegree > 2)$ are considered as distinguished nodes. This class also has a copy constructor to duplicate pcdnNode objects. Like the 'Node' class, this class has a 'removeNeighbours' method which takes as its input node 'n' and a cycle, and removes those neighbours of the node 'n', that do not belong to the cycle. The difference between this method and the one in the 'Node' class is that this method will also update the corresponding automaton of the PcdnNode.

The 'Automaton' class extends the 'Graph' class and initializes the automaton object. The 'updateAutomaton' is a method that takes a list of edges as its input and then updates the automaton graph after removing all of those edges. This method will update the lists of nodes, edges, nodeMap and edgeMap as well. After removing edges, if a node does not have any edge, it should be removed. The 'getSharedEdges' method will take two automaton objects as its input and will return the list of the shared edges between them. The 'createAutomatonNode' and 'createAutomatonEdge' methods will respectively create a new node and edge as their names suggest.

The 'AutomatonNode' class extends the 'Node' class, so it inherits all of the methods and attributes in the 'Node' class. Each 'AutomatonNode' object has the 'initial' attribute, which specifies whether or not the current node is an initial node in the automaton graph. There is a method in the 'AutomatonNode' class that can specify this attribute. We have a copy constructor to duplicate 'AutomatonNode' objects and to compare them with each other; we have overridden the 'equals' and 'hashCode' methods.

'AutomatonEdge' extends the 'Edge' class, and adds one more attribute to 'AutomatonEdge' objects, which is a 'name'. We have also defined a copy constructor in this class for duplication purposes and overridden 'equals' and 'hashCode' methods as well.

The 'nodePair' class is a helper class which has three generic attributes, node1, node2 and syncNode. The attributes node1 and node2 could be the first and second nodes in the synchronous product and syncNode could contain a node in the synchronous product

graph. We have overridden the 'equals' and 'hashCode' methods, so we can compare 'nodePair' objects with each other.

The 'Dependency' class extends the 'Graph' class. This class has a method named 'initialize' which generates the dependency graph of a PCDN. It first finds all of the isolated cycles of the PCDN graph, and then in each cycle calculates the synchronous product between consecutive nodes in that cycle and compares the result with the next node. All of the nodes in the synchronous product that have shared events only with the next node in the isolated PCDN will create the dependency graph. All of the nodes in the synchronous product are pairs of nodes of the PCDN graph, and to create the dependency graph, we will connect the first node in a node pair to the second node, by a directed no-name edge. There is a method named 'synchronousProduct' that calculates the synchronous products of two automata and returns an automaton as its output. This class also has a method called 'createDependencyNode' which takes the name of the PCDN node and its state, and creates a new DependencyNode object.

'DependencyNode' extends the 'PcdnNode' class and has the 'state' attribute. Each node in the PCDN graph has a corresponding automaton and the state will specify the node name of the automaton. We have overridden the 'equals' and 'hashCode' methods in this class in order to be able to compare 'DependencyNode' objects with each other.

'SuperGraph' does not inherit from the 'Graph' class because it contains undirected graphs. Table 4.3 presents a list of attributes along with their explanations for the SuperGraph class. This class has a method named 'initialize' which creates the supergraph of the dependency graph. It finds all of the distinct cycles of the dependency graph, and associates a node number to each distinct cycle. Each distinct cycle would be one supernode of the supergraph; supernodes are connected to each other if the corresponding distinct cycles have common nodes. We have used a depth-first search algorithm to find cycles in the dependency graph. We first find all of the maximal connected consistent subgraphs of the dependency graph by computing all possible assignments of states to distinguished subprocesses (in order to satisfy consistency), and then finding those subgraphs that satisfy fullness.

The 'SuperNode' class assigns a cycle to each supernode and contains lists of input and ouput nodes of the dependency graph. It also has a list of the neighbours of the supernode. Another attribute is 'Edges' which is a list of the edges that are connected to this supernode. If a cycle in the dependency graph violates the consistency condition, its 'forbidden' attribute will be set to true, and it will be omitted from the calculations.

The 'SuperEdge' class assigns an 'id' to each edge and contains a list of two supernodes that are connected to each other via this superedge.

Attributes	Explanation
lastId	It is needed for creation of new nodes and edges.
pcdn	This is the location of the PCDN '.xmd' file.
directoryModels	This is the location of the directory of the automata models.
nodes	It is a list of all of the nodes in the Graph.
nodeMap	This is a map between a Node object and its ID.
edges	This is a list of all of the edges in the Graph
edgeMap	This is a map between an edge object and its ID.
dp	dp is the dependency graph.
subGraphs	It contains all of the maximally connected full, consistent subgraphs.

Table 4.3: List of the attributes of the SuperGraph class

Chapter 5

Case Study

5.1 Simple Traffic Network vs. Complex Traffic Network

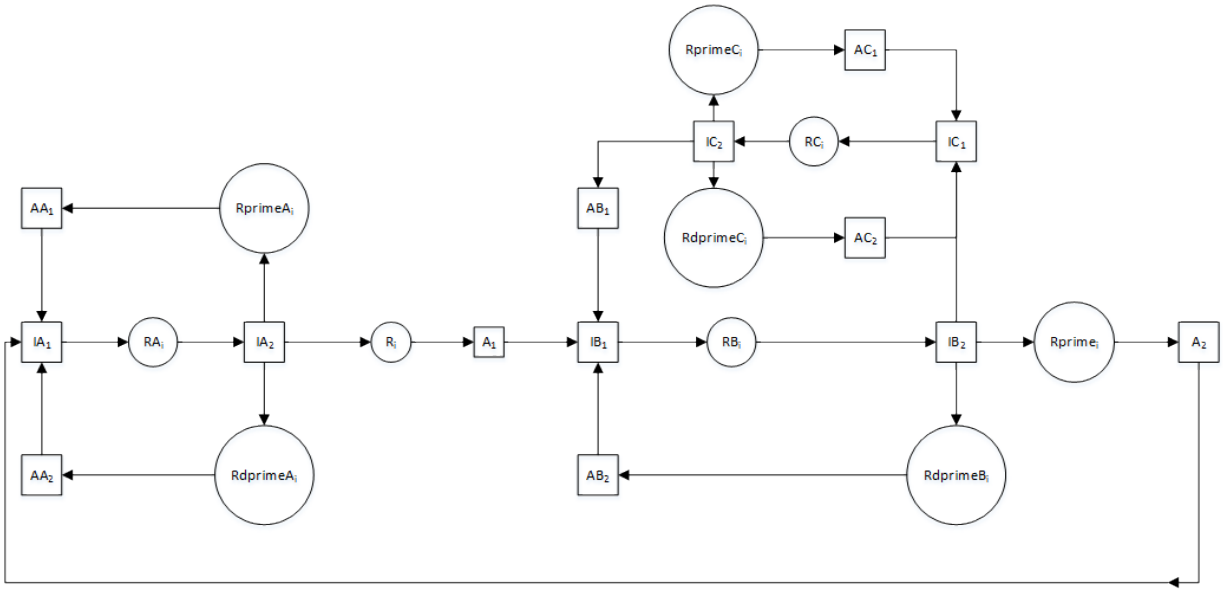


Figure 5.1: Parameterized-chain network graph

In this section we consider a more complex version of the train network example of [30]. The model consists of distinguished subprocesses that represent the intersections within

the network and of linear parameterized segments representing routes of arbitrary length. It would be seen that the existence of generalized circular waits will depend on the lengths of these routes.

Figure 5.1 is the PCDN graph and represents a network of six intersections and seven routes. A train will enter the network from the IA_1 "input node" and travel through the network using any of the specified routes. (Directions of the movements are indicated by the arrows) Each space in the network will get filled by the arrival of a train and will empty upon its leaving. IA_1, IB_1 and IC_1 are the input nodes and IA_2, IB_2 and IC_2 are the output nodes. Rx_i, $Rprimex_i$, $Rdprimex_i$ (x could be A, B and C, except $RprimeB_i$), R_i and $Rprime_i$ are the parameterized nodes. Distinguished nodes have been represented using square nodes in the graph. The traffic network represented in figure 5.1 could be arbitrarily large with many distinguished and parameterized nodes. For the purpose of this example the network contains seven routes, each represented by a parameterized node, which means their length could be arbitrarily large. Each intersection is also represented by a distinguished node. To be consistent with the simpler version of this network we will assume every train comprises two cars and will occupy two spaces at a time. In order to satisfy the assumptions underlying the reachability analysis of [30], trains are modelled as entering input nodes in a single event; they are also modelled as leaving the network in a single event. Otherwise, they pass through spaces on the routes one wagon at a time. Each intersection is blocked after the entrance of a train and will accept new trains only upon the departure of the first train. Suppose that a train enters the network from intersection IA_1, and then continues to the main route. Upon arrival at the next intersection, IA_2, it has three choices: it could go to the upper or lower route to come back to intersection IA_1, or it could continue to reach to the next intersections, IB_1 and IB_2. Again the train has three different choices. It could continue to the main route to return to the IA_1 intersection, or it could choose the lower or upper route. By choosing the lower route, it will come back directly to the IB_1 intersection, but by choosing the upper route it will enter the next intersection, IC_1, and again there is a choice to make, upper route or lower route, both of which will eventually reach the next intersections IC_2 and IB_1. We will instantiate the PCDN network using three subprocesses in each parameterized segment, which will allow us to analyse fully the parameterized network [30].

All of the input nodes including IA_1, IB_1 and IC_1 intersections, have a structure similar to figure 5.2, with a slight relabelling of event names (replacing 'a' in IA_1 to 'b' or 'c', to achieve IB_1 and IC_1 graphs respectively.) In intersection IA_1, the entrance of the train from outside of the network has been denoted by a local event named ia. The only difference among these input nodes is that IB_1 and IC_1 do not have the local events ib and ic, respectively because trains cannot enter the network from those intersections.

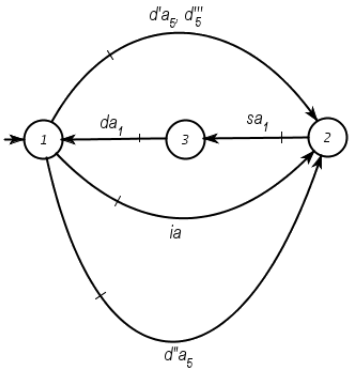


Figure 5.2: Input node

Their shared events with the previous space are denoted by d'''_5, d_5 and d''_5 in IA_1, IB_1 and IC_1, respectively. Shared events $d'a_5$, $d''a_5$ and d'''_5 represent entrance of the train from top, bottom and previous space of the main route, respectively. This intersection will be emptied when the first and second wagons of the train leave this intersection by sa_1 and da_1 events, respectively. The main route after the first input node has been represented by a parameterized node RA_i.

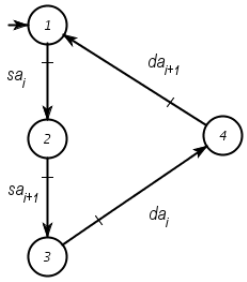
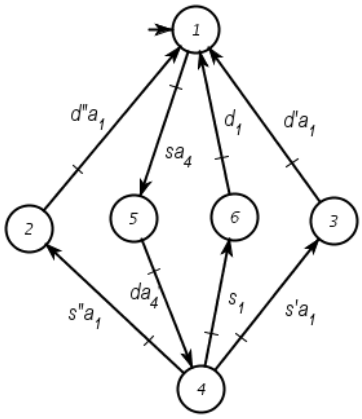


Figure 5.3: Parameterized node representing the main route

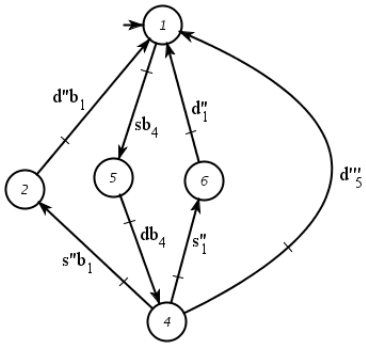
In figure 5.3 the first and second wagon will arrive at the i^{th} space by events sa_i and da_i and they will leave by events sa_{i+1} and da_{i+1} respectively. Other routes which have been modelled by a parameterized node have the same structure to this one and can be constructed by proper relabelling of the above model.

The intersection IC_2 has the same structure with a slight relabelling of the event names.

Figure 5.4a depicts the structure of the output nodes IA_2 and IC_2. The first and second wagon will enter this intersection via events sa_4 and da_4 from the main route. After that wagons can leave this intersection in three different ways; they could go to the lower route



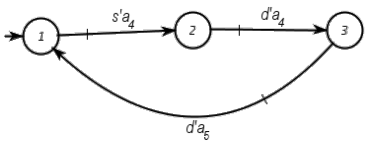
(a) IA_2 and IC_2



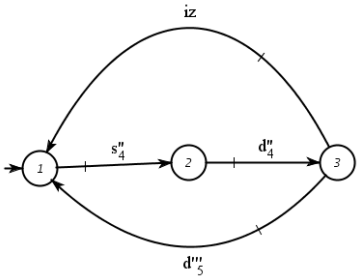
(b) IB_2

Figure 5.4: Output nodes

using $s''a_1$ and $d''a_1$ events or they could continue along the main route using s_1 and d_1 events or finally they could exit to the upper route using $s'a_1$ and $d'a_1$ events, which is the only difference between IA_2 and IC_2, on the oner hand, and IB_2 on the other hand, as depicted in figure 5.4b. Intersection IC_2 has the same structure as IA_2 because they are both connected to a parameterized node in their upper route, while IB_2 is connected to a distinguished node in its upper route, and input nodes can accept trains in just one event.



(a) AA_1, AA_2, AB_1, AB_2, AC_1, AC_2, A_1



(b) $A2$

Figure 5.5: Distinguished nodes

Figure 5.5 depicts the models of the other distinguished nodes in the PCDN network. All of the distinguished nodes depicted in figure 5.5a are the same except for a slight change in event names. This space gets filled by entrance of two wagons via $s'a_4$ and $d'a_4$ events

and become empty by the $d'a_5$ event. The only difference in the A_2 node is that it has one more local event for trains to leave the network, as depicted in 5.5b.

5.2 The Results

Figure 5.6 represents the dependency graph of the traffic network. This dependency graph contains fourteen different consistent and full subgraphs.

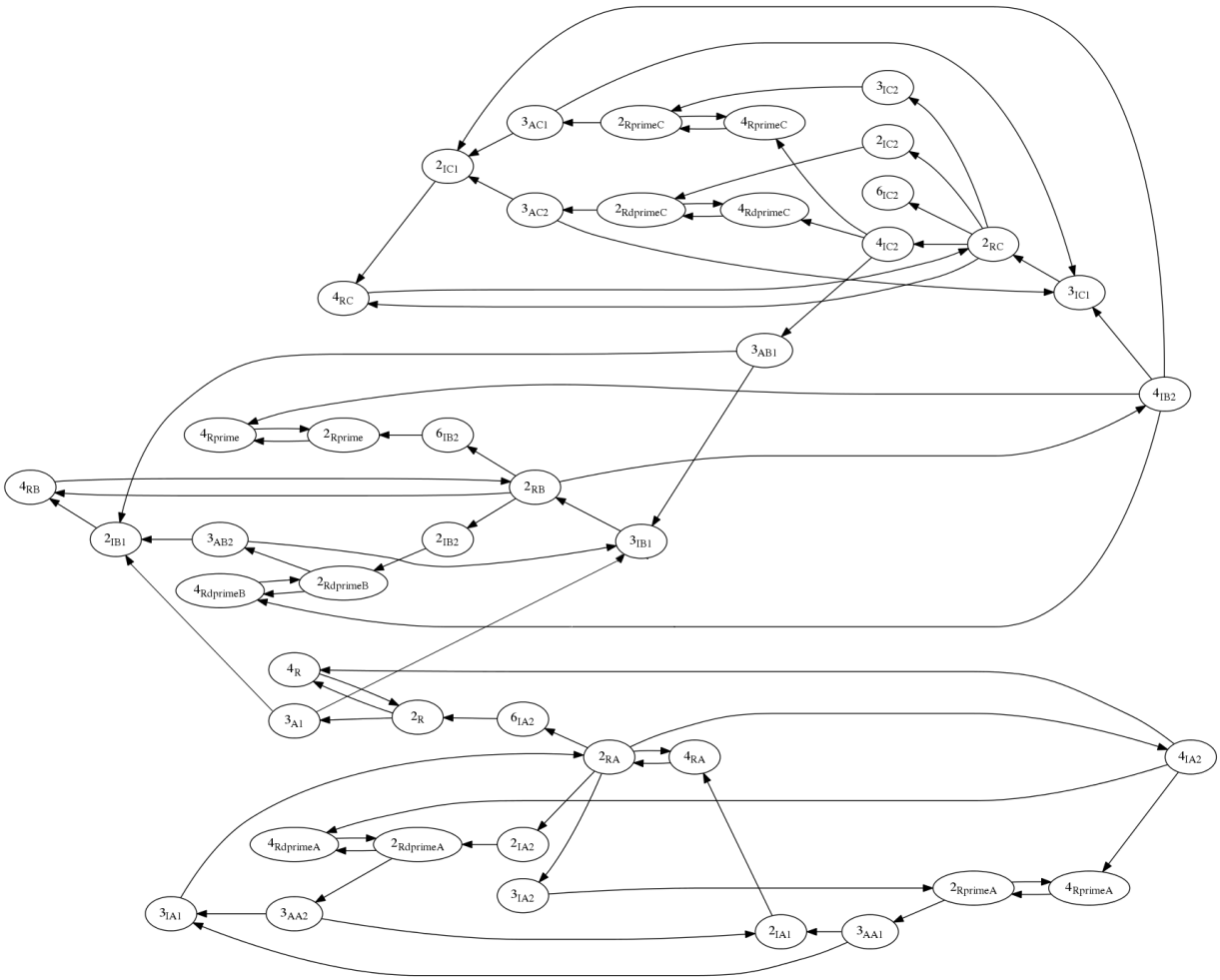
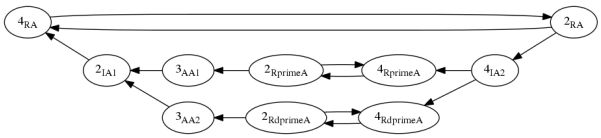


Figure 5.6: Dependency graph

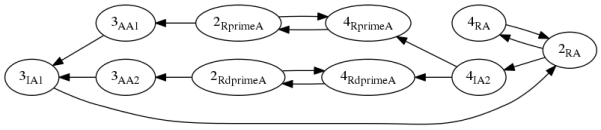
To satisfy the consistency condition, subgraphs must include an input node and also

must contain just one state of any distinguished node. For example, in figure 5.6, the loop between nodes $2_{RPrimeA}$ and $4_{RPrimeA}$ does not satisfy the first condition of the consistency condition, so it does not represent a circular wait. The states of the nodes are represented by state numbers in the dependency graph. For example, in 3_{IA_1} the state of IA_1 has been represented by 3, while in $4R$ the state of the parameterized node is 4. In figure 5.6, a consistent subgraph cannot contain both nodes 3_{IA_1} and 2_{IA_1} or nodes 3_{IC_1} and 2_{IC_1}, because a subprocess cannot be in two different states at the same time. To satisfy the fullness condition, for each output node in the subgraph, if it has any shared event with its neighbours in the PCDN graph in that specific state, there should be an edge to any state of that neighbour in the subgraph to satisfy the fullness condition.

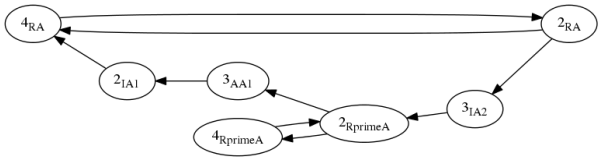
For the purposes of this example, a dangling loop is a cycle of nodes corresponding to parameterized processes that shares only a single node with the rest of the subgraph; any other cycle of such nodes is an embedded loop. Based on the number of 'embedded' and 'dangling loops' in each subgraph, we can divide fourteen subgraphs of the dependency graph to five different subgraphs as depicted in figure 5.7.



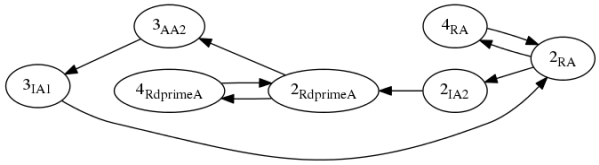
(a) Subgraph with three embedded loops



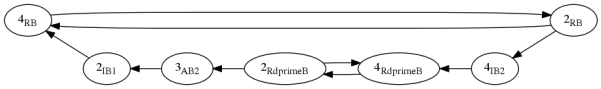
(b) Subgraph with two embedded loops and one dangling loop



(c) Subgraph with one embedded and one dangling loop



(d) Subgraph with two dangling loops



(e) Subgraph with two embedded loops

Figure 5.7: Five distinct full, consistent subgraphs of the dependency graph 5.6

All of the subgraphs of figure 5.7 satisfy the consistency conditions by having at least one input node and at most one state of any distinguished node. For example, the subgraph in figure 5.7c satisfies the consistency conditions by including an input node 2_{IA_1} and not

having more than one state of any distinguished node. It also satisfies the fullness condition by having an edge from the output node 3_{IA_2} to $2_{RprimeA}$. As shown in figure 5.7c, the 2_{RA} and 4_{RA} loop requires an even number of parameterized nodes while the loop containing the nodes $2_{RprimeA}$ and $4_{RprimeA}$ requires an odd number of parameterized nodes to reach a partial deadlock in the network. In other words, the partial deadlock can occur only if the number of spaces on the upper return route from IA_2 back toward IA_1 is odd.

Figure 5.7b represents another full, consistent subgraph of the dependency graph. This subgraph has just one input node, 3_{IA_1}, and does not have more than one state of any distinguished node. The output node 4_{IA_2} has shared events with its neighbours $RdprimeA$ and $RprimeA$ in the PCDN graph when it is in state 4. So in subgraph 5.7b, there are edges with those neighbours to satisfy the fullness condition. By the same reasoning, the loops between the nodes $2_{RprimeA}$ and $4_{RprimeA}$ and between the nodes $2_{RdprimeA}$ and $4_{RdprimeA}$ should be instantiated with an even number of parameterized nodes to reach a partial deadlock in the network. That is, the number of spaces on both routes from IA_2 towards IA_1 must be even. On the other hand, the loop between the nodes 2_{RA} and 4_{RA} requires an odd number of parameterized nodes for instantiation, and it shows the need of an odd number of spaces in the return path from IA_1 to IA_2 to reach partial deadlock in the network.

As depicted in the first four subfigures of 5.7b, the same loop may appear as a dangling loop in one full, consistent subgraph but an embedded loop in others, depending on the states of distinguished subprocesses. Hence, the parity of the length of a route that is required for partial deadlock depends on the states of distinguished subprocesses.

We altered some of the configurations of the automaton models to see how the changes would affect the results. First, there is no difference in the results when the trains can enter the network via any of the input nodes rather than just via one input node (IA_1). In the current configuration trains can exit the network from A_2, so there is no A_2 in the dependency graph, nor any subgraph. If we let the trains leave the network also via node AB_1, in the dependency graph and subgraphs there would be no state of A_2 or AB_1, as expected.

Figure 5.8 shows the dependency graph of this new configuration. This new PCDN network has sixteen different maximal, full, consistent subgraphs, and we can divide them into the five distinct categories, shown in figure 5.7.

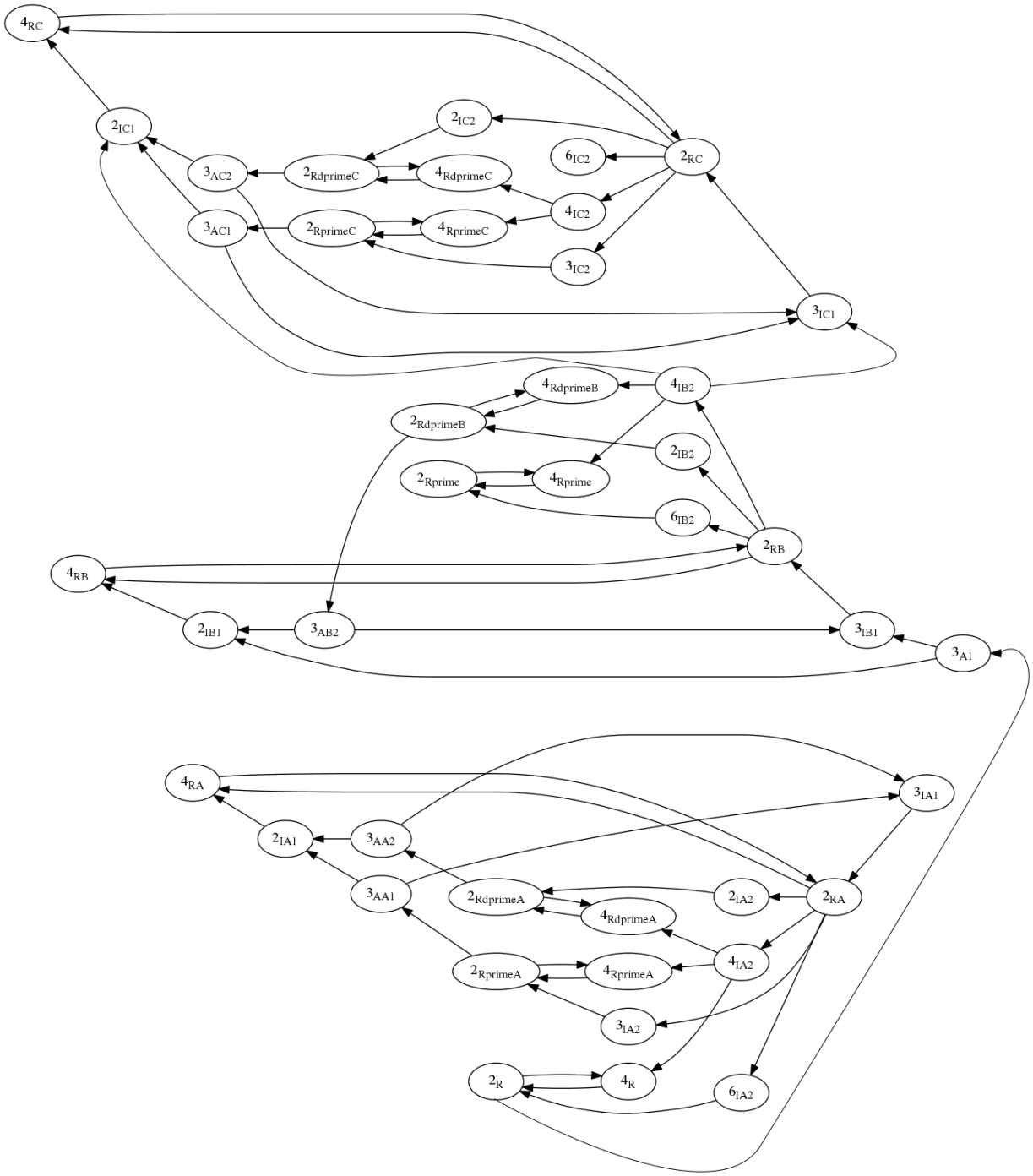


Figure 5.8: Dependency graph

Chapter 6

Concluding Remarks and Future Work

This thesis describes a tool used to implement a decision procedure for the existence of reachable generalized circular waits in the class of parameterized networks called Parameterized-Chain DES networks. Such a network consists of a fixed set of distinguished subprocesses linked by parameterized subnetworks of linear topology. By the nature of parameterized networks, there may be infinitely many such generalized circular waits. In the framework of [30, 32], all the generalized circular waits are represented by a finite set of full, consistent subgraphs of the dependency graph of a network; indeed, each full, consistent subgraph can be interpreted as a regular language, each word of which encodes a generalized circular wait. This thesis goes beyond the framework of [30, 32] by allowing networks to include multiple input processes.

In the present thesis, a consistent subgraph of the dependency graph must contain as one of its nodes a state of an input subprocess. The tool identifies such subgraphs by first finding all cycles within the dependency graph, and representing such cycles as nodes of a "supergraph". Nodes of the supergraph are connected by an edge if the corresponding cycles intersect. Maximal consistent subgraphs are then computed by starting with cycles that include states of input processes, and then by repeatedly merging other cycles with these, as long as the resulting subgraphs do not contain multiple states of any single distinguished subprocess. The tool then eliminates any maximal consistent subgraphs that fail to satisfy the fullness property, yielding the set of all maximal full, consistent subgraphs.

Because of the requirement of consistency, our problem is a subproblem of the forbidden pairs problem which is NP-complete, and indeed the complexity of the algorithm is

exponential in the number of distinguished subprocesses. Suppose that we have n distinguished subprocesses with m different states for each one of them, in this case the number of possibilities for the consistency condition is m^n; but in reality it is lesser than this and the dependency graph will determine the exact number of states for each distinguished subprocess. By the NP-completeness of the forbidden pairs problem it is unlikely that a subexponential algorithm exists. On the other hand, the complexity of our algorithm is of course independent of the number of parameterized subprocesses. In this sense the approach of this thesis avoids the problem of combinatorial explosion.

The full, consistent subgraphs that the algorithm computes represent counterexamples to deadlock-freedom of the network. As we showed in the thesis, the tool makes it easy to study the effect of modifications of the network on its set of deadlocks. A potential topic for research is to automate the search for minimal modifications of the network that eliminate deadlock.

References

[1] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions? In *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234. Springer Berlin / Heidelberg, 2001.

[2] Paul C. Attie and E. Allen Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20:51–115, January 1998.

[3] Hans Bherer, Jules Desharnais, and Richard St-Denis. Control of parameterized discrete event systems. *Discrete Event Dynamic Systems*, 19:213–265, 2009.

[4] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983.

[5] Gary Chartrand, Linda Lesniak, and Ping Zhang. *Graphs & digraphs*. CRC Press, 2010.

[6] E. Emerson and Vineet Kahlon. Model checking large-scale and parameterized resource allocation systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 55–69. Springer Berlin / Heidelberg, 2002.

[7] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *In 17th International Conference on Automated Deduction*, pages 236–255, 2000.

[8] E. Allen Emerson and Vineet Kahlon. Model checking large-scale and parameterized resource allocation systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 251–265, London, UK, 2002. Springer-Verlag.

[9] E. Allen Emerson and Vineet Kahlon. Parameterized model checking of ring-based message passing systems. In *In Proc. of CS04, volume 3210 of LNCS*. Springer, 2004.

[10] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 85–94, New York, NY, USA, 1995. ACM.

[11] M.P. Fanti and MengChu Zhou. Deadlock control methods in automated manufacturing systems. *Systems, Man and Cybernetics, Part A: IEEE Transactions on Systems and Humans*, 34(1):5–22, 2004.

[12] Eli Fox-Epstein. *Forbidden Pairs Make Problems Hard*. Bachelor Thesis, 2011.

[13] Petr Kolman and Ondej Pangrc. On the complexity of paths avoiding forbidden pairs. *Discrete Applied Mathematics*, 157(13):2871 – 2876, 2009.

[14] Sava Krstic. Parametrized system verification with guard strengthening and parameter abstraction. In *Proceedings of 4th Workshop on Automated Verification of Infinite-State Systems*, AVIS '05, 2005.

[15] Yongjian Li. Mechanized proofs for the parameter abstraction and guard strengthening principle in parameterized verification of cache coherence protocols. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1534–1535, New York, NY, USA, 2007. ACM.

[16] F. Lin and W. M. Wonham. Decentralized supervisory control of discrete-event systems. *Information Sciences*, 44(3):199–224, 1988.

[17] Charles E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, 6(3):515–536, 1989.

[18] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[19] Kedar S. Namjoshi and Richard J. Trefler. Analysis of dynamic process networks. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 164–178, 2015.

[20] Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 589–606, 2016.

[21] S. Nazari and J.G. Thistle. Blocking in fully connected networks of arbitrary size. *IEEE Transactions on Automatic Control*, 57(5):1233 –1242, May 2012.

[22] Siamak Nazari. *Analysis of Parameterized Networks*. PhD Thesis, Electrical and Computer Engineering, University of Waterloo, 2008.

[23] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2011.

[24] Antti Siirtola and Juha Kortelainen. Algorithmic verification with multiple and nested parameters. In *Formal Methods and Software Engineering*, volume 5885 of *Lecture Notes in Computer Science*, pages 561–580. Springer Berlin / Heidelberg, 2009.

[25] J.G. Thistle. Supervisory control of discrete event systems. *Mathematical and Computer Modelling*, 23(1112):25 – 53, 1996.

[26] N. Viswanadham, Y. Narahari, and T.L. Johnson. Deadlock prevention and deadlock avoidance in flexible manufacturing systems using petri net models. *Robotics and Automation, IEEE Transactions on*, 6(6):713 –723, 1990.

[27] W. M. Wonham. Lecture notes on supervisory control of discrete event systems, 2012. Available at http://www.control.utoronto.ca/cgi-bin/dldes.cgi, [Jan. 30, 2013].

[28] Hananya Yinnone. On paths avoiding forbidden pairs of vertices in a graph. *Discrete Applied Mathematics*, 74(1):85 – 92, 1997.

[29] M.H. Zibaeenejad. Weak invariant simulation and analysis of parameterized networks. November 2014. http://hdl.handle.net/10012/8982.

[30] M.H. Zibaeenejad and J.G. Thistle. A framework for analysis of generalized parameterized networks. Submitted for journal publication.

[31] M.H. Zibaeenejad and J.G. Thistle. Weak invariant simulation and its application to analysis of parameterized networks. *Automatic Control, IEEE Transactions on*, 59(8):2024–2037, Aug 2014.

[32] M.H. Zibaeenejad and J.G. Thistle. Dependency graph: An algorithm for analysis of generalized parameterized networks. In *American Control Conference (ACC), 2015*, pages 696–702, July 2015.