# Structural Decomposition of STGs – an Approach for Modular Asynchronous Circuit Design

Dissertation

by

Benedictus Benyamin Kangsah

1. Tutor : Prof. Dr.-Ing. Jochen Beister
2. Tutor : Prof. Dr. Walter Vogler
Chairman of examination board : Prof. Dr.-Ing. Wolfgang Kunz

# Strukturelle Dekomposition von STGs – ein modularer Ansatz zum Entwurf ungetakteter Schaltwerke

**vom**

**Fachbereich Elektrotechnik und Informationstechnik**
**der Technischen Universität Kaiserslautern**
**zur Verleihung des akademischen Grades**
**Doktor der Ingenieurwissenschaften (Dr.-Ing.)**

**genehmigte Dissertation**

**von**

**Benedictus Benyamin Kangsah M.Sc.**

**geb. Jakarta (Indonesien)**

**D 386**

1. Betreuer : Prof. Dr.-Ing. Jochen Beister
2. Betreuer : Prof. Dr. Walter Vogler
Vorsitzender der Prüfungskommission : Prof. Dr.-Ing. Wolfgang Kunz
Dekan des Fachbereichs : Prof. Dr.-Ing. Hans D. Schotten
Tag der mündlichen Prüfung : 07. November 2014

# Kurze Zusammenfassung

Mit immer komplexeren asynchronen (ungetakteten) Schaltwerken – als Teilen heutiger Systems-On-a-Chip – hat auch die Größe und Komplexität ihrer Verhaltensspezifikationen mit Signalflankengraphen ("Signal Transition Graphs", STGs) zugenommen. Dieses erschwert und macht es manchmal unmöglich, die Schaltung mit dem von Signalflankengraph spezifizierten Verhalten mit einem Werkzeug wie *petrify* [CKK+96] oder *CASCADE* [BEW00] zu synthetisieren.

In der vorliegenden Arbeit wird daher ein Verfahren vorgeschlagen, welches zuerst den STG zerlegt, was zu einer modularen Umsetzung [KWVB03] [KVWB05] führt. Damit kann der Syntheseaufwand verringert werden, da eine Explosion des Zustandsraumes ("state explosion") vermieden werden kann und Wiederverwendbarkeit von Bibliothekselementen gegeben ist. Ein Ansatz für die Zerlegung eines Signalflankengraphen wird in [VW02] [KKT93] [Chu87a] vorgestellt. Der Zerlegungsalgorithmus von Vogler und Wollowski [VW02] basiert auf dem Algorithmus von Chu [Chu87a], ist aber allgemeiner anwendbar als die Algorithmen in [KKT93] [Chu87a], deren formale Korrektheit in [VW02] bewiesen wurde.

Nach der Einführung (Kapitel 1) werden in Kapitel 2 zunächst die Platz/Transitions-Netze (P/T-Netze) vorgestellt. Ihre zur Verhaltensbeschreibung nebenläufiger dynamischer Systeme erforderlichen Eigenschaften, vor allem Lebendigkeit und Beschränktheit, werden erörtert, ebenso die Besonderheiten der Signalflankengraphen als Unterklasse der P/T-Netze. Es folgen die Prinzipien der Zerlegung in Komponenten und deren Parallelkomposition auf STG- und Schaltungsebene.

Eine Motivation für die Zerlegung eines P/T-Netzes ergibt sich schon bei der Analyse, wenn z.B. herausgefunden werden soll, ob es lebendig ist oder nicht. Dazu müsste man alle erreichbaren Markierungen ermitteln. Bei einem großen, hoch nebenläufigen Netz könnte deren Anzahl so groß werden ("state explosion"), dass sie nicht mehr zu handhaben sind.

Aber auch wenn eine Schaltung ausgehend von ihrem Signalflankengraphen synthetisiert werden soll, kann die notwendige Bestimmung aller erreichbaren

Markierungen zur nicht mehr handhabbaren "state explosion" führen. Viele Synthesealgorithmen haben exponentielle Komplexität bei der Ableitung der erreichbaren Zustände. Man kann zwar versuchen, dem "state explosion"-Problem mit einem heuristischen Algorithmus mit polynomialer Komplexität beizukommen. Aber wenn das Problem groß ist, erhält man ein nichtoptimales Ergebnis. Daher wird die Zerlegung des P/T-Netzes zum Lösen der oben genannten Probleme verwendet, in der Hoffnung, dass Gesamtaufwand und Kosten für die Synthese der Komponenten deutlich kleiner werden als für die Behandlung des großen Netzes. Eines dieser Verfahren ist das Signalflanken-Zerlegungsverfahren von [VW02]. Es wird in Kapitel 3.2 beschrieben.

Das Zerlegungsverfahren aus [VW02] kann noch weiter verbessert werden, um damit Signalflankengraphen für reale Anwendungen synthetisierbar zu machen und bessere Zerlegungsergebnisse zu bekommen. Einige Verbesserungsvorschläge für [VW02] werden in Kapitel 4 behandelt. Diese Verbesserungen werden in [KVWB04] vorgeschlagen, einige von ihnen werden formal in [VK04] bewiesen. Mit diesen Verbesserungen können nicht nur Signalflankengraphen von realen Anwendungen zerlegt werden, sondern es können auch bessere Zerlegungsergebnisse erhalten werden. Dummy-Transitionen und strukturelle Auto-Konflikte sind dabei erlaubt. Durch die Übersetzung von Hardware-Beschreibungssprachen in Signalflankengraphen [BL00] werden oft Dummy-Transitionen eingeführt. Strukturelle Auto-Konflikte mit Steuerplätzen treten in einer nicht deterministischen Spezifikation (z.B. die Spezifiation eines VME-Bus-Controller) oder in einer Arbiter-Spezifikation [Wol97] [YKKL94] häufig auf. Das Problem mit strukturellen Auto-Konflikten wird durch Einführung von Transition-Fusion gelöst. Probleme mit nicht-sicheren Dummy-Transitionen werden durch die Umwandlung in sichere Dummy-Transitionen gelöst. Diese und das Entfernen der schleifen-behafteten Dummy-Transitionen verringern die Häufigkeit von Backtracking in dem Algorithmus, wodurch man bessere Zerlegungsergebnisse bekommen kann. Die Effizienz des Algorithmus wird auch durch Kontraktion von global irrelevanten Signalen vor der Zerlegung und durch die Neuordnung der zu kontrahierenden Transitionen erhöht.

Die Zerlegungsverfahren aus [VW02] basieren auf Netzreduktion zum Finden einer Ausgangsblock-Komponente. Es ist sehr arbeitsintensiv, eine initiale Spezifikation zu reduzieren, bis die End-Komponente gefunden wird. Diese Reduktion ist nicht immer möglich, was dazu führt, dass weitere irrelevante Eingangsvariablen zu den relevanten Eingangsvariablen der Komponente hinzugefügt werden müssen. Dies führt zu einer unnötig großen Spezifikationen, was auch zu einer unnötig groß implementierten Schaltungen führt. Statt dieser Reduktion wird in Kapitel 5 dieser Dissertation ein neuer Ansatz präsentiert, indem das ursprüngliche Netz zuerst in strukturelle Komponenten – stark zusammenhängende Zustandsmaschinen ("strongly connected state machines", SCSMs) – zerlegt wird. Eine initiale Ausgangsblock-Komponente ist durch die Zusammensetzung der strukturellen Komponenten herauszufinden. Danach bekommt man eine endgülti-

ge Ausgangsblock-Komponente durch Netz-Reduktion. Durch die Nutzung dieses Ansatzes wird die Begrenzung der Netzreduktionsoperation überwunden, was zu kleineren End-Komponenten als in [VW02] führt. Zusätzlich ist diese Methode in der Praxis einfach anzuwenden.

Da wir meistens mit der Struktur eines Netzes beschäftigt sind, ist es sinnvoll, eine strukturelle Abstraktion des Netzes vorzunehmen. In Kapitel 6 dieser Arbeit wird ein struktureller Abstraktionsalgorithmus [Kan03] vorgestellt. Ein SCSM Subnetz ist in den meisten Fällen mit dem Strukturgraphen effizienter zu finden, als das Platz-Transition- (P/T-) Netz direkt zu traversieren. Der Strukturgraph eines gewöhnlichen P/T-Netzes kann nicht nur für die Suche nach SCSM-Subnetzen oder zur Kontraktion von Transitionen in der Mitte des Knotens verwendet werden, sondern auch für andere Algorithmen, welche die Knoten eines P/T-Netzes traversieren. Daher wird vorgeschlagen, einen Strukturgraphen als eine abstrakte graphische Datenstruktur für ein P/T-Netz bei der Umsetzung solcher Algorithmen zu verwenden. Einige Anwendungen von Strukturgraphen und deren experimentelle Ergebnisse können in [War05] und [Taw04] gefunden werden.

Schließlich diskutiert Kapitel 7 die Anwendung des STG-Dekompositionsalgorithmen im ungetakteten Schaltungsentwurf. "Speed independent"- (SI-) Schaltungen werden zuerst diskutiert. Danach werden 3D-Schaltungen, die aus einem erweiterten Burst-Mode-Spezifikationen ("extended burst mode", XBM) synthetisiert werden, diskutiert. Der [VW02]-Algorithmus ist gut für die Umsetzung in SI-Schaltungen, da eine solche Umsetzung das Netz nicht zu stark reduziert. Eine zu starke Reduktion könnte Eingangssignale entfernen, die wichtig sind, um ein kompletten Zustandskodierung zu erreichen. Es kann also dazu führen, dass keine komplette Zustandskodierung gefunden werden kann. Stattdessen ist der SCSM-Subnetz-basierte Algorithmus gut für die Umsetzung in 3D-Schaltungen, da dieser kleinere Komponenten liefert und weniger Probleme mit Schleifen-behafteten Transitionen hat, die oft in XBM-Spezifikationen aufgrund von Pegel-Transitionen vorhanden sind.

Ein Algorithmus zum Übersetzen einer Signalflankengraph-Spezifikation in eine XBM-Spezifikation wurde zuerst in [BEW99] vorgeschlagen. Dieser Algorithmus leitet von der Signalflankengraph-Spezifikation einen endlichen Automaten ab und übersetzt dann die Automaten in eine XBM-Spezifikation. Obwohl die XBM-Spezifikation einen Automaten darstellt, lässt sie gewisse Nebenläufigkeiten zu. Diese können direkt übersetzt werden. Ein Algorithmus, der direkt eine Signalflankengraph-Spezifikation in eine XBM-Spezifikation übersetzt, wird in Kapitel 7.3.1 vorgestellt. Dieser Algorithmus verbessert die Effizienz des in [BEW99] vorgeschlagenen Verfahrens. Allerdings können nicht alle Signalflankengraph-Spezifikationen in eine XBM-Spezifikation übersetzt werden, da die XBM-Spezifikation nur eine Teilmenge der im Signalflankengraph möglichen Nebenläufigkeiten erlaubt. Die Zerlegung des Signalflankengraphen kann verwendet werden, um eine Signalflankengraph-Spezifikation in eine XBM-Spezifikation übersetzbar

zu machen. Zerlegung löst aber nicht alle Probleme, da auch nach der Zerlegung einige Komponenten nicht in XBM-übersetzbar sind.

Am Ende werden mit DESI, ein Werkzeug zum Zerlegen von Signalflankengraphen und dessen Zerlegungsergebnisse vorgestellt. Es wird in Kapitel 7.4.1 gezeigt, dass mit DESI zerlegte Signalflankengraphen, welche eine große Anzahl von erreichbaren Markierungen aufweisen, effektiv zerlegt werden können. Mit Werkzeugen wie Petrify ist eine solche effektive Synthese nicht möglich, wenn die Anzahl der erreichbaren Markierung groß ist, da die Schaltung mehr Platz auf dem Chip benötigt wird. Es werden auch mehr Rechenressoucen und Zeit benötigt. Dies machte es unmöglich die Signalflankengraph-Spezifikation einen 7-stufigen FIFO-Speichersteuerung zu synthetisieren. Solche Probleme mit Rechenressourcen und Zeit gibt es bei der Zerlegung des Signalflankengraphen nicht. Die daraus resultierende Fläche ist nicht nur kleiner, sondern für eine kleine Anzahl von erreichbaren Markierungen, wie beispielsweise beim NEI-Arbiter kann man auch von der Zerlegung des Signalflankengraphen profitieren, indem man Bibliothekselemente wie z.B. ME-Elemente aus der Spezifikation extrahiert.

*to my parent*

*Johannes Sulaiman Kangsah and Maria Natalia Goretti*

# Acknowledgments

I would like to thanks Prof. Jochen Beister, my doctor father, who gives me advices, comments and corrections for my work, also he gives me the infrastructure and financial support to do research in his chair. Ralf Wollowski and Prof. Walter Vogler, who give me the chance to do research in the STG-decomposition project with them. Roland Hecker, who gives me financial and spiritual support at the hard time of my study. Prof. Wolfgang Kunz, who makes my promotion happen. My colleague, Karsten Laux, Meinrad Fiedler and Peter Kosack, who give feedbacks about my research. Peter Tawdross, Surya Warman, and Zhong Wei Li, for testing and implementing part of the proposed algorithm in this dissertation. Roland Hecker, Max Thalmeier and Stephan Herzog, who give me continuous motivation and spiritual support until the end phase of this dissertation. Also for Indonesian friends and neighbours, Ivan Solihin, Bahter and Maureen Bukit who help me a lot during the end phase. Dekanat Mr Hauck and Guthail for the promotion formalities. All the member of Stammtisch "der Freunde des Entwurfs programmierter Systeme" and the colleagues of Fa. Wipotec who cheering me up until the end phase of my promotion. For my wife, Ninasari, and the children, Hosea, Micha and Stella for their patience and support.

# Contents

# Chapter 1

# Introduction

Specification of asynchronous circuit behaviour becomes more complex as the complexity of today's System-On-a-Chip (SOC) design increases. This also causes the Signal Transition Graphs (STGs) – interpreted Petri nets for the specification of asynchronous circuit behaviour – to become bigger and more complex, which makes it more difficult, sometimes even impossible, to synthesize an asynchronous circuit from an STG with a tool like *petrify* [CKK+96] or *CASCADE* [BEW00].

It has, therefore, been suggested to decompose the STG as a first step; this leads to a modular implementation [KWVB03] [KVWB05], which can reduce synthesis effort by possibly avoiding state explosion or by allowing the use of library elements. A decomposition approach for STGs was presented in [VW02] [KKT93] [Chu87a]. The decomposition algorithm by Vogler and Wollowski [VW02] is based on that of Chu [Chu87a] but is much more generally applicable than the one in [KKT93] [Chu87a], and its correctness has been proved formally in [VW02].

This dissertation begins with Petri net background described in chapter 2. It starts with a class of Petri nets called a place/transition (P/T) nets. Then STGs, the subclass of P/T nets, is viewed. Background in net decomposition is presented in chapter 3. It begins with the structural decomposition of P/T nets for analysis purposes – liveness and boundedness of the net. Then STG decomposition for synthesis from [VW02] is described.

The decomposition method from [VW02] still could be improved to deal with STGs from real applications and to give better decomposition results. Some improvements for [VW02] to improve decomposition result and increase algorithm efficiency are discussed in chapter 4. These improvement ideas are suggested in [KVWB04] and some of them are have been proved formally in [VK04].

The decomposition method from [VW02] is based on net reduction to find an output block component. A large amount of work has to be done to reduce an initial specification until the final component is found. This reduction is not always possible, which causes input initially classified as irrelevant to become relevant input for the component. But under certain conditions (e.g. if structural

auto-conflicts turn out to be non-dynamic) some of them could be reclassified as irrelevant. If this is not done, the specifications become unnecessarily large, which intern leads to unnecessarily large implemented circuits. Instead of reduction, a new approach, presented in chapter 5, decomposes the original net into structural components first. An initial output block component is found by composing the structural components. Then, a final output block component is obtained by net reduction.

As we cope with the structure of a net most of the time, it would be useful to have a structural abstraction of the net. A structural abstraction algorithm [Kan03] is presented in chapter 6. It can improve the performance in finding an output block component in most of the cases [War05] [Taw04]. Also, the structure net is in most cases smaller than the net itself. This increases the efficiency of the decomposition algorithm because it allows the transitions contained in a node of the structure graph to be contracted at the same time if the structure graph is used as internal representation of the net.

Chapter 7 discusses the application of STG decomposition in asynchronous circuit design. Application to speed independent circuits is discussed first. After that 3D circuits synthesized from extended burst mode (XBM) specifications are discussed. An algorithm for translating STG specifications to XBM specifications was first suggested by [BEW99]. This algorithm first derives the state machine from the STG specification, then translates the state machine to XBM specification. An XBM specification, though it is a state machine, allows some concurrency. These concurrencies can be translated directly, without deriving all of the possible states. An algorithm which directly translates STG to XBM specifications, is presented in chapter 7.3.1. Finally DESI, a tool to decompose STGs and its decomposition results are presented.

# Chapter 2

# Petri Net Background

Petri nets have captured a large amount of interest since they were introduced by C.A.Petri [Pet66]. Many researchers have spent their effort in the theoretical and practical use of petri nets. Its graphical representation makes it easier for humans to capture the information it represents than text based representations do; petri nets are used in practice for modelling; e.g. to model the behaviour, especially concurrent behaviour, of dynamic systems. Not only that, they are a precise formal mathematical notation, which is a great advantage. Because of its formality, the model can be analyzed to determine, whether it has the properties needed for synthesis. If the model can be synthesized, then it can be implemented; e.g. as a circuit. Being a precise and compact formal model of concurrent behaviour, Petri nets have become a much used way of specifying the complex behaviour of systems which consist of components that work concurrently.

A petri net is a bipartite directed graph. A bipartite graph is a graph with two kinds of nodes, such that no arc connects two nodes of the same kind. In a directed graph, the arc connecting two nodes has direction. In petri nets, a *node* is either a place or a transition. In the graph representation, places are represented as circles and transitions as boxes. The places can be marked by tokens.

## 2.1 Place/Transition Nets

In this dissertation, a class of petri nets called a place/transition (P/T) nets is used. Unlike elementary nets, where places can only have Boolean markings (token/no token), or coloured nets, where tokens are distinguished by their so-called colours, the places of P/T nets can be marked by an integer number ($\in \mathbb{N}_0$) of otherwise indistinguishable tokens [BC92]. The dynamics are modelled by the flow of tokens along the directed arcs to and from firing transitions. The number of token flowing over an arc will be called its *weight*.

**2.1.1.** DEFINITION. A place/transition net $N$ is a tuple $(P, T, F, M_0, W)$ where
$P$ is the set of places,
$T$ is the set of transitions, ($P$ and $T$ are disjoint)
$F \subseteq P \times T \cup T \times P$ is the set of flow relation represented by directed arcs,
$M_0 : P \to \mathbb{N}_0$ is the initial marking,
$C : P \to \mathbb{N}$ is the place capacity,
and $W : P \times T \cup T \times P \to \mathbb{N}_0$ assigns weight to every pair $(p, t)$ and $(t, p)$. The weight denotes the number of tokens that must flow from $p$ to $t$, repsectively from $t$ to $p$, whenever transition $t$ is fired. If $p$ and $t$ are connected by a directed arc, i.e. if $(p, t)$ or $(t, p)$ belong to $F$, a weight $> 0$ is assigned to the arc. If no arc connects $p$ and $t$, i.e. if neither $(p, t)$ nor $(t, p)$ belongs to $F$, then $W$ assigns weight 0 to both pairs.

Sometimes when applying a structural operation to $N$ which only considers its initial marking, $M_0$ of $N$ is shortly denoted as $M_N$.

Because only P/T nets are discussed here, P/T nets will be referred to only as nets later in this dissertation. If a net $N$ or $N'$, etc. is introduced, it is assumed that implicitly this introduces its components $(P, T, F, M_0, W)$ or $(P', T', F', M_0', W')$, etc. For each node $x \in P \cup T$ the *preset* of $x$ is ${}^\bullet x = \{y \mid (y, x) \in F\}$ and the *postset* of $x$ is $x^\bullet = \{y \mid (x, y) \in F\}$.

By the *weak* firing rule, a transition $t$ is *enabled* under a marking $M (M : P \to \mathbb{N}_0)$, denoted by $M[t\rangle$, if $\forall p \in {}^\bullet t : W(p, t) \leq M(p)$. If $M[t\rangle M'$, then we say $t$ can be *fired* under $M$, yielding the follower marking $M'$, in which $\forall p \in P : M'(p) = M(p) + W(t, p) - W(p, t)$. In contrast to the *strong* firing rule, the post places of $t$ are not checked for, whether they have enough capacity to receive the tokens. Because of this, there is no need to consider the capacity of the places in the weak firing rule, and it is assumed that the capacity of the places is infinite. Because only the weak firing rule is used in this dissertation, when a transition is mentioned as firing, this will mean it fires under the weak rule.

If a finite sequence of firing transitions $w \in T^*$ ($T^*$ is a set of firing sequences) is enabled under a marking $M$, denoted by $M[w\rangle$, and yields the follower marking $M'$ when occurring, this will be denoted by $M[w\rangle M'$. A marking $M$ is called *reachable* if $\exists w \in T^*$ such that $M_0[w\rangle M$. The set of markings reachable from $M_0$ is denoted as $R_{M_0}$. It is represented graphically by a *reachability graph*.

A step is a set consisting of a single transition or of several concurrent and simultaneously firing transitions. If steps of more than one transition are to be considered, the reachability graph must be extended to become a step graph [Bei00]. A *state graph* with its *reachable states* is derived from the reachability graph or step graph for synthesis purposes.

**2.1.2.** DEFINITION. Let $N$ be a P/T net.

- $p \in P$ is $n$-bounded ($n \in \mathbb{N}$) iff for all reachable markings $M \in R_{M_0}$, $M(p) \leq n$.

- $N$ is *n-bounded* iff for all places $p \in P$, $p$ is $n$-bounded.
- $N$ is *safe* iff $N$ is 1-bounded.

**2.1.3.** Definition. Let $N$ be a P/T net.

- $t \in T$ is live in $M_0$ iff $\forall M \in R_{M_0}, \exists M[w\rangle M'$ such that $t$ is enabled under $M'$.
- $N$ is *live* iff for all $t \in T$, $t$ is live in $M_0$.

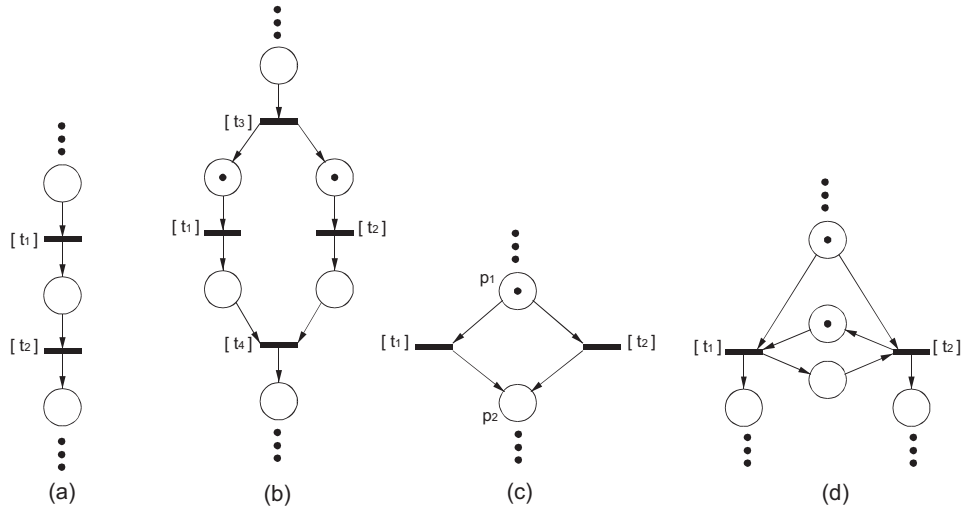


Figure 2.1. Basic relations in petri nets: (a) causal dependence (b) $t_1, t_2$ are concurrent (c) $t_1, t_2$ are in a structural conflict which is also a dynamic conflict (d) $t_1, t_2$ are in a structural conflict which is not dynamic

A transition is called a *fork* transition iff $|t^\bullet| > 1$ and it is called a *join* transition iff $|^\bullet t| > 1$; e.g. $t_3$ is a fork transition and $t_4$ is a join transition in Fig. 2.1b. A place is called a *choice* place iff $|p^\bullet| > 1$, and it is called a *meeting* place iff $|^\bullet p| > 1$; e.g. $p_1$ is a choice place and $p_2$ is a meeting place in Fig. 2.1c.

The terms fork, join, choice, and meeting, characterize not only the structure of the net. They also have a great influence on the dynamics of the net. A fork transition places tokens on each of its post places after firing. A join transition, instead, removes tokens from all of its pre places before firing. In contrast to a transition that can produce or consume tokens, places are passive. They can only distribute or gather tokens. A choice place distributes the tokens it has to one of its post transitions. A meeting place, instead, gathers tokens from its pre transitions.

Note that a transition (place) can be both a fork and a join transition (a choice and a meeting place). Hence, a transition can be classified either as a simple

transition or a fork transition or a join transition, or a fork and join transition. The same with the place: a place can be classified either as a simple place or a choice place or a meeting place or a choice and meeting place. This place and transition classification is important later when net abstraction is discussed.

There are three kinds of basic relations that we can represent by petri nets, namely:

1. The *dependence* relation: Transition $t_2$ depends on transition $t_1$ if ${t_1}^\bullet \cap {}^\bullet t_2 \neq \emptyset$ (see Fig. 2.1a). Firing $t_1$ will *give concession* to $t_2$.

2. The *independence* (*concurrency*) relation: Transition $t_1$ and $t_2$ are concurrent or (mutually) independent if there are enough tokens to enable $t_1$ and $t_2$ under a marking M (see Fig. 2.1b). Note that in Fig. 2.1c, if we add a token to $p_1$, we will have $t_1$ concurrent to $t_2$.

3. the *exclusion (conflict)* relation: Transitions $t_1$ and $t_2$ are in *structural conflict* if there is a choice place $p \in P, p \in {}^\bullet t_1 \cap {}^\bullet t_2$. They could also be in *dynamic conflict* if there is a reachable marking $M$ which enables $t_1$ and $t_2$, and firing $t_1$ will disable $t_2$ or vice versa (*deconcession*). In Fig. 2.1c, $t_1$ and $t_2$ are in structural and dynamic conflict; but in Fig. 2.1d, $t_1$ and $t_2$ are in structural but not dynamic conflict.

A net $N$ is an *ordinary* net, if $\forall (x, y) \in F : W((x, y)) = 1$, i.e. if all arc weights are restricted to 1. The following are subclasses of ordinary nets.

- A net $N$ is a *state machine (SM)* if each transition has exactly one input place and one output place ($\forall t \in T, |t^\bullet| = |{}^\bullet t| = 1$).

- A net $N$ is a *marked graph (MG)* if each place has exactly one input transition and one output transition ($\forall p \in P, |p^\bullet| = |{}^\bullet p| = 1$).

- A net $N$ is a *free choice (FC) net* if $\forall p \in P$ : if $|p^\bullet| > 1$ then ${}^\bullet(p^\bullet) = \{p\}$.

- A net $N$ is an *extended free choice (EFC) net* if $\forall p, p' \in P$ : if $p^\bullet \cap {p'}^\bullet \neq \emptyset$ then $p^\bullet = {p'}^\bullet$

An essential property of FC nets is that if $t_1$ and $t_2$ share a common input place then it can never be the case that one of them is enabled while the other is not. That is, every marking enables either both of them or none of them. Hence, in a FC net, every structural conflict also is a dynamic conflict in a safe net. In the case of a non free choice conflict, the dynamic behaviour of the net, i.e. its reachable markings $R_{M_0}$, have to be derived first, before it can be determined whether the conflict is dynamic or not. This is the important difference between free choice and non free choice conflicts.

An EFC net can be transformed into an FC net as suggested by [Bes87]. Therefore, only FC nets will be considered further on this dissertation. Fig. 2.2 shows an example of a transformation of an EFC net into an FC net.

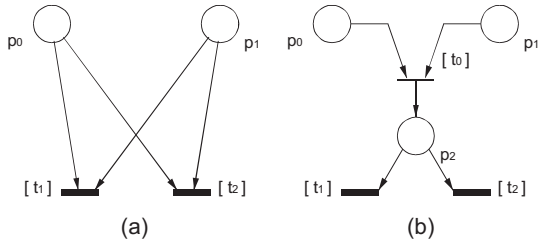


Figure 2.2. An EFC net (a), transformed into an FC net (b)

**2.1.4.** DEFINITION. A *(directed) path* of a net $N$ is an alternating sequence of places and transitions $(x_1, ..., x_i, x_{i+1}, ..., x_n)$, $n > 1$, in which $x_i \in P \cup T$ for $1 \leq i \leq n$, $(x_i, x_{i+1}) \in F$ and all $x_i$s are distinct for $1 \leq i < n$. A path is called *circle path* iff $x_1 = x_n$. A circle path is called a *loop* iff $n = 3$.

**2.1.5.** DEFINITION. A net $N$ is *strongly connected* iff $\forall x, y \in P \cup T$ in which $x \neq y$, there is a directed path from $x$ to $y$.

In a strongly connected net, there is always a path from one node to any other node in the net.

**2.1.6.** PROPOSITION. *If there is a marking $M_0$ such that an ordinary net $N$ is live and bounded, then $N$ is strongly connected.*

**Proof:** See Satz 14.5 in [Sta90]. □

To be strongly connected is only a necessary condition for a net to be live and bounded. Hence, there are strongly connected nets which are not live or not bounded; see Fig. 2.3 for examples.

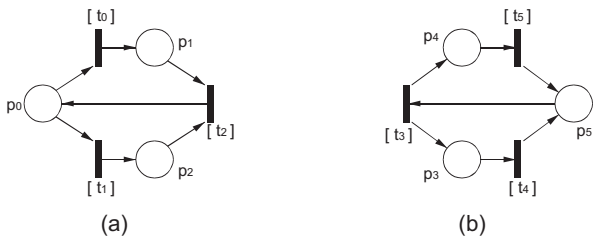


Figure 2.3. Strongly connected nets that are not (a) structurally live or (b) structurally bounded

In practice, nets may occur that are not strongly connected. They follow the pattern: start routine, process, stop routine (see Fig. 2.4a). But, such nets can be made strongly connected by adding dummy transitions which are connected

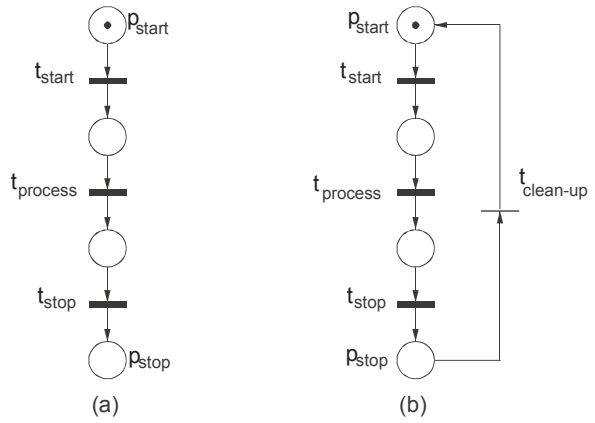


Figure 2.4. (a) not strongly connected net (b) strongly connected net

with all the stop places as pre places and all the start places as post places (see Fig. 2.4b). Clean-up transitions – e.g. the *dummy* transition in Fig. 2.4b – restore the initial state of the net from a final state.

Sometimes, it is easier first to consider the underlying structure of $N$. Only after the structural properties have been found, then the dynamic properties of the net – starting from initial marking $M_0$ – are considered. This approach is used for example to find liveness and boundedness properties of $N$. The following are the structural definitions of liveness and boundedness of $N$: $N$ is *structurally bounded* iff $\exists k \in I\!N : \forall M_0, N$ is $k$-bounded. $N$ is *structurally live* iff $\exists M_0$ such that $N$ is live.

**2.1.7.** Proposition. *A marked graph has a live and safe initial marking iff it is strongly connected.*

**Proof:** See Satz 14.11. in [Sta90]. □

**2.1.8.** Proposition. *A state machine net is live and safe iff it is strongly connected and has only one token.*

**Proof:** see Folgerung 14.13. in [Sta90]. □

The following definition is needed to find structural subnets of $N$.

**2.1.9.** Definition. $N'$ is a *partial subnet* of an ordinary net $N$ ($N' \leq N$) iff $P' \subseteq P$, $T' \subseteq T$ and $F' \subseteq F \cap ((P' \times T') \cup (T' \times P'))$.
$N'$ is a *subnet* of an ordinary net $N$ ($N' \subseteq N$) if $P' \subseteq P$, $T' \subseteq T$, and $F' = F \cap ((P' \times T') \cup (T' \times P'))$.

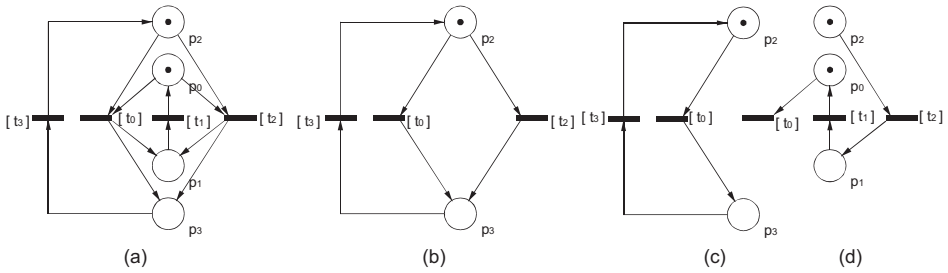


Figure 2.5. (a) A net $N$, (b) an SCSM subnet of $N$, (c) a subnet of $N$, (d) a partial subnet of $N$

Later, in decomposition of the net, a *component* which is responsible for producing a specified output is derived from a subnet. Fig. 2.5 shows examples for definition 2.1.9.

**2.1.10.** Definition. An ordinary net $N$ is *state machine decomposable (SMD)* iff there exists a collection of SCSM subnets $N_i (1 \leq i \leq a)$ of $N$ such that $P = \cup P_i, T = \cup T_i, F = \cup F_i$. $\{N_1, ..., N_a\}$ is called a *cover* of $N$, and it is said that $N$ is *covered* by $\{N_1, ..., N_a\}$. $N' \subseteq N$ is the *SMD-subnet* of N if $N'$ is an SMD net.

In an SMD net, the interaction between SCSM subnets is established through a subset of transitions which are called *synchronization transitions.*

**2.1.11.** Definition. Let $N$ be an SMD net. A transition $t \in T$ is a *synchronization transition* iff $|{}^\bullet t| > 1$ or $|t^\bullet| > 1$.

**2.1.12.** Proposition. *Let $N$ be an SMD net. Every synchronization transition $t \in T$ belongs to at least two different SCSM subnets of any cover of $N$.*

**Proof:** In an SCSM subnet, each transition has only one pre- and one post place. Hence, if there is a synchronization transition in $N$, it should be owned by at least two different SCSM subnets of any cover of $N$. □

**2.1.13.** Definition. Let $N_1$ be a partial subnet of an ordinary net $N$

A path $H = (x_1, ..., x_r), r > 1$ of $N$ is a *handle* of $N_1$ iff $H \cap (P_1 \cup T_1) = \{x_1, x_r\}$.

Note that from Def. 2.1.4, $x_1, x_r$ could be the same node.

The handle is classified according to its first and last node; hence, a handle can be a *PP-, PT-, TP-, or TT-handle.*

Intuitively, a marked graph can be built from a single transition by successively adding TT-handles. The resulting net is a strongly connected marked graph which

has a live and safe initial marking according to proposition 2.1.7. Also, a state machine can be built from a single place by successively adding PP-handles. The resulting net is a strongly connected state machine which is live and safe if it has only one token according to proposition 2.1.8. Therefore, PP- and TT-handles are known as good handles.

In contrast, PT- and TP-handles could cause structurally non-live or non-bounded nets. Therefore they are called bad handles. In Fig. 2.3a, adding the PT-handle $(p_0, t_0, p_1, t_2)$ to the circle path $(t_1, p_2, t_2, p_0, t_1)$ results in a structurally non-live net. The net in Fig. 2.3b is structurally non-bounded because of the TP-handle $(t_3, p_4, t_5, p_5)$ of the circle path $(p_3, t_4, p_5, t_3, p_3)$.

The notions of siphons (formally called deadlocks) and traps, both introduced by Commoner [Com72], have very interesting and useful properties concerning the liveness of a net.

A *siphon* $S$ is a set of places in $N$ such that every transition which puts a token on some place in $S$ requires at least one token from some place in $S$. Hence, a siphon loses tokens each time a transition which has a pre place but no post place in $S$ fires. This implies that if a siphon contains no tokens, it will receive no tokens from any possible firing sequence. Every transition having a pre place in the empty siphon will never be enabled.

A *trap* $\Theta$ is a set of places in $N$ such that every transition which takes a token from $\Theta$ puts at least one token back into $\Theta$. Hence, a trap gains tokens each time a transition which has a post place but no pre place in $\Theta$ fires. This implies, that once a trap is marked (i.e. contains at least one token), it will remain marked.

The formal definitions of siphons and traps are as follows.

**2.1.14.** Definition. Let $N$ be an ordinary net. $S \subseteq P$ is a *siphon* iff $S \neq \emptyset$ and ${}^\bullet S \subseteq S^\bullet$. A siphon $S$ is *minimal* iff there exists no siphon $S'$ such that $S' \subset S$. $\Theta \subseteq P$ is a *trap* iff $\Theta \neq \emptyset$ and $\Theta^\bullet \subseteq {}^\bullet\Theta$. $ST \subseteq P$ is a *siphon − trap* iff $ST \neq \emptyset$ and $ST$ is a siphon which is also a trap. Hence, ${}^\bullet ST \subseteq ST^\bullet$ and $ST^\bullet \subseteq {}^\bullet ST$; i.e. ${}^\bullet ST = ST^\bullet$.

**2.1.15.** Definition. $N' \subseteq N$ is the *siphon(S)-subnet* of N induced by siphon $P'$ with $T' = {}^\bullet P'$.
$N' \subseteq N$ is the *trap(T)-subnet* of N induced by trap $P'$ with $T' = P'^\bullet$.
$N' \subseteq N$ is the *siphon-trap(ST)-subnet* of N induced by siphon-trap $P'$ with $T' = {}^\bullet P' \cup P'^\bullet$.

The set $\{p_0, p_1, p_2\}$ in Fig. 2.6 is a siphon $S$, because the pre transitions of $S$, $\{t_0, t_1, t_2, t_3\}$, are a subset of $\{t_0, t_1, t_2, t_3, t_4\}$, which are the post transitions of $S$. The S-subnet induced by $S$ is the one in the box marked "siphon" (see Fig. 2.6). If $t_4$ is fired, then all the transitions in the S-subnet cannot be fired anymore.

The set $\{p_3, p_4, p_5\}$ in Fig. 2.6 is a trap $\Theta$, because the post transitions of $\Theta$, $\{t_5, t_6, t_7, t_8\}$, are a subset of $\{t_4, t_5, t_6, t_7, t_8\}$, which are the pre transitions of $\Theta$.

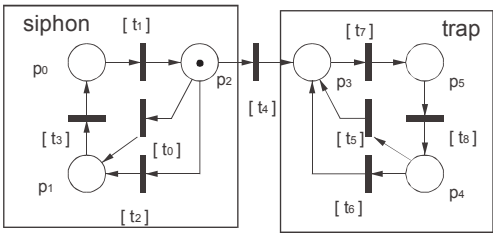


Figure 2.6. Siphon and trap example

The T-subnet induced by $\Theta$ is the one in the box marked "trap" (see Fig. 2.6). If $t_4$ is fired, then all the transitions in the T-subnet could be fired forever.

If a transition $t_9$ with arcs $(p_3, t_9)$ and $(t_9, p_2)$ are added to the net in Fig. 2.6, then all the places in the net form a siphon-trap, and the ST-subnet of the siphon-trap is the net itself. In a siphon-trap net, there is token flow neither out of nor into the net.

**2.1.16.** Definition. Let $N$ be a net and $N'$ a partial subnet of $N$.

A path $(t, p)$ in $N$ is a *meeting path* of $N'$ iff $p \in P'$ and $(t, p) \notin F'$. A meeting path $(t, p)$ can be extended to a handle iff there exists a handle $(x_1, ..., x_r) = (x_1, ..., t, p)$ of $N'$; $x_r = p$; if $r = 2$ then $x_1 = t$.

**2.1.17.** Proposition. *In a strongly connected net, a meeting path can always be extended to a handle.*

**Proof:**

From Def. 2.1.16, a meeting path $(t, p)$ of $N'$ has an arc $(t, p) \notin F'$. If $t \notin T'$ then there is a directed path from $x (x \in N')$ to $t$ because $N$ is strongly connected (see Def. 2.1.5); i.e. we have a handle $(x, ..., t, p)$. If $t \in T'$ then we have a TP-handle $(t, p)$. □

The following definition of a redundant place was introduced by [Ber87] and is adapted for decomposition by [VW02].

**2.1.18.** Definition. A place $p$ of a net $N$ is (structurally) *redundant* if there is a set of places $Q \subset P$ with $p \notin Q$, a valuation [1] $V : Q \cup \{p\} \to \mathbb{N}$ and some $c \in \mathbb{N}_0$ with the following properties for all $t \in T$:

1. $V(p)M_N(p) - \sum_{q \in Q} V(q)M_N(q) = c$
   (For the initial marking, the valuated number of tokens on $p$ is greater than or equal to the sum of the validated numbers of tokens on the places belonging to $Q$.)

[1] The valuation $V(r)$ of a place $r \in Q \cup \{p\}$ is given to every token on $r$, flowing into $r$, and out of $r$

2. $V(p)(W(t,p) - W(p,t)) - \sum_{q \in Q} V(q)(W(t,q) - W(q,t)) \geq 0$
   (When transition $t$ occurs, the growth of the valuated number of tokens on $p$ is greater than or equal to that of the places of $Q$.)

3. $V(p)W(p,t) - \sum_{q \in Q} V(q)W(q,t) \leq c$
   (The difference between the valuated number of tokens on $p$ and those of places belonging to $Q$ necessary to give concession to $t$ must be less than or equal to this difference in the initial marking.)

**2.1.19.** Definition. Place $p$ is a *general duplicate* of place $q$, if $\forall t : W(p,t) - W(q,t) = c, W(t,p) - W(t,q) \geq c$, and $M_N(p) - M_N(q) \geq c$.

**2.1.20.** Proposition. *General duplicate place is a redundant place.*

**Proof:** Substitute equation 1 to equation 3 of Def. 2.1.18 gives the following equation: $V(p)(M_N(p) - W(p,t)) - \sum_{q \in Q} V(q)(M_N(q) - W(q,t)) \geq 0$. We have $Q = \{q\}$ and for $V(p) = V(q) = 1$, the above equation become $M_N(p) - M_N(q) \geq W(p,t)) - W(q,t)$ which is fulfilled per Def. 2.1.19; equation 2 of Def. 2.1.18 become $W(t,p) - W(t,q) \geq W(p,t) - W(q,t)$ which is also fulfilled per Def. 2.1.19. □

**2.1.21.** Definition. Place $p$ is an *extended duplicate* of place $q$, if $\forall t : W(t,p) = W(t,q), W(p,t) = W(q,t)$ and $M_N(p) \geq M_N(q)$.

**2.1.22.** Proposition. *Extended duplicate place is a redundant place.*

**Proof:** Extended duplicate place is a general duplicate place with $c = 0$. Hence based on proposition 2.1.20, extended duplicate place is a redundant place. □

**2.1.23.** Definition. In ordinary net, place $p$ is a *practical duplicate* of place $q$, if ${}^\bullet p = {}^\bullet q$, $p^\bullet = q^\bullet$ and $M_N(p) \geq M_N(q)$.

**2.1.24.** Proposition. *Practical duplicate place is a redundant place.*

**Proof:** Practical duplicate place is an extended duplicate place, i.e. if $t \in {}^\bullet p$ then $t \in {}^\bullet q$ and $W(t,p) = W(t,q) = 1$; if $t \notin {}^\bullet p$ then $t \notin {}^\bullet q$ and $W(t,p) = W(t,q) = 0$; if $t \in p^\bullet$ then $t \in q^\bullet$ and $W(p,t) = W(q,t) = 1$; if $t \notin p^\bullet$ then $t \notin q^\bullet$ and $W(p,t) = W(q,t) = 0$. Hence based on proposition 2.1.22, practical duplicate place is a redundant place. □

## 2.2 Signal Transition Graphs

As event-driven systems, asynchronous circuits respond immediately and in general concurrently to rising and falling edges of their binary input signals $x$, namely by generating edges of their output signals $y$. The interaction between circuit and environment across their interface can be modeled from the causal point of view by signal transition graphs (STGs). The first proposals were made by [Wen77], [RY85], and [Chu86], with extensions and generalization by [VYCLdM94], [Wol97], and [WB00]. In this dissertation, unless otherwise mentioned only STGs without extensions and generalizations are considered.

A controller is a circuit that synchronize the operations performed by an operational unit – e.g. counter, ALU, etc. – through a protocol. Fig. 2.7 shows part of the timing diagram of a VME bus controller from [CKK+02] (only for the $dsr+$ case). The STG describing the complete timing diagram is shown in Fig. 2.8.

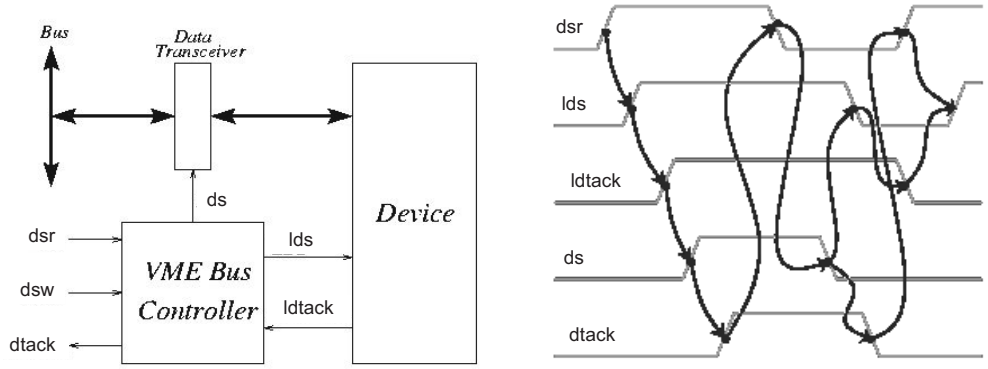


Figure 2.7. Timing diagram example

Firing a transition in the STG means an occurrence of a signal edge; therefore, transitions in an STG are labelled with signals from some alphabet $\Sigma \times \{+, -\}$ or with the empty word $\lambda$. $\{+, -\}$ denote the edges: $s+$ means a rising edge of $s$ and $s-$ means a falling edge of $s$.

The types of transitions in this dissertation are:

- input transitions,
- output transitions,
- divining transitions – transitions that have been silenced,
- internal transitions – transitions of internal signals that are unobservable at the system interface and required for synthesis,

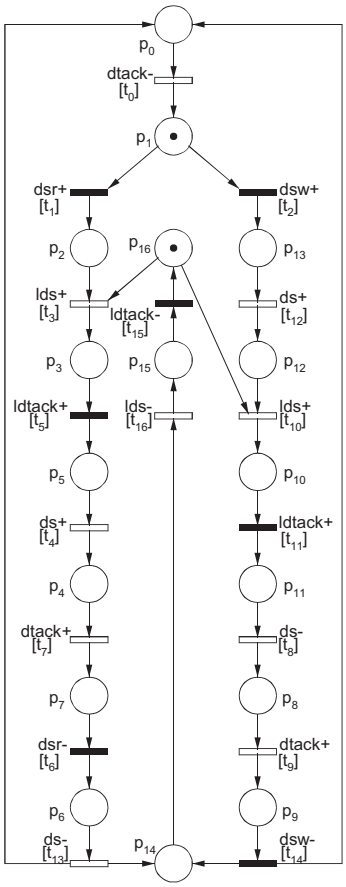


Figure 2.8. VME bus controller STG

- dummy transitions do not represent signal changes and are labelled with the empty word $\lambda$ – used to simplify petri net structure and to make the graph easier to understand.

In a graph, transitions are represented as a filled box (input), an empty box (output), a gray box (internal), a line (dummy), or box with two lines (divining). See Fig. 2.9 for illustration.

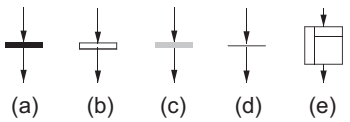


Figure 2.9. (a) An input transition, (b) an output transition, (c) an internal transition, (d) a dummy transition, (e) a divining transition

**2.2.1.** Definition. An STG $N$ is a tuple $(P, T, F, M_0, W, l, In, Out)$ where
$P$ is the set of places,
$T$ is the set of transitions, ($P$ and $T$ are disjoint)
$F \subseteq P \times T \cup T \times P$ is the flow relation, i.e. the set of directed arcs,
$M_0 : P \to \mathbb{N}_0$ is the initial marking,
$C : P \to \mathbb{N}$ is the place capacity,
$W : P \times T \cup T \times P \to \mathbb{N}_0$ assigns weights to the pairs $(p, t)$ and $(t, p)$. Pairs belonging to $F$ (the directed arcs) are assigned weights $> 0$; pairs $\notin F$ (non-existent arcs) are assigned weight 0,
$l : T \to In \times \{+, -\} \cup Out \times \{+, -\} \cup \{\lambda\}$ is the label of transitions, where
$In \subseteq \Sigma$ is the set of input signals,
$Out \subseteq \Sigma$ is the set of output signals,
and $In$ and $Out$ are disjoint.

The labelling of transition sequences can be extended as follows; $l(t_1...t_n) = l(t_1)...l(t_n)$, where the empty word is deleted automatically. A sequence $v$ of signal edges is *enabled* under a marking $M$, denoted by $M[v\rangle\rangle$, if there is some transition sequence $w$ with $M[w\rangle$ and $l(w) = v$; $M[v\rangle\rangle M'$ if $M[w\rangle\rangle M'$. If $M = M_0$, then $v$ is called a *trace*. The *language* $L(N)$ is the set of all traces. Two STGs are *language equivalent* if they have the same traces.

Due to the physical nature of the signal, signal edges are required to alternate. An STG is *consistent* if for all signals $s \in l$, in every trace of the STG, the edges $s+$ and $s-$ alternate, and there are no two traces where $s+$ comes first in the one and $s-$ in the other.

Two different transitions $t_1$ and $t_2$ are *enabled concurrently* under a marking $M$ if $W(., t1) + W(., t2) \leq M$, i.e. if there are enough tokens for both transitions to be fired together. If both transitions have the same label $s_{edge}$, then $s_{edge}$

is *enabled auto-concurrently* under $M$. An STG has no *auto-concurrency*, if no $s_{edge}$-transitions are enabled auto-concurrently under any reachable marking.

If two different transitions $t_1$ and $t_2$ are in structural conflict and are labelled with the same $s_{edge}$, then they are in *structural auto-conflict*, and the STG has such a conflict. If $t_1$ is an input and $t_2$ an output transition, then they form a *structural input-output conflict* and the STG has such a conflict. If both $t_1$ and $t_2$ are output transitions labelled with different signals, then they form a *structural output-output conflict*, and the STG has such a conflict.

If two different transitions $t_1$ and $t_2$ are in dynamic conflict and are labelled with the same $s_{edge}$, then they are in *dynamic auto-conflict*, and the STG has such a conflict.

Transitions in *free choice conflict* are transitions that have the same pre places: ${}^\bullet t_1 = {}^\bullet t_2$. In a safe net, these transitions are in dynamic conflict: if one transition is enabled, then the others also are enabled. If there is another place in pre place of $t$ besides the choice places, then this place is called *control place*. Another example of dynamic conflict are transitions in structural conflict where the choice place(s) are the only pre places of at least one transition: ${}^\bullet t_1 \cap {}^\bullet t_2 = {}^\bullet t_1$ or ${}^\bullet t_1 \cap {}^\bullet t_2 = {}^\bullet t_2$; i.e. there is a transition without control places. In contrast, if each of the transitions in structural conflict has a control place in addition to the conflict place(s), then these transitions would not be in dynamic conflict, unless there is a reachable marking $M$ that enables two or more of them.

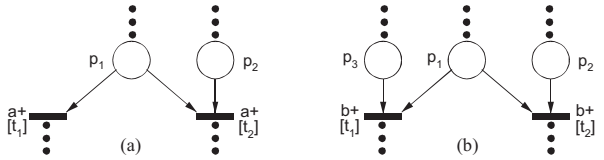


Figure 2.10. Structural auto-conflict

Fig. 2.10 shows two STGs with a structural auto-conflict. The STG in Fig. 2.10a has a structural auto-conflict between the transitions $t_1$ and $t_2$ (both labelled $a+$); it also is a dynamic auto-conflict, because the choice place $p_1$ is the only pre place of $t_1$ – $t_1$ has no control place, so that when $t_2$ is enabled, $t_1$ is also enabled. In contrast, the structural auto-conflict between $t_1$ and $t_2$ (both labelled $b+$) in Fig. 2.10b is not necessarily dynamic because each transition in conflict has a control place; perhaps there is no reachable marking under which both transitions are enabled.

An STG is *deterministic* if it has no internal transitions and if for each of its reachable markings and each $s_{edge}$, there is at most one $s_{edge}$-labelled transition enabled under the marking, i.e., no auto-concurrency and no dynamic auto-conflict.

An STG is *synthesizable* [2] if it is deterministic and consistent. It is also required that the synthesizable STG has no dynamic input-output conflict, because such an STG cannot be synthesized into a reliable (i.e. hazard free) digital circuit; except with some timing constraint. Most synthesis tools also require STGs to be live and safe to be synthesizable.

The following backgrounds are needed for STG decomposition.

**2.2.2.** DEFINITION. A *secure t-contraction* (based on [And83]) of STG $N$, resulting in $\overline{N}$, is done as follows:

(a) Merge each pre place of $t$ with each post place of $t$:
$\overline{P} = \{p \mid p \in P - {}^\bullet t \cup t^\bullet\} \cup \{(p, p') \mid p \in {}^\bullet t, p' \in t^\bullet\}$
$M_{\overline{N}}((p, p')) = M_N(p) + M_N(p')$
$\overline{W}((p, p'), t_1) = W(p, t_1) + W(p', t_1), \forall t_1 \in T$
$\overline{W}(t_1, (p, p') = W(t_1, p) + W(t_1, p'), \forall t_1 \in T$

(b) Delete $t$ and all its incident loop arcs:
$\overline{T} = T - t$
$\overline{W}((p, p'), t) = 0$
$\overline{W}(t, (p, p')) = 0$

For simplicity, in the graph, the merge place of $p_1$ and $p_2$ is denoted as $p_{1\_2}$ instead of $(p_1, p_2)$. Fig. 2.11 shows an example for step by step contraction. Fig. 2.11a is the initial STG, upon which secure contraction of transition $t$ will be performed. Fig. 2.11b is the STG after merging each $p \in {}^\bullet t = \{p_1\}$ with each $p \in t^\bullet = \{p_2, p_3\}$. Hence we have $p_{1\_2}$ with $M_{\overline{N}}(p_{1\_2}) = M_N(p_1) + M_N(p_2) = 1$ and $p_{1\_3}$ with $M_{\overline{N}}(p_{1\_3}) = M_N(p_1) + M_N(p_3) = 1$. Note that when merging places, the incident arcs of the place are kept intact. Fig. 2.11c is the STG after deleting loop-only transition $t$ and all its incident loop arcs.

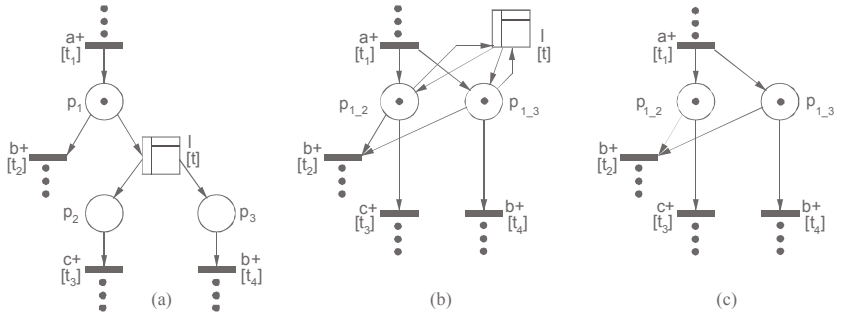


Figure 2.11. Secure $t$-contraction

[2] means "synthesizable without hazards and timing constraints", e.g. that a hazard-free asynchronous circuit without timing constraints can be built

Simulations are a well-known important device for proving language inclusion or equivalence. A *simulation from* $N_1$ *to* $N_2$ is a relation $\mathcal{S}$ between markings of $N_1$ and $N_2$ such that $(M_{N_1}, M_{N_2}) \in \mathcal{S}$ and for all $(M_1, M_2) \in \mathcal{S}$ and $M_1[t\rangle M_1'$ there is some $M_2'$ with $M_2[l_1(t)\rangle\rangle M_2'$ and $(M_1', M_2') \in \mathcal{S}$. If such a simulation exists, then $N_2$ can go on simulating all signals of $N_1$ forever. If there exists a simulation from $N_1$ to $N_2$, then $L(N_1) \subseteq L(N_2)$.

A relation $\mathcal{B}$ is a *bisimulation* between $N_1$ and $N_2$ if it is a simulation from $N_1$ to $N_2$ and $\mathcal{B}^{-1}$ is a simulation from $N_2$ to $N_1$. If such a bisimulation exists, we call the STGs *bisimilar*; intuitively, the STGs can work side by side such that in each stage each STG can simulate the signals of the other.

In the following definition of *parallel composition* $\|$, the distinction between input and output signals is important and will be considered. The idea of parallel composition is that the composed systems run in parallel synchronizing on common signals. Since a system controls its outputs, it is not allowed that a signal is an output of more than one component; input signals, on the other hand, can be shared. An output signal of one component can be an input of one or several others, and in any case it is an output of the composition (see Fig. 2.12).

The *parallel composition* $\|$ of STGs $N_1$ and $N_2$ is defined if $Out_1 \cap Out_2 = \emptyset$. Then, let $A = (In_1 \cup Out_1) \cap (In_2 \cup Out_2)$ be the set of common signals. In Fig. 2.12b, these common signals are $(X''', Y_2'', Y_1'')$. In the parallel composition $N = N_1 \| N_2$, each $s_{edge}$-labelled transition $t_1$ of $N_1$ is combined with each $s_{edge}$-labelled transition $t_2$ from $N_2$ if $s_{edge} \in A$. In the formal definition of parallel composition, $*$ is used as a dummy element, which is formally combined e.g. with those transitions that do not have their label in the synchronization set $A$. ($*$ is assumed not to be a transition or a place of any net.) Thus, $N$ is defined by

$$
\begin{aligned}
P &= P_1 \times \{*\} \cup \{*\} \times P_2 \\
T &= \{(t_1, t_2) \mid t_1 \in T_1, t_2 \in T_2, l_1(t_1) = l_2(t_2) \in A\} \\
&\quad \cup \{(t_1, *) \mid t_1 \in T_1, l_1(t_1) \notin A\} \\
&\quad \cup \{(*, t_2) \mid t_2 \in T_2, l_2(t_2) \notin A\} \\
W((p_1, p_2), (t_1, t_2)) &= \begin{cases} W_1(p_1, t_1) \text{ if } p_1 \in P_1,\ t_1 \in T_1 \\ \text{or} \\ W_2(p_2, t_2) \text{ if } p_2 \in P_2,\ t_2 \in T_2 \end{cases} \\
W((t_1, t_2), (p_1, p_2)) &= \begin{cases} W_1(t_1, p_1) \text{ if } p_1 \in P_1,\ t_1 \in T_1 \\ \text{or} \\ W_2(t_2, p_2) \text{ if } p_2 \in P_2,\ t_2 \in T_2 \end{cases}
\end{aligned}
$$

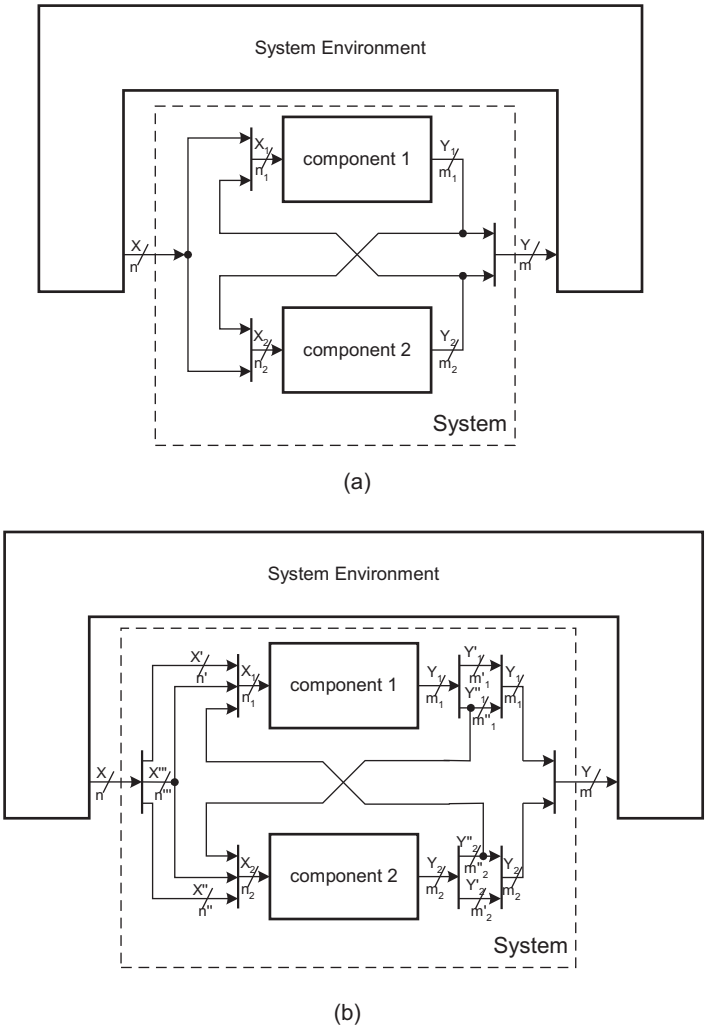


Figure 2.12. (a) parallel composition block diagramm and (b) its detail

$$
\begin{aligned}
l((t_1,t_2)) &= \begin{cases} l_1(t_1) & \text{if } t_1 \in T_1 \\ l_2(t_2) & \text{if } t_2 \in T_2 \end{cases} \\
M_N &= M_{N_1} \dot{\cup} M_{N_2}, \text{ i.e. } M_N((p_1,p_2)) = \begin{cases} M_{N_1}(p_1) & \text{if } p_1 \in P_1 \\ M_{N_2}(p_2) & \text{if } p_2 \in P_2 \end{cases} \\
In &= (In_1 \cup In_2) - (Out_1 \cup Out_2) \\
Out &= Out_1 \cup Out_2
\end{aligned}
$$

For simplicity, in the graph, a place of the composition is denoted as $p$ instead of $(p,*)$ or $(*,p)$; also transition is denoted as $t$ instead of $(t,*)$ or $(*,t)$, and $t_{1\_2}$ instead of $(t_1,t_2)$. Also, to keep the example small, the transitions are only labelled with signals instead of signal edges. An example of parallel composition follows.

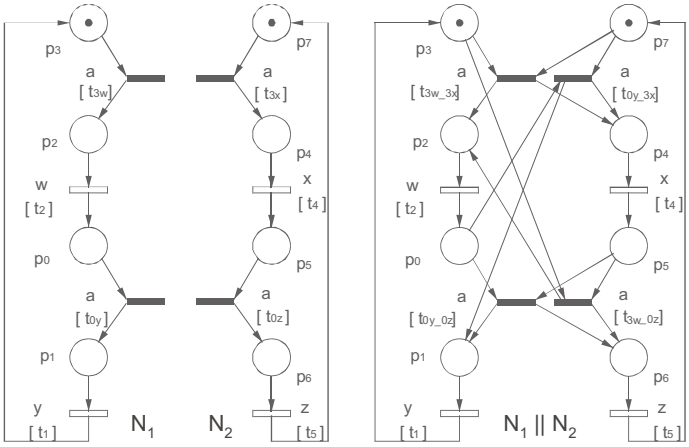


Figure 2.13. Parallel composition example

In the example, there are two transitions with label $a$ in $N_1$: $t_{3w}$ and $t_{0y}$, and two in $N_2$: $t_{3x}$ and $t_{0z}$. Each transition with label $a$ in $N_1$ should be synchronized with each transition with label $a$ in $N_2$. Therefore, in $N_1 \parallel N_2$, there are four transitions with label $a$: a synchronization of $t_{3w}$ with $t_{3x}$ ($t_{3w\_3x}$), $t_{3w}$ with $t_{0z}$ ($t_{3w\_0z}$), $t_{0y}$ with $t_{3x}$ ($t_{0y\_3x}$) and $t_{0y}$ with $t_{0z}$ ($t_{0y\_0z}$). Note that though there is a synchronization between $t_{0y}$ and $t_{3x}$, and between $t_{3w}$ and $t_{0z}$, the synchronized transitions $t_{0y\_3x}$ and $t_{3w\_0z}$ in $N_1 \parallel N_2$ will never be fired. However, all these possible pairings have to be done, because it is not known a priori which $s$-labelled transition of $N_2$ will occur together with which $s$-labelled transition of $N_1$.

One can consider the place set of the composition as the disjoint union of the place sets of the components. Therefore, markings of the composition (regarded as multisets) can be considered as the disjoint union of markings of the components. Sometimes – e.g. if a bisimulation-like correctness definition (see section 3.2.1) is

used –, it is useful to look only at the restriction $M|_{P_i}$ of a marking $M$ of the composition.

A marking $M_1 \dot{\cup} M_2$ of the composition is also denoted by $(M_1, M_2)$. By definition of $\|$, the firing $(M_1, M_2)[(t_1, t_2)\rangle(M_1', M_2')$ of $N(N = N_1 \parallel N_2)$ corresponds to the firings $M_i[t_i\rangle M_i'$ in $N_i$, $i = 1, 2$; here, the firing of $*$ means that the empty transition sequence fires. Therefore, all reachable markings of $N$ have the form $(M_1, M_2)$, where $M_i$ is a reachable marking of $N_i$, $i = 1, 2$.

If the components do not have internal transitions, then their composition has none. To see that $N$ is deterministic if $N_1$ and $N_2$ are, consider different transitions $(t_1, t_2)$ and $(t_1', t_2')$ with the same label that are enabled under the reachable marking $(M_1, M_2)$. The transitions should differ in at least one component; suppose $t_1, t_1'$ are different transitions in $N_1$. It cannot be the case that $t_1$ is a transition while $t_1' = *$, because we would have $l((t_1, t_2)) \in In_1 \cup Out_1$ but $l((t_1', t_2')) \notin In_1 \cup Out_1$; i.e. $l((t_1', t_2')) \notin A$. Therefore, $t_1$ and $t_1'$ are different transitions with the same label. Because $(t_1, t_2)$ and $(t_1', t_2')$ are enabled under the reachable marking $(M_1, M_2)$, $t_1$ and $t_1'$ are enabled under the reachable marking $M_1$, which contradicts that $N_1$ is deterministic. But note that $N$ might have structural auto-conflicts even if none of the $N_i$ has. The example in Fig. 2.13 shows this case.

Up to isomorphism, composition is associative and commutative. Therefore, the parallel composition of a family (or collection) $(C_i)_{i \in I}$ of STGs can be denoted as $\|_{i \in I} C_i$, provided that no signal is an output signal of more than one of the $C_i$. The markings of such a composition will be denoted by $(M_1, \ldots, M_n)$ if $M_i$ is a marking of $C_i$ for $i \in I = \{1, ..., n\}$ where $n$ is the number of an output block in a partition.

# Chapter 3

# Existing Decomposition Methods

In the time of imperialism in this world, the strategy *'divide et impera'* is used by a conqueror to conquer a large country with a small force. He first divides the large country into two or more parts governed by opposing parties, and then conquer them, one by one. The same strategy is used by many researchers to conquer (to analyze and synthesize) a large net (with high concurrency degree) by dividing it into two or more small nets. I.e. by decomposition, a net is divided into subnets.

The first motivation for decomposing a P/T net is to analyze it. For example, to find whether a net is live or not, one must first derive its reachable markings. This task is difficult and sometimes impossible for a large, highly concurrent net due to the *state explosion* problem; i.e. the number $r$ of reachable markings is too large to handle.

As P/T nets are used more and more to model systems, another problem is encountered if a circuit is to be synthesized from the net. Because one should derive the reachable markings of the net as the first synthesis step, the same problem as when analyzing the net occurs, i.e. the state explosion problem. Not only that, many of the synthesis algorithms that derive the implementation from reachable states have exponential complexity. This task is difficult and sometimes impossible for a large number of reachable states. One can try to alleviate this problem by using a heuristic algorithm which has polynomial complexity. But if the problem is large, sometimes one can find only a solution that represents a local minimum in solution space, i.e. a non-optimal result. Therefore, the decomposition of P/T nets is used to address the above problems, hoping that the overall effort and cost of synthesizing the components will be significantly smaller than for handling the large net.

This chapter begins with the structural decomposition of P/T nets for analysis purposes – liveness and boundedness of the net. Then STG decomposition for synthesis from [VW02] is described.

## 3.1 P/T Net Decomposition: Analysis Purpose

The early P/T net decomposition is aimed at analyzing the liveness and boundedness of the net. The researchers first looked for the smallest structures of P/T nets that can be used to characterize liveness and boundedness. This is the cause for the birth of the siphon and trap definitions, which were first suggested by [Com72]. [Com72] and [Hac72] used siphons and traps to prove liveness and safeness of FC nets. Similar approach was also used by [BT87] and [EBS89].

A siphon $S$ which contains a trap $\Theta$ which is marked at $M_0$ will always contain a token; hence the transitions of $S^\bullet$ can always be fired. If all the siphons in the net contain a trap which is marked at $M_0$, then all the transitions in the net can be fired. For an FC net, this also means that the net is live.

**3.1.1.** Proposition. *An FC net $N$ is live iff every minimal siphon of $N$ contains a trap which is marked at $M_0$.*

**Proof:** see [Hac72]. □

**3.1.2.** Proposition. *A live FC net $N$ is safe iff it is covered by SCSM-subnets which have exactly one token each at $M_0$.*

**Proof:** see Theorem 6.5. in [BT87]. □

Note that proposition 3.1.2 implies strong connectedness of $N$.

**3.1.3.** Proposition. *In a live and safe FC net $N$, if $S$ is a minimal siphon of $N$, then $S$ is also a trap.*

**Proof:** see Lemma 6.9. in [BT87]. □

**3.1.4.** Proposition. *In a live and safe FC net $N$, a subnet $N_S$ induced by a minimal siphon $P_S$ is an SCSM-subnet of $N$.*

**Proof:** see Lemma 6.10. in [BT87]. □

**3.1.5.** Corollary. *Let $N$ be a live and safe FC net. Every SCSM-subnet of $N$ is marked at $M_0$ and there exists an SCSM-cover of $N$, such that each $N_i$ belonging to the SCSM-cover has exactly one token at $M_0$.*

**Proof:** $N$ is live. Therefore, every minimal siphon contains a trap which is marked at $M_0$ (proposition 3.1.1). $N$ also is safe; therefore every subnet $N_S$ induced by a minimal siphon $P_S$ is an SCSM-subnet of $N$ (proposition 3.1.4). Hence, being both live and safe, FC net $N$ is covered by SCSM subnets which have exactly one token each at $M_0$ (proposition 3.1.2). □

Using a graph theoretical approach, [BL89] characterize minimal siphons in P/T nets. Based on this characterization, [EBS89] derive a minimal siphon characterization for FC nets.

**3.1.6.** PROPOSITION. *Let $N$ be an FC net, $P_S \subseteq P$ a siphon of $N$ and $N_S$ the subnet of $N$ induced by $P_S \cup {}^\bullet P_S$. $P_S$ is minimal iff $P_S$ is strongly connected and we have $|{}^\bullet t \cap P_S| = 1$ for every transition $t \in N_S$.*

**Proof:** see [EBS89] □

Based on proposition 3.1.6, [EBS89] propose an algorithm to find the subnet $N_S$ induced by minimal siphon $S$. The algorithm concentrates on a specific part of the net. Beginning with a seed node of the net, it gradually adds parts of the net (handles) until the subnet is found.

**Algorithm** *S-subnet*
**Input:** A strongly connected FC net with a *seed* place $\overline{p}$
**Output:** S-subnet induced by the minimal siphon $P_S$
1. $P_S := \{\overline{p}\}$; $T_S := \emptyset$; $F_S := \emptyset$;
2. **while** $\exists p \in P_S$ and $\exists t \in {}^\bullet p$ such that $(t,p) \notin F_S$ **do**
3. *get_meeting_path_handle*$(N_S, N, p, t, N_H)$;
4. $(*$ $N_H$ (which is a path in $N$ but not a path in $N_S$) is a handle of $N_S$ that begins with a node in $N_S$ and ends with $t, p$ $*)$
5. $P_S := P_S \cup P_H$; $T_S := T_S \cup T_H$; $F_S := F_S \cup F_H$;
6. $(*$ **end of while** there is still a meeting path of $N_S$ $*)$

The algorithm *S-subnet* begins with $N_S$ which has only one seed place $p$. The S-subnet $N_S$ is found by iteratively adding a meeting path handle to the current $N_S$, so long as the resulting $N_S$ still has meeting paths.

The net in Fig. 2.6 is taken as an example. $p_2$ is the seed place. After adding $p_2$ to $N_S$, there is a meeting path $(t_1, p_2)$ that should be extended to a handle. The extension is the handle $H_1 = (p_2, t_0, p_1, t_3, p_0, t_1, p_2)$. After adding $H_1$ to $N_S$, there is still a meeting path $(t_2, p_1)$ that should be extended to a handle. The extension is the handle $H_2 = (p_2, t_2, p_1)$. After adding $H_2$ to $N_S$, there is no more meeting path to be found. Hence, $N_S$ (the net in the siphon box of the Fig. 2.6) is an S-subnet and $P_S$ is a minimal siphon of $N$.

The algorithm *S-subnet* generates S-subnets with the following properties [EBS89]:

1. The resulting net is strongly connected, because at the beginning there is only one place and the operation of adding a meeting handle (see definition 2.1.16 and proposition 2.1.17) preserves strong connectedness.

2. Every transition $t$ in $T_S$ has exactly one incoming arc in $N_S$. $t$ has at least one incoming arc due to strong connectedness, and $t$ has at most one incoming arc because the added meeting handle always ends in a place.

3. Because all the meeting paths of $p \in P_S$ are taken, we have $T_S = {}^\bullet P_S$; and because not all choice paths of $p \in P_S$ are taken, we have $T_S \subseteq P_S{}^\bullet$. $N_S$ is an S-subnet of $N$; i.e. all the arcs $f \in F$ between nodes in $N_S$ should also be in $F_S$. Assume this is not the case, which means there is an arc $f \in F$ between nodes in $N_S$ that is not in $F_S$. It could be an arc from a place $p \in P_S, |p^\bullet > 1|$ to a transition $t \in T_S, |{}^\bullet t > 1|$, which cannot be the case in an FC net; or it could be an arc from a transition $t \in T_S$ to a place $p \in P_S$, which cannot be the case because all the meeting paths have been taken.

From the above properties, it is obvious that the proposition 3.1.6 is true; i.e. $P_S$ is a minimal siphon of $N$.

Using the rank theorem, [Esp90] improved the liveness and boundedness characterization of an FC net. [KB92] used this improvement and proposed a polynomial algorithm to decide liveness and boundedness of an FC net. It also decides state machine decomposability by finding minimal siphons of the FC net using the [EBS89] algorithm and checks whether the net induced by the minimal siphon is a state machine.

An improved version of the [Esp90] algorithm proposed by [Kem93] yields a linear-time algorithm for finding the minimal siphons of a strongly connected FC net. The C++ implementation of the [Kem93] algorithm and experiment results can be seen in [War05].

## 3.2 STG Decomposition: Synthesis Purpose

[Chu87a], [Chu87b], and [KKT93] suggest decomposition methods for STGs, but these approaches can only deal with very restricted net classes. [Chu87a] only decomposes live and safe FC nets, which cannot model controlled choices or arbitration, and makes further restrictions; e.g. each transition label is allowed only once (which makes the STG deterministic in the sense of language theory), and conflicts can only occur between input signals. The conference version [Chu87b] restricts attention even further to marked graphs, which have no conflicts at all.

The method in [Chu87a] and [Chu87b] constructs for each output signal $s$ a component $C_i$ that generates this signal; $C_i$ has as inputs all signals that – according to the net structure – may directly cause $s$ to change. The component is obtained from the STG $N$ by contracting all transitions belonging to the signals

that are neither input nor output signals for this component. This contraction is required to be tr-preserving (as defined in [Chu87a]) , and it might be necessary to add further signals to the inputs to ensure this.

In [Chu87a] [Chu87b], it is stated that the parallel composition of the $C_i$ – i.e. the *(modular) implementation* – has the same language as $N$; in the restricted setting of [Chu87a] [Chu87b], this is the same as having isomorphic reachability graphs. Clearly, this isomorphism is very strict and not a necessary requirement for an implementation to be correct (see section 3.2.1). On the other hand, language equivalence is too weak in general, since it ignores which choices are possible during a run, and in particular it ignores deadlocks as will be shown in section 3.2.1.

A similar decomposition method is described in [KKT93]; only marked graphs with only output signals are considered. In contrast to [Chu87a], a component can generate several output signals, and different components can generate the same signal; this gives more flexibility, but the latter feature necessitates additional components for collecting occurrences of the same signal generated by different components.

[BW93] and [Wol97] use a decomposition approach like [Chu87a] to implement a modular asynchronous controller. Also, [KGJ96] use a decomposition approach for fork/join machines, which are a restricted form of FC STGs. After decomposition, the results are implemented as distributed burst mode circuits.

### 3.2.1 Vogler-Wollowski algorithm

The decomposition method from Vogler-Wollowski [VW02] is based on [Chu87a]. In contrast to other methods, it can be applied without restriction to the graph theoretic structure of the given STG. It can even deal with arc weights greater than 1 and unsafe nets. It only restricts STG $N$ to be deterministic and synthesizable. Also, the main difference between the algorithm [VW02] and others is the correctness proof. The algorithm [VW02] is proved to be correct and can be applied to labelled P/T nets which are more general than STGs. The following are the features of the algorithm [VW02]:

- The composition of the $C_i$ is ensured to be free of what Ebergen calls computation interference [Ebe92] (see Fig. 3.7 for a computation interference example), where one component produces an output that is an unspecified input for another.

- Only behaviour where the environment behaves as specified by the original STG $N$ is considered, i.e. the composition of the components might specify additional inputs, but these and any subsequent behaviour is ignored since they cannot occur if the implementation runs in an appropriate environment. The same is done e.g. in [Dil88] and [Ebe92], so both these features are not new – but new in the context of STG decomposition.

These features are achieved with a bisimulation-like correctness definition. Bisimulation is chosen instead of language equivalence, because bisimulation can distinguish between deterministic and non-deterministic STG. The STGs in Fig. 3.1 are language equivalent, i.e. they have the same language {$\lambda$, *send, send receive*}. But they are not bisimilar. $N$ can simulate $N'$ and $N''$. But $N'$ cannot simulate $N$ ($N'$ can deadlock after *send*), also $N''$ cannot simulate $N$ ($N''$ can deadlock after firing internal $\lambda$-transition).

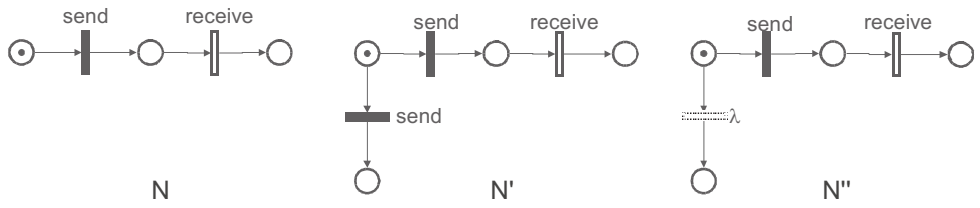


Figure 3.1. Language equivalent STGs which are not bisimulation equivalent

For deterministic STGs, language equivalence and bisimulation coincide. $N'$ and $N''$ in Fig. 3.1 are not deterministic STG; $N'$ is not deterministic due to the dynamic auto conflict and $N''$ is not deterministic due to the internal $\lambda$-transition.

The algorithm in [VW02] decomposes a deterministic STG $N$ into component STGs $C_i$ based on a given partition of the set of output variables, such that each $C_i$ is responsible for one block of the partition. $N$ should have no structural auto conflicts to be deterministic and no structural input-output-conflicts to be synthesizable. The partition is required to be *feasible*, i.e. the outputs in structural output-output conflict should be in the same block. Each $C_i$ is then extracted from an initial STG $N_i$ (a copy of STG $N$), care being taken to keep only the relevant input signals, which may be global inputs or outputs of other components. We say that a signal $c$ is a *relevant input signal* for an output signal $x$, if there is a $c$-labelled transition $t$ which *gives concession* to an $x$-labelled transition $t'$. This is the case if $t^\bullet \cap {}^\bullet t' \neq \emptyset$, $c \in In \cup Out$, and $x \in Out$. Deconcession is not considered because the output should be feasible and there is no structural input-output conflict in $N$. The other signals are irrelevant for $C_i$, and their corresponding transitions are silenced into divining transitions ($l(t) = \lambda$), which are securely contracted from STG $N_i$. The procedure to extract $C_i$ is as follows:

1. Before contracting a divining transition $t$, the *preconditions* should be checked for: that the pre and post arcs of $t$ have weight 1, and that $t$ forms no loop with any place. Fig. 3.2a shows an STG that violates the arc weight requirement: $W(p_1, t) = 2$; Fig. 3.2b shows an STG that violates the no loop requirement: $t$ forms a loop with $p_3$. Hence, none of the two fulfil the preconditions for $t$-contraction.

   It should also be checked whether the *secure $t$-contraction requirement* is fulfilled: either $({}^\bullet t)^\bullet \subseteq \{t\}$ (either $t$ has no preplaces or every preplace of

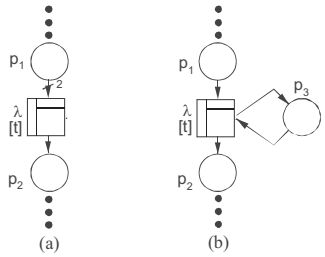


Figure 3.2. Violation of preconditions for $t$ contraction

$t$ has only $t$ as its posttransition); or ${}^\bullet(t^\bullet) = \{t\}$ and $\forall p \in t^\bullet : M_N(p) = 0$ (every postplace of $t$ has only $t$ as its pretransition and has no token at initial marking). A divining transition that violates this secure $t$-contraction requirement is *a non-secure transition*. Fig. 3.3 shows STGs that violate the no choice place in the pre place requirement: $({}^\bullet t)^\bullet = \{t, t_2\} \not\subseteq \{t\}$; the one in Fig. 3.3a also violates the no meeting place in post place requirement: ${}^\bullet(t^\bullet) = \{t_1, t\} \neq \{t\}$. This violation can cause a *backfiring* if $t$ is contracted; after $t$ is contracted, firing $t_1$ will enable $t_2$. The one in Fig. 3.3b violates the post place requirement: $\nexists p \in t^\bullet$. After firing $t$ in Fig. 3.3b, the net is dead. But, if $t$ is contracted, the net is never dead. The one in Fig. 3.3c violates the no marking in post place requirement: $M_N(p_2) = 1$. This violation can cause a dead net to become alive. After contracting $t$ in Fig. 3.3c, $t_2$ is enabled. Hence, none of the STGs in Fig. 3.3 fulfills the secure $t$-contraction requirements.

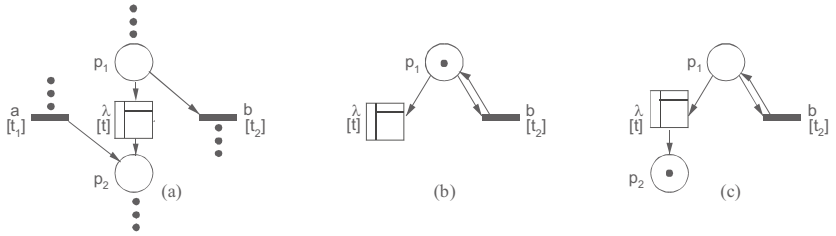


Figure 3.3. Violations of secure $t$-contraction requirements

If there is a precondition for $t$-contraction or a secure $t$-contraction requirement that cannot be fulfilled, then *backtracking* is performed, i.e. the $s_{edge}$-label of $t$ and every other $t \in T_i$ labelled with $s$ is restored; and $s$ is considered as a relevant signal for $C_i$.

2. If all the above requirements are fulfilled, then a secure transition contraction (see definition 2.2.2) can be performed.

3. After contraction, it should be checked whether the secure $t$-contraction has

caused a dynamic auto conflict or not. If it caused a dynamic auto conflict, then backtracking should be done. Fig. 2.11c shows a case where backtracking becomes necessary; because after secure $t$-contraction of $t$ labelled $\lambda$ (Fig. 2.11b), we have $t_2$ labelled with $b+$ and $t_4$ labelled with $b+$ in a structural auto conflict which is also a dynamic auto conflict.

4. If there is a redundant place (see definition 2.1.18) after contraction, it should be removed together with its incident arcs.

Steps 1-4 are performed on all divining transitions in STG $N_i$. At the end, after changing transitions with label $a \in Out_N, a \notin Out_i$ into input transitions, the component $C_i$ is obtained.

Secure transition contraction and redundant place deletion is the *admissible operations* for [VW02] method.

As an example, the environment STG $N$ for the wechselpuffer example from Beister is taken (see Fig.3.4a). For the *Aout*-component, only transitions with signal *Rout* are relevant – transitions with label *Aout* are $t_1$ and $t_8$; $t_7$ gives concession to $t_1$, $t_9$ to $t_8$; $t_7$ and $t_9$ are labelled with *Rout*. After making a copy of $N$ as initial *Aout*-component, all transitions with irrelevant signal label are silenced (see Fig.3.4b; the labels of the silenced transitions are kept for convenience). Note that trying to contract $t_3$ or $t_4$ at the beginning will cause backtracking, because they are non-secure transitions. After secure contraction of $t_6, t_{11}, t_{10}$, and $t_0$ (see Fig.3.4c), the redundant place $p_{2\_14\_13\_0\_1}$ is deleted. Now, $t_3$ and $t_4$ can be contracted securely (see Fig.3.4d). Further contraction of $t_2$ or $t_5$ is not possible, because they are non-secure transitions. Hence, backtracking should be done for both of them; i.e. *Rm* is added as relevant input signal. After backtracking, the end *Aout*-component is obtained (see Fig.3.4e).

The [VW02] algorithm has been proved to be correct according to the following definition 3.2.1.

**3.2.1.** Definition. A collection of deterministic components $(C_i)_{i\in I}$ is a *correct decomposition* or a *correct implementation* of a deterministic STG $N$, if the parallel composition $C$ of the $C_i$ is defined, $In_C \subseteq In_N$, $Out_C \subseteq Out_N$ and there is a relation $\mathcal{B}$ between the markings of $N$ and those of $C$ with the following properties.

1. $(M_N, M_C) \in \mathcal{B}$
2. For all $(M, M') \in \mathcal{B}$, we have:
   (a) If $a \in In_N$ and $M[a\rangle\rangle M_1$, then either $a \in In_C$ and $M'[a\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some $M_1'$ or $a \notin In_C$ and $(M_1, M') \in \mathcal{B}$.
   (b) If $x \in Out_N$ and $M[x\rangle\rangle M_1$, then $M'[x\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some $M_1'$.

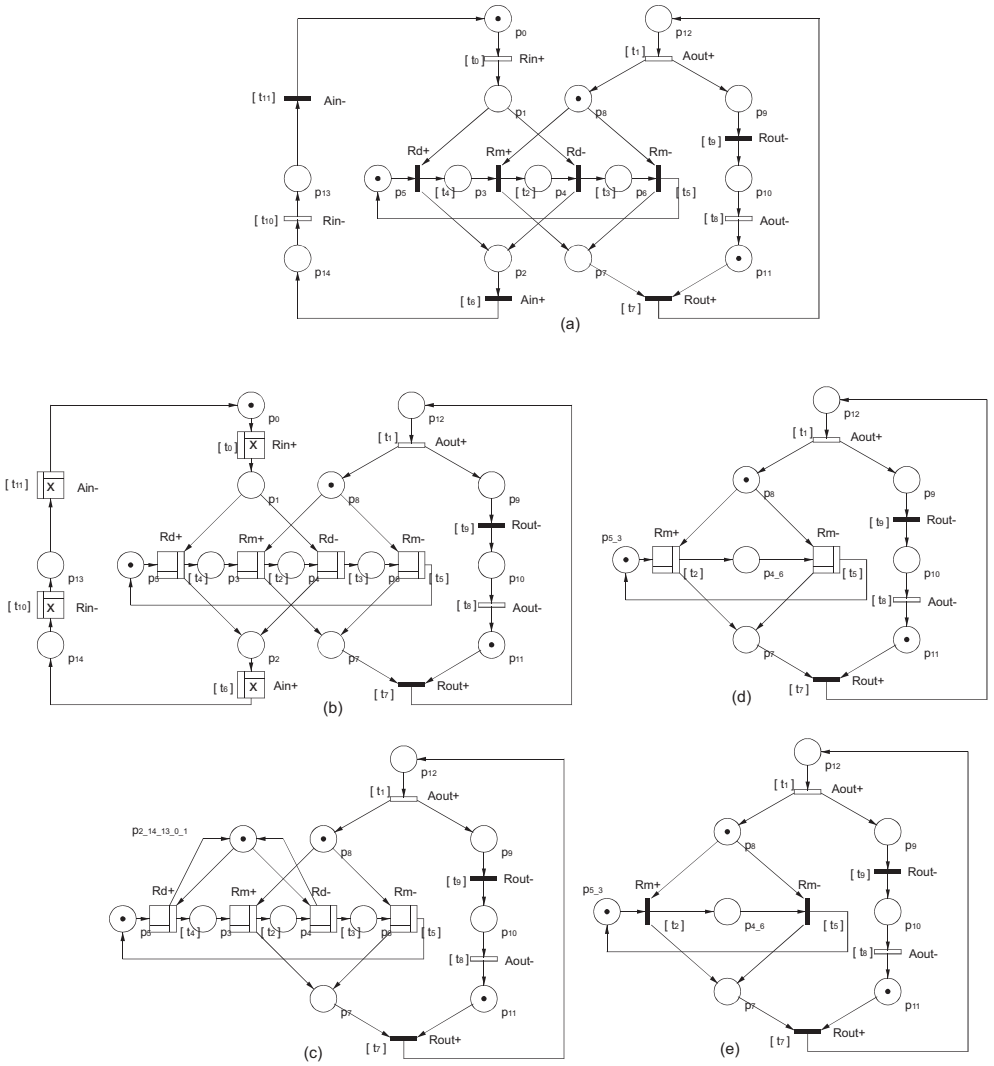


Figure 3.4. Finding *Aout*-component for wechselpuffer example

(c) If $x \in Out_C$ and $M'[x\rangle\rangle M'_1$, then $M[x\rangle\rangle M_1$ and $(M_1, M'_1) \in \mathcal{B}$ for some $M_1$.

(d) If $x \in Out_i$ for some $i \in I$ and $M' \mid_{P_i} [x\rangle\rangle$, then $M'[x\rangle\rangle$ (no *computation interference*).

Here, and whenever a collection $(C_i)_{i\in I}$ is encountered in the following, $P_i$ stands for $P_{C_i}$, $Out_i$ for $Out_{C_i}$ etc.

In the definition 3.2.1, $C$ is allowed to have fewer input signals than $N$; the reasons are as follows: There might be some input signals that are not relevant for producing the right outputs; whereas $N$ makes some assumptions on the environment regarding these inputs, $C$ does not – hence, the environment might produce these signals any time, but they are ignored. Such irrelevant input signals are called *globally irrelevant* inputs.

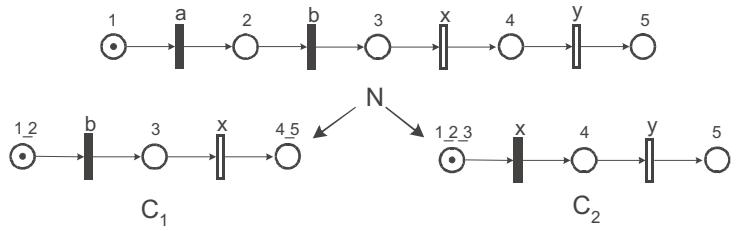


Figure 3.5. The globally irrelevant input $a$ is not implemented in $C_1$ and $C_2$

Fig. 3.5 shows a very simple example of an STG $N$ and a decomposition into two components $C_1$ and $C_2$ that can be constructed by the [VW02] algorithm and have $\mathcal{B} = \{(1, (1\_2, 1\_2\_3)), (2, (1\_2, 1\_2\_3)), (3, (3, 1\_2\_3)), (4, (4\_5, 4)), (5, (4\_5, 5))\}$ – here a marking of $N$ or a component is identified with its single marked place; the marking of $C$ is denoted by the marking of the components $(M_1, M_2)$.

Clause 2a of definition 3.2.1 says that an input allowed by the specification is also allowed by $C$ (or ignored). In Fig. 3.5, $b$ is an input of $N$ and also an input of $C$ ; If $(2)[b\rangle\rangle(3)$ in $N$, then $(1\_2, 1\_2\_3)[b\rangle\rangle(3, 1\_2\_3)$ in $C$ and $(3, (3, 1\_2\_3))$ is also in $\mathcal{B}$.

In Fig. 3.5, $a$ is only an input of $N$ but not of any component, but still the components are a correct implementation because $(1, (1\_2, 1\_2\_3)) \in \mathcal{B}$ and after $(1)[a\rangle\rangle(2)$, $(2, (1\_2, 1\_2\_3))$ is also in $\mathcal{B}$.

Clause 2b of definition 3.2.1 says that the specified outputs can be generated by $C$. In Fig. 3.5, for $(3, (3, 1\_2\_3)) \in \mathcal{B}$ and $x \in Out_N$; If $(3)[x\rangle\rangle(4)$ in $N$, then $(3, 1\_2\_3)[x\rangle\rangle(4\_5, 4)$ in $C$ and $(4, (4\_5, 4))$ is also in $\mathcal{B}$.

Clause 2c of definition 3.2.1 says that $C$ produces only the outputs which are specified in $N$. In Fig. 3.5, for $(3, (3, 1\_2\_3)) \in \mathcal{B}$ and $x \in Out_C$; If $(3, 1\_2\_3)$ $[x\rangle\rangle$ $(4\_5, 4)$ in $C$ , then $(3)[x\rangle\rangle(4)$ in $N$ and $(4, (4\_5, 4))$ is also in $\mathcal{B}$.

Remarkably, there is no clause requiring a match for inputs of $C$. If $M'[a\rangle\rangle M_1'$ for some input $a$, then either $M[a\rangle\rangle M_1$, in which case the uniquely defined $M_1'$ and $M_1$ match (in $\mathcal{B}$ after firing $a$), or the input is not specified; in the latter case, the environment is not supposed to supply it, such that this potential behaviour of $C$ which will never occur in an appropriate environment can be ignored, i.e. one that satisfies the assumption of the specification.

The usefulness of this feature is demonstrated by the simple example in Figure 3.6: $C_1$ and $C_2$ are an intuitively correct decomposition of $N$ (obtained by [VW02] algorithm), since together they answer an input $a$ by $x$ and a following $b$ by $y$, just as specified. But in $C_1 \parallel C_2$, which is just the disjoint union of $C_1$ and $C_2$, $b$ is enabled initially in contrast to $N$. Note that this implies that $N$ and $C_1 \parallel C_2$ are not language equivalent, as e.g. required in [Chu87a] [Chu87b].

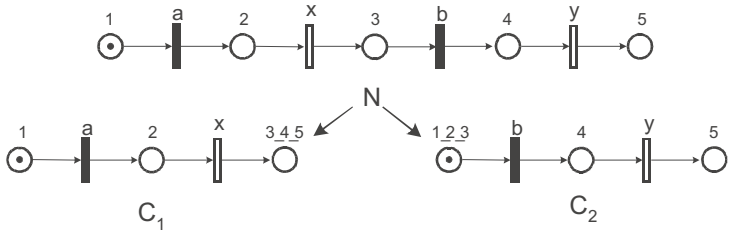


Figure 3.6. Specification does not need to simulate input of $C_1 \parallel C_2$

The fourth clause, 2d of definition 3.2.1 is a requirement that could easily be overlooked: if, in state $M'$, $C_i$ on its own could generate an output $x$ that is an input of some $C_j$, but not allowed there, then there simply exists no $x$-labelled transition enabled under $M'$ due to the definition of parallel composition; but $x$ is under the control of $C_i$, so it might certainly produce this output, and we must make sure that it is present in $C$, i.e. does not lead to a failure of $C_j$ for instance.

An example is shown in Figure 3.7. The parallel composition $C$ of $C_1$ and $C_2$ looks very much the same as $N$, and they certainly have the same language - they are even bisimilar. But putting circuits for $C_1$ and $C_2$ together, $C_1$ will possibly produce output $x$ after receiving input $a$; firing $x$ – which is an input transition in $C_2$ – after $a$ has been fired is not allowed in $C_2$ and can cause a failure in $C_2$. In $C$, $x$ cannot occur after $a$ alone; this occurrence is also not specified in $N$, and therefore $C_1$ and $C_2$ should not be a correct decomposition of $N$ – and they are indeed not one due to the fourth clause. The correct decomposition of $N$ is shown in Fig. 3.8.

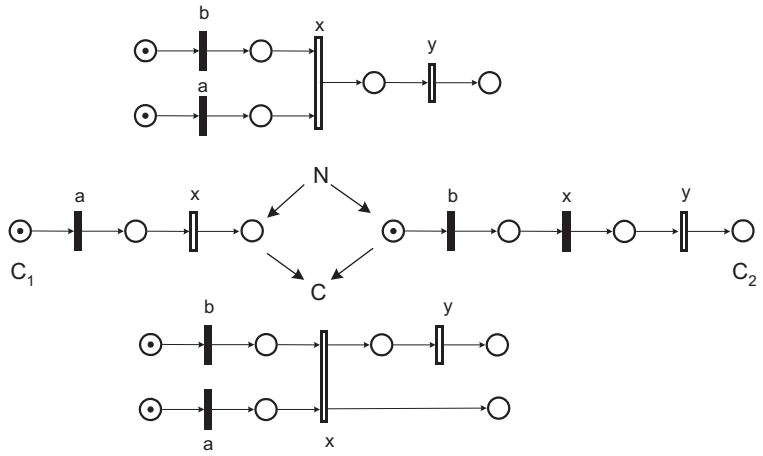


Figure 3.7. A computation interference example

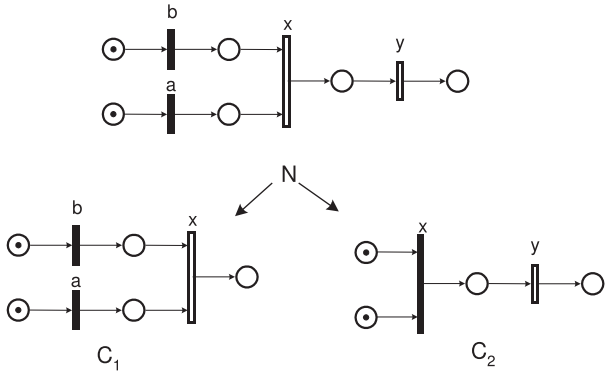


Figure 3.8. Correct decomposition of $N$ for example in the Fig. 3.7

# Chapter 4

# Improvements for [VW02] algorithm

The [VW02] algorithm requires the initial STG to have no auto-concurrency, no dynamic auto-conflict and no dummy transitions. Auto-concurrency of a signal is physically impossible. In [VW02], structural instead of dynamic conflict is checked for. This can be over cautious because there may actually be no dynamic conflict in spite of the structural one. The exclusion of dummy transitions also is too restrictive in practice. For example, the dummy transitions normally used for synchronization are deterministic and can be removed from the initial STG by secure $t$-contraction before decomposition.

To increase algorithm efficiency and achieve better results, backtracking should be avoided if possible. This can be achieved by changing the order of contraction, deleting loop-only dummy transitions and transforming a non-secure $t$-contraction into a secure one. Also efficiency of the algorithm can be increased by reuse of intermediate component.

Following are improvements that will increase applicability of the algorithm, give better decomposition results and increase algorithm efficiency.

## 4.1 Grouping and ordering divining transitions

The efficiency of the algorithms and the quality of their results depend on the order in which the transitions are contracted. Good strategies are those where all of the transition belonging to a group characterized by certain features are contracted one by one before contraction of another group is started. There are two useful groupings: grouping by the same signal label, and grouping of transitions in conflict with one another.

Grouping transitions by signal label is important in the case of backtracking. Backtracking is done by adding transitions with the same signal label $l(t)$ as transition $t$ whose removal caused backtracking. Therefore, contracting transitions with the same signal label has an advantage that by backtracking – adding back transitions with signal label $s$, the algorithm needs only to continue from

the intermediate component before the group with signal $s$ is contracted. This strategy can increase algorithm efficiency, because there is no need to begin from the initial net or from any other intermediate component except the one from which the first transition with signal label $s$ was contracted. Another advantage of grouping transitions by signal label: intermediate decomposition results can also be reused for other components. For example, the intermediate component after contracting globally irrelevant input signals can be reused as the initial component for all the components; i.e. there is no need to begin with the initial net. For more detail about reuse, see section 4.2.

Recall that backtracking should be done if a transition $t$ cannot be contracted. This is the case if:

1. $t$ does not fulfill the preconditions for contraction due to an arc weight $> 1$ or a loop on a divining transition.

2. $t$ is a non-secure transition, or

3. contracting $t$ causes a new structural auto-conflict.

One can see that backtracking is mostly due to transitions in conflict. Hence, it might also be useful, to begin by contracting transitions which are in conflict with a group of transition with the same signal label. So, if backtracking is needed it can be done directly, before other transitions in the group are contracted. With this strategy, some contraction can be saved. This strategy however can lead to unnecessary backtracking as shown below.

Grouping transitions with the same signal label can only increase algorithm efficiency; but grouping transitions which are in conflict can also yield better decomposition results. This grouping is imposed to find a loop-only divining transition or a duplicate transition which can only be found with correct grouping and ordering of divining transitions; i.e. the divining transitions which are not in conflict should be contracted first. Without grouping, unnecessary backtracking may have to be done, which results in a larger component than needed. For example, see Fig. 4.12a: if $t_3$ is contracted first, then backtracking should be done because $t_3$ is a non-secure transition. Also, in Fig. 4.12c: if $t_7$ or $t_4$ is contracted first, then backtracking should be done because, $t_7$ and $t_4$ are non-secure transitions. Therefore, transitions in conflict should be contracted last.

In some cases, after contracting other transitions, transitions which were non-secure become secure. This case needs a reordering approach: if there is a secure $t$-contraction requirement that cannot be fulfilled during contraction, then the transition is *reordered* by attaching it to the end of the queue of transitions to be contracted. The same $t$ is reordered again, only if the queue has become shorter since the last reordering of $t$, i.e. if at least one other divining transition has been contracted; if not, backtracking should be done, because $t$ cannot be securely contracted. The example for this case is shown in the wechselpuffer example

(Fig. 3.4). In the example, if $t_3$ or $t_4$ is contracted first, then backtracking should be done, because $t_3$ and $t_4$ are non-secure transitions. But, after contracting other transitions, $t_3$ and $t_4$ become secure; i.e. unnecessary backtracking are avoided.

From the above discussion, the general procedure to assure better decomposition results and good efficiency is as follows: First, group divining transitions that have the same signal. From each group, divining transitions that are in conflict are ordered so as to be contracted first. During contraction, if a transition $t$ is non-secure, then insert $t$ and all the transitions that have the same signal label at the end of the queue. When all divining transitions (except the newly inserted ones in the queue) have been contracted, transitions in the queue which are in conflict are grouped together and ordered so as to be contracted last. If there is a secure $t$-contraction requirement that cannot be fulfilled during contraction, then the transition is reordered (placed at the end of the queue of transitions to be contracted). The same $t$ is reordered again, only if the queue has become shorter since the last time $t$ was moved; if not, backtracking should be done, because $t$ cannot be securely contracted.

## 4.2 Reuse of intermediate components

Instead of beginning each output block from the initial STG, one can sometimes reuse the intermediate component from an output block as the initial component for other output blocks. This strategy will be called *reuse of intermediate components* strategy. Reuse of intermediate components can increase the efficiency of decomposition algorithm. It is suggested in [VW02] and [VK07], and proved to be correct in [SV05]. The intermediate component from an output block can be reused as initial component for other output blocks only if they have common irrelevant signals. The intermediate component reached after removing dummy transitions and globally irrelevant signals can be reused by all output blocks in a partition; it will be called *the global intermediate component*.

If reuse of intermediate components is planned, then from the copy of the initial STG, only transitions with common irrelevant signal labels are silenced. As a consequence, if a transition cannot be contracted and backtracking should be done, it does not mean that the signal $l(t)$ should be added as relevant for all output blocks. For example, in the case that $t$ cannot be contracted due to a new structural auto-conflict between $t_1$ and $t_2$, only output blocks where $l(t_1)$ is relevant should add $l(t)$ as relevant signal.

An example of a global intermediate component is locked2 in Fig. 4.13. Fig. 4.13a shows the initial net with the dummy transitions $t_{25}$ and $t_{26}$ already silenced, and its global intermediate component in Fig. 4.13b. But on inspection, input signals $a$ and $D$ seem to be globally irrelevant. This is tested by extracting the $rdy$-component (Fig. 4.13c,d,and Fig. 4.14a with removal of $acka$, $ackd$, and $s$). But trying to contract $t_0$ ($a+$) or $t_1$ ($D+$) – see Fig. 4.14c and d – results in a new

structural auto-conflict between $t_3$ and $t_{14}$ (both labelled $c+$). Hence, $a$ and $D$ are *not* globally irrelevant. As shown in Fig. 4.14b, they are relevant signals of the $rdy$-component.

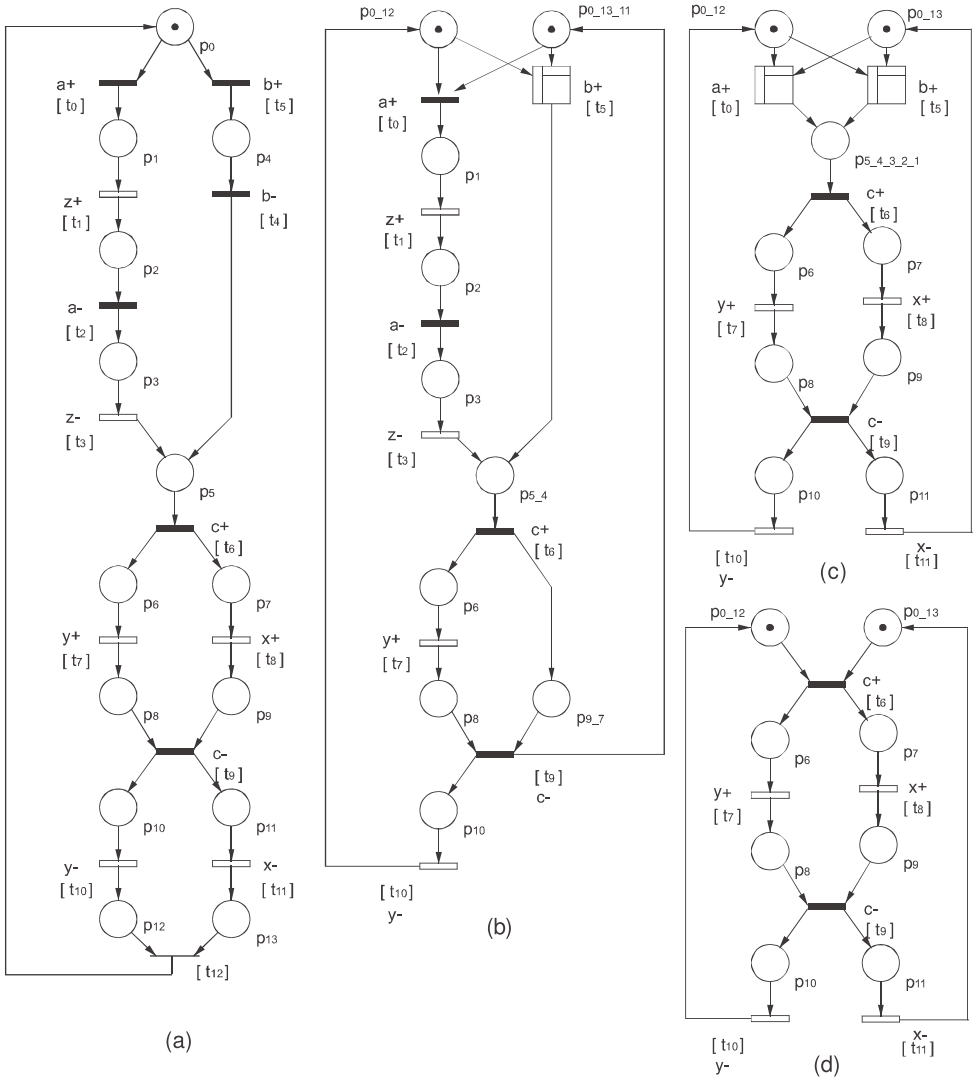


Figure 4.1. Reuse of intermediate component

Another example for the reuse of an intermediate component is shown in Fig. 4.1. In the initial net (Fig. 4.1a), one finds the relevant signals $c$ for the $x$- and the $y$-component and $a$ for the $z$-component. Hence, the potentially irrelevant signals are $a, b, z, y$ for the $x$-component, $a, b, z, x$ for the $y$-component and $b, c, x, y$ for the $z$-component. The only potential globally irrelevant signal, then seems to be $b$.

An apparently global intermediate component is obtained by deleting dummy transition $t_{12}$ and the apparently globally irrelevant signal $b$. But after contract-

ing $t_4$, $t_5$ cannot be deleted because it has become a non-secure transition(see Fig. 4.1b).

A good candidate for reuse is shown in Fig. 4.1d, obtained by contracting $z+$, $z-$, and $a-$ from Fig. 4.1b, and $a+$ and $b+$ from the resulting intermediate component Fig. 4.1c. For further decomposition, the $xy$-component in Fig. 4.1d can be used for extracting the $y$-component, and reused for obtaining the $x$-component. Another possibility for reuse is between $y$- and $z$-component, which has common irrelevant signals $x, b$. But this does not bring too much advantage, because after contracting $t_4$ and transitions $t_8$ and $t_{11}$ with signal label $x$, $t_5$ cannot be contracted because it has become insecure , hence backtracking must restore the $b-$ transition $t_4$, $x+$ transition $t_8$ and $x-$ transition $t_{11}$ (see Fig. 4.1b).

The greatest advantage is got by *reuse the intermediate component* after contracting transitions with signal label $z, a, b$. After contracting transitions with signal label $z$ $(t_1, t_3)$, transitions with signal label $a, b$ are placed at the end of a queue because $t_0$ and $t_5$ are non-secure transitions. The transitions with signal label $a, b$ are ordered such that transitions which are not in conflict, are contracted first; i.e. $(t_2, t_4, t_0, t_5)$. After contracting $t_2$ and $t_4$, $t_5$ is found to be a duplicate divining transition of $t_0$ and vice versa (see Fig. 4.1c). $t_5$ can be deleted and $t_0$ can be contracted securely. Hence, as a rule of thumb for *reuse intermediate component*: choose output blocks which have the largest number of common irrelevant signals. The result is shown in Fig. 4.1d.

Note that the causes of backtracking are the arc weight, the choice place, and the meeting place. Hence in a marked graph (a subclass of ordinary nets), which has no choice or meeting place, there will be no backtracking; i.e. the decomposition result is deterministic; it depends only on the output block partition. This has been proved formally in [SVJ05].

## 4.3 Deleting loop-only dummy transitions

If we securely contract a dummy transition $t$, not only may $t$ become a *loop-only transition* – a transition $t$ such that ${}^\bullet t = t^\bullet$, and $t$ forms a loop with every $p \in {}^\bullet t$ – but another dummy transition $t'$ might also become a loop-only transition. Hence we can also delete $t'$ and all its incident loop arcs as we normally do during secure contraction; for we don't care whether the loop-only dummy transition fires or not, or how many times it fires. Fig. 4.2b shows an STG with a dummy transition $t'$ that became a loop-only transition when $t$ was securely contracted in the STG in Fig. 4.2a. We can safely delete $t'$, because we care only that $c+$ fires.

In an initial STG, there should be no loop-only transitions, except loop-only dummy transitions that we can remove safely. If there is a loop-only input transition, then the STG is not consistent. If it is a loop-only output transition, then the STG is not only inconsistent, but also not hazard free due to dynamic

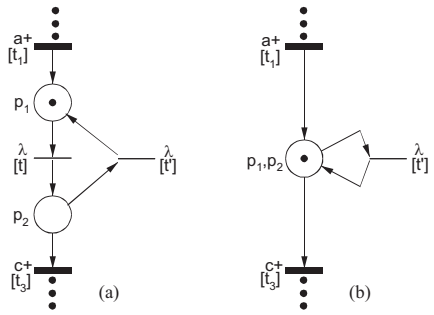


Figure 4.2. STG (a) initial, (b) after contracting $t$

input-output conflicts if it is in conflict with input transition.

## 4.4 Deleting duplicate transitions

One of the improvements in [VK07], [VK05] over [VW02] is the deletion of duplicate divining transitions. But they may also be duplicate input transitions: a transition $t$, and another transition $t'$ with the same label $l(t') = l(t)$ which has arcs to and from the same places and with the same arc weight as $t$. Such duplicate transition are redundant and can be removed. If they are not removed, then there will be auto-conflicts which force the decomposition algorithm to backtrack. Backtracking will result in a larger component, and this is avoided by deleting all duplicate transitions with the same $s_{edge}$ label except one.

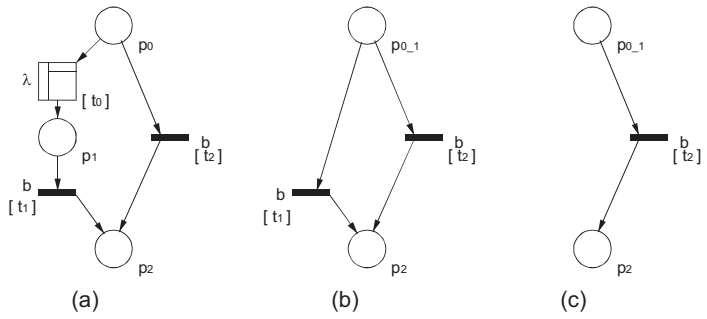


Figure 4.3. (a) a net $N$ (b) after contracting transition $t_0$, (c) after deleting duplicate transition $t_1$

Fig. 4.3 shows an example of a duplicate input transition $s_{edge}$ label. Note that there will be no duplicate output transitions unless they are already present in the initial net.

**4.4.1.** Proposition. *Deleting duplicate input transitions is an admissible operation.*

**Proof:** Having the same input places, $t$ and $t'$ will be enabled together. Being in dynamic auto-conflict, either $t$ but not $t'$, or $t'$ but not $t$ will be fired. Because $t$ and $t'$ have the same output places, the same marking will be reached regardless of whether $t$ or $t'$ is fired. Obviously, then, removal of $t$, or alternatively of $t'$ will make no difference; i.e. the removal of one of the two transitions will introduce neither new conflicts nor cause new concessions to be given to output transitions by divining concessions. In fact, deleting duplicate input transition $t$ of input transition $t'$ also transforms the intermediate component $N$ into a bisimilar intermediate component $N'$; because $M[l(t)\rangle\rangle M_1$ in $N$ can be simulated by $M'[l(t')\rangle\rangle M_1'$ in $N'$; and $M'[l(t')\rangle\rangle M_1'$ in $N'$ can be simulated by either $M[l(t)\rangle\rangle M_1$ or $M[l(t')\rangle\rangle M_1$ in $N$. □

Furthermore, the results after deleting duplicate input transitions are synthesizable; i.e. it does not introduce new auto-concurrency, new auto-conflicts, or new input-output conflicts and it preserves consistency.

## 4.5 Transition fusion

Only a deterministic STG is synthesizable. Therefore, it is the specifier's responsibility to ensure that there is no dynamic auto-conflict in his or her specification. Therefore it is suggested to ask the specifier whether a specified auto-conflict is dynamic or not. If it is dynamic, then the specifier must either change his or her specification or allow internal transitions to be inserted.

If it is just a structural conflict, then the specifier can leave the specification as it is, to be handled by the improved decomposition algorithm [VK07] or allow the transition in structural conflict to be fused. Transition fusion [KVWB04] yields a more straightforward specification by fusing pseudo auto-conflicts. Each transition which is in structural auto-conflict should have so-called control places; otherwise the conflict is dynamic. Transition fusion fuses transitions which are in structural conflict, but have control places, into a new transition, and changes the original transitions into dummy ones. Transition fusion can be applied to the initial net or an intermediate component (if a new structural auto-conflict arises).

The following is the procedure to fuse two transitions in structural auto-conflict, but with control places. Let $N$ be an STG in which $t_1, t_2 \in T$ are labelled with the same $s_{edge}$, ${}^\bullet t_1 \cap {}^\bullet t_2 = S \neq \emptyset$, ${}^\bullet t_1 - S \neq \emptyset$, ${}^\bullet t_2 - S \neq \emptyset$; and for all conflict places $p_c \in S$, let $W(p_c, t_1) = 1$, $W(p_c, t_2) = 1$. We will have $N_{fuse}$ after fusing $t_1$ and $t_2$ of $N$ as follows:

1. Copy $N$, to obtain an initial $N_{fused}$.

2. Add new merge places from the control places, add a new place $p_{new}$:
   $P_{fused} = \{p \mid p \in P\} \cup \{(p_1, p_2) \mid p_1 \in {}^\bullet t_1 - S, p_2 \in {}^\bullet t_2 - S\} \cup \{p_{new}\}$
   $M_{N_{fused}}((p_1, p_2)) = M_N(p_1) + M_N(p_2)$

$W_{fused}((p_1, p_2), t) = W(p_1, t) + W(p_2, t), \forall t \in T - \{t_1, t_2\}$
$W_{fused}(t, (p_1, p_2)) = W(t, p_1) + W(t, p_2), \forall t \in T - \{t_1, t_2\}$
$M_{N_{fused}}(p_{new}) = 0$

3. Add new fuse transition $t_{new}$ with the same label as $t_1$, add an arc from $t_{new}$ to $p_{new}$; change the label of $t_1$ and $t_2$ to the empty label:
$T_{fused} = T \cup \{t_{new}\}$
$l_{fused}(t_{new}) = l_N(t_1)$
$W_{fused}(t_{new}, p_{new}) = 1$
$l_{fused}(t_1) = \lambda$
$l_{fused}(t_2) = \lambda$

4. Introduce causality to the new fuse transition by adding an arc from each merge place to $t_{new}$:
$W_{fused}((p_1, p_2), t_{new}) = 1$

5. Add arcs from the choice places $p_c \in S$ to $t_{new}$, remove the arcs from the choice places to $t_1$ and $t_2$; and add arcs from $p_{new}$ to $t_1$ and $t_2$:
$W_{fused}(p_c, t_{new}) = 1$
$W_{fused}(p_c, t_1) = 0$
$W_{fused}(p_c, t_2) = 0$
$W_{fused}(p_{new}, t_1) = 1$
$W_{fused}(p_{new}, t_2) = 1$

After transition fusion, $t_1$ and $t_2$ are securely contracted. If $t_1$ or $t_2$ must not be contracted because contraction would not be secure, then we undo the fusion. An example follows.

Fig. 4.4a is the VME bus controller STG (repeated again for convenience). The STG has a structural auto-conflict between $lds+$-labelled transitions $t_3$ and $t_{10}$ with control places $p_2$ and $p_{12}$. Hence we fuse the transitions as follows: First, we add the merge place $p_{2\_12}$ ($M_{N_{fused}}(p_{2\_12}) = M_N(p_2) + M_N(p_{12}) = 0$), arcs from $t_1$ and $t_{12}$ to $p_{2\_12}$, and the new place $p_{17}$ ($M_{N_{fused}}(p_{17}) = 0$). After that, we add the new transition $t_{17}$ with label $lds+$, add an arc from $t_{17}$ to $p_{17}$, and change the labels of $t_3$ and $t_{10}$ to $\lambda$ (see Fig. 4.5a). We also move the causality to the new fuse transition by adding an arc from $p_{2\_12}$ to $t_{17}$ (see Fig. 4.5b). Finally, we add an arc from $p_{16}$ to $t_{17}$, remove the arcs from $p_{16}$ to $t_3$ and $t_{10}$, and add arcs from $p_{17}$ to $t_3$ and $t_{10}$. Fig. 4.5c shows the STG after transition fusion. Fig. 4.4b shows the STG after secure contraction of $t_3$ and $t_{10}$.

If contraction of $t_1$ and $t_2$ introduces another structural auto-conflict with control places, then repeat transition fusion. If the pre transitions of the control places are a subset of, or the same as the pre transitions of the merge place from the previous fusion, and if the post transitions of the control places are only the transitions in structural auto-conflict, then we don't need to create another merge place for the current fusion; because causality is already preserved by the merge

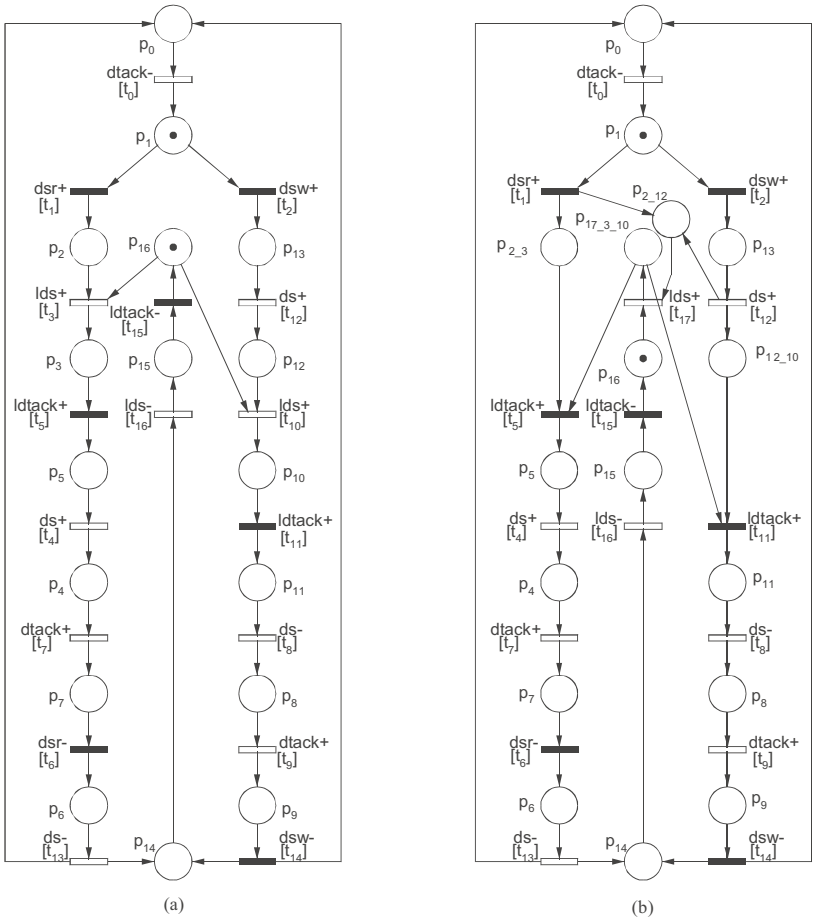


Figure 4.4. VME STG (a) initial, (b) after fusing and then contracting $t_3$ and $t_{10}$

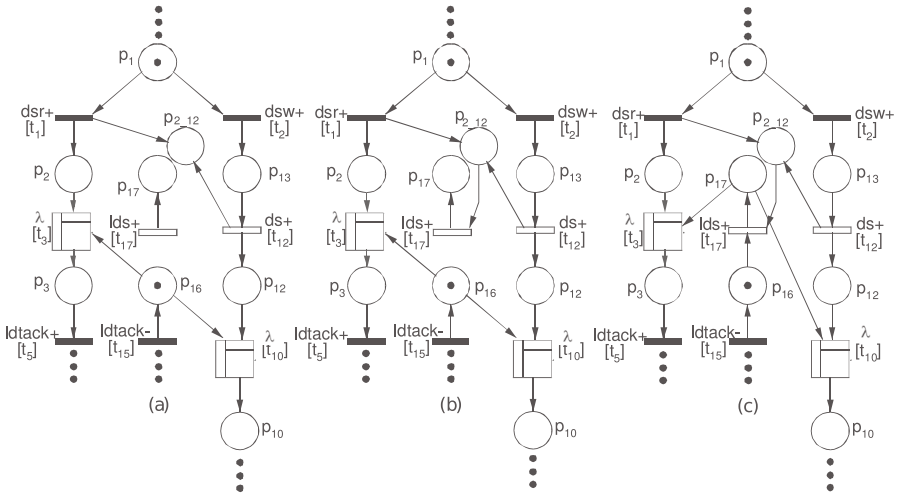


Figure 4.5. Steps of the fusion of $t_3$ and $t_{10}$

place from the previous fusion. Let $N$ be an STG in which $t_3, t_4 \in T_{fused}$ are labelled with the same $s_{edge}$, ${}^\bullet t_3 \cap {}^\bullet t_4 = S \neq \emptyset$, ${}^\bullet t_3 - S \neq \emptyset$, ${}^\bullet t_4 - S \neq \emptyset$; and for all conflict places $p'_c \in S$, let $W(p'_c, t_3) = 1$, $W(p'_c, t_4) = 1$. We need no new merge place if $p_3 \in {}^\bullet t_3 - S$, $p_4 \in {}^\bullet t_4 - S$, ${}^\bullet p_3 \cup {}^\bullet p_4 \subseteq (p_1, p_2)^\bullet$, $p_3{}^\bullet = t_3$, and $p_4{}^\bullet = t_4$ and we fuse $t_3, t_4$ of $N_{fused}$ as follows:

1. Copy $N_{fused}$ to obtain an initial $N'_{fused}$.

2. Add new place $p'_{new}$:
   $P'_{fused} = P \cup \{p'_{new}\}$
   $M_{N'_{fused}}(p'_{new}) = 0$

3. Add new transition $t'_{new}$ with the same label as $t_3$, add an arc from $t'_{new}$ to $p'_{new}$; change the label of $t_3$ and $t_4$ to $\lambda$:
   $T'_{fused} = T \cup \{t'_{new}\}$
   $l'_{fused}(t'_{new}) = l_N(t_3)$
   $W'_{fused}(t'_{new}, p'_{new}) = 1$
   $l'_{fused}(t_3) = \lambda$
   $l'_{fused}(t_4) = \lambda$

4. Add arc from choice place to $t'_{new}$, remove the arcs from choice places $p'_c \in S$ to $t_3$ and $t_4$; and add arcs from $p'_{new}$ to $t_3$ and $t_4$:
   $W'_{fused}(p'_c, t'_{new}) = 1$
   $W'_{fused}(p'_c, t_3) = 0$
   $W'_{fused}(p'_c, t_4) = 0$
   $W'_{fused}(p'_{new}, t_3) = 1$
   $W'_{fused}(p'_{new}, t_4) = 1$

The STG in Fig. 4.4b now has a new structural auto-conflict between the *ldtack+*-labelled transitions $t_5$ and $t_{11}$ with control places $p_{2\_3}$ and $p_{12\_10}$. Hence we can fuse these transitions, but this time we don't need to create a new merge place because ${}^\bullet p_{2\_3} \cup {}^\bullet p_{12\_10} = {}^\bullet p_{2\_12} = \{t_1, t_{12}\}$; and $p_{2\_3}{}^\bullet \cup p_{12\_10}{}^\bullet = \{t_5, t_{11}\}$ which are the transitions in structural auto-conflict (causality is already preserved by the merge place from the previous contraction). The transition fusion procedure is easier. We need only to add a new place $p_{18}$ ($M_{N'_{fused}}(p_{18}) = 0$), add the new transition $t_{18}$ with label *ldtack+*, add an arc from $t_{18}$ to $p_{18}$, and change the labels of $t_5$ and $t_{11}$ to $\lambda$. Finally, we add an arc from $p_{17\_3\_10}$ to $t_{18}$, remove the arcs from $p_{17\_3\_10}$ to $t_5$ and $t_{11}$, and add arcs from $p_{18}$ to $t_5$ and $t_{11}$. Fig. 4.6a shows the STG after transition fusion. Fig. 4.6b shows it after secure contraction of $t_5$ and $t_{11}$. There is no other structural auto-conflict in the specification. Hence, no more transition fusion is needed.

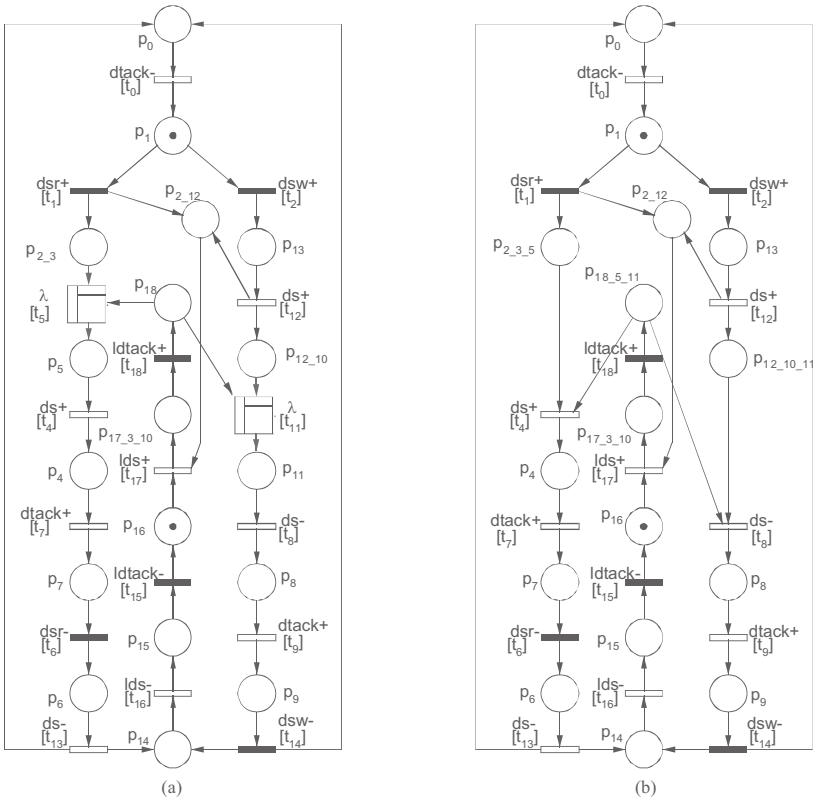


Figure 4.6. VME STG. (a) after fusing $t_5$ and $t_{11}$, (b) after contracting $t_5$ and $t_{11}$

In an intermediate component, there may be a new structural auto-conflict between transitions $t_1$ and $t_2$ such that $t_1$ and $t_2$ do not give concession or deconcession to any output transition; such a conflict is called a *new pseudo auto-conflict*.

In a FC net, if there is a dynamic auto-conflict in the initial net, then the specifier can change his or her specification by transition fusion; or if there is a new pseudo auto-conflict in the intermediate component, then transition fusion can help to obtain a better end component.

The procedure is almost the same as above: Let $N$ be an STG in which $t_1, t_2 \in T$ are labelled with the same $s_{edge}$, ${}^\bullet t_1 \cap {}^\bullet t_2 = \{p\}$, ${}^\bullet(p^\bullet) = \{p\}$, $t_1, t_2$ of $N$ is fused as follows:

1. Copy $N$ to obtain an initial $N'$.

2. Add new place $p'_{new}$:
   $P' = P \cup \{p'_{new}\}$
   $M_{N'}(p'_{new}) = 0$

3. Add new transition $t'_{new}$ with the same label as $t_1$, add an arc from $t'_{new}$ to $p'_{new}$; change the label of $t_1$ and $t_2$ to $\lambda$:
   $T' = T \cup \{t'_{new}\}$
   $l'(t'_{new}) = l_N(t_1)$
   $W'(t'_{new}, p'_{new}) = 1$
   $l'(t_1) = \lambda$
   $l'(t_2) = \lambda$

4. Add arc from choice place $p$ to $t'_{new}$, remove the arcs from $p$ to $t_1$ and $t_2$; and add arcs from $p'_{new}$ to $t_1$ and $t_2$:
   $W'(p, t'_{new}) = 1$
   $W'(p, t_1) = 0$
   $W'(p, t_2) = 0$
   $W'(p'_{new}, t_1) = 1$
   $W'(p'_{new}, t_2) = 1$

In Fig. 4.7a, $a$ and $b$ are globally irrelevant signals. To find the global intermediate component $t_0, t_{10}, t_1$, and $t_{12}$ should be contracted. But contracting these transitions introduces a new structural auto-conflict between $t_3$ and $t_2$ (see Fig. 4.7c). This conflict is a new pseudo auto-conflict because $t_3$ and $t_2$ do not give concession or deconcession to any output transition. Hence, transition fusion could be applied. The new place $p_{15}$ and the new transition $t_{16}$ with label $c+$ are added. Arcs from $p_{0\_1\_2}$ to $t_{16}$, from $t_{16}$ to $p_{15}$ and from $p_{15}$ to $t_3$ and $t_2$ are added. The arcs from $p_{0\_1\_2}$ to $t_3$ and $t_2$ are removed, and $t_3$ and $t_2$ are silenced (see Fig. 4.7d). The net after contracting $t_3$ and $t_2$ is shown in Fig. 4.7e, which is smaller than if backtracking due to pseudo new auto-conflict were done.

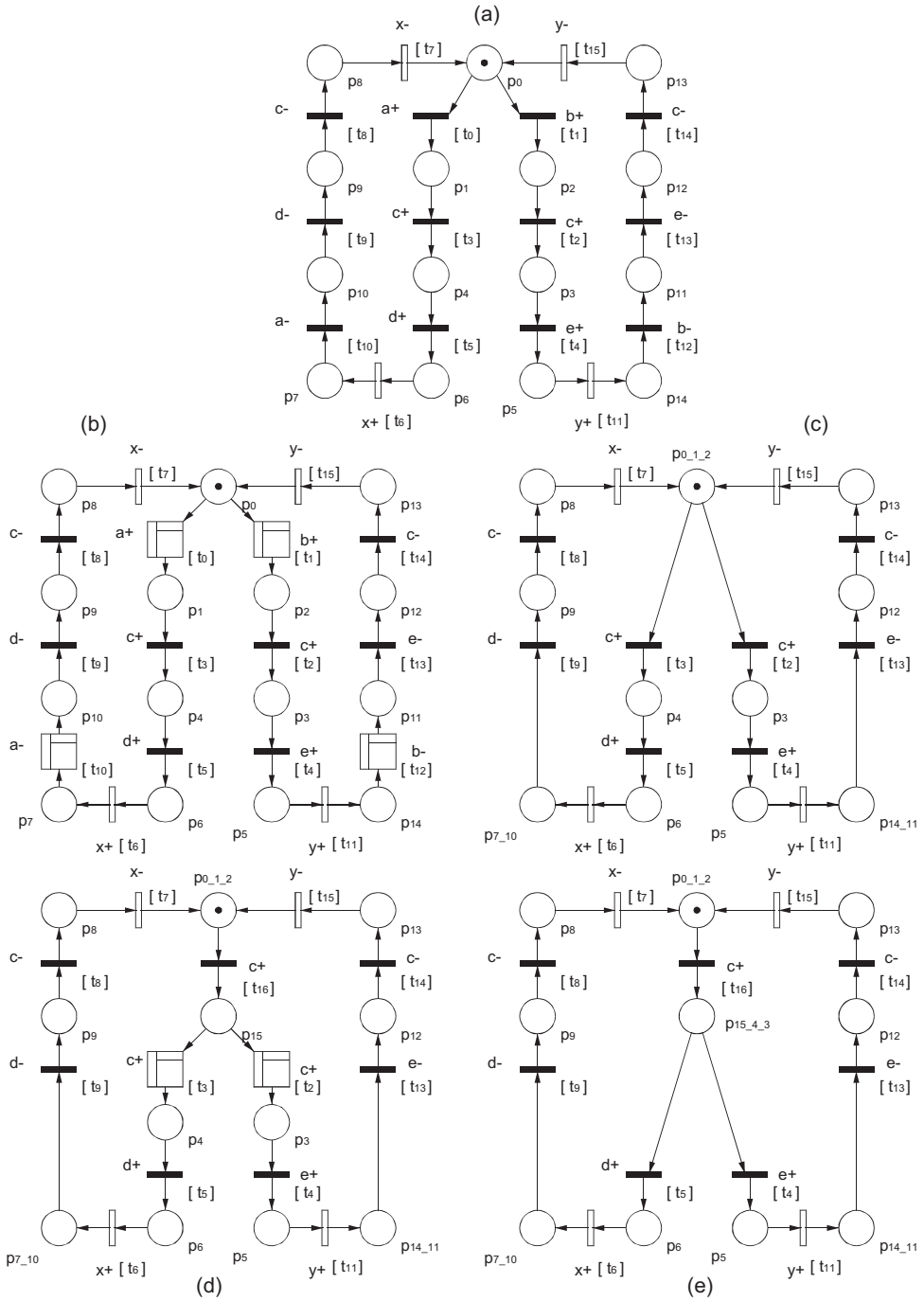


Figure 4.7. New pseudo auto-conflict example

## 4.6 Inserting internal signals

There are cases where an internal signal should be inserted into the specification (see also [Wol97]). For example, if there is a dynamic auto-conflict between output transitions, then the circuit should arbitrate between them. By arbitration, the internal cicuit should first decide which of the simultaneous requests will be granted. To find this arbitration component, an internal signal needs to be inserted into the specification; because it is not allowed to decompose a specification with a dynamic auto-conflict.

Now, it is proposed that such a component could be further decomposed (even if it is responsible for only a single output signal) into smaller subcomponents that communicate, also using newly introduced signals that remain internal to the larger component.

The internal transition inserted should not enable or disable an input transition because the environment cannot observe the internal transition and such enabling or disabling can cause circuit malfunction. To have a safe insertion without changing the interface of the system, transition refinement is introduced for internal signal insertion. The output transition is refined into an internal transition which gives concession to the output transition. The signal label for the internal transition should be consistent. Therefore, toggle transitions are introduced here for the internal signals. Toggle transitions were suggested by [VYCLdM94]. Fig. 4.8 shows the refinement of a toggle transition into a pair of edge transitions.

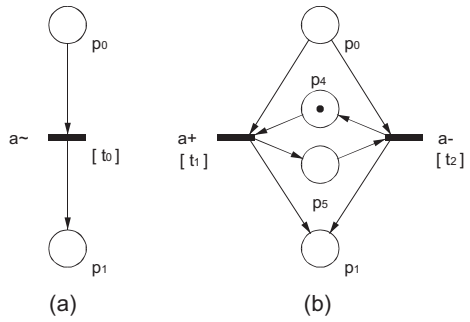


Figure 4.8. (a) toggle transition $t_0$ (b) refinement of $t_0$

The only constraint for inserting internal toggle transitions is that output transitions should not be in structural input-output conflict with another transition. Formally: for output transitions $t_1, t_2$ in dynamic auto-conflict – $t_1, t_2$ not in structural input-output conflict with another transition $t$ –, $t_1$ is refined into $t_{1\_int}$ with $l(t_{1\_int}) = 'int_1 \sim'$ , $p_{1\_int}$ with $M_N(p_{1\_int}) = 0$ and $t_{1\_out}$ with $l(t_{1\_out}) = l(t_1)$; $t_2$ is refined into $t_{2\_int}$ with $l(t_{2\_int}) = 'int_2 \sim'$, $p_{2\_int}$ with $M_N(p_{2\_int}) = 0$ and $t_{2\_out}$ with $l(t_{2\_out}) = l(t_2)$. Add arcs from $t_{1\_int}$ to $p_{1\_int}$, from $t_{2\_int}$ to $p_{2\_int}$, from $p_{1\_int}$ to $t_{1\_out}$, and from $p_{2\_int}$ to $t_{2\_out}$.

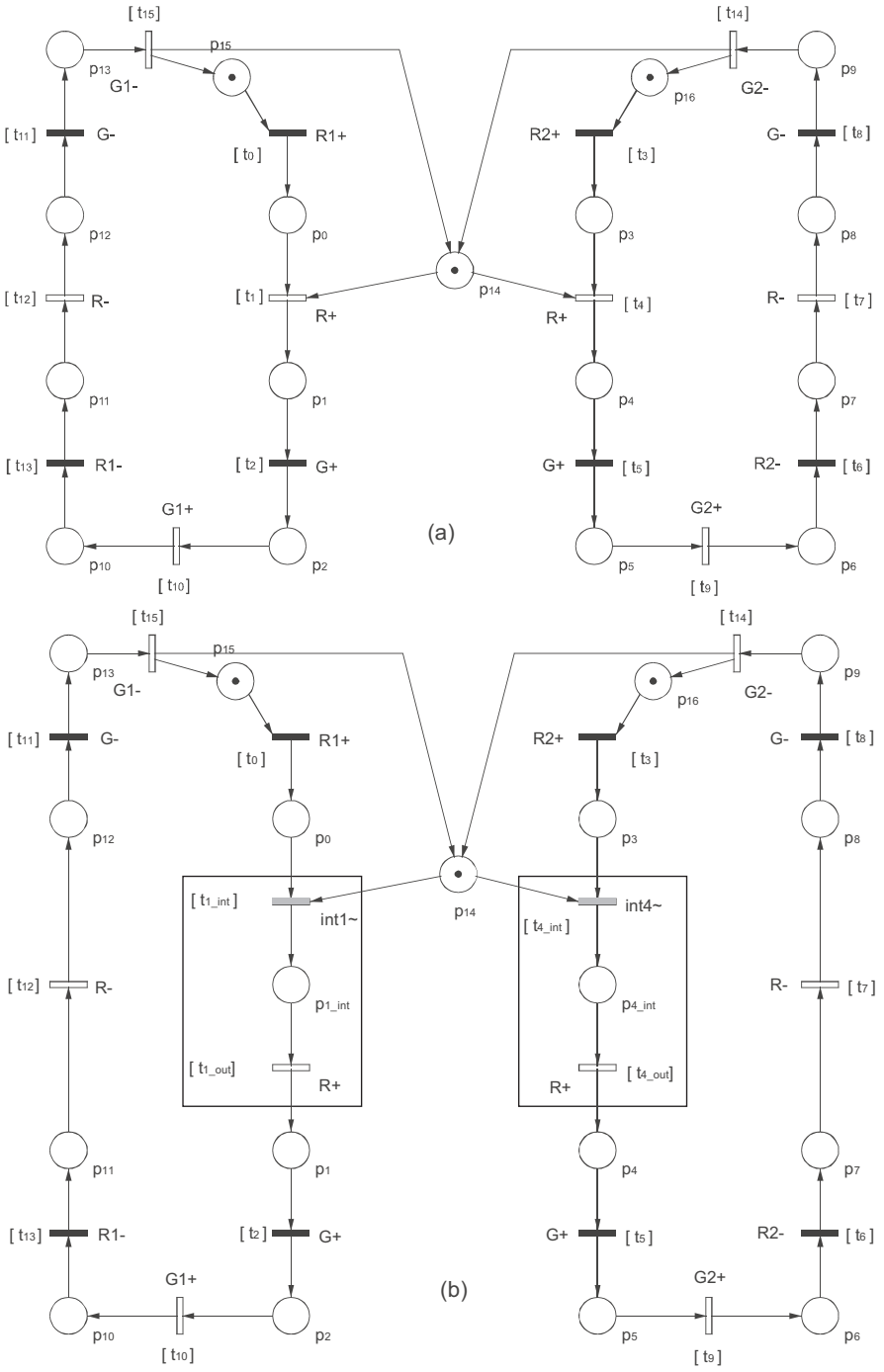


Figure 4.9. (a) NEI-arbiter [Wol97] (b) adding internal toggle transition to NEI-arbiter

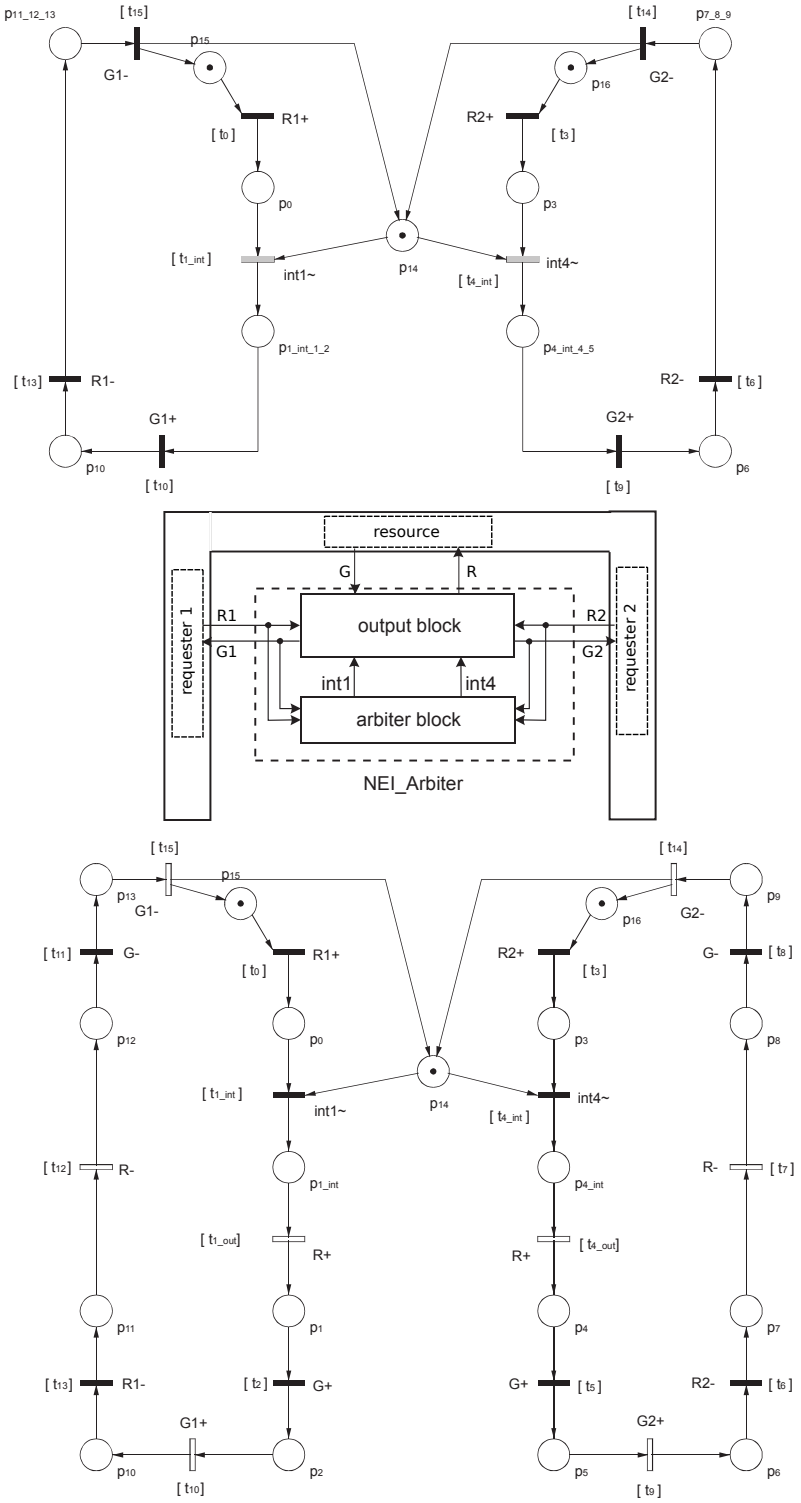


Figure 4.10. Decomposed NEI_arbiter from Fig. 4.9b into the arbitration net (above) and the residual net (below)

In the following example, the NEI_arbiter from [Wol97] has a structural auto-conflict between output transitions $t_1$ and $t_4$ (see Fig. 4.9a). $t_1$ is refined into $t_{1\_int}$ and $t_{1\_out}$. $t_4$ is refined into $t_{4\_int}$ and $t_{4\_out}$ (see Fig. 4.9b). After decomposition, the circuit is divided into two blocks: the one that produces outputs $R, G1$, and $G2$ for the environment, and the arbitration block (see Fig. 4.10). This division is important, because the arbitration circuit needs special treatment for implementation and it is normally available as a library element. With internal signal insertion and decomposition, the designer does not need to reinvent the arbiter; he or she just take the one from the library and connects it to the rest of the circuit.

Inserting internal toggle transitions is easy and is permissible in most cases. But the resulting circuit maybe more complex than if internal edge transitions were inserted. The problem with inserting internal edge transition is where to insert the opposite edge; because such insertion should preserve consistency and the interface of the circuit – i.e. it should not give concession or deconcession to an input transition. In the case of the NEI_arbiter example, inserting the internal edge transitions is possible ( see [Wol97] ).

## 4.7 Securing non-secure t-contractions

Before contracting a transition, we should check whether the contraction would be secure; if it isn't, we should backtrack. But in some cases, non-secure contractions can be made secure by splitting places and duplicating transitions, so that backtracking can be avoided (see also [Wol97]). The contraction of $t$ is not secure if a pre place of $t$ is a *choice place* and a post place of $t$ is a *meeting place* – $({}^\bullet t)^\bullet \neq \{t\}$ and ${}^\bullet(t^\bullet) \neq \{t\}$. Such a non-secure contraction can be transformed into a secure one by the following procedure (see Fig. 4.11 for an example):

1. Copy $N$ to obtain an initial $N_{sec}$
2. Split the meeting place $p_{meet}$ by adding $p'_{meet}$, where $p_{meet} \in t^\bullet$, ${}^\bullet p_{meet} - \{t\} = R$ and $R \neq \emptyset$:
   $P_{sec} = P \cup \{p'_{meet}\}$
   $M_{N_{sec}}(p'_{meet}) = M_N(p_{meet})$
   $M_N(p_{meet}) = 0$
   $W_{sec}(t', p'_{meet}) = W(t, p_{meet}),\ \forall t' \in R$
   $W_{sec}(t', p_{meet}) = 0,\ \forall t' \in R$
3. Duplicate the post transitions $t_{dup}$ of the meeting place where $t_{dup} \in p_{meet}{}^\bullet$ and ${}^\bullet t_{dup} = \{p_{meet}\}$, by adding copies $t'_{dup}$:
   $T_{sec} = T \cup \{t'_{dup}\}$
   $W_{sec}(p'_{meet}, t'_{dup}) = W(p_{meet}, t_{dup})$
   We only need to add post arcs for $t'_{dup}$:
   $W_{sec}(t'_{dup}, p) = W(t_{dup}, p),\ \forall p \in P$

If $t_{dup}$ is a fork transition, then transformation cannot be done because it will cause a new structural auto-conflict.

Fig. 4.11a shows an STG that has a non-secure dummy transition $t$: the pre place $p_1$ of $t$ is a conflict place and the post place $p_2$ of $t$ is a combination place. Besides, there is a token on $p_2$. Hence to make dummy transition $t$ secure, $p_2$ should no longer be a combination place, and there should no longer be a token on it. First, split the combination place $p_2$ by adding a new place $p'_2$. Remove the token from $p_2$ and place it on $p'_2$ instead (see Fig. 4.11b). Then we can add the transition $t'_3$ which duplicates $t_3$ (not a non-secure dummy transition), add an arc from the new place $p'_2$ to duplicate transition $t'_3$, and an arc from $t'_3$ to $p_3$. Now the dummy transition $t$ can be securely contracted (see Fig. 4.11c).

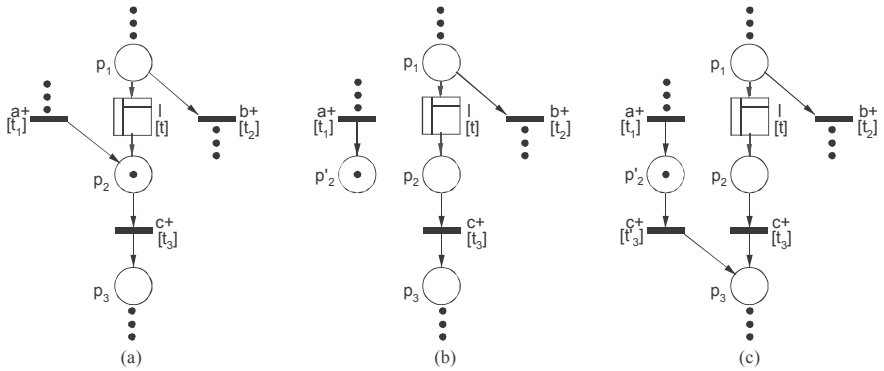


Figure 4.11. Securing a non-secure t-contraction

## 4.8 Vogler-Kangsah algorithm

The [VW02] method required the initial STG to be deterministic: no internal $\lambda$-transition, no dynamic auto conflict. The no dynamic auto conflict requirement is assured by not allowing structural auto conflict. This is sometimes over cautious, because not every structural auto conflict is dynamic. A practical example like the vme bus controller (see Fig. 2.8) cannot be handled by the [VW02] algorithm – due to the structural auto conflict between $t_3$ and $t_{10}$. Also the no dynamic input-output conflict requirement for synthesizability is assured by not allowing structural input-output conflicts. Dummy transitions in [VW02] are treated as internal transitions which further restrict the application of the method to STGs without dummy transitions.

The [VW02] method can be applied not only to STGs but also to labelled P/T nets. However, this generalization has some drawbacks. Instead of signal edges, only signals are considered which cause structural auto conflicts between transitions with labels $a+$ and $a-$ to be considered as dynamic auto conflicts. If

this structural auto conflicts occur in the initial STG, then the STG cannot be handled by the [VW02] algorithm. If it occurs in the intermediate component, then backtracking should be done, which will give a result larger than needed. Also, signal consistency is not yet considered in [VW02].

All the above problems are solved in [VK07], improving the decomposition results and broadening the applicability of the method. The improvements are:

- Structural auto conflicts and input-output conflicts are allowed in the initial net if they are not dynamic. Because structural input-output conflicts are allowed, not only an input transition which gives concession to an output transition should be considered as relevant, but also the one which removes concession (deconcession).

- Dummy transitions are allowed in the initial net if they could be contracted securely.

- Instead of signals, signal edges are considered. Also, the end component is proved to be consistent if the initial net is consistent.

- A new admissible operation is added in [VK07]: the deletion of redundant divining transitions. There are two kinds of redundant divining transitions, namely:

  - *loop-only divining transitions* $t_l$ if each place $p \in {}^\bullet t_l \cup t_l{}^\bullet$ forms a loop with $t_l$ and the arcs of the loop have the same weight. (Fig. 4.12a,b);
  - *duplicate divining transition* $t_d$ if there is another divining transitions $t'$ which has arcs to and from the same places with the same weight as $t_d$ (Fig.4.12c,d).

  Note that deleting loop-only divining transitions is also used by secure contraction.

- The correctness proof has been restructured by Vogler, making it easier to add further possible admissible operations. The restructuring is based on the following definition and preposition.

**4.8.1.** Definition. Let $N$ be an STG that has the following properties:

1. There is no auto-concurrency in $N$.
2. There is no dynamic auto conflict in $N$.
3. There is no structural $\lambda$-output conflict.
4. If $t_2$ is an output transition and $t_1{}^\bullet \cap {}^\bullet t_2 \neq \emptyset$, then $t_1$ is not a divining transition.

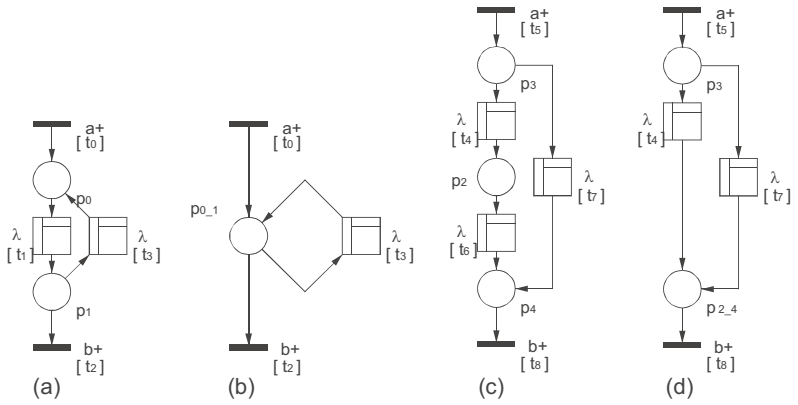


Figure 4.12. (a) A net $N$, (b) net after contracting $t_1$ from $N$, (c) a net $N'$, (d) net after contracting $t_6$ from $N'$

An operation is *pre-admissible* if, whenever applied to $N$, it results in an STG $N'$ that also has the above properties.

**4.8.2.** Proposition. *Let a pre-admissible operation be given that, applied to some member of a family $(C_i)_{i\in I}$ satisfying the third and fourth properties of definition 4.8.1, transforms some $C_j$ to a bisimilar $\overline{C}_j$ with the same input and output signals. Then the operation is admissible.*

**Proof:** [VK07] □

Secure transition contraction, deletion of redundant places, and deletion of redundant divining transitions are admissible operations. Applying only admissible operations to find each output block component, assures decomposition correctness according to definition 3.2.1.

- Vogler also proves that the algorithm is terminate, that it gives correct decomposition results according to definition 3.2.1 and that the result is synthesizable if the initial STG is synthesizable.

Structural auto conflicts could be found in the initial STG, and they could occur if there are *new conflict pairs* – two transitions $t_1, t_2$ with $t_1 \neq t \neq t_2$ are a *new conflict pair* if ${}^\bullet t \cap t_1{}^\bullet \neq \emptyset$ and $t^\bullet \cap {}^\bullet t_2 \neq \emptyset$ in $N$ or vice versa – which forms *new structural auto conflicts* in the intermediate component after secure contraction of transition $t$. The following strategies are introduced that show how to deal with the no dynamic auto conflict requirement in the initial STG and in the intermediate component.

- *Conservative strategy*: This strategy is used in [VW02] which considers structural auto conflicts as dynamic. It is easy to check for structural auto

conflicts, since they can only appear in form of new conflict pairs. But, this strategy is over cautious in some cases.

- *Specifier-dependent strategy*: This strategy (and also the following two strategies) allows structural auto conflicts in the initial net if the specifier guarantees that they are not dynamic – it has been proved by Vogler that these structural auto conflicts will not change into dynamic ones if admissible operations are applied. However, if a new structural auto conflict appears in an intermediate component, then backtracking should be done.

- *Interactive strategy*: This strategy (and the following strategy), try to avoid unnecessary backtracking due to new structural auto conflicts in intermediate components, provided they are not dynamic. The user is asked for confirmation whether the new structural auto conflict is dynamic or not. If the user confirms that the auto conflict is not dynamic, then no backtracking is needed; otherwise backtracking should be done.

- *Risky strategy*: Here, all the new structural auto conflicts are assumed to be non-dynamic.

Vogler has proved that any structural auto-conflict which is dynamic in an intermediate component will survive up to the end component as a dynamic auto-conflict or as auto-concurrency. Hence, any mistake – allowing dynamic auto-conflict – made in the interactive or risky strategy can be found after generating the reachability graph of the end component or by trying to synthesize the end component.

The *locked*2 STG in Fig. 4.13a cannot be handled by the [VW02] method due to dummy transitions $t_{25}$ and $t_{26}$. With the improved algorithm, dummy transitions are allowed if they can be contracted securely. $t_{25}$ and $t_{26}$ can be contracted securely (see Fig. 4.13b). Hence, they are all allowed, and the specification can be handled by the [VK07] algorithm. To find the *rdy*-component of STG *locked*2, the first step should be to find the transitions that give concession or deconcession to $t_{13}$ and $t_{18}$. $t_8$ labelled $c$, $t_{11}$ labelled $acka$, and $t_{12}$ labelled $ackd$ give concession to $t_{13}$; $t_{21}$ labelled $c$ gives concession to $t_{18}$. Hence, transitions with signal labels $c, acka$, and $ackd$ are relevant for the *rdy*-component. Transitions with signal labels $a, D$, and $S$ are irrelevant and are silenced (see Fig. 4.13c). After contracting $t_4, t_9, t_5, t_{10}, t_{22}$, (see Fig. 4.13d) and $t_2$ the intermediate *rdy*-component is shown in Fig. 4.14a. An attempt to contract $t_0$ or $t_1$ forces backtracking due to a new structural auto-conflict between $t_3$ and $t_{14}$ (see Fig. 4.14c for the contraction of $t_0$). Hence, for the specifier-dependent and the interactive strategy, signals $a$, and $D$ must also be considered as relevant; the final *rdy*-component is shown in Fig. 4.14b.

Using the risky strategy, the final *rdy*-component is shown in Fig. 4.14d. Note that the mistake made when using the risky strategy – allowing the new

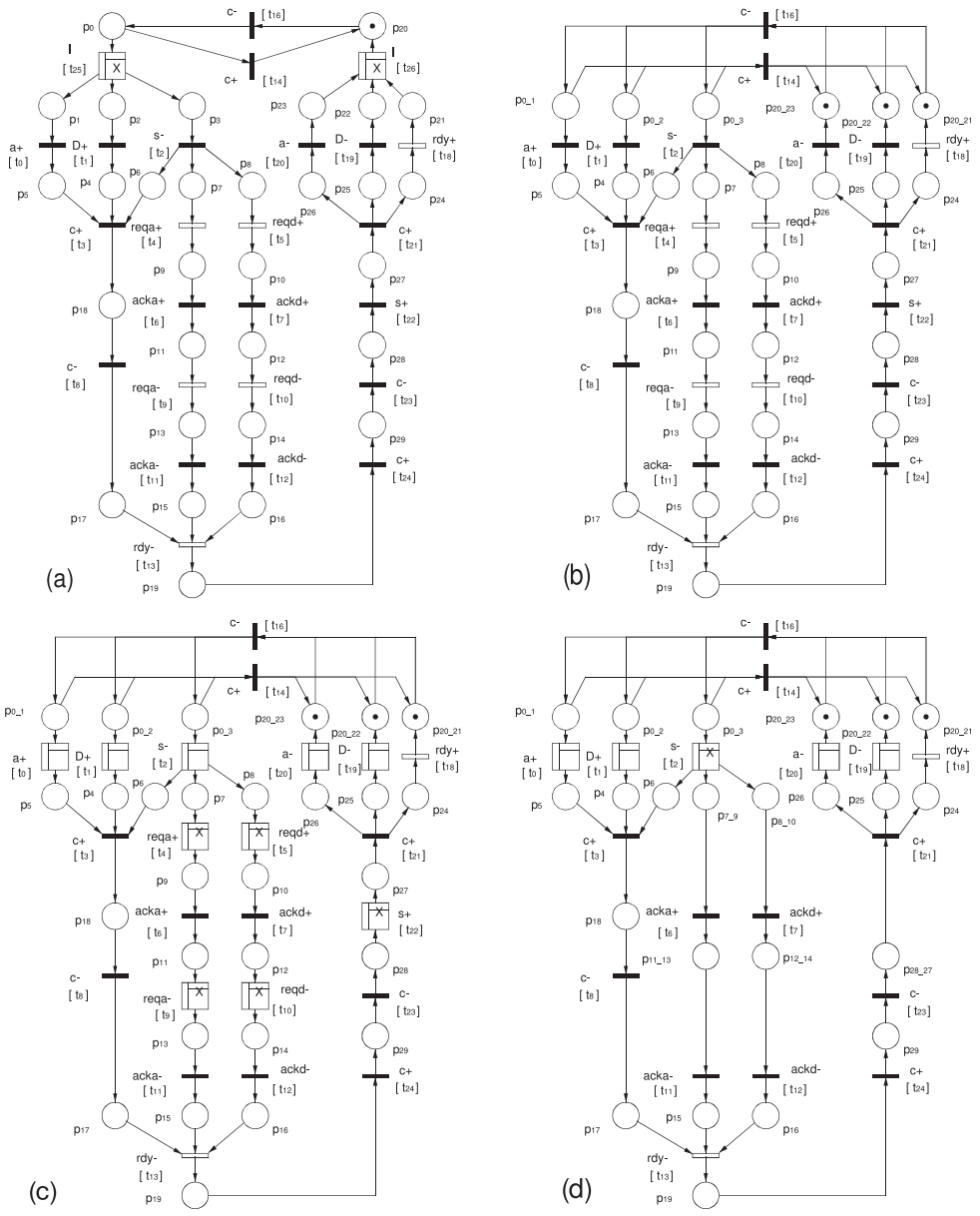


Figure 4.13. Locked2 example: (a) dummy transitions are marked for contraction, (b) dummy transitions are contracted, (c) irrelevant transitions for *rdy*-component are silenced, (d) after contracting transitions without conflict place

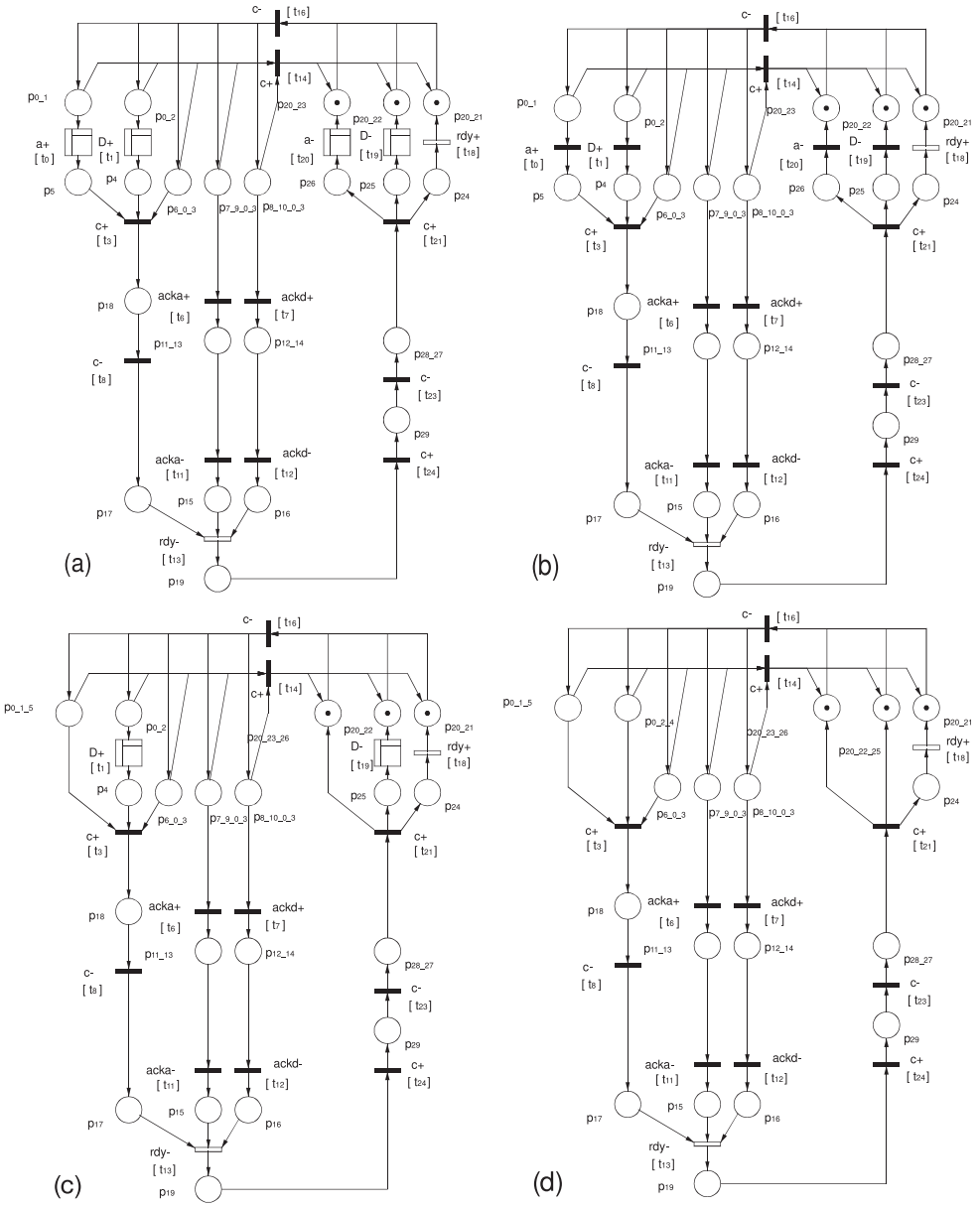


Figure 4.14. Locked2 example (continued): (a) after contracting transitions with signal label $reqa, reqd, s$, (b) $rdy$-component after backtracking transitions with signal label $a, D$, (c) net with structural auto-conflict after contracting transitions with signal $a$, (d) risky $rdy$-component with dynamic auto-conflict between $t_3$ and $t_{14}$ (both labelled $c+$)

structural auto-conflict between $t_3$ and $t_{14}$, which is dynamic – is preserved and can be checked in the final $rdy$-component; i.e. the final $rdy$-component has a structural and dynamic auto-conflict between $t_3$ and $t_{14}$.

# Chapter 5

# STG Decomposition with SMD-subnets as Initial Components

To decompose an STG $N$ by the [VW02] method, first a copy of $N$ is taken as initial component $N_i$; then $N_i$ is reduced until the end component is found. For a large net, this means that many operations must be performed to obtain the final (smaller) component. Secure transition contraction takes an important role in this reduction. If contraction cannot be done, then backtracking must be performed. Backtracking due to a new structural auto conflict is unavoidable. It must be done in order to make the component synthesizable. Backtracking due to a non-secure transition that can change the property of the net is also unavoidable. E.g. as shown in Fig. 3.3b and c, non-secure contraction can change the liveness property of the net – a dead net may become live. But backtracking due to a non-secure $t$-contraction could be avoided; e.g. non-secure transitions $t_2, t_5$ in the wechselpuffer example (see Fig. 3.4d). Also structural preconditions of secure contraction – e.g. no loop with any place – limit the use of secure contraction. These limitations of secure contractions increase the number of apparently relevant signals – unnecessarily, as will be shown in later sections.

Another example is async99* in Fig. 5.1, a modification of the net in [BEW99]. The dynamic input-output conflict between $t_0$ and $t_2$, and between $t_1$ and $t_9$ is a feature of extended burst mode specification which includes timing constraints such that the dynamic input-output conflict will never happen. Also, input-output concurrency has been added in Fig. 5.1. For the $z$-component, the relevant input signals are $a$ – $t_6$ gives concession to $t_7$ and $t_3$ gives concession to $t_5$ – and $phi$ – $t_{12}$ gives concession to $t_7$. Signals $y, x$, and $c$ are irrelevant. Hence, the corresponding transitions are silenced. But, contracting $t_0$ or $t_1$ – both of them have signal label $c$ – is not possible, because both of them are non-secure transitions. Also, contracting $t_9$ – with signal label $y$ – is not possible because $t_9$ forms a loop with $p_1$. The same with $t_2$ – which has signal label $x$ and forms a loop with $p_0$. Hence, after backtracking is performed for signals $c, y$ and $x$, the resulting $z$-component is the same as in Fig. 5.1, but with signals $x$ and $y$ as input

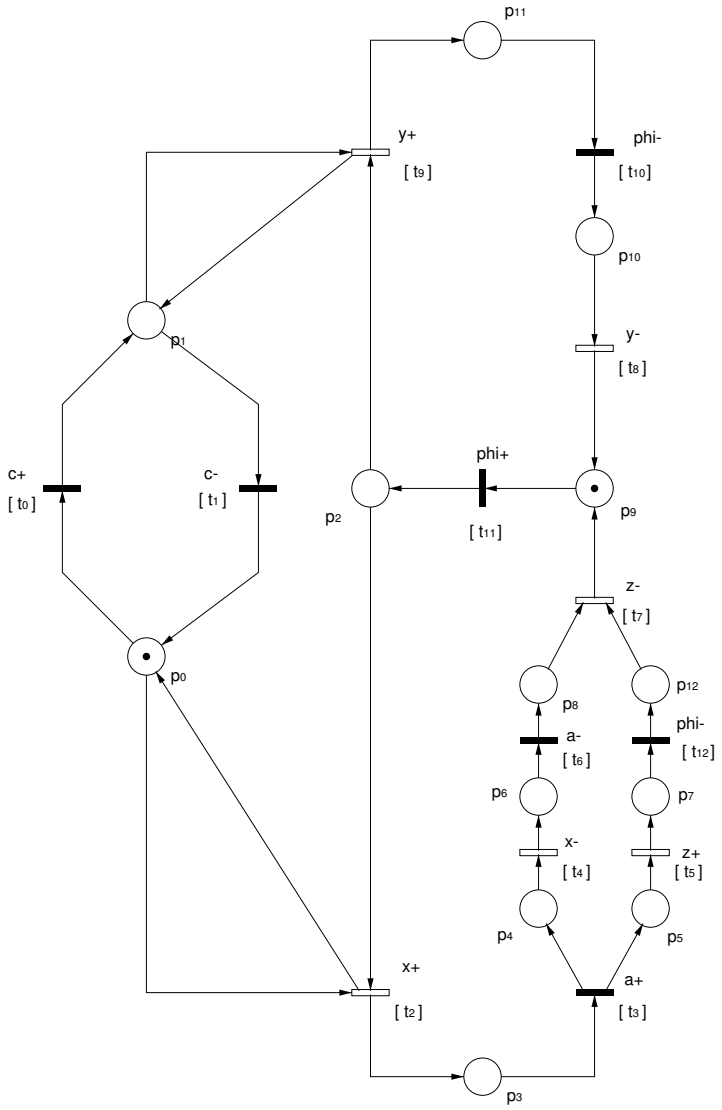


Figure 5.1. Async99* [BEW99]

signals.

## 5.1 The SMD-subnet method

One way to overcome secure contraction limitation and to increase algorithm efficiency is by directly taking a subnet of $N$ as initial component. Taking the subnet would reduce the number of transitions to be contracted and therefore the number of falsely relevant signals due to secure contraction limitation.

In data-path-and-controller system design, the system is divided into operational unit – also called datapath by many authors – and controller unit. The controller governs the interaction between agents in the operational unit and the interaction between system and environment. The specification of this interaction is also called the *communication protocol*, shortly *protocol*.

An agent of the operational unit and the environment normally operate sequentially. Concurrency is due to several agents operating autonomously. The interaction between agents should be synchronized by the communication protocol. The protocol between two agents usually is a state machine, due to the sequential nature of the agents themselves. If the same protocol is repeated, then it will become a strongly connected state machine (SCSM). Hence, in practice a controller specification is a synchronization of SCSMs; i.e. it is an state machine decomposable (SMD) net.

**5.1.1.** Definition. Let $N_1 = (P_1, T_1, F_1, M_{N_1}, l_1)$ and $N_2 = (P_2, T_2, F_2, M_{N_2}, l_2)$ be nets. $N = (P, T, F, M_N, l)$ is a *union* of $N_1$ and $N_2$ iff

$$\begin{aligned} P &= P_1 \cup P_2 \\ T &= T_1 \cup T_2 \\ F &= F_1 \cup F_2 \\ \forall p \in P : M_N(p) &= \begin{cases} M_{N_1}(p) & \text{if } p \in P_1 \\ M_{N_2}(p) & \text{if } p \in P_2 \end{cases} \\ \forall t \in T : l(t) &= \begin{cases} l_1(t) & \text{if } t \in T_1 \\ l_2(t) & \text{if } t \in T_2 \end{cases} \end{aligned}$$

Note: If $p \in P_1 \cap P_2$, then $M_{N_1}(p) = M_{N_2}(p) = M_N(p)$, and if $t \in T_1 \cap T_2$, then $l_1(t) = l_2(t) = l(t)$.

**5.1.2.** Definition. A union of $N_1$ and $N_2$ yields a *synchronization net* $N$ iff $\forall p \in P$, if $p \in P_1$ then ${}^\bullet p \cup p^\bullet \subseteq T_1$; and if $p \in P_2$ then ${}^\bullet p \cup p^\bullet \subseteq T_2$.

From definition 5.1.2, by synchronization, the places in $N_1$ and $N_2$ do not change their type; e.g. a choice place in $N_1$ (or $N_2$) remains a choice place in $N$. But a transition may change its type: in $N_1$, let $P_1' = {}^\bullet t \cup t^\bullet \subset P_1$ and in $N_2$, let $P_2' = {}^\bullet t \cup t^\bullet \subset P_2$. If there are places $p \in P_1' \Delta P_2'$, i.e. places belonging only to $P_1'$

or only to $P_2'$, then a normal transition $t$ becomes a fork or join transition in the union of $N_1$ and $N_2$. As an example, the net $N$ in Fig. 5.2 is a synchronization of $N_1$ and $N_2$. Transitions $t_0, t_1$ and $t_2$ are synchronization transitions.

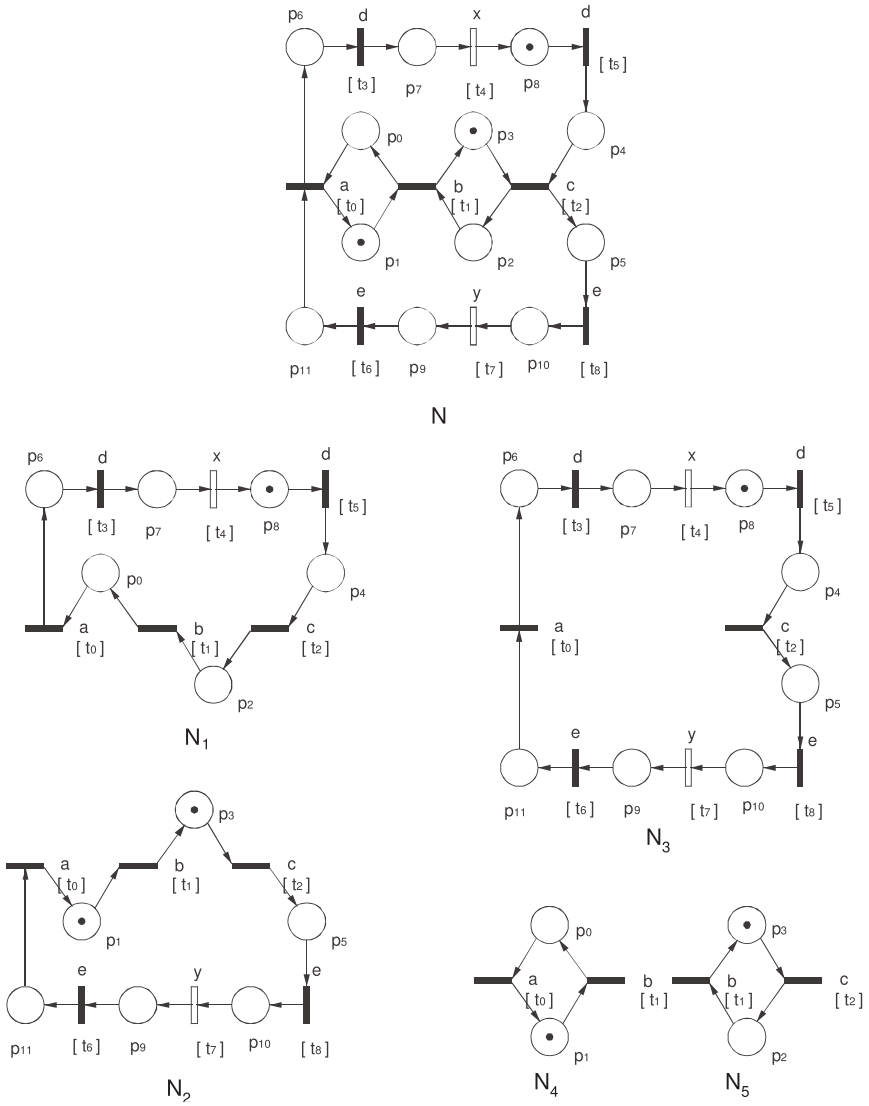


Figure 5.2. A net $N$ and its SCSM subnets

**5.1.3.** Definition. An SCSM subnet $N_i$ of $N$ is a *safe SCSM* iff $N_i$ has only one token at the initial marking.

**5.1.4.** Definition. A *cover* of $N$ is a set $\chi$ of SCSM subnets of $N$ such that their synchronization results in $N$. $\chi$ is a *safe cover* of $N$ iff each SCSM subnet in $\chi$ is safe.

The cover $\chi_1 = \{N_3, N_4, N_5\}$ of $N$ in Fig. 5.2 is a safe cover. But the cover $\chi_2 = \{N_1, N_2\}$ of $N$ in Fig. 5.2 is not safe; because the SCSM $N_2$ in Fig. 5.2 is not safe.

As will be shown later, the safe cover condition is imposed to ensure synthesizability of the component.

**5.1.5.** Definition. An SCSM subnet $N_i$ of $N$ is *redundant* in the cover $\chi$ iff the synchronization of all the SCSM subnets in $\chi$ results in the same net $N$ as the synchronization of all SCSM subnets except $N_i$.

A cover without redundant SCSM subnets is *irredundant*.

The cover $\chi_3 = \{N_1, N_3, N_4, N_5\}$ of $N$ in Fig. 5.2 is a safe cover. $N_1$ is redundant in $\chi_3$, because synchronizing $N_1, N_3, N_4, N_5$ results in the same net $N$ as synchronizing $N_3, N_4, N_5$. One can also see that each part of $N_1$ (in this case, each TT-handle of $N_1$) is also covered by either $N_3$ or $N_4$ or $N_5$ – $t_0, p_6, t_3, p_7, t_4, p_8, t_5, p_4, t_2$ in $N_1$ is also covered by $N_3$, $t_1, p_0, t_0$ by $N_4$ and $t_2, p_2, t_1$ by $N_5$.

In Fig. 5.3, there are two safe covers of $N$ without redundant SCSM subnets: $\chi_1 = \{N_1, N_2\}$ and $\chi_2 = \{N_3, N_4\}$. However, the cover $\chi_3 = \{N_1, N_2, N_3\}$ has a redundant SCSM subnet $N_3$. This shows that whether an SCSM subnet is redundant or not depends on in which cover it is contained.

Because the SMD-subnet method operates only with the structure of the net, it is necessary to have explicit consistency for each signal $s \in (In \cup Out)$ in order that the resulting component is synthesizable.

**5.1.6.** Definition. A safe SCSM subnet $N_m$ of $N$ is a *consistent SCSM* for signal $s$, if $\forall t \in T \mid l(t) = s, t \in T_m$.
A signal $s \in (In \cup Out)$ has *explicit consistency* if for all safe covers $\chi$ of $N$, signal $s$ is consistent in $N'$, where $N'$ is the synchronization net of SCSMs that include $s$-transitions (transitions with signal label $s$); Observe that this implies that a consistent SCSM for $s$ is covered by $N'$.

The only safe cover for the net $N$ in Fig. 5.4 is $\chi = \{N_1, N_2\}$. Signal $a$ has explicit consistency for $\chi$; because $a$ is consistent in the synchronization net $N$ of $N_1$ and $N_2$ (both $N_1$ and $N_2$ include $a$-transitions). However, signals $x$ and $b$ have no explicit consistency for $\chi$; because, signal $x$ and $b$ are not consistent in $N_1$ which has $x$-transitions and $b$-transitions. Note that $N$ has no consistent SCSM subnets for signals $x$ and $b$.

Although the synchronization of the (safe) subnet $N_1$ and the unsafe subnet $N_2$ results in the safe net $N$, the restriction to safe cover is necessary because an unsafe SCSM cannot be safely contracted. On the other hand, a live and safe free-choice net $N$ can always be covered by a safe cover.

In the net $N$ (see Fig. 5.5), signal $a$ has no explicit consistency for $\chi_1 = \{N_1, N_2, N_3, N_4, N_5, N_6, N_7\}$ because $a$ is consistent only if the consistent SCSM

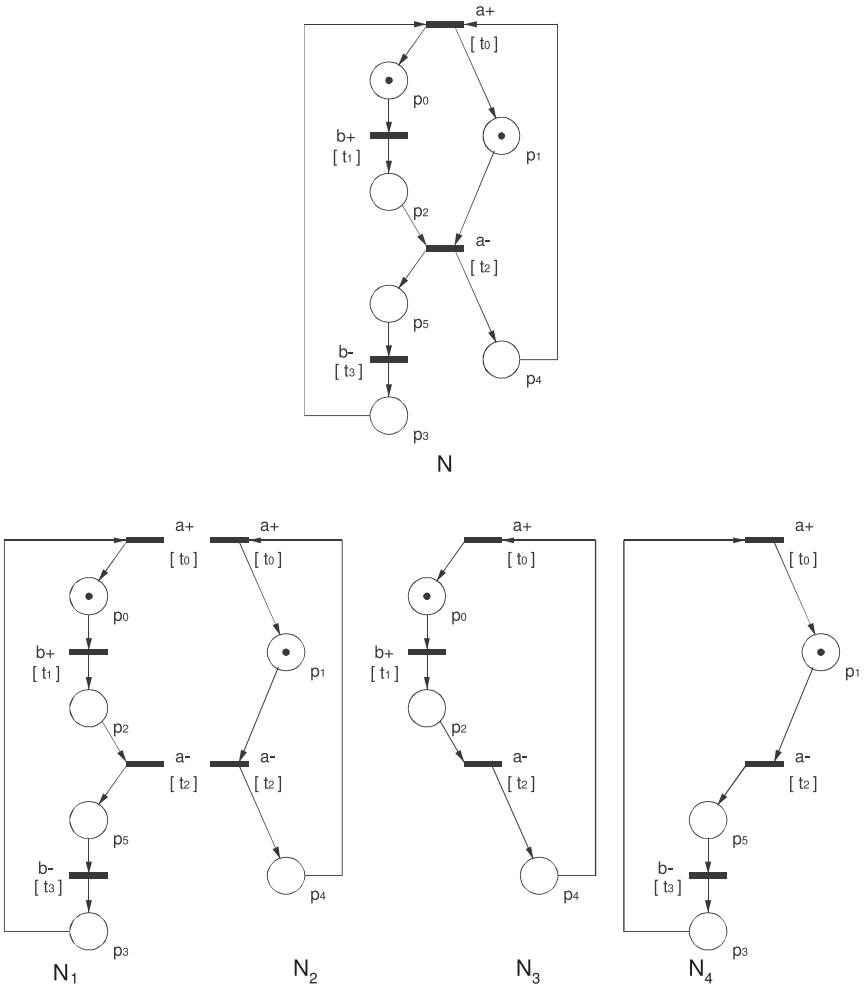


Figure 5.3. A net $N$ and its SCSM subnets

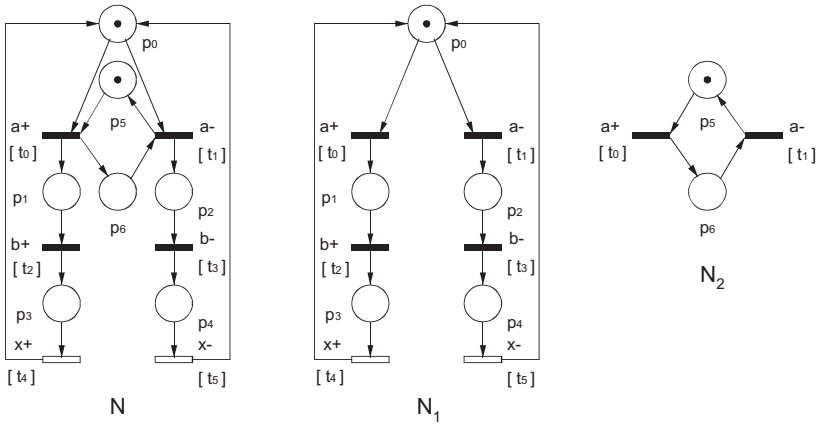


Figure 5.4. A net $N$ and its SCSM subnets

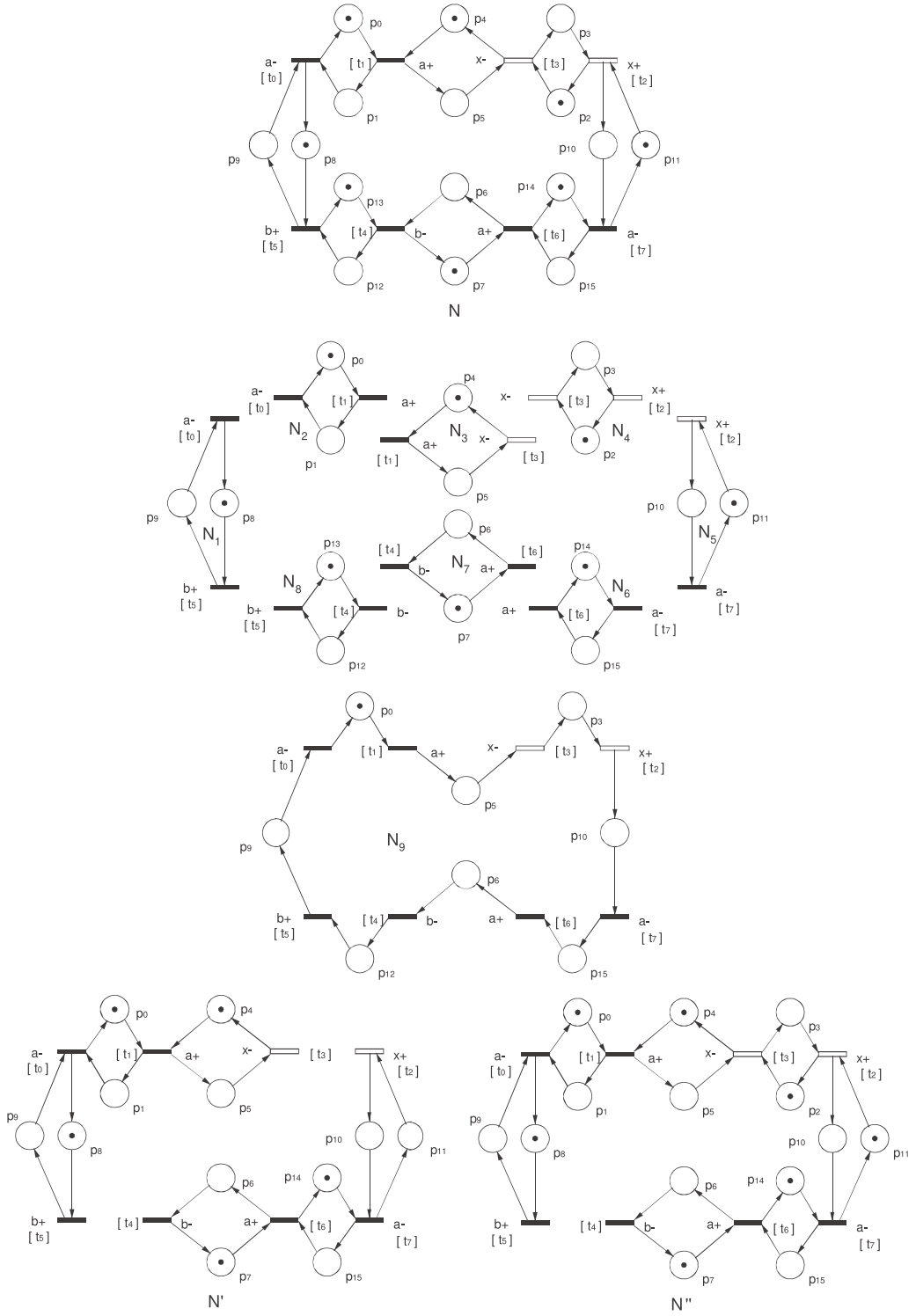


Figure 5.5. A net $N$, its SCSM subnets $N_1$ to $N_9$ and SMD subnets $N'$ and $N''$

$N_9$ for $a$ is covered by the synchronization net of SCSMs that include $a$-transitions. This is not the case for $N''$, which is a synchronization of SCSM $N_4$ and SCSMs $N_1, N_2, N_3, N_5, N_6$ and $N_7$, which include every $a$-transition.

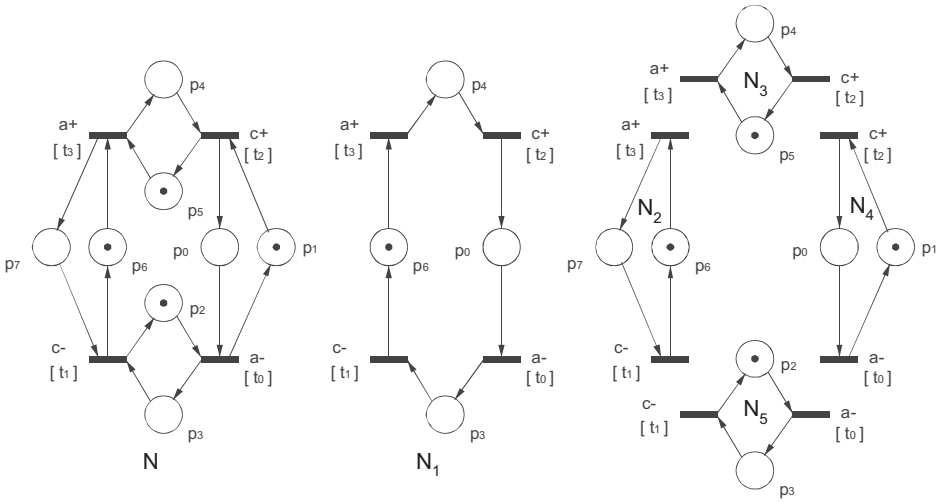


Figure 5.6. A net $N$ with its safe SCSM subnets

The net in Fig. 5.6 has a safe cover $\chi$ without redundant SCSMs: $\chi = \{N_2, N_3, N_4, N_5\}$. Signals $a$ and $c$ respectively are consistent only if $N_1$ is covered by a synchronization net of SCSMs that include $a$ and $c$ transitions. This is the case for $N$, which is the synchronization net of SCSMs $N_2, N_3, N_4$ and $N_5$, which include every $a$- and every $c$-transition.

The net in Fig. 5.3 has safe covers without redundant SCSMs: $\chi_1 = \{N_1, N_2\}$ and $\chi_2 = \{N_3, N_4\}$. Obviously, signal $a$ has explicit consistency – $N_1, N_2, N_3$ and $N_4$ are consistent SCSMs for $a$. This also shows that a consistent SCSM for a signal is not necessarily unique. Signal $b$ also has explicit consistency for $\chi_1$ – $N_1$ is the only SCSM with $b$-transitions which is consistent for signal $b$ –, and for $\chi_2$ – signal $b$ is consistent in $N$, which is a synchronization net of SCSMs with $b$-transitions: $N_3$ and $N_4$.

For synthesis-purpose decomposition it is not necessary to have a live initial specification that preserves the behaviour of the initial net and the synthesizability of the components. Figure 5.7a shows an example of a net which is not live with its cover (Figure 5.7b,c). The SCSMs of the cover also are a correct decomposition – that preserves behaviour – into a $w, x$- and a $y, z$-component. Both components are live and therefore synthesizable, but the composition behaviour has a deadlock.

Though in principle it is not necessary to have a live initial specification, the SMD-subnet method being based on SCSMs, requires the initial net to be live. Non-live specifications are not synthesizable, therefore most synthesis tools require live initial specifications; they test a given specification for liveness and

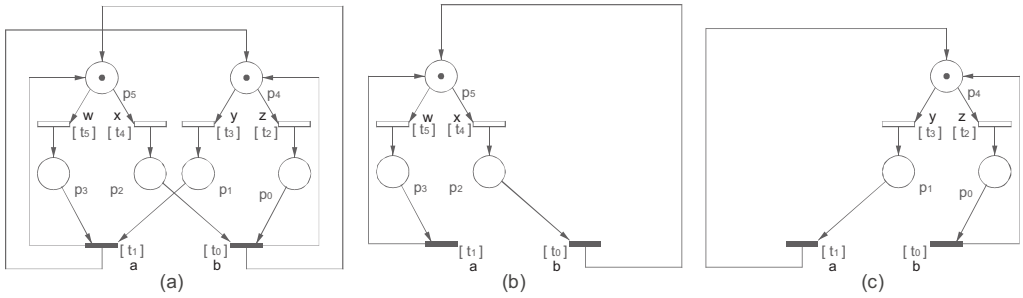


Figure 5.7. (a) A net which is not live; (b) $w, x$-component; (c) $y, z$-component

reject if it is non-live. Meaningful specifications for controllers are live; liveness can be tested for before decomposition and synthesis. The decomposer receives the specification that has been proven to be live, and decomposes the specification in a way such that the behaviour is preserved and the components are live. Hence, there is no need to waste effort by allowing a non-live specification.

The safeness requirement is imposed because the STG is used to model controller behaviour. The argument for non-safeness is to allow counter behaviour. Counting should not be modelled by an STG. Instead, it ought to be delegated to an operational unit (counter); the control of the counting process is what should be specified by an STG.

**5.1.7.** Definition. An STG $N$ is *well specified* iff $N$ is a live and safe net which is synthesizable and $N$ has explicit consistency for every input and output signal.

The net $N$ in Fig. 5.4 is a live and safe SMD net, but it is not well specified, because signals $b$ and $x$ have no explicit consistency for any cover. Also, the net in Fig. 5.5 is a live and safe SMD net, but it is not well specified, because signal $a$ has no explicit consistency(and even is in concession) for $\chi_1 = \{N_1, N_2, N_3, N_4, N_5, N_6, N_7\}$. The nets $N$ in Fig. 5.6 and Fig. 5.3 are well specified.

As in [VW02], *relevant signals* for an output block $B$ are relevant input signals for all output signals $b \in B$ and the output signals in $B$. $N_B$, the initial specification for an output block $B$, is a synchronization of SCSM subnets in the safe cover of $N$ which has transitions labelled with signals relevant for $B$.

**5.1.8.** Definition. An SCSM subnet $N_i$ of $N$ is a *relevant* SCSM for an output block $B$ iff $\exists t \in T_i$ such that $t$ is labelled with a signal relevant for $B$. Let an *SMD-subnet* $N_B$ be a synchronization net $\chi_c$ of relevant SCSM subnets for $B$ ($\chi_c \subseteq \chi$ of a well specified net $N$). $N_B$ is a *correct initial component* for an output block $B$ iff:

1. $N_B$ is a live and safe net,

2. $N_B$ is synthesizable after contracting all transitions in $N_B$ with irrelevant signal labels.

For an output block $B$, $\chi_c$ could also be found by removing the set $\chi_{nc}$ of non-relevant SCSMs from the safe cover $\chi$ of $N$. $N_B$ is obtained by synchronizing all SCSM subnets in $\chi_c$. Removing SCSM subnets from $\chi$ will increase concurrency in $N_B$, because there is a transition $t$ in the component which is a synchronization transition in $N$, that does not need to wait for synchronization anymore; i.e. $t$ becomes a normal transition. $t$ cannot be a transition labelled with a relevant signal; because if it is, then all the SCSM subnets containing $t$ are taken; i.e. $t$ would still be a synchronization transition, which is not the case.

This increase of concurrency will affect only the firing of transitions with relevant input signals, but not the firing of transitions labelled with output signals from $B$. For the output transition is enabled only if the concession transition with relevant signal label is fired. The increase of concurrency which affects only the firing of transitions with relevant input signals is not a problem because correct decomposition, by definition, does not require input transitions of the component to be simulated by $N$.

Note that if the environment behave as per specification $N$ then there will be no increment of concurrency.

Instead of obtaining $N$ by synchronizing the SCSM subnets of a safe, life and relevant cover $\chi$ of $N$, $N$ or an equivalent net can also be obtained by parallel composition of correct initial components.

**5.1.9.** Proposition. *Parallel composition $C$ of correct initial components of $N$ will result in $N$ or an equivalent net – a correct decomposition of $N$ as per definition 3.2.1.*

**Proof:** In this proof, the marking of parallel composition is considered as disjoint union of the marking of components. The proof of each item in definition 3.2.1 is as follow:

1. $(M_N, M_C) \in \mathcal{B}$ ; this is because the components are obtained by removing SCSMs from the safe cover of $N$.

2. For each $(M, M') \in \mathcal{B}$:

   (a) If $M[a\rangle\rangle M_1$ and $a \in In_N$, then there are two possibilities: If $a \in In_C$ then $M'[a\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some $M_1'$; the components which have $a$ can simulate $a$ of $N$, because each component is derived from $N$ and live; if $a$ is fired in $N$, then $a$ could also be fired in each component which has $a$. If $a \notin In_C$ then $(M_1, M') \in \mathcal{B}$.

   (b) If $M[x\rangle\rangle M_1$ and $x \in Out_N$, then $M'[x\rangle\rangle M_1'$ and $(M_1, M_1') \in \mathcal{B}$ for some $M_1'$; the component which has $x$ can simulate $x$ of $N$, because

each component is derived from $N$ and live; if $x$ is fired in $N$, then $x$ could also be fired in the component which produces $x$.

(c) If $x \in Out_C$ and $M'[x\rangle\rangle M'_1$, then $M[x\rangle\rangle M_1$ and $(M_1, M'_1) \in \mathcal{B}$ for some $M_1$; $N$ can simulate $x$ which is produced by a component, because the component which produces $x$ is built of SCSM subnets which have relevant input signals for $x$; i.e. $x$ is enabled in the live and safe $x$-component, only if all the concession transitions of $x$ have been fired – which is also the case for $N$.

(d) If $x \in Out_i$ for some $i \in I$ and $M'|_{P_i}[x\rangle\rangle$, then $M'[x\rangle\rangle$; there will be no computation interference, because an output $x$ is produced only if all the concession transitions of $x$ have been fired in the $x$-component which is live and safe; i.e. all the other components also are ready to fire $x$.

□

In the following proofs, let an SMD-subnet $N'$ be obtained by synchronizing all the SCSMs in $\chi'$ where $\chi'$ is a safe cover of $N'$ after removing an SCSM subnet $N_i$ from the cover $\chi$ of $N$.

**5.1.10.** Lemma. *There will be no new conflict in the SMD-subnet $N'$*

**Proof:** Assume there is a new conflict in $N'$ between transitions $t_1, t_2$ with choice place $p$. Then there is already a choice place $p$ among the pre places of conflict transitions $t_1, t_2$ in $N$. But this is not possible because synchronizing $N_i$ with $N'$ will not change the choice place $p$ into a normal place (see definition 5.1.2). □

**5.1.11.** Proposition. *There will be no new auto-conflict or new input/output conflict in SMD-subnet $N'$*

**Proof:** Direct by lemma 5.1.10: Because there will be no new conflict , then there will be no new auto-conflict or new input/output conflict in SMD-subnet $N'$. □

As mentioned before, the SMD-subnet method tends to increase concurrency and therefore may cause auto-concurrency and non-consistency. This is shown by the following example: The non-safe SCSM subnet $N_2$ in Fig. 5.2 is also the initial $y$-component of $N$ with the non-safe cover $\chi_1 = \{N_1, N_2\}$ – the $e$- and $y$-transitions occur only in $N_2$; $N_2$ is the only relevant SCSM for $y$. This non-safeness introduces a new auto-concurrency(between $t_6$ and $t_8$). Hence, $N_2$ is not synthesizable.

Signal $a$ in the net $N$ (see Fig. 5.5) has no explicit consistency for $\chi_1 = \{N_1, N_2, N_3, N_4, N_5, N_6, N_7\}$. The initial component for the output signal $x$ is

$N''$ which is a synchronization of the nets in $\chi_1$ without the non-relevant SCSM $N_9$. Lack of explicit consistency for signal $a$ introduces auto-concurrency and non-consistency; e.g. it is possible to fire $a+$ $[t_6]$ and $a-$ $[t_0]$ at the same time (not consistent); it is also possible to fire $a+$ $[t_6]$ and $a+$ $[t_1]$ at the same time (auto-concurrency and non-consistency). Hence, $N''$ is not synthesizable.

Therefore, safeness and explicit consistency is imposed to assure that the resulting net $N'$ after deleting an SCSM subnet $N_i$ from $N$ has consistency and no new auto-concurrency.

**5.1.12.** Proposition. *Consistent STGs are non-auto-concurrent.*

**Proof:** see Lemma 3.1. [Esp03] □

**5.1.13.** Proposition. *There will be no new auto-concurrency, and consistency is preserved in SMD-subnets of $N'$.*

**Proof:** For a well specified net, consistency is preserved by the SMD-subnet method because of explicit consistency and safeness. This also ensures that no new auto-concurrency is introduced by the SMD-subnet method due to proposition 5.1.12. □

Propositions 5.1.11 and 5.1.13 assure that an initial component $N_B$ of a well specified net is synthesizable. The only thing left to show is that $N_B$ is also live and safe, which unfortunately cannot be fulfilled by the class of SMD nets. An example for this is shown in Fig. 5.8; after deleting circle path $(p_0, t_0, p_1, t_1, p_0)$ the resulting net is not live. The class of P/T nets that can be handled by the SMD-subnet method, i.e. result in a live and safe initial component, is the class of live and safe FC nets. Note that this includes live and safe marked graphs.

Following is the proof that applying the SMD-subnet method to a live and safe FC net $N$ results in a live and safe initial component.

**5.1.14.** Lemma. *Every SCSM subnet of $N'$ also is an SCSM subnet of $N$*

**Proof:** Assume there is a new SCSM subnet of $N'$ which is not in the set of SCSM subnets of $N$; i.e. there is a partial subnet of $N'$ that is not in $N$. This is not possible because $N'$ is obtained by synchronizing all the SCSMs in $\chi'$, where $\chi'$ is the cover of $N'$ after removing an SCSM subnet from the cover $\chi$ of $N$. Hence, there is no new SCSM subnet, and it might be equal if the removed SCSM subnet is redundant in $\chi$; i.e. Every SCSM subnet of $N'$ also is an SCSM subnet of $N$. □

**5.1.15.** Proposition. *Let $N$ be a live and safe FC net with a safe cover $\chi$. The SMD-subnet $N'$ is also a live and safe FC net.*

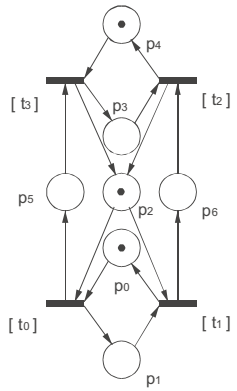


Figure 5.8. Regulating net [ES89]

**Proof:** $N$ is a live and safe FC net. From corollary 3.1.5, every SCSM subnet of $N$ is marked at $M_0$ and each $N_j \in \chi$ has exactly one token at $M_0$. $N'$ is also live and safe because of lemma 5.1.14. Therefore every SCSM subnet of $N'$ is also marked at $M_0$; and due to $\chi' \subset \chi$, every $N_k \in \chi'$ also has exactly one token at $M_0$. $N'$ is FC, because if not then there is a transition $t$ in conflict with choice place $p$ and control place $p'$ in $N'$, and adding $N_i$ will not change this; i.e. $t, p, p'$ would also be in $N$. But this cannot be the case, because $N$ is an FC net. □

**5.1.16.** Theorem. *Applying the SMD-subnet method to a well specified FC net results in a correct initial component as per definition 5.1.8*

**Proof:** Liveness and safeness is assured by proposition 5.1.15. Synthesizability is assured by proposition 5.1.11 and 5.1.13. □

## 5.2 Free choice net extension

Free choice (FC) net extension is suggested to decide liveness and safeness of non-FC nets with polynomial complexity. The principle is to introduce an operation that transforms a non-FC net $N'$ into an FC net $N$ and to prove that the transformation would preserve liveness and safeness. This extension can be used to extend the class of nets that can be handled by the SMD-subnet method.

### 5.2.1 Regulation circle path

[DE95] suggests to extend an FC net into a non-FC net with alternating choice by adding a regulation circle path.

**5.2.1.** DEFINITION. Let $T_U = \{t_1, ..., t_m\}$ be a subset of transition set $T$ of a net $N$ with identical pre places, i.e. ${}^\bullet T_U = {}^\bullet t$ for every $t \in T_U$. For every transition $t_i$ of $T_U$, we define a new place $p_i$.
The net $N_U$: $P_U = \{p_1, ..., p_m\}$, $T_U$, $F_U = \{(p_1, t_1), (t_1, p_2), ..., (p_m, t_m), (t_m, p_1)\}$ is called a *regulation circle path* of $N$. For the circle path to be live, exactly one place $p_i$ must be marked with a token: $\exists p_i : M_{N_U}(p_i) = 1$

This regulation circle path is easily recognized in a non-FC net $N$: First, find transitions in conflict where each transition has only one control place, then check whether the control places and the transitions in conflict form a circle path. As an example, $t_2, t_5$ in Fig. 3.4e are transitions in conflict where $p_{5\_3}$ is the single control place of $t_2$ and $p_{4\_6}$ the single control place of $t_5$. $t_2, p_{4\_6}, t_5, p_{5\_3}$ form a circle path $(p_{5\_3}, t_2, p_{4\_6}, t_5, p_{5\_3})$, and only $p_{5\_3}$ is marked. Hence, $(p_{5\_3}, t_2, p_{4\_6}, t_5, p_{5\_3})$ is a regulation circle path.

**5.2.2.** PROPOSITION. *Let $N$ be a net with a live and bounded marking, and $N'$ be obtained from $N$ by synchronization with a regulation circle path. Then $N'$ also has a live and bounded marking.*

**Proof:** see [DE95] □

Proposition 5.2.2 can be used to extend the SMD-subnet method if a net becomes a live and safe FC net $N$ after removing regulation circle paths. For $N'$ has a live and safe marking, and after removing an SCSM from a safe cover of $N'$, a live and safe FC net $N$ is obtained (fulfilled definition 5.1.8.1).

As an example, consider the net in Fig. 3.4e. Without the regulation circle path $(p_{5\_3}, t_2, p_{4\_6}, t_5, p_{5\_3})$, the resulting net is an FC net which is live and safe; therefore, it can be handled by the SMD-subnet method.

The example in Fig. 5.8 cannot be handled by the SMD-subnet method; because without the regulation circle path $(p_0, t_0, p_1, t_1, p_0)$ the resulting net is an FC net which is not live.

### 5.2.2 Level SCSM

In the async99* example (see Fig. 5.1), the path $(t_0, p_1, t_1, p_0, t_0)$ is a circle path with loop arcs between $p_1$ and $t_9$, and between $p_0$ and $t_2$. The transitions $t_0$ and $t_1$ have the same signal label $c$ with different signal edges. A token in one of the places $p_0$ and $p_1$ represents the current level of signal $c$ (whether it is 0 or 1) because the circle path with the loop arcs is the only consistent SCSM for $c$.

**5.2.3.** DEFINITION. A safe SCSM $N_i$ is a *level SCSM* for signal $s$ iff $N_i$ is the only consistent SCSM for signal $s$ and there is a *level circle path* $(t_0, p_0, t_1, p_1, t_0)$ with loop arcs formed by some *sample transitions* $T_s (t_0, t_1 \notin T_s)$ with $p_1$ and $p_0$, where the transitions $t_0$ and $t_1$ have the same signal label $s$ with different signal

edges. The places $p_0$ and $p_1$ which represent the level of the signal are called *level places*.

A level circle path can be found easily in a non-FC net by deleting all loop arcs; the level circle path which is disjoint from the rest of the net with its loop arcs is a level SCSM if it is a safe SCSM. In the async99* example (see Fig. 5.1), after removing the loop arcs $(p_1, t_9)$, $(t_9, p_1)$, $(p_0, t_2)$, and $(t_2, p_0)$ the level circle path $(t_0, p_0, t_1, p_1, t_0)$ is found. After adding loop arcs $(p_1, t_9)$ $(t_9, p_1)$, $(p_0, t_2)$, and $(t_2, p_0)$ the resulting SCSM is a level SCSM for signal $c$, because it is a safe SCSM and it is the only consistent SCSM for signal $c$.

**5.2.4.** PROPOSITION. *Synchronizing a level SCSM to a live and safe net $N$ results in a net $N'$ which is also live and safe.*

**Proof:** A level SCSM is live because a level SCSM is a safe SCSM which, according to proposition 2.1.8, is also live. $N$ interacts with a level SCSM through sample transitions $T_s$. When a transition $t \in T_s$ is fired, it removes and then replaces a token on a level place $p$. Because of this, the token never leaves the level SCSM, which assures that there always will be a token on the level places (the level SCSM itself is live). Therefore, a transition $t \in T_s$ is live in $N'$ if it is also live in $N$.
$N'$ is safe because firing a $t \in T_s$ does not increase the number of tokens in the level SCSM and $N$. □

Proposition 5.2.4 can be used to extend SMD-subnet method if a net becomes a live and safe FC net $N$ after removing the level SCSM. This is because $N'$ is a live and safe net, and after removing an SCSM from a safe cover of $N'$, a live and safe FC net $N$ is obtained (fulfilled definition 5.1.8.1).

After removing the level SCSM from the net in Fig. 5.1, the resulting net is a live and safe FC net. Hence, the net can be handled by the SMD-subnet method.

### 5.2.3 Release of non-FC nets

Hack [Hac74] suggests an extension of the FC net into an SMD net by means of a net release method. A similar FC net extension to a so-called *state machine allocatable* (SMA) net is suggested by [JV80] and [Sta90]. Also, Esparza [ES91] has suggested a similar FC extension of live and safe nets based on handles.

**5.2.5.** DEFINITION. Let $N$ be an ordinary net with $p \in P, t \in T$ such that $|p^\bullet| > 1, |{}^\bullet t| > 1$ and $t \in p^\bullet$. The arc $(p, t)$ becomes *released* iff we modify the net in the following way:
$P' = P \cup \{p'\}$; $M_0(p') = 0$
$T' = T \cup \{t'\}$; $l(t') = \lambda$
$F' = F - \{(p, t)\} \cup \{(p, t'), (t', p'), (p', t)\}$

A net is in *released form* when all arcs from a place to a transition satisfying the conditions above have been released.

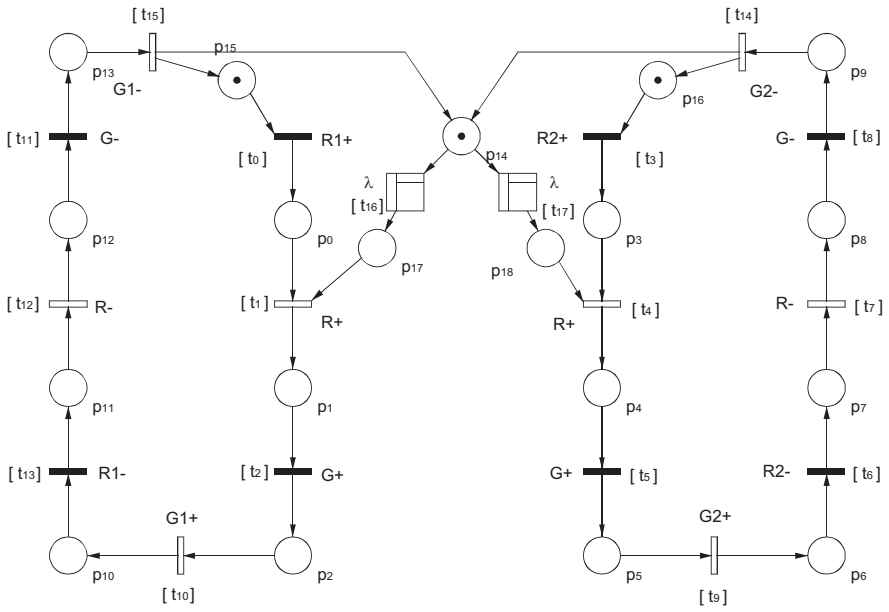


Figure 5.9. The released net of the NEI_Arbiter example in Fig. 4.9a

As an example, the NEI_Arbiter net in Fig. 4.9a has the released form shown in Fig. 5.9 – $(p_{14}, t_1)$ is released into $(p_{14}, t_{16}, p_{17}, t_1)$; $(p_{14}, t_4)$ is released into $(p_{14}, t_{17}, p_{18}, t_4)$.

**5.2.6.** PROPOSITION. *Let $N'$ be the released form of a net $N$, then $N'$ is an FC net. Further, if $N'$ has a live and safe marking, so has $N$.*

**Proof:** see [Hac74]. □

After releasing the NEI_Arbiter net, the resulting net is a live and safe FC net. Hence, the net can be handled by the SMD-subnet method.

## 5.3 Finding an SCSM cover algorithm

From proposition 3.1.4, in a live and safe FC net $N$, a subnet $N_S$ induced by a minimal siphon $P_S$ is an SCSM subnet of $N$. Therefore, the algorithm to find a minimal siphon could be used to find a safe SCSM subnet from a live and safe FC net. The algorithm *find_safe_scsm_subnet* is derived from the algorithm *S-subnet* [EBS89] with certain additions for safe SCSMs: If there already is a token in the partial subnet $N_S$, then there should be no token in any place of

the meeting path handle $N_H$ given by the subroutine *get_meeting_path_handle*; otherwise, there should be no more than one token in $N_H$ (line 5 - line 6 of the algorithm *find_safe_scsm_subnet*).

**Algorithm** *find_safe_scsm_subnet*
**Input:** a live and safe FC net $N$ which is strongly connected, with a *seed* place $\overline{p}$
**Output:** a safe SCSM subnet $N_S$
1. $P_S := \{\overline{p}\}$; $T_S := \emptyset$; $F_S := \emptyset$;
2. **while** $\exists p \in P_S$ and $\exists t \in {}^\bullet p$ such that $(t,p) \notin F_S$ **do**
3. *get_meeting_path_handle*$(N_S, N, p, t, N_H)$;
4. $(*$ $N_H$ is a handle that begin with a node $x$ in $N_S$ and ends with $t, p$ $*)$
5. $(*$ if $\exists p' \in P_S$ and $M_0(p') \neq 0$ then $\forall p'' \in P_H, M_0(p'') = 0$ $*)$
6. $(*$ otherwise, $\sum_{i=1}^{|P_H|} M_0(p_i) \leq 1$ $*)$
7. $P_S := P_S \cup P_H$; $T_S := T_S \cup T_H$; $F_S := F_S \cup F_H$;
8. $(*$ end of while there is a meeting path of $N_S$ $*)$

The SCSM subnet $N_i$ of the live and safe FC net $N$ found by the algorithm *find_safe_scsm_subnet* is synchronizable, because $P_i$ is a minimal siphon which is also a trap (see proposition 3.1.3); therefore, the SCSM subnet $N_i$ has the property of an ST-subnet (see definition 2.1.15): $T_i = {}^\bullet P_i \cup P_i^\bullet$. This fulfils the requirement for synchronization in Definition 5.1.2.

**Algorithm** *find_scsm_cover*
**Input:** a live and safe FC net $N$ which is strongly connected
**Output:** a safe cover $\chi$
1. $P_{taken} = \emptyset$;
2. **for** every place $\overline{p} \in P$ that has a token **do**
3. *find_safe_scsm_subnet*$(N, \overline{p}, N_S)$;
4. add the safe SCSM $N_S$ to $\chi$; $P_{taken} = P_{taken} \cup P_S$;
5. $(*$ end of for all places $\overline{p} \in P$ that have a token $*)$
6. **while** $(P - P_{taken}) \neq \emptyset$ **do**
7. take a place $p \in (P - P_{taken})$;
8. *find_safe_scsm_subnet*$(N, p, N_S)$;
9. add the safe SCSM $N_S$ to $\chi$; $P_{taken} = P_{taken} \cup P_S$;
10. $(*$ end of while $(P - P_{taken}) \neq \emptyset$ $*)$

In a live and safe FC net, a safe cover exists (see proposition 3.1.2). This also means, that the algorithm *find_safe_scsm_subnet* can always find a safe SCSM from the given seed place.

The algorithm *find_scsm_cover* begin with places in $N$ that contain a token. For each place with a token, the corresponding safe SCSM subnet is found (line 2 - line 5). After that, for each place not yet covered, the corresponding safe SCSM subnet is found (line 6 - line 10). When all the places of $N$ are covered, then $\chi$ is the safe cover of $N$.

**5.3.1.** PROPOSITION. *The cover $\chi$ found by the algorithm find_scsm_cover is a safe cover of $N$*

**Proof:** As argued above, each $N_i \in \chi$ is an ST-subnet with $T_i = {}^\bullet P_i \cup P_i^\bullet$. $N$ itself is also an ST-subnet, because $P = \cup_{i=1}^{|\chi|} P_i$ is the union of minimal siphons which also are traps; i.e. $P$ is also a siphon which is a trap and has the property $T = {}^\bullet P \cup P^\bullet$. Hence we have ${}^\bullet P \cup P^\bullet = {}^\bullet(\cup_{i=1}^{|\chi|} P_i) \cup (\cup_{i=1}^{|\chi|} P_i)^\bullet = \cup_{i=1}^{|\chi|}({}^\bullet P_i \cup P_i^\bullet) = \cup_{i=1}^{|\chi|} T_i = T$. Because each $N_i \in \chi$ is a subnet, we have $F = \cup_{i=1}^{|\chi|} F_i$. □

As an example for the algorithm *find_scsm_cover*, the net in Fig. 5.2 is taken. The places $p_1$,$p_3$, and $p_8$ each have a token. First, the algorithm *find_safe_scsm_subnet* is called, with $p_1$ taken as seed place (line 3 of the algorithm *find_scsm_cover*). $P_S$ of the partial subnet $N_S$ has only one element, $p_1$. There is a meeting path $(t_0, p_1)$ that is not in $F_S$. Hence, the meeting path handle $N_H$ of $N_S$ should be found (line 3 of the algorithm *find_safe_scsm_subnet*). Because a place $p_1 \in P_S$ has a token, all the places in $P_H$ should be empty (line 5 of the algorithm *find_safe_scsm_subnet*). The meeting path handle $(p_1, t_1, p_0, t_0, p_1)$ is found, resulting in $N_4$ after addition to $N_S$. $N_4$ has no other meeting path. Hence, the algorithm *find_safe_scsm_subnet* terminates. $N_4$ is a safe SCSM. Next, the algorithm *find_safe_scsm_subnet* is called with $p_3$ as seed place. The meeting path handle $(p_3, t_2, p_2, t_1, p_3)$ is found resulting in $N_5$. Then, the algorithm *find_safe_scsm_subnet* is called for the third time, with $p_8$ as seed place. The possible meeting path handles are $N_{H_1}$ which results in $N_1$, and $N_{H_3}$, which yields $N_3$. Choosing $N_3$ the algorithm *find_scsm_cover* terminates because every $p \in P$ is already covered; it has found the safe cover $\chi_1 = \{N_3, N_4, N_5\}$.

Taking $N_1$ (via $N_{H_1}$) instead of $N_3$ leaves the places $p_5, p_{10}, p_9$ and $p_{11}$ uncovered. This means, the algorithm *find_safe_scsm_subnet* must be called again, this time with $p_5$ as the seed place (line 8 of the algorithm *find_scsm_cover*). In this case, the meeting path handle must contain one token, because $p_5$ has no token (line 6 of the algorithm *find_safe_scsm_subnet*). The meeting path handle is $N_{H_3}$ yielding $N_3$. The algorithm *find_scsm_cover* terminates with the safe cover $\chi_2 = \{N_1, N_3, N_4, N_5\}$. This example shows that it is possible to have redundant SCSMs in a safe cover by the algorithm *find_scsm_cover*. It also shows that which of several possible safe covers is found by the algorithm depends on the seed place given as input to the *find_scsm_cover* algorithm.

The [EBS89] algorithm is a linear time implementation algorithm, suggested by [Kem93]. The C++ implementation of net covering with minimal siphons based on the [Kem93] algorithm and some experimental results can be found in [War05]

## 5.4 SMD subnet algorithm

Combining the fragments discussed in the previous section, the algorithm for decomposing an STG with SMD-subnets as initial components is described as follows.

**Algorithm** *SMD-Subnet*
**Input:** a net $N$, a feasible partition $\Pi$ (a set of output blocks $B$)
**Output:** a set of $N(B)$ (components of $N$ based on output blocks $B$)

1. $\chi := \emptyset$;
2. **if** $N$ is not an FC net **then**
3. find regulation circle paths; add to $\chi$;
4. find level SCSMs; add to $\chi$;
5. release the net $N$ into $N'$;
6. ($*$ end of if $N$ is not an FC net $*$)
7. **if** $N'$ is not a well specified FC net **then**
8. report($N$ is not a well specified FC net); exit;
9. *find_scsm_cover*$(N', \chi)$;
10. **for** every output signal $o$ in $Out$ **do**
11. $\Sigma_c(o) =$ *find_concessioner*$(N, o)$;
12. **for** every output block $B$ in $\Pi$ **do**
13. $\Sigma_c(B) = \emptyset$;
14. **for** every output signal $o$ in $B$ **do** $\Sigma_c(B) = \Sigma_c(B) \cup \Sigma_c(o)$;
15. $\Sigma_{nc}(B) = \{In \cup Out\} - \{\Sigma_c(B) \cup B\}$;
16. ($*$ end of for every output block $B$ in $\Pi$ $*$)
17. **for** every output block $B$ in $\Pi$ **do**
18. $\chi_{nc}(B) := \emptyset$; $\chi_c(B) := \emptyset$;
19. **for** every $N_s \in \chi$ **do**
20. **if** $\exists t \in T_s$ and signal label of $t \in \Sigma_c(B)$ **then** $\chi_c(B) := \chi_c(B) \cup N_s$;
21. ($*$ end of for all $N_s \in \chi$ $*$)
22. $N(B)$ is a synchronization of all SCSM subnets in $\chi_c(B)$;
23. $N'(B) = N(B)$; $\chi_{nc}(B) := \chi - \chi_c(B)$;
24. **repeat**
25. $backtracking = false$; $T_l = \emptyset$;
26. **for** every non-concessioner signal $\sigma$ in $\Sigma_{nc}(B)$ **do**
27. ($*$ change irrelevant transition into $\lambda$-transition $*$)
28. **for** every transition $t \in T_{N(B)}$ labelled $\sigma$ **do**
29. $l(t) = \lambda$; $T_l = T_l \cup \{t\}$;
30. ($*$ end of for every non-concessioner signal $\sigma$ in $\Sigma_{nc}(B)$ $*$)
31. **for** every non-concessioner signal $\sigma$ in $\Sigma_{nc}(B)$ **do**
32. **for** every transition $t \in T_{N(B)}$ labelled $\sigma$ **do**
33. **if** !*net_reduction*$(N(B), t, T_l)$ **then** $backtracking = true$; break;
34. ($*$ end of for every transition $t \in T_{N(B)}$ labelled $\sigma$ $*$)

35. **if** *backtracking* **then**
36. $\Sigma_{nc}(B) = \Sigma_{nc}(B) - \{\sigma\}$; $\Sigma_c(B) = \Sigma_c(B) \cup \{\sigma\}$;
37. **for** every $N_s \in \chi_{nc}(B)$ **do**
38. **if** $\exists t' \in T_S$ labelled $\sigma$ **then**
39. synchronize $N'(B)$ with $N_s$; $\chi_c(B) := \chi_c(B) \cup N_s$;
40. $(*$ end of for every all $N_s \in \chi_{nc}(B)$ $*)$
41. $\chi_{nc}(B) := \chi - \chi_c(B)$; $N(B) = N'(B)$; break;
42. $(*$ end of if *backtracking* $*)$
43. $(*$ end of for every non-concessioner signal $\sigma$ in $\Sigma_{nc}(B)$ $*)$
44. **until** !*backtracking*
45. $(*$ end of for every output block $B$ in $\Pi$ $*)$

The algorithm *SMD-Subnet* begins with the transformation into an FC net, if $N$ is non-FC net (line 2 - line 6). If by transformation, a safe SCSM (a regulation circle path or a level SCSM) is found, then this is added to a safe cover $\chi$. The net $N'$ resulting from transformation is an FC net. Then, $N'$ is checked whether it is well specified or not (line 7 - line 8). If $N'$ is not well specified then the algorithm terminates without a cover; otherwise a safe cover of $N'$ is found. After that, the set of concessioner signals $\Sigma_c(o)$ of each output signal $o$ in *Out* is determined (line 10 - line 11). From that, the set of concessioner, $\Sigma_c(B)$, and non-concessioner signals $\Sigma_{nc}(B)$ is found for each output block $B$ in the feasible partition (line 12 - line 16). Subsequently, the component for each output block $B$ in the feasible partition $\Pi$ is obtained as follows (line 17 - line 45):

1. After adding relevant SCSMs to $\chi_c(B)$ (line 19 - line 21), the initial component $N(B)$ is found by synchronizing all the relevant SCSMs in $\chi_c(B)$. $\chi_{nc}(B)$ is the safe cover $\chi$ without $\chi_c(B)$. $N(B)$ is copied to $N'(B)$ for backtracking purposes.

2. After irrelevant transitions are silenced (line 26 - line 30), the *net_reduction* algorithm is called for each divining transition in $N(B)$ (line 31 - line 43). The *net_reduction* algorithm does secure $t$-contractions and deletes redundant places and divining transitions if any exist in the net $N(B)$ resulting from secure $t$-contraction. If a $t$ cannot be contracted, then backtracking should be done. Backtracking can be done in the SMD-subnet method by synchronizing the SCSMs that have transitions labelled $l(t)$ in $\chi_{nc}(B)$, with $N'(B)$ (line 37 - line 40). Then, $N'(B)$ is copied to $N(B)$.

3. Repeat step 2, while backtracking is needed.

First, the FIFO net in Fig. 5.10 is taken as an example. Because the FIFO net is a marked graph, lines 2 to 6 are skipped. It is also well specified. Hence, a safe cover can be found. Fig. 5.11 shows a possible safe cover. For the *Ain*-component, relevant signals are *A*1 and *Ain*. Hence, $N_2, N_3, N_4$ are the relevant SCSMs for the

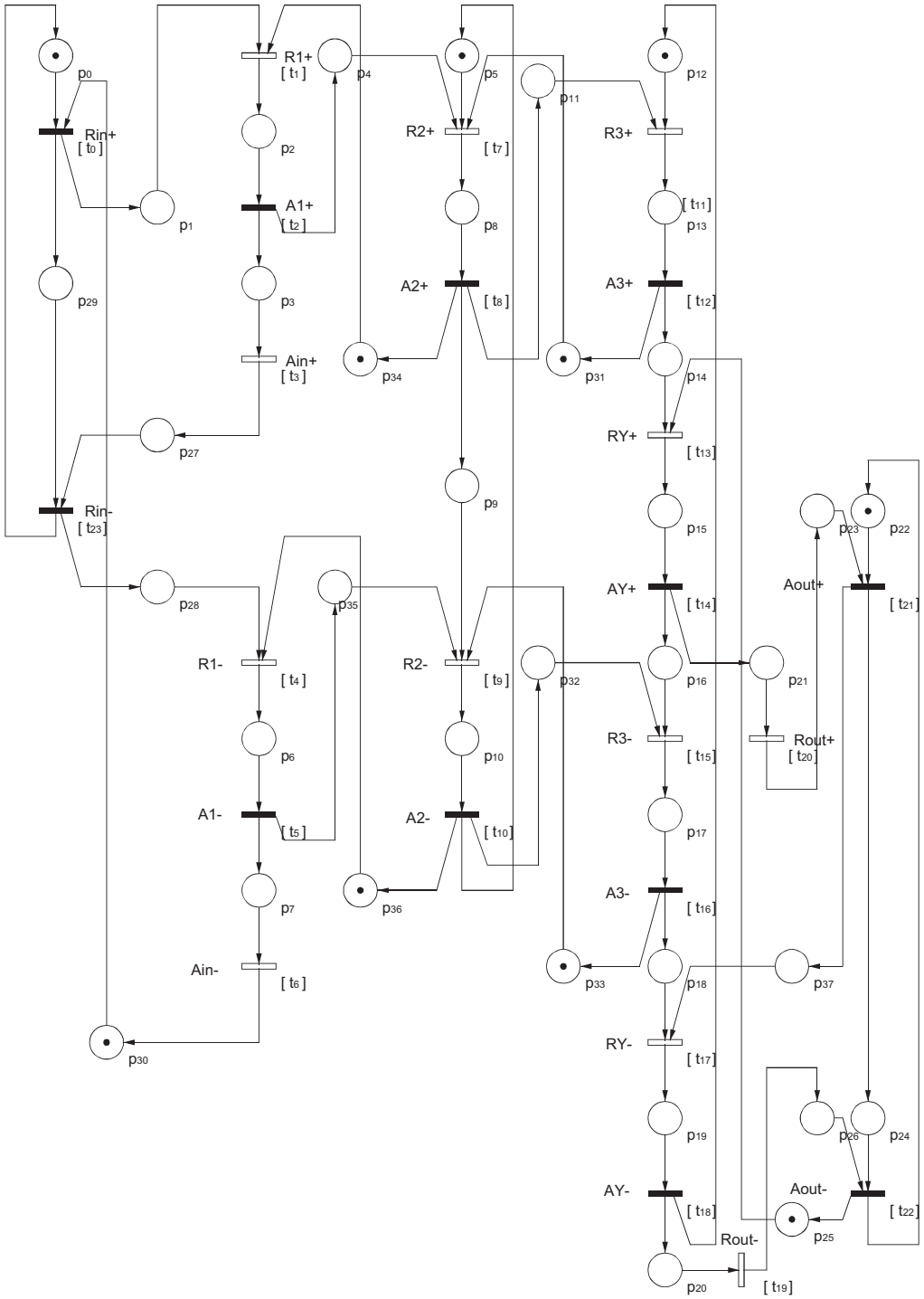


Figure 5.10. FIFO example [BW93]

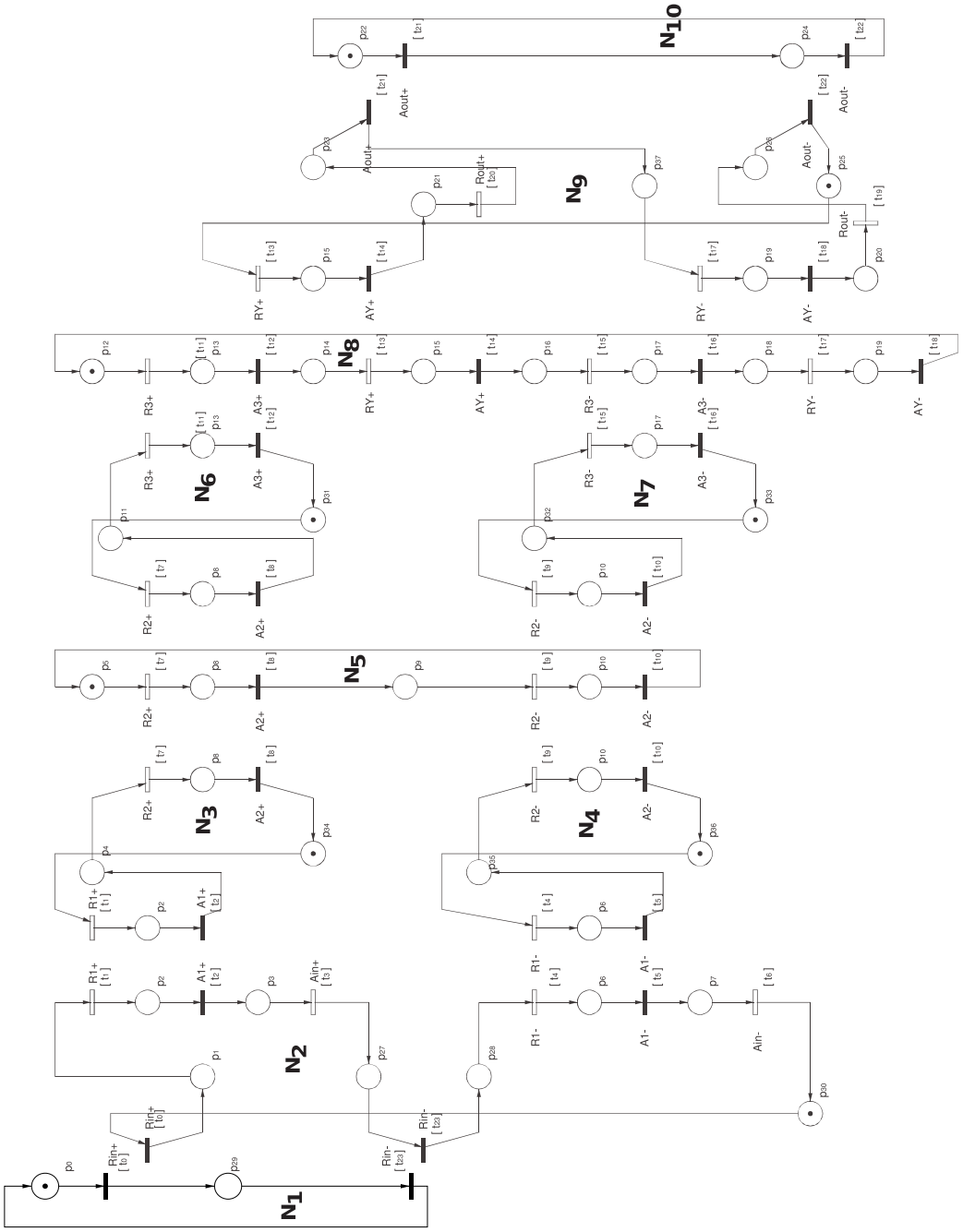


Figure 5.11. A safe cover of the FIFO example

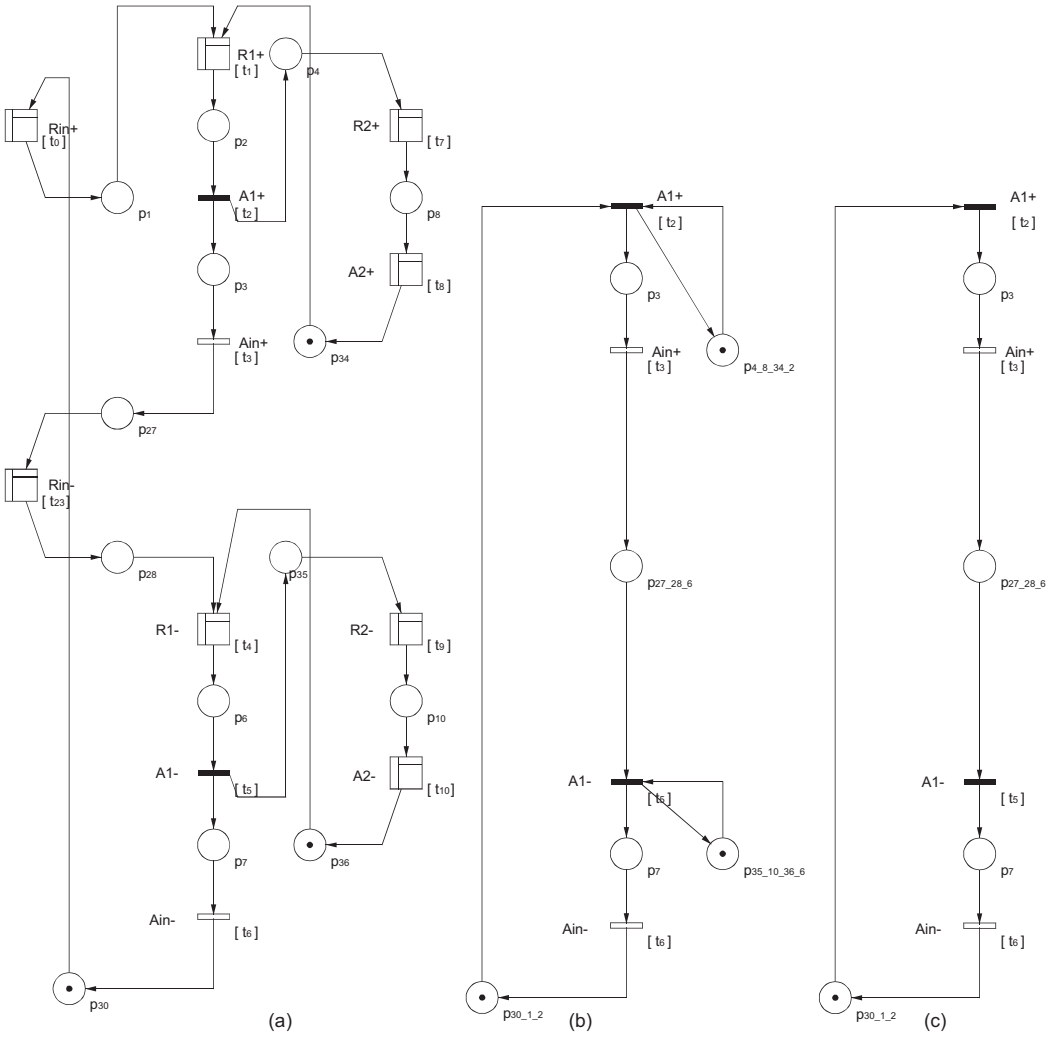


Figure 5.12. (a) Initial *Ain*-component, (b) intermediate *Ain*-component after contracting all irrelevant transitions, (c) final *Ain*-component after deleting redundant places $p_{4\_8\_34\_2}$ and $p_{35\_10\_36\_6}$

*Ain*-component. The SMD-subnet for the *Ain*-component after $N_2, N_3$ and $N_4$ have been synchronized and irrelevant transitions silenced, is shown in Fig. 5.12a. After contracting transitions with labels $R1$ $(t_1, t_4)$, $R2$ $(t_7, t_9)$, $A2$ $(t_8, t_{10})$, and $Rin$ $(t_0, t_{23})$, the result is shown in Fig. 5.12b. The final *Ain*-component is shown in Fig. 5.12c after deleting the redundant places $p_{4\_8\_34\_2}$ and $p_{35\_10\_36\_6}$. Compared with the [VW02] method (where the initial *Ain*-component is the complete FIFO net with all transitions except the $A1$- and *Ain*-transitions silenced; step by step examples for the *Ain*-component are shown in the AG-Beister website) and the silenced transitions in the SMD component can be contracted more easily than those of the complete FIFO-net.

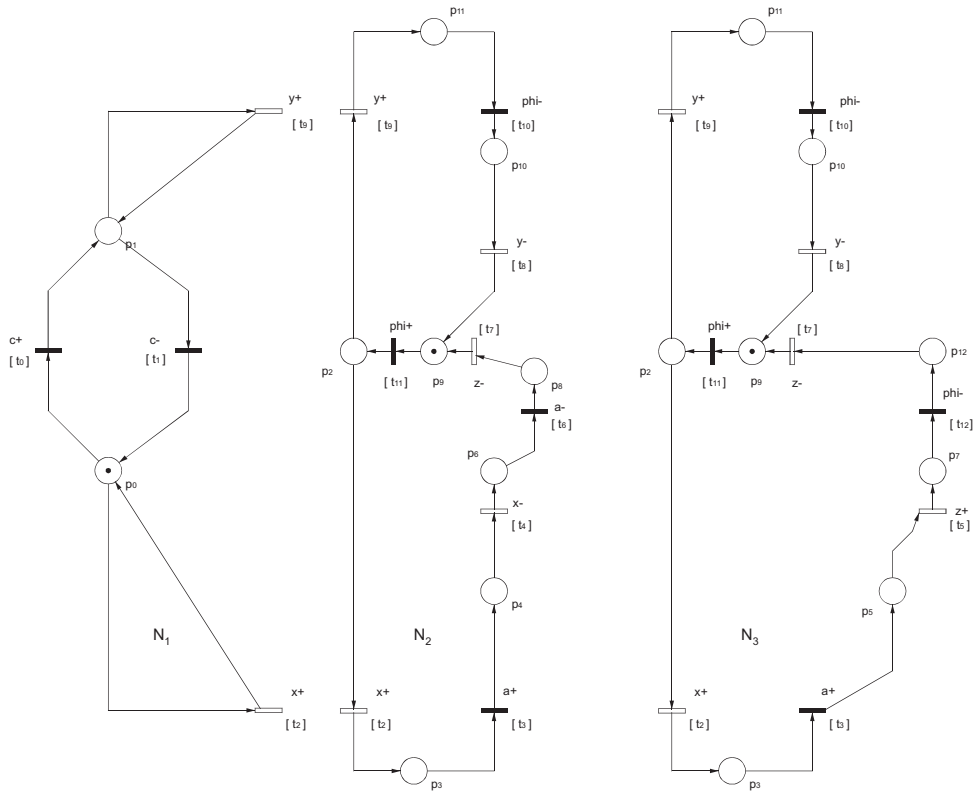


Figure 5.13. A safe cover of the async99* example (in Fig. 5.1)

Another example is async99* shown in Fig. 5.1. Without its level SCSM($N_1$ in Fig. 5.13), the async99* net becomes a live and safe FC net. Hence, the SMD-subnet method can be applied. A safe cover $\chi = \{N_1, N_2, N_3\}$ of async99* is shown in Fig. 5.13. Relevant signals for the $z$-component are $a, phi$, and $z$. Hence, the relevant SCSMs for the $z$-component are $N_2$ and $N_3$. The SMD-subnet for $z$-component after $N_2$ and $N_3$ are synchronized and irrelevant transitions are silenced is shown in Fig. 5.14a. After contracting transitions with labels $y$ $(t_9, t_8)$ and $x$ $(t_2, t_4)$, the result is shown in Fig. 5.14b. Compared with the [VW02] result

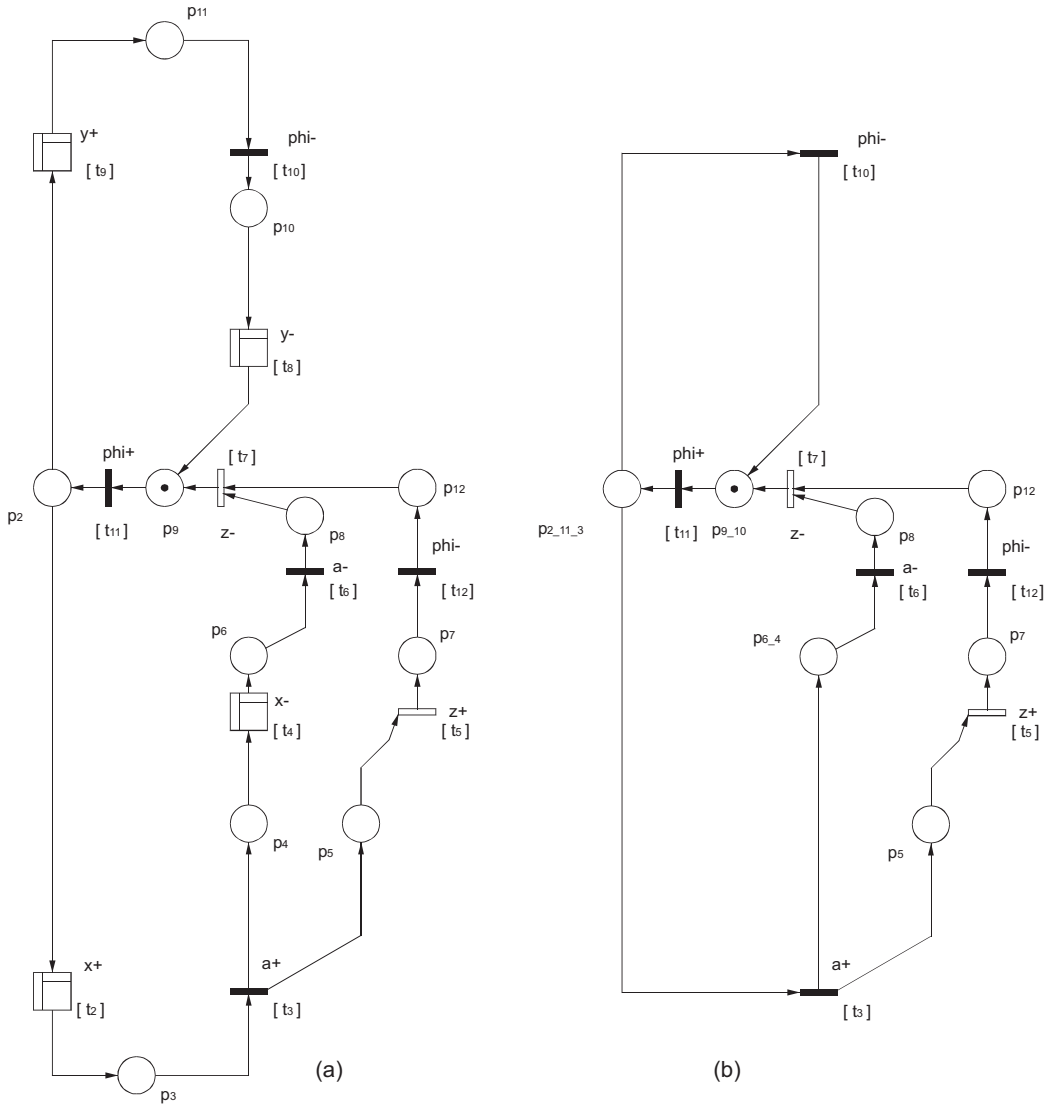


Figure 5.14. The async99* example, (a) SMD-subnet for the $z$-component, (b) final $z$-component

for the $z$-component (the net in Fig. 5.1 with signals $x, y$ as input signals), the SMD-subnet method produces a much smaller result.

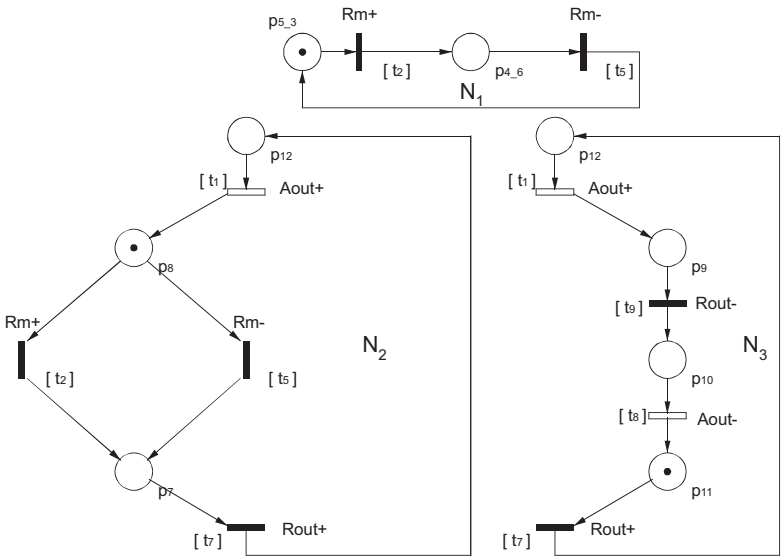


Figure 5.15. A safe cover of the net in Fig. 3.4

Without the regulation net, the net in Fig. 3.4e is a live and safe FC net. Hence, the SMD-subnet method can be applied. A safe cover $\chi = \{N_1, N_2, N_3\}$ of the net is shown in Fig. 5.15. Relevant signals for the *Aout*-component are *Rout* and *Aout*. Hence, the relevant SCSMs for this component are $N_2$ and $N_3$. The initial SMD-subnet for the *Aout*-component is the net in Fig. 5.16a ($N_2$ and $N_3$ have been synchronized, and irrelevant transitions were silenced). Transition $t_2$ becomes a duplicate of $t_5$ and can be deleted. After deletion (Fig. 5.16b), $t_5$ can be securely contracted, resulting in the final *Aout*-component (Fig. 5.16c) without *Rm* as relevant input signal. Compared with [VW02] result for the *Aout*-component (the same net in Fig. 3.4e), the SMD-subnet method produces a significantly smaller result.

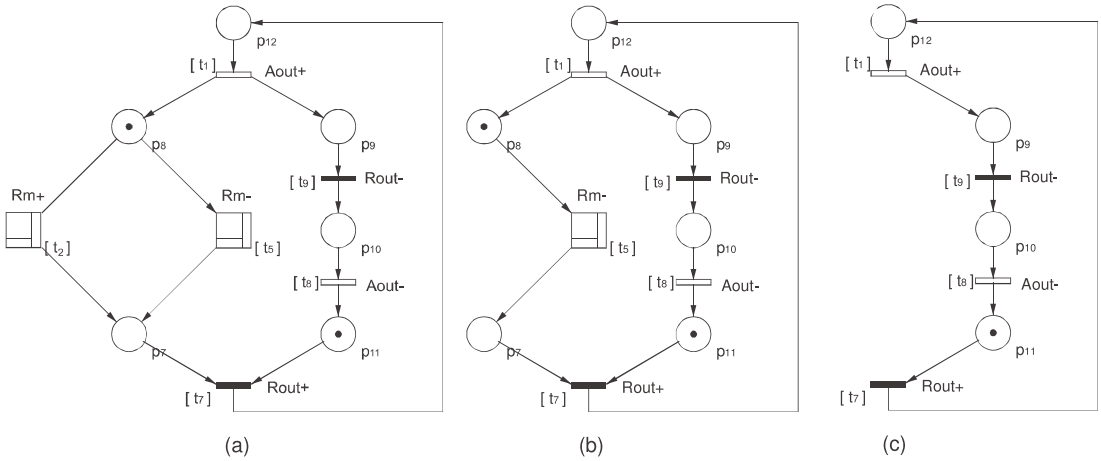


Figure 5.16. (a) Initial SMD-subnet for *Aout*-component, (b) after deleting duplicate transition $t_2$, (c) after contracting $t_5$

# Chapter 6

# P/T-net Abstraction into Structure Graphs

When searching for SCSM subnets (see section 5.3), place and transition nodes in a P/T-net should be traversed to find the meeting path handle. Clearly, the algorithm's complexity depends on the number of nodes in the net. For example, to find the meeting path handle for place $p_8$ in Fig. 5.2, the nodes $p_8, t_5, p_4, t_2, p_2, t_1, p_0, t_0, p_6, t_3, p_7, t_4$ should be traversed first.

Only nodes with a branch play a role in deciding how the graph will be traversed; Nodes without branching have no role in this case. A sequence of nodes without branch – for example, in Fig. 5.2 the sequence of nodes $p_6$, $t_3$, $p_7$, $t_4$, $p_8$, $t_5$, $p_4$ of $N_1$ could be grouped together into a super node $\omega_1$ as shown in Fig. 6.1a.

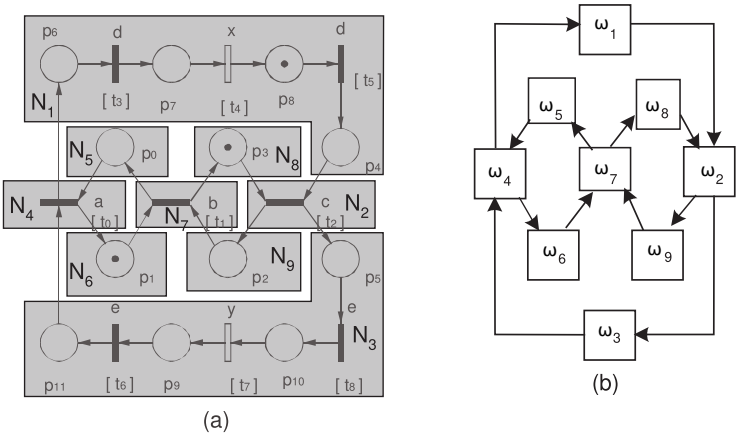


Figure 6.1. (a) a net $N$ from Fig. 5.2 (b) structure graph $S_N$ of $N$

## 6.1 Structure Graphs

A graph which abstracts nodes in an ordinary P/T-net will be called a *structure graph*. There are no places or transitions in the structure graph. Instead, *super node* will be introduced.

**6.1.1.** Definition. A node (place or transition) of the P/T-net which has more than one pre arc or post arc is a *branch node*. Nodes with a single pre and a single post arc are called *non-branch nodes*.

Merge places, conflict places, fork transitions, and join transitions are branch nodes according to this definition.

*Non-branch nodes* are a place or a transition with only one pre- and one post arc (Fig. 6.2).


Figure 6.2. Non-branch node types

**6.1.2.** Definition. A *super node* $\omega$ is the abstraction of a maximal-length alternating sequence of P/T-transitions and places without internal branches. The length of the sequence is denoted by $|\omega|$, with $|\omega| \geq 1$. The maximal-length condition ensures that the structure graph of a P/T-net is unique.
Branching is confined to the pre arcs of the first or *head node* and to the post arcs of the last or *tail node* of the sequence. *Middle nodes*, i.e. nodes between head and tail node, are non-branch nodes.

The first or head node of a super node can be a branch node with more than one pre arc (Fig. 6.3a-b), or a node with a single pre arc which is one of several post arcs of the preceding node (Fig. 6.3c-d).


Figure 6.3. Head node types

The last or tail node of a super node can be a branch node with more than one post arc (Fig. 6.4a-b), or a node with a single post arc which is one of two or more pre arcs of the successor node (Fig. 6.4c-d).


Figure 6.4. Tail node types

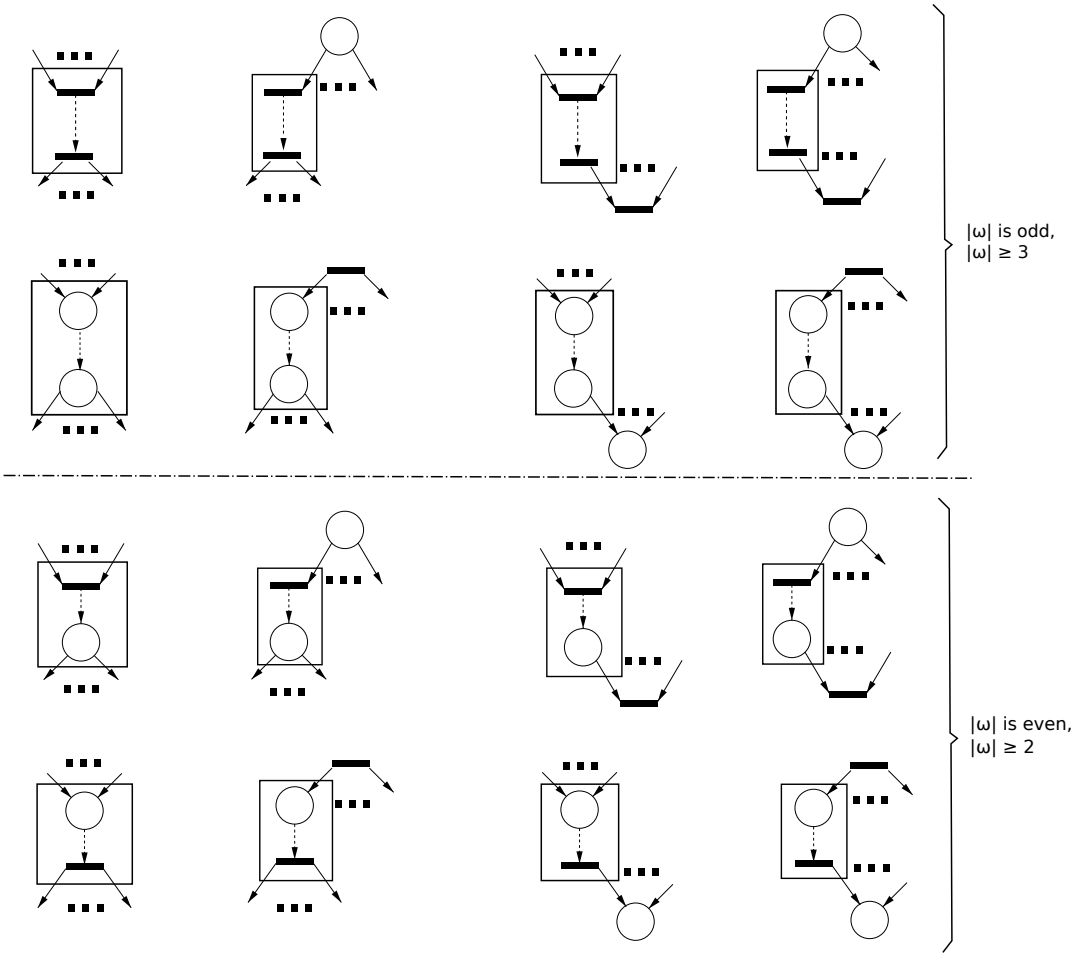


Figure 6.5. The 16 types of possible super nodes

The *middle node* types are the two non-branch node types shown in Fig. 6.2.

Each head node type can be paired with each tail node type, resulting in the complete set of 16 basic super node types. In Fig. 6.5 they have been arranged in 4 groups (rows) of 4 types each. Subtypes can be distinguished by their length, $|\omega|$, the main distinction being that between odd and even lengths (upper and lower half of Fig. 6.5).

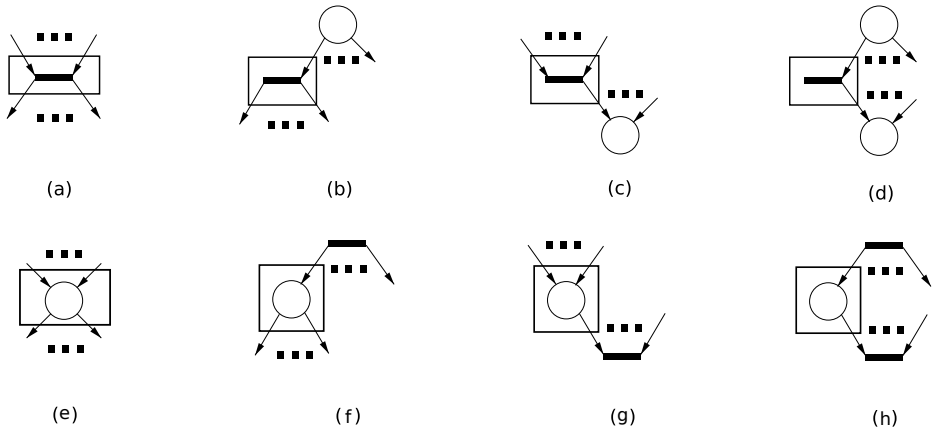


Figure 6.6. Super nodes $\omega$ with $|\omega| = 1$

Head and tail node can be one and the same node: no middle nodes, $|\omega| = 1$, odd. There are eight such super node types, see Fig. 6.6, obtained from the upper (odd length) half of Fig. 6.5 by merging the head and tail transitions respectively places.

For $|\omega| = 2$, (even head node, tail node, no middle node) there also are eight super node types, obtainable by joining head and tail node by an arc in the lower half of Fig. 6.5.

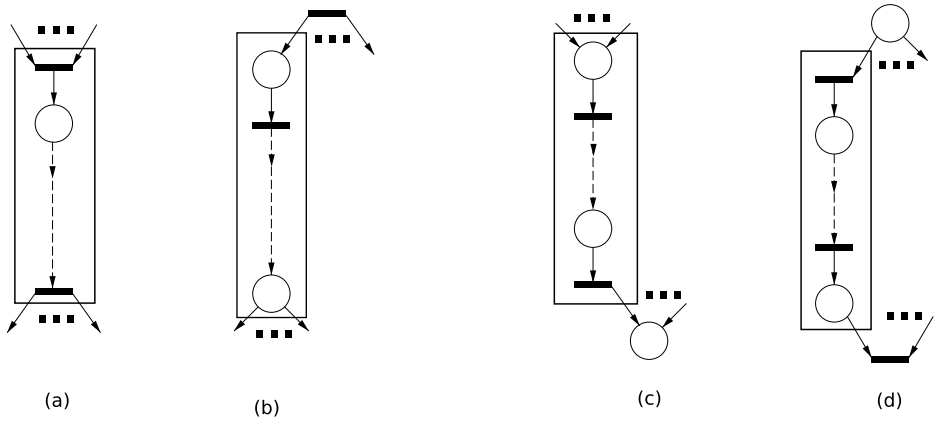


Figure 6.7. (a) and (b) Super nodes $\omega$ with $|\omega| \geq 3$, odd; (c) and (d) Super nodes $\omega$ with $|\omega| \geq 4$, even.

Finally, Fig. 6.7 shows one type from each of the four groups of Fig. 6.5 for $|\omega| \geq 3$ (both odd and even).

The most general characterization of super nodes is by their possible interconnections with predecessor and successor super nodes. Inspection of Fig. 6.5 reveals that only four such interconnection structures are possible. All four appear in every row of Fig. 6.5. They are (see Fig. 6.8):

- several pre arcs to the head end of the super node in question from preceding super nodes (predecessors) and several post arcs from its tail end to successor super nodes (successors) (Fig. 6.8a);
- only one pre arc to the head end from a predecessor with several post arcs and several post arcs from its tail end to successors (Fig. 6.8b);
- several pre arcs to its head from predecessors and only one post arc from its tail to a successor with several pre arcs (Fig.6.8c), and
- only one pre arc to its head from a predecessor with several post arcs, and only one post arc from its tail to a successor with several pre arcs (Fig. 6.8d).

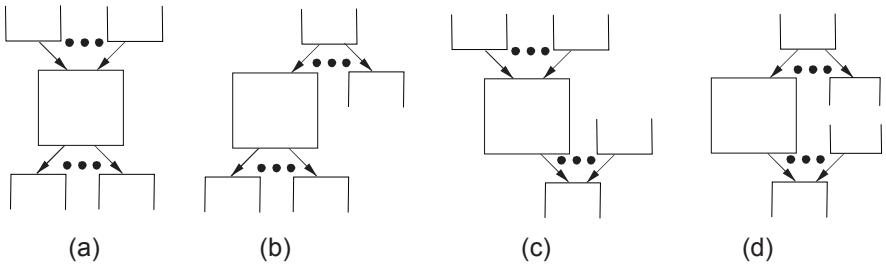


Figure 6.8. Possible interconnection structures of super nodes.

**6.1.3.** Definition. The *structure graph* $S_N$ of an ordinary P/T net $N$ is a tuple $(\Omega, F)$ where
$\Omega$ is the set of super nodes according to Def. 6.1.2, and
$F \subseteq \Omega \times \Omega$ is the flow relation.

A super node $\omega$ is represented as a rectangle in the structure graph.

As an example, the structure graph for the net in Fig. 5.2 is shown in Fig. 6.1b. The meeting path handle for place $p_8$ in the sequence $N_1$ associated with super node $\omega_1$ can be found more quickly now: Only super nodes $N_1, N_2, N_9, N_7, N_5, N_4$ need to be traversed. This is only half the number of nodes that have to be traversed without abstraction.

**Algorithm** *CreateStructureGraph*

**Input:** a strongly connected P/T-net $N = (P, T, F)$ with branch nodes
**Output:** structure graph $S_N = (\Omega, F)$

```
1.  The status of all nodes (places and transitions) of N is set to not visited;
2.  find a node n which has more than one pre arc;
3.  status(n) := visited;
4.  create a new super node ω (as yet incomplete) with element n as head node;
5.  insert ω into the set Ω;
6.  push ω onto stack;
7.  while stack is not empty do
8.     pop a super node ω from stack;
9.     take a node from ω as current node n;
10.    if n has only one post arc then
11.       while ∃n' ∈ n•, n' is a non-branch node do
12.          add n to ω; n := n';
13.       (* end of while ∃n' ∈ n•, n' is not a branch node *)
14.       if n' has only one pre arc then add n to ω; n := n';
15.    else
16.       (* ω is complete *)
17.    (* end of if n has only one post arc *)
18.    for all node n* ∈ post nodes of n do
19.       if status(n*) is visited then
20.          find an ω' in Ω with head(ω') = n*;
21.       else (* n* is not visited *)
22.          status(n*) := visited;
23.          create a new super node ω' with element n* as head node;
24.          insert ω' into the set Ω;
25.          push new ω' into stack;
26.       (* end of if status(n*) is visited *)
27.       insert an arc from ω to ω' into F;
28.    (* end of for all node n* ∈ post nodes of n *)
29. (* end of while stack is not empty *)
```

The algorithm *CreateStructureGraph* begins with a branch node $n$ which has more than one pre arc (see Fig. 6.9a); because the net is strongly connected and there is a branch in the P/T net, such a node can always be found. $n$ is visited and inserted into a super node $\omega$. Then, the super node $\omega$ is inserted into $\Omega$ and placed in the stack. While there is a super node $\omega$ in the stack, the following is done (line 7 - line 29):

- (line 8 - line 17) Take a super node $\omega$ from the stack. $\omega$ has only one node $n$. If $n$ has more than one post arc, then $\omega$ has only this node (see Fig. 6.9e); otherwise $n$ is a head node with one post arc and all the sequential nodes of $n$ which are not a branch nodes, are added to $\omega$ (see Fig. 6.9b) until there

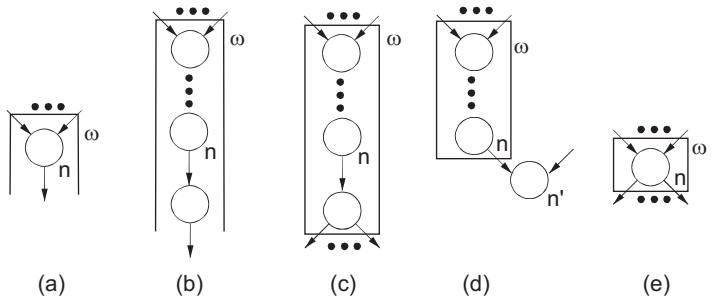


Figure 6.9. Illustration for the algorithm *CreateStructureGraph*

is a sequential node $n'$ of $n$ which is a branch node. If $n'$ has only one pre arc then $n'$ is a tail node and is added to $\omega$ (see Fig. 6.9c); otherwise the pre node of $n'$ is the tail node (see Fig. 6.9d).

- (line 18 - line 28) For each post node $n*$ of tail node of $\omega$, add an arc from the super node $\omega$ to another super node $\omega'$ which has $n*$ as head node. If $n*$ has not been visited, then $n*$ is visited and inserted into a super node $\omega'$. Subsequently, the super node $\omega'$ is inserted into $\Omega$ and placed in the stack.

Note that, though the algorithm *CreateStructureGraph* could begin with a different initial node, it always creates the same structure graph for the P/T net.

Consider the net $N$ in Fig. 6.10a as an example for the algorithm *CreateStructureGraph*. First, $t_9$ which has more than one pre arc is visited and is inserted into a super node $\omega_1$. $\omega_1$ is placed in the stack. As the only node in stack, $\omega_1$ is taken and is inserted into $\Omega$. Because $t_9$ has only one post arc ($t_9$ is the head node), the next node of $t_9$, $p_0$, which is not a branch node, is added to $\omega_1$.

The only post node of $p_0$ is the branch node $t_0$. It is inserted into $\omega 1$ because it has only one pre arc, and becomes the tail node of $\omega_1$ because of it two post arcs.

Then each of the two post nodes of $t_0$, $p_1$ and $p_2$, both unvisited are connected to $\omega_1$ as the head nodes of new super nodes $\omega_3$ and $\omega_2$, respectively. $\omega_2$ and $\omega_3$ then are placed on the stack.

Next, $\omega_3$ with head node $p_1$ is removed from the stack and inserted into $\Omega$. The construction of $\omega_3$ is then continued by adding the sequence $t_1$, $p_3$, $t_2$, $p_4$, $t_3$, $p_5$, $t_4$, $p_6$ is added node by node to $\omega_3$. $p_6$ is the last node that can be added because it is connected to the branch node $t_9$, which has more than one pre arc and is the head node of $\omega_1$.

Finally, the last super node in the stack, $\omega_2$ with head node $p_2$, is taken from the stack and inserted into $\Omega$. Because $p_2$ has only one post arc ($p_2$ is the head node), the sequence of non-branch nodes $t_5, p_7, t_6, p_8, t_7, p_9, t_8, p_{10}$ is added to $\omega_2$. $p_{10}$ is the last node that can be added to $\omega_2$ because the only next node of $p_{10}$, $t_9$,

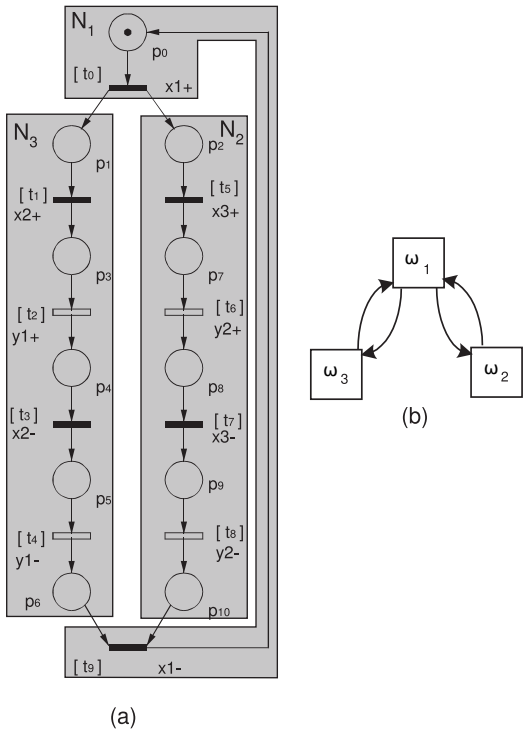


Figure 6.10. (a) a net $N$ (b) its structure graph $S_N$

is a branch node with more than one pre arc. Then, $\omega_2$ is connected to $\omega_1$ which has a visited node $t_9$. The structure graph is shown in Fig. 6.10b.

Fig. 6.11 depicts the wechselpuffer example from the Fig. 3.4a. The corresponding structure graph is shown in Fig. 6.12.

## 6.2 Contracting middle node transitions

The structure graph is not only useful for finding SCSM subnets, but also for contracting middle node transitions. Before contracting a middle node transition $t$, it needs to be ensured that $t$ can be securely contracted.

**6.2.1.** Proposition. *A middle node transition $t$ in a super node fulfills preconditions for contraction and the requirement for secure $t$-contraction.*

**Proof:** According to definition 6.1.2, $t$ is not a branch node; hence, $t$ has only one pre place $p$ and one post place $p'$. $p$ and $p'$ are different places because as a middle node, the $t$ forms no loop with any place. The arc weight requirement is fulfilled because the structure graph is obtained from an ordinary P/T net. $p$ could be either another middle node or a head node; in both cases, $p$ has only one post transition. Also, $p'$ could be either another middle node or a tail node; in both

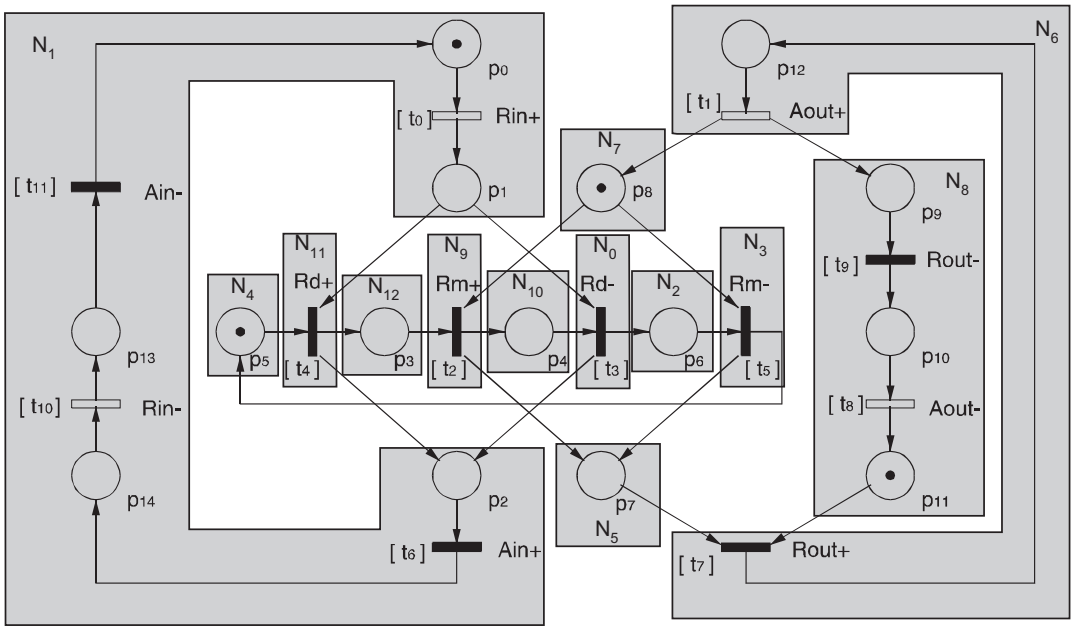


Figure 6.11. Wechselpuffer example

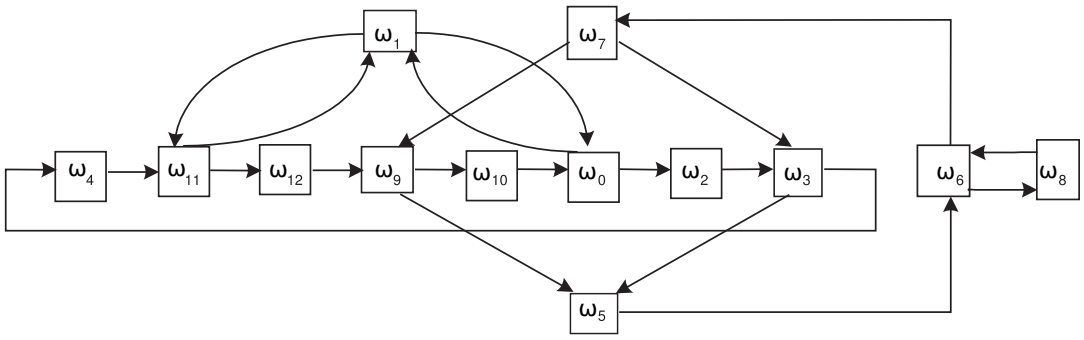


Figure 6.12. Wechselpuffer structure graph

cases, $p'$ has only one pre transition. Therefore, $t$ fulfills the secure $t$-contraction requirement. □

From proposition 6.2.1, a middle node transition $t$ can be securely contracted according to definition 2.2.2. Because $t$ has only one pre place $p$ and one post place $p'$, the contraction is trivial; i.e. by adding a merged place $p''$ which has as its pre transitions the pre transitions of $p$, and as post transitions the post transitions of $p'$, and an initial marking equal to the sum of the tokens in $p$ and $p'$; and then removing $t, p, p'$ and all their incident arcs.

The above concept for contracting a middle node transition can be extended to contracting a sequence of middle node transitions $T = (t, ..., t')$, which has the only pre place of $t$ as its pre place and the only post place of $t'$ as its post place. Instead of contracting each transition successively, all the transitions in $T$ are contracted in one step; i.e. by adding a merged place $p''$ which has as pre transitions the pre transitions of $p$, as post transitions the post transitions of $p'$ and an initial marking equal to the sum of tokens in the place nodes $P = (p, ..., p')$; and by then removing all the transition in $T$, all the places in $P$, and all their incident arcs.

An example is the net in Fig. 3.4b, which has the same super nodes as the net in Fig. 3.4a. The super node $N_1$ (see Fig. 6.11) has a sequence of middle node divining transitions $t_6, t_{10}, t_{11}, t_0$, the pre place $p_2$ and the post place $p_1$. Contracting all the divining transitions in one step results in the net in Fig. 3.4c, which is obtained by adding a merged place $p_{2\_14\_13\_0\_1}$ with pre transitions $t_4, t_3$, post transitions $t_4, t_3$, and the initial marking equal to the sum of tokens in $p_2, p_{14}, p_{13}, p_0$ and $p_1$, which is one. $t_6, t_{10}, t_{11}, t_0, p_2, p_{14}, p_{13}, p_0, p_1$ and their incident arcs are removed from the net.

Before the above approach is applied to the structure graph, another possible structure of a super node all of whose transition nodes are divining transitions should be considered:

1. A super node $\omega$ with nodes $p, ..., p'$ will become a super node $\omega'$ with a merged place $p''$ (see Fig. 6.13a); $M_0(p'')$ is equal to sum of tokens in all the places from $p$ to $p'$, inclusively.

2. A super node $\omega$ with nodes $p, ..., t$ will become a super node $\omega'$ with nodes $p$ and $t$ (see Fig. 6.13b); $M_0(p)$ is the sum of tokens in the places in $\omega$.

3. A super node $\omega$ with nodes $t, ..., p$ will become a super node $\omega'$ with nodes $t$ and $p$ (see Fig. 6.13c); $M_0(p'')$ is the sum of tokens in the places in $\omega$.

4. A super node $\omega$ with nodes $t, ..., t'$ will become a super node $\omega'$ with nodes $t, p$, and $t'$ (see Fig. 6.13d); $M_0(p'')$ is the sum of tokens in the places in $\omega$.

Only in the case of a super node $\omega$ with nodes $p, ..., p'$ (see Fig. 6.13a), the head node and the tail node will be removed and replaced by merge place $p''$. One

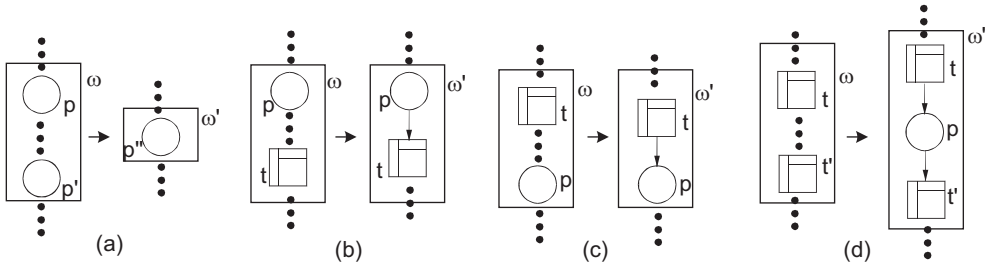


Figure 6.13. Possible structures of super nodes with divining transitions only

might think that in this case, an arc should be added from each transition in ${}^\bullet p$ to $p''$ and from $p''$ to each transition in $p'^\bullet$; the existing arc from each transition in ${}^\bullet p$ to $p$ and from $p'$ to each transition in $p'^\bullet$ should be removed. But this should be done only for the P/T net $N$, not for its structure graph $S_N$. This is because pre arcs of $p''$ are the pre arcs of head node $p$, and the post arcs of $p''$ are the post arcs of tail node $p'$, so these arcs can simply be left in place. Only the internal arcs of $\omega$ need to be removed.

In the other three cases (see Fig. 6.13b,c,d) the head and tail nodes of $\omega$ are the same as those of $\omega'$ (after contraction of the middle node transition). The pre arcs of the head node and the post arcs of the tail node would not be removed in the first place. This is another advantage of the structure graph which encapsulates nodes inside a super node.

In the case where there are non-divining transitions in the super node, the super node is traversed piecewise, i.e. from head node to the first relevant transition, between successive relevant transitions, and from the last relevant transition to the tail node. This procedure is implemented by the following algorithm.

**Algorithm** *TrivialContraction*
**Input:** A structure graph $S_N$
**Output:** A structure graph $S'_N$
1. $S'_N = S_N$;
2. **while** $\exists \omega \in \Omega'$ with divining transitions and $|\omega| > 2$ **do**
3. $n$ is the head node of $\omega$;
4. **while** $\exists n' \in n^\bullet$ **do**
5. $M_0(p_{mid}) = 0$;
6. **while** $n'$ is not a relevant transition or a tail node of $\omega$ **do**
7. **if** $n'$ is a place **then** $M_0(p_{mid}) = M_0(p_{mid}) + M_0(n')$;
8. $n' = n'^\bullet$;
9. ($*$ end of while $n'$ is not a relevant transition or a tail node of $\omega$ $*$)
10. **if** $n$ is a place and $n'$ is a place **then**
11. add a place node $n''$ into $\omega'$;
12. $M_0(n'') = M_0(n) + M_0(p_{mid}) + M_0(n')$;

13. **elseif** $n$ is a place and $n'$ is a transition **then**
14. $M_0(n) = M_0(n) + M_0(p_{mid})$;
15. add the node $n, n'$ into $\omega'$;
16. **elseif** $n$ is a transition and $n'$ is a place **then**
17. $M_0(n') = M_0(p_{mid}) + M_0(n')$;
18. add the node $n, n'$ into $\omega'$;
19. **else** $(*$ $n$ is a transition and $n'$ is a transition $*)$
20. add the node $n, p_{mid}, n'$ into $\omega'$;
21. $(*$ end of if $n$ is a place and $n'$ is a place $*)$
22. $n = n'$;
23. $(*$ end of while $\exists n' \in n^\bullet$ $*)$
24. $\omega = \omega'$;
25. $(*$ end of while $\exists \omega$ with divining transition and $|\omega| > 2$ $*)$

Applied to the net in Fig. 3.4a and Fig. 6.11), this algorithm describes the transformation from Fig. 3.4b to Fig. 3.4c, where both nets have the same structure graph (see Fig. 6.12); only the sequences within the super nodes are affected.

The algorithm *TrivialContraction* begins with a super node $\omega$ containing one or more middle divining transition nodes; $n$ is the head node of $\omega$. While there is a node to the post node of $n$, proceed as follows (line 4 - line 23): Traverse the sequence of nodes until a node $n'$ which is a relevant transition node or the tail node is reached (line 6 - line 9). While traversing, count the number of tokens in the place nodes (line 7). The possible $n$ and $n'$ are the following.

- (line 10 - line 12) Both nodes are places. In this case, a place node $n''$ is added to $\omega'$ and marked with a number of tokens equal to the sum of the tokens found from $n$ to $n'$.

- (line 13 - line 15) In case $n$ is a place and $n'$ is a transition, $n$ and $n'$ are inserted into the super node $\omega'$; the number of tokens in $n$ is the sum of tokens in the traversed place nodes including $n$.

- (line 16 - line 18) In case $n$ is a transition and $n'$ is a place, $n$ and $n'$ are inserted into the new super node $\omega'$; the number of tokens in $n'$ is the sum of tokens in the traversed place nodes including $n'$.

- (line 19 - line 21) Both nodes are transitions. In this case, $n, n'$ and a new place node $p_{mid}$ is inserted into the new super node $\omega'$ (line 20); the number of tokens in $p_{mid}$ is the sum of tokens in the traversed place nodes.

After processing a part of the super node $\omega$, $n$ is replaced by $n'$ and the above is done until the tail node is reached. When the tail node is reached, then $\omega$ is replaced by $\omega'$ and another super node which has a middle divining transition node is taken until no super node with a middle divining transition node is left.

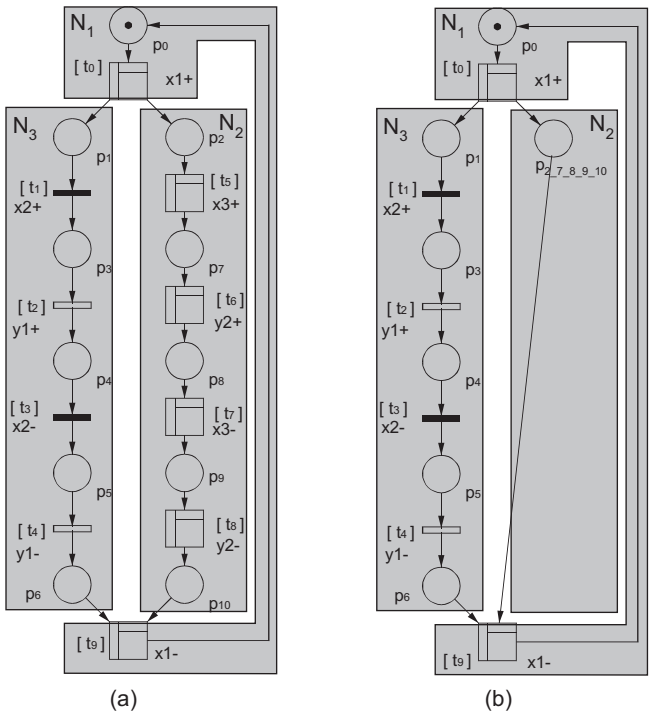


Figure 6.14. (a) Initial net for the $y1$-component, (b) after contracting middle node divining transitions

As an example, the net in Fig. 6.10a is taken. The initial net for the $y1$-component is shown in Fig. 6.14a. For $N_3$, nothing has to be done because it contains no divining transition. For $N_2$, place $p_2$ is the head node, place $p_{10}$ is the tail node, and all the middle transitions are divining transitions. Hence, $p_{2\_7\_8\_9\_10}$ replaces all the nodes in $N_2$ and has no initial marking because the sum of tokens in the places $p_2, p_7, p_8, p_9, p_{10}$ is zero. For $N_1$, there is no middle divining transition. Hence, there is no change made to the node $N_1$. The net after contracting middle node transitions is shown in Fig. 6.14b.

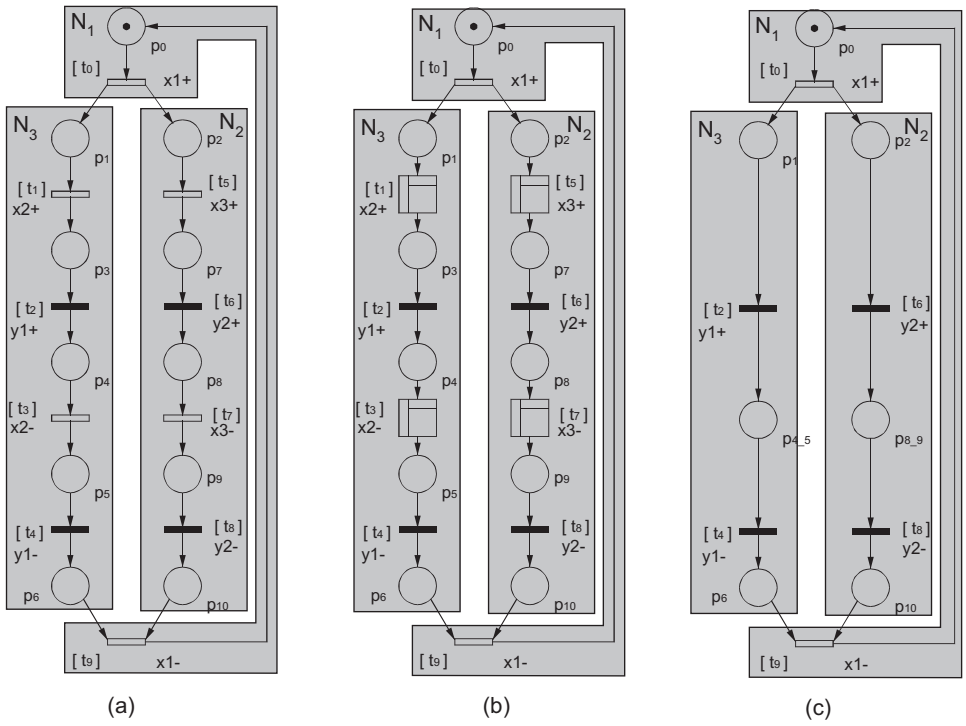


Figure 6.15. (a) A net $N$, (b) initial net for $x1$-component, (c) net after contracting middle node divining transitions

Another example is the net in Fig. 6.15a which is the environment version of the net in Fig. 6.10a. The initial net for the $x1$-component is shown in Fig. 6.15b. For $N_1$, nothing has to be done because it contains no divining transitions. But $N_2$ and $N_3$ are interesting. Because the structure of $N_3$ is the same as $N_2$, only $N_2$ will be discussed. $N_2$ contains divining as well as relevant middle-node transitions. Hence, the nodes is processed piecewise as follows: $(p_2, t_5, p_7, t_6)$, then $(t_6, p_8, t_7, p_9, t_8)$ and lastly $(t_8, p_{10})$. The first part contains the head node $p_2$, the end node $t_6$ and the middle divining transition $t_5$. Therefore, $p_2$ and $t_6$ are added to $N_2'$ with $M_0(p_2) := M_0(p_2) + M_0(p_7) = 0$. The second part begins with $t_6$, ends with $t_8$ and containsthe middle divining transition $t_7$. Therefore, $p_{8\_9}, t_8$ is added to $N_2'$ with $M_0(p_{8\_9}) = M_0(p_8) + M_0(p_9) = 0$. The third part has the first node

$t_8$, the end node $p_{10}$ and no middle nodes. Therefore, no change is made, $t_8$ and $p_{10}$ are added to $N_2'$ with $M_0(p_{10}) := M_0(p_{10}) = 0$. The net $N$ after contracting middle node transitions in $N_2$ and $N_3$ is the final $x1$-component and is shown in Fig. 6.15c.

# Chapter 7

# STG Decomposition in Asynchronous Circuit Design

## 7.1 Asynchronous Circuits

There are two types of digital circuits: combinational and sequential circuits. In combinational circuits, outputs depend only on inputs. In sequential circuits, the outputs also depend on the internal state.

In designing a sequential digital circuit, one can choose to design

- a *synchronous circuit*: a circuit with a global synchronization signal, e.g. clock; or
- an *asynchronous circuit*: a circuit with
  - dedicated synchronization signals generated by communicating component circuits in accordance with a protocol (e.g. request/acknowledge, completion signals) or
  - no dedicated synchronization signals at all within the circuit or within one of its component circuits.

The synchronous circuit clock signal usually pulses at regular intervals and is distributed throughout the circuit, thus serving to keep all of the components synchronized with the clock in each step of processing. All processing in each step must complete within the clock period; otherwise the circuit will fail to function properly.

Nowadays, the use of a global clock presents difficult problems [FN01] [DN95] [CKK+02] [SF01], namely:

1. **Clock skew**: Due to different delays along the clock signal propagation paths, the clock signal will not reach all components simultaneously. If the difference is too great, a "late" component may already be "seeing" the

internal state change of an "early" component when the clock pulse reaches it, and as a consequence the circuit will enter a wrong next state. In today's chip technology where interconnection delay dominates propagation delay and decreasing clock period, it needs a large effort to assure that the clock reaches every component with a tolerable skew.

2. **Power consumption**: In synchronous CMOS circuits, the clock signal causes many CMOS components to consume power even when they are not performing any useful work. In mobile equipment like hand phone, a low power consumption is needed. This is difficult to achieve with a synchronous circuit.

3. **Electromagnetic interference and noise**: In a synchronous circuit, the clock signal triggers periodic bursts of activity. These bursts induce electromagnetic interference and noise. This can be particularly problematic for nearby sensitive analog components. Therefore, a synchronous system is not applicable to safety equipment which does not allow any electromagnetic interference to another system; or to equipment which does not allow any electromagnetic emission for security reasons.

4. **Worst-case timing**: In order to use a single clock signal, all components in the circuit must operate within the same period of time. As a result, the clock period must be equal to the processing time needed by the slowest component. This reduces circuit efficiency because all the other component must always wait for the slowest component.

Traditionally, sequential digital circuit designers partition the specification into two main parts:

1. **the data path** which includes units that perform data transformation and data storage.

2. **the controller** which synchronizes the operations performed by data path units with a communication protocol.

In this dissertation only asynchronous controller design is considered.

An asynchronous circuit design is generally modeled using the standard distinction between *functionality* and *timing*. The functionality of a controller is modelled by the communication protocol. The timing is modelled by a delay model and an operating mode.

The delay model assumed for synthesis is important for an asynchronous circuit. It involves the following aspects:

- Model delay is associated with physical elements.

  - *Gate* delay model: ideal (delay-free) gate followed by a delay on its output, delay-free wires; i.e. real wire delays included in the delay of the preceding gate.
  - *Wire* delay model: ideal gates; real gate delays *included* in the model wire delays of the gate's fanout.

- Delay value: The delay may be assumed to be either *unbounded*, i.e. an arbitrary finite delay, or *bounded*, i.e. lying within given min/max bounds. The unbounded delay model is pessimistic, because in practice, the delay value is known and could be used for circuit minimization.

- Physical delay behaviour (see Fig. 7.1): A *pure* delay model does not alter the input signal, it is only delayed by $\Delta$. An *inertial* delay model filters out pulses that have a width less than a threshold value $\delta$, wider pulses are delayed by $\Delta$ (=$\delta$ Fig. 7.1). Inertial delay is more realistic for modelling a gate, because a gate cannot absorb and react directly for a small input change.

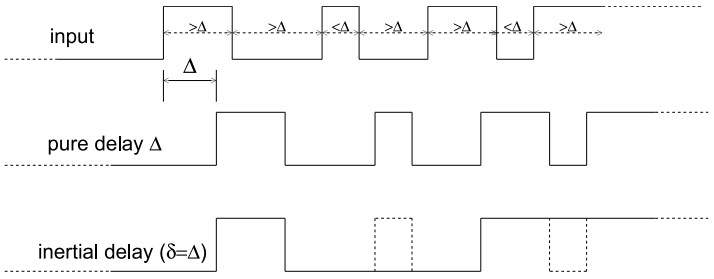


Figure 7.1. Example of a gate with delay $\Delta$, with $\Delta$ modelled as pure delay and as inertial delay with threshold value $\delta = \Delta$.

An *operating mode* describe interaction between an asynchronous controller and its environment. The following are types of operating mode:

- *fundamental mode (FM)*: in FM, communication protocol is done in two time intervals: communication and computation that are strictly separated and alternated. First *communication* occurs, when the environment sends new inputs within a time interval $\delta_1$ to the circuit. Then *computation* occurs, when the circuit reacts to the new inputs by providing new outputs. In this time interval $\delta_2$, the environment must not send any new input until the circuit has stabilized after a computation which is signaled by the produced outputs (see Fig. 7.2a). If there is only a single input change (SIC) during communication $\delta_1$ then it is called *SIC-FM*; otherwise if there are multiple input changes (MIC) during communication $\delta_1$, then it is called *MIC-FM*. FM forces to use the bounded delay model, because the time interval should be computed to fulfill the time constraint.

- *burst mode (BM)*: BM is based on the MIC-FM. The difference is, in BM the outputs can be produced while the circuit is still in computation phase, as long as it is known that the environment will be slow enough not to change the inputs while computation is still in progress (see Fig. 7.2b). This is possible, because the environment also needs time to compute its output (input for the circuit).

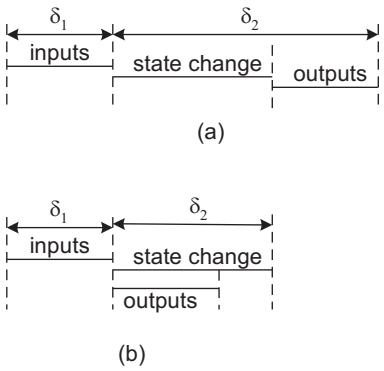


Figure 7.2. (a) FM and BM (b) BM

- *input/output (I/O) mode*: in I/O mode, the environment responds to circuit outputs without timing constraints; i.e. there is no communication or computation time interval. The only constraint is the communication protocol between the circuit and its environment.

## 7.2 Speed Independent Circuits

Speed independent (SI) circuits are based on the Muller circuits [MB59] which are designed under the unbounded gate delay model and I/O mode. Hence, SI circuits function correctly regardless of gate delay. Wire delay is assumed to be negligible. This assumption is optimistic, because in today's chip technology, wire delay is larger than gate delay. This assumption also requires the circuit designer to ensure the wire delay between implemented gates either to be negligible or cause no externally observable spurious behaviour.

To synthesize a Muller circuit, the high-level specification (e.g. STG) should be translated into a reachability graph first. Next, the reachability graph is examined to determine if a circuit can be generated using only the specified input and output signals.

Two markings in a reachability graph have *unique state coding (USC)* if they have different values of input and output signals. An STG has USC if all marking pairs in the reachability graph have USC. Two markings in a reachability graph have *complete state coding (CSC)* if they either have USC or do not have USC

but do have the same output signals excited in both markings. An STG has CSC if all marking pairs in the reachability graph have CSC.

When an STG does not have CSC, the implied value for some output signals cannot be determined by simply considering the values of the input and output signals. This ambiguity leads to a state of confusion for the circuit. Therefore this CSC problem should be solved by adding an internal signal. The inserted internal signal should not change the interface of the circuit; i.e. it should not give concession to any input transition.

An *irreducible CSC* problem occurs if there is no way to insert an internal signal to obtain CSC, without changing the interface of the circuit. This happens if there is a complementary set of input signal changes – e.g. $a+, a-$ or $a-, a+$ – which have not been acknowledged by the circuit by an internal signal change, i.e. there is no output change in between. In the case of an irreducible CSC problem, either the specification must be modified or the environment must be constrained.

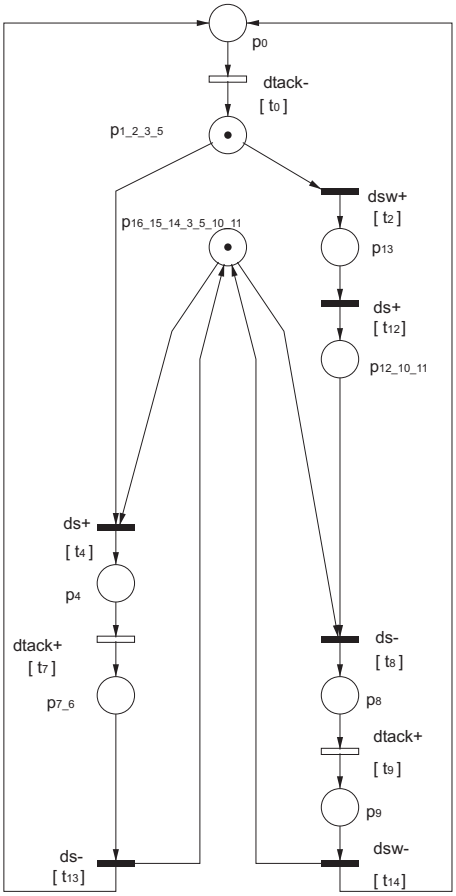


Figure 7.3. dtack component for vmecon example in Fig. 2.8

For example, the STG in Fig. 7.3 has no CSC and USC, because after firing

transitions $M_0[dsw+\rangle M_1[ds+\rangle M_2[ds-\rangle M_3$, $M_1$ and $M_3$ are found to have the same value of input and output signals – both markings have $dsw = 1$ and $ds = dtack = 0$; and at $M_1$ there is no output is excited, but at $M_3$, $dtack$ is excited. The STG also has irreducible CSC at marking $M_1$ and $M_3$, because there is no way to insert an internal state signal in between without changing the interface.

Note that the speed independent circuit design from [CKK+02] uses more restrictions than the one from Muller; i.e. that the output should be persistent in all possible behaviours of the protocol under a given environment. An STG is said to be *output persistent* if for any pair of output signals $x*$ and $y*$ ($'*'$ could be either $'+'$ or $'-'$), $\nexists M \in R_{M_0}, M[x*\rangle$ and $M[y*\rangle$ such that $M[x*\rangle M'$ and $y*$ is not enabled in $M'$. An SI circuit synthesized from an STG which is output persistent is guaranteed to have no hazard under the pure delay model.

## 7.3 3D Circuits

The extended burst mode (XBM) specification [YD99] is an extension of the burst mode (BM ) specification [Now93]. From an XBM specification, the synthesized *3D circuit* is based on the Huffman circuit [Huf64] which is designed under the bounded wire delay model and fundamental mode. However, the 3D circuit operates in burst mode. Also, the XBM circuit is guaranteed to work properly under the pure and the inertial delay model.

An XBM specification is a labelled directed graph which is defined as follows:

**7.3.1.** Definition. An *extended burst-mode (XBM)* specification is a tuple $(S, F, I, O, C, s_{in}, s_{out}, f_{cond}, s_0, In, Out, Cond)$ where
$S$ is the set of states,
$F \subseteq S \times S$ is the set of state transitions,
$Condi$ is the set and $Cond$ the tuple of conditional input signals,
$Inp$ is the set and $In$ the tuple of triggering input signals ($Inp \cap Condi = \emptyset$),
$Inpall = Inp \cup Condi$,
$Outp$ is the set and $Out$ the tuple of triggering output signals ($Inpall \cap Outp = \emptyset$),
$I \subseteq \{0, 1, *\}^{|In|}$ is the set of $In$ tuples,
$O \subseteq \{0, 1, *\}^{|Out|}$ is the set of $Out$ tuples,
$C \subseteq \{0, 1, *\}^{|Cond|}$ is the set of $Cond$ tuples,
$s_{in} : S \to I$ associates to each state the unique $In$ tuple under which it is reached,
$s_{out} : S \to O$ a unique $Out$ tuple to each state,
$f_{cond} : F \to C$ assigns to each state transition the conditions (the $Cond$ tuple) under which it occurs,
$s_0 \in S$ is the initial state,
an $'*'$ is a don't care which can have a value of either $'0'$ or $'1'$.

According to the above definition, every state is assigned the unique input under which it is entered and the output value it generates. This requires every state in the XBM specification to be entered at a single *unique entry point*. For example, the set of valid entry points to state 1 from state 0 in Fig. 7.4a is $\{01011, 01111\}$, but from state 3 to state 1 is $\{01011\}$. Thus the unique entry point condition is not met in this specification. A specification satisfying the unique entry point condition is shown in Fig.7.4b.

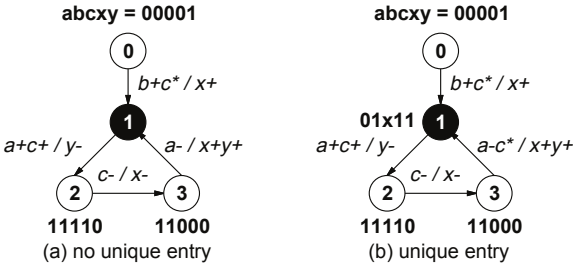


Figure 7.4. Unique entry point example [Yun94]

For convenience, normally the arc is labelled with an *input burst* function $l_{in} : F \to \wp(In \times \{+, -, *\})$ and an *output burst* function $l_{out} : F \to \wp(Out \times \{+, -\})$, where $\wp(P)$ is the set of all subsets of $P$. They are derived from $s_{in}$ and $s_{out}$ as follows: for any arc $(u, v) \in F$, an $i$-th input $in_i \in In$ is in the $l_{in}((u, v))$ if $s_{in}(u) \neq s_{in}(v)$ in the $i$-th position or $s_{in}(v) = *$ in the $i$-th position. A $'+'$ is concatenated to $in_i$ if the $i$-th input value of $s_{in}(u) = 0$ and $s_{in}(v) = 1$ or the $i$-th input value of $s_{in}(u) = *$ and $s_{in}(v) = 1$, else a $'-'$ is concatenated to $in_i$ if the $i$-th input value of $s_{in}(u) = *$ and $s_{in}(v) = 0$ or the $i$-th input value of $s_{in}(u) = 1$ and $s_{in}(v) = 0$, otherwise a $'*'$ is concatenated to $in_i$ (*directed don't care transition of* $in_i$*). Directed don't care transitions allow one to specify that an input change may or may not happen in a given input burst. The idea is that some inputs in a burst may be allowed to change once* monotonically along a sequence of bursts rather than having to change in a particular burst. The same (without don't care) is done for the output. Also for any arc $(u, v) \in F$, an $i$-th input $cond_i \in Cond$ is in the $l_{in}((u, v))$ if $f_{cond}((u, v)) \neq *$ in the $i$-th position. A $'+'$ is concatenated to $cond_i$ if the $i$-th input value of $f_{in}((u, v)) = 1$, otherwise a $'-'$ is concatenated to $cond_i$. Conditional signal transition is placed in the bracket to differ from other transition. If $\exists l_{in}((u, v)) = \emptyset$, then the specification is not correct; because a state transition can only happen if there is an input value change (input transition). One can see in Fig. 7.5a the XBM specification as per definition 7.3.1 and its more convenient representation in Fig. 7.5b.

An edge transition which is not immediately preceded by a directed don't care transition for the same signal is called *compulsory transition*. An input burst should have at least one compulsory transition; i.e. $l_{comp} : F \to \wp(In \times \{+, -\})$, an $i$-th input $in_i \in In, in_i \in l_{in}((u, v))$ is in $l_{comp}((u, v))$ iff $s_{in}(u)$ and $s_{in}(v)$ in

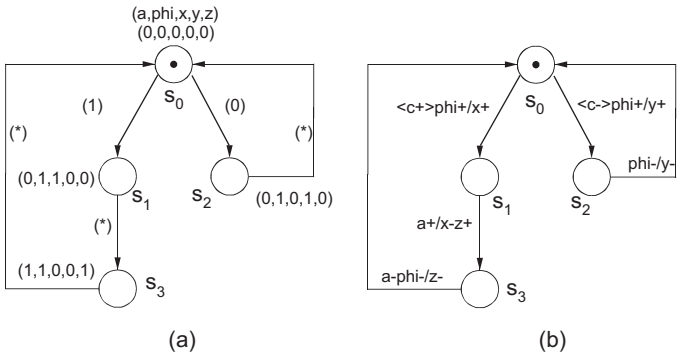


Figure 7.5. (a) an XBM specification, (b) a more convenient XBM specification

the $i$-th position are not equal to $'*'$. If $\exists l_{comp}((u,v)) = \emptyset$ then the specification is not correct; because there should be at least one compulsory transition in an input burst. Without $dackn+$ in transition from state 3 to 4, the specification in Fig. 7.6 would not be correct because $fain-$ in transition from state 3 to 4 is not a compulsory transition: it is immediately preceded by a directed don't care $fain*$ in transition from state 2 to 3.

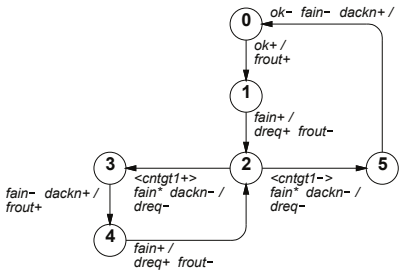


Figure 7.6. Compulsory transition example [YD99]

Input bursts of every pair of state transitions emanating from the same state must satisfy the *distinguishability constraint* such that for every pair of input bursts $i$ and $j$ emanating from the same state, either the conditions are mutually exclusive or the set of compulsory transitions in $i$ is not a subset of the set of all possible input transitions in $j$.

In Fig. 7.7a, the input burst from state 0 to 1 and the input burst from state 0 to 2 fulfill the distinguishability constraint, because their conditions $< c+ >$ and $< c- >$ are mutually exclusive. In Fig. 7.7b, input $i = \{b+\}$ causes the transition from state 0 to state 2. Input burst $j$ (leading from state 0 to 1) contains the compulsory transition $a+$ and may also contain the occurrence $b+$ of don't care transition $b* : j \in \{\{a+\},\{a+,b+\}\}$. $i = \{b+\}$ is a subset of $\{a+,b+\}$. The indistinguishability constraint is violated. If $a+$ occurs after $b+$, the XBM machine will end up in state 2 instead of state 1. In Fig. 7.7c, $b+$ is the

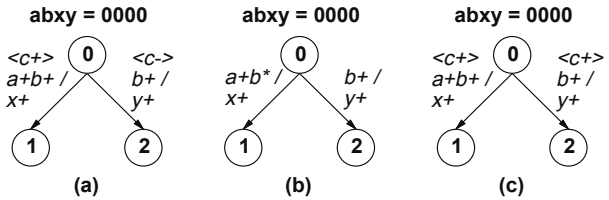


Figure 7.7. Distinguishability constraint example [YD99]

only compulsory transition of input burst $k = \{b+\}$ from state 0 to 2. $\{a+, b+\}$ is a possible input change of input burst $l$ from 0 to 1. $k = \{b+\}$ is a subset of $l = \{a+, b+\}$. Also, the conditions of input burst $k$ and $l$ are the same. Hence input bursts $k$ and $l$ do not fulfill the distinguishability constraint.

A 3D circuit works as follows: in a given state, when all the specified conditional signals have stabilized and all the specified edge signals in the input burst have appeared, the circuit asserts the specified output changes, moves to a new state, and then is ready for the next input burst. Specified edges in the input burst may appear in arbitrary temporal order. Each signal transition which is specified as a directed don't care transition may change its value monotonically at any time, even while outputs are changing. Output changes may be generated in any order.

The conditional signals must stabilize to correct levels before any compulsory edge in the input burst appears and must hold their values until after all of the input edges appear. The minimum delay from the conditional stabilizing to the first compulsory edge is called the *setup time*. Similarly, the minimum delay from the last input edge to the conditional change is called the *hold time*. Actual values of setup and hold times of conditional signals with respect to the first compulsory edge and the last input edge depend on the implementation. The period starting at the specified setup time before the first compulsary edge and ending at the specified hold time after the last input edge is called the *sampling period*. Conditional signal values need not be stable outside of the specified sampling periods.

### 7.3.1 From STG specification to XBM specification

There are methods [Wol97] – based on an elementary method [Bei00] – and [LSV93], which try to use an STG as specification due to its expressiveness for circuits with bounded wire delays. This is because STGs can specify a wider range of problems than can be solved by an XBM circuit and every XBM specification has an STG specification. In this chapter, another way is described to obtain circuits with bounded wire delay from an STG specification; i.e. by translating an STG specification into an XBM specification.

The idea to translate an STG specification into an XBM specification was first

suggested by [BEW99]. Not like [BEW99] where the translation was performed after deriving the state machine from the STG specification, the algorithm presented in this chapter translates directly from the STG. An XBM specification, though it is a state machine, allows some concurrency, namely: input and output burst, don't care transitions and conditional signals. These concurrencies can be translated directly, without explicitly deriving all of the possible states, and therefore the method is more efficient than [BEW99].

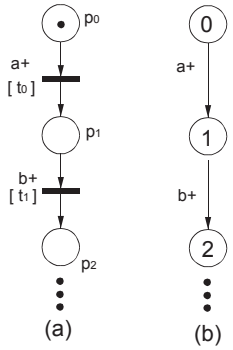


Figure 7.8. Sequence of input transitions: (a) STG, (b) its XBM translation

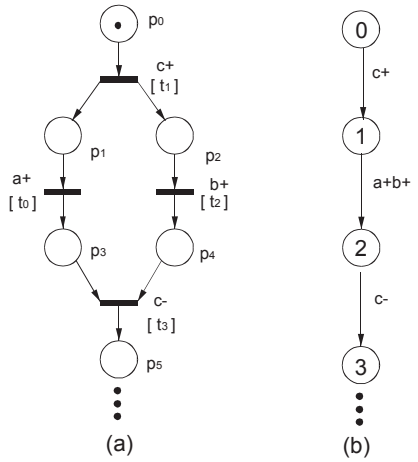


Figure 7.9. Concurrent input transitions: (a) STG, (b) its XBM translation

Let's see first, what is allowed in an STG that can be translated to XBM:

- Sequence of input transitions – see Fig. 7.8.
- Concurrent input(output) transitions – see Fig. 7.9(Fig. 7.10). Note: a sequence of transitions that are concurrent with each other is not allowed.

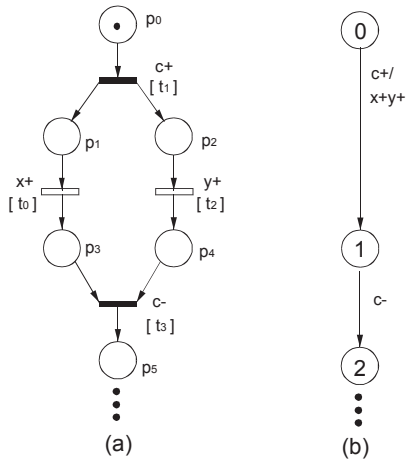


Figure 7.10. Concurrent output transitions: (a) STG, (b) its XBM translation

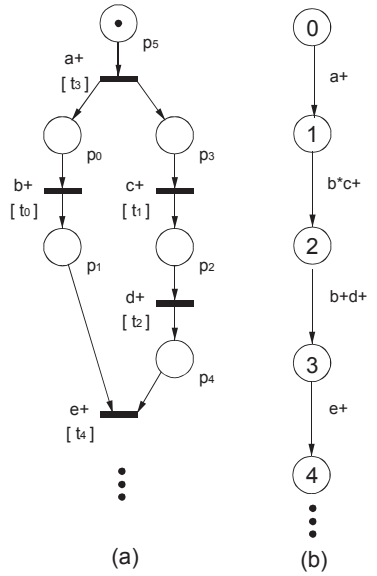


Figure 7.11. Input only don't care transition $t_0$: (a) STG, (b) its XBM translation

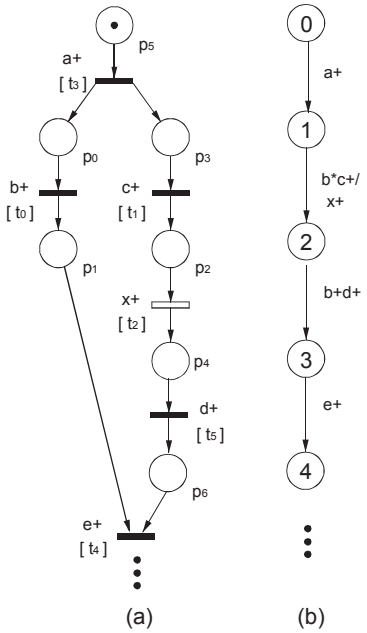


Figure 7.12. Don't care transition $t_0$ with output: (a) STG, (b) its XBM translation

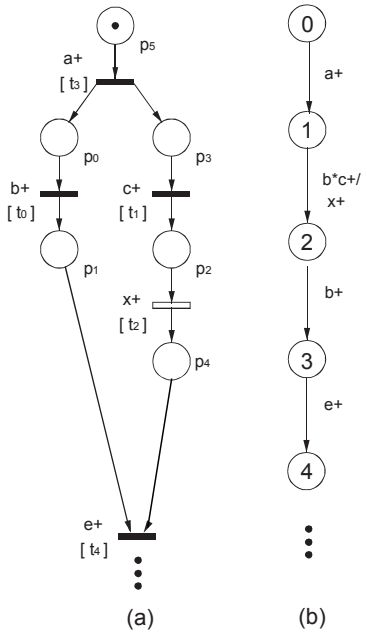


Figure 7.13. Don't care transition $t_0$ with output in the tail (a) STG, (b) its XBM translation without compulsory transition

- Don't care concurrent transitions with a transition sequence – see Fig. 7.11 for input only don't care transition;see Fig. 7.12 for don't care with output. Note: the output transition must be in the middle of the transition sequence. If the output transition is in the head of the transition sequence, then there will be an input-output conflict in the XBM translation so that the specification cannot be synthesized. If the output transition is in the tail of transition sequence, then the resulting XBM transition will have no compulsory transition. For example, after translating the STG in Fig. 7.13a, the resulting XBM in Fig. 7.13b has the transition $2 \rightarrow 3$ without compulsory transition.

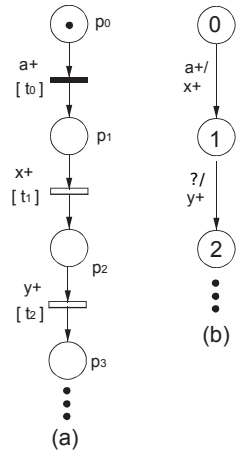


Figure 7.14. Sequence of output transitions: (a) STG, (b) its XBM translation without compulsory transition

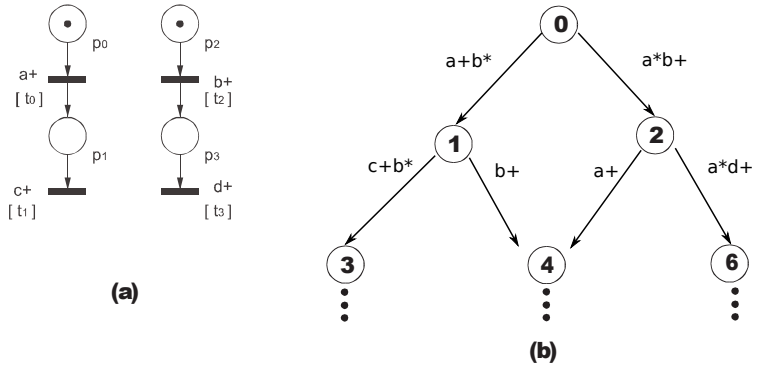


Figure 7.15. Sequence of input transitions that are concurrent with each other: (a) STG, (b) part of its XBM translation without compulsory transition

The following are not allowed in STGs that are to be translated to XBM

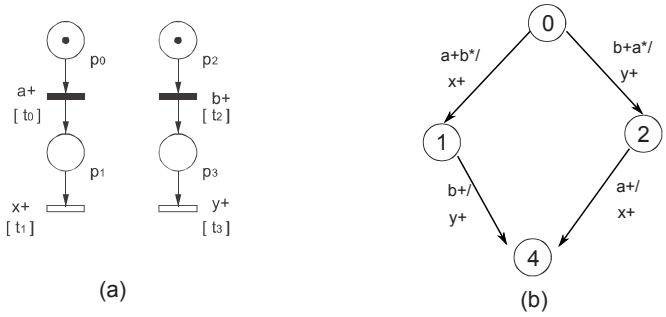


Figure 7.16. Sequence of input and output transitions that are concurrent with each other: (a) STG, (b) its XBM translation without compulsory transition

- Input-output conflict (concurrency) because the resulting XBM after translation is not synthesizable. Note: dynamic input-output conflicts (concurrency) are found first by "playing the token game". Don't care transitions concurrent with a transition sequence are an exception of input-output concurrency that is allowed.

- Output-output conflicts: The XBM resulting by translation has transitions without compulsory transitions.

- Sequences of output transitions (SOT): The resulting XBM has transitions without compulsory transitions. For example, after translating the STG in Fig. 7.14a, the resulting XBM in Fig. 7.14b has transition $1 \rightarrow 2$ without any compulsory transition.

- Sequences of transitions that are concurrent with each other. Because the resulting XBM after translation either has transitions without compulsory transitions or has input-output concurrency. For example, after translating the STG in Fig. 7.15a, the resulting XBM in Fig. 7.15b has transitions $1 \rightarrow 4$ and $2 \rightarrow 6$ without compulsory transitions. For another example with output transitions, see the STG in Fig. 7.16a. The resulting XBM in Fig. 7.16b has transitions $1 \rightarrow 4$ and $2 \rightarrow 4$ without compulsory transitions.

The algorithm *STG2XBM* needs a bounded STG as input. This is because only a bounded net has a finite set of reachable markings, and only an STG with a finite set of reachable markings can be synthesized into a circuit. The STG is also required to be consistent because only consistent STG have unique entry points [1]. Since an XBM specification does not have an equivalent for a $\lambda$ input transition, $\lambda$ input transitions are used for synchronization in STGs; $\lambda$ output transitions are also allowed because they represent state transitions without output in the XBM specification.

[1]Consistency of the STG is necessary but not sufficient, the resulting XBM still needs to be checked whether it has unique entry points.

A conditional signal only needs to be stable while its value is being sampled by the input burst, otherwise it can change its value by alternating $'+'$ and $'-'$ transitions. Therefore, this behaviour can be modelled in the STG as a level SCSM (see section 5.2.2). For example, the *async∗* STG in Fig. 5.1 has a level SCSM for conditional signal $c$. To increase algorithm efficiency, level SCSMs are removed from the STG and the conditions are placed in the sample transitions. This way, the number of reachable markings is reduced by half.

Firing an output transition at an initial marking is not possible, because the input burst would be empty, which is not allowed in XBM specifications [2]. This problem is treated in line 2 to line 3 of the algorithm *STG2XBM*.

**Algorithm** *STG2XBM*
**Input:** an ordinary STG $N$ which is bounded and consistent
**Output:** an extended burst mode machine
1. Replace all level SCSM with conditional signal;
2. **if** there is an output transition which is enabled by $M_0$ **then**
3. report and exit algorithm;
4. map the marking $M_0$ into XBM initial state $S$;
5. put $M_0$ into stack;
6. **while** there is still a marking $M$ in the stack **do**
7. find $S$ from $M$ in the map;
8. **if** there is a choice of input transitions which are enabled by $M$ **then**
9. for each choice *FireAndTranslate2XBM*;
10. **else**
11. *FireAndTranslate2XBM*;
12. (∗ end of if there is a choice of input transition which are enabled by $M$ ∗)
13. (∗ end of while there is still a marking $M$ in the stack ∗)
14. merge XBM state with a $\lambda$ input transition; (∗ see Fig. 7.17 ∗)

**Algorithm** *FireAndTranslate2XBM*
**Input:** an ordinary STG $N$ which is bounded and consistent
1. **if** there are input and output transitions which are enabled by $M$ **then**
2. report and exit algorithm; (∗ input-output concurrency ∗)
3. $D$ = all don't care input transition enabled in $M$;
4. $I$ = all the input transition enabled in $M$ - $D$;
5. fire all the input transitions in $I$, resulting in $M'$;
6. **if** $|I| > 1$ and there are transitions which are enabled by $M'$ **then**
7. report and exit algorithm; (∗ concurrent sequence of transitions ∗)
8. $O = \emptyset$
9. **if** there is an output transition which is enabled by $M'$ **then**

[2] This goes for every type of asynchronous circuit specification. The circuit remains in its inital stable state (including the initial output) until it is started by the first input change.

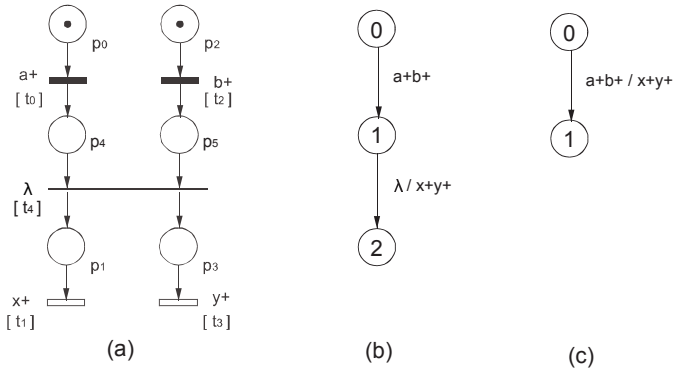


Figure 7.17. Input and output burst: (a) STG, (b) its XBM translation before and (c) after merger

10. **if** there is input transitions other than in $D$ which are enabled by $M'$ **then**
11. report and exit algorithm; ($*$ input-output concurrency $*$)
12. **if** output transitions in conflict **then** report and exit algorithm;
13. $O$ = all the output transition enabled in $M'$;
14. fire all the output transitions in $O$ resulting in $M''$;
15. $M_{next} = M''$;
16. **if** there is an output transition which is enabled by $M''$ **then**
17. report and exit algorithm; ($*$ SOT $*$)
18. **if** $|O| > 1$ and there are transitions enabled by $M''$ **then**
19. report and exit algorithm; ($*$ concurrent sequence of transitions $*$)
20. **else** ($*$ there is no output transition which is enabled by $M'$ $*$)
21. $M_{next} = M'$;
22. ($*$ end of if there is an output transition which is enabled by $M'$ $*$)
23. **if** there is no mapping of $M_{next}$ in the map **then**
24. map the marking $M_{next}$ into XBM state $S'$;
25. put $M_{next}$ into stack;
26. ($*$ end of if there is no mapping of $M_{next}$ in the map $*$)
27. label the transition $S \rightarrow S'$ with the label of transition in $D$, $I$ and $O$;

Because XBM specification allows only specific concurrency which is a subset of the concurrencies possible in STG specifications, not all STGs can be translated directly to XBM specifications. Sometimes, decomposition is needed to make it translatable as suggested by [BEW99]. [BEW99] suggest a state machine decomposition approach by trying to reduce the state machine vertically (and horizontally if needed). This vertical reduction assume that the irrelevant input signal as don't care. The vertical reduction is done with the method suggested by Graselli and Luccio [GF66]. The implementation of this state machine decomposition based on [BEW99] and [GF66] can be seen in [Kan02].

Also, STG decomposition can be used to make an STG specification translatable to an XBM specification. For example, the wechselpuffer STG in Fig. 3.4a cannot be translated directly into an XBM specification. Because after $Rin+$, $Rd+$, $Rm+$ are fired in sequence, $Ain+$ and $Rout+$ are concurrently enabled and give concession concurrently to $Rin-$ and $Aout+$. This is the case of not allowed output-output concurrency. After decomposition, the $Aout-$ component (see Fig. 5.15c) can be translated to the XBM specification.

But decomposition does not always solve the problem, because even after decomposition some components may not be XBM translatable. For example, the $z$-component of the async99* STG in Fig. 5.14b cannot be translated into a valid XBM specification due to input-output concurrency between $z+$ and $a-$ after firing $a+$.

## 7.4 DESI

DESI (**DE**composer of **SI**gnal Transition Graphs) is implemented based on the algorithm [VK07] [VW02] which starts with a given partition of the set of output variables: each $C_i$ is responsible for one block of the partition. The $C_i$s are then extracted from the STG by transition contraction and deletion of redundant places [VW02], as well as deletion of loop-only and duplicate $\lambda$-transitions [VK07], care being taken to keep only the relevant input signals, which may be global inputs or outputs of other components. Step by step examples of STG decomposition can be seen in AG-Beister website.

Deleting redundant places can increase efficiency of the algorithm by contracting a transition. But to find a redundant place as per definition 2.1.18 takes much more effort than the gain in efficiency. Hence, DESI removes only special cases of redundant places: loop-only and (extended) duplicate places.

- A *loop-only* place is a marked place $p$, such that $p$ and $t$ form a loop with arcs of weight 1 for all $t \in {}^\bullet p \cup p^\bullet$. The example STG in Fig. 7.18a will have a loop-only place $(p_1, p_2)$ after secure $t$-contraction.

- Place $p$ is an *(extended) duplicate* of place $q$ if $\forall t : W(t,p) = W(t,q), W(p,t) = W(q,t)$ and $M_N(p) \geq M_N(q)$. After secure $t$-contraction, the example STG in Fig. 7.18b will have a place $(p_1, p_2)$ in which $M_{\overline{N}}((p_1, p_2)) = 1$ and $M_{\overline{N}}(p_3) = 0$. Hence $(p_1, p_2)$ is a duplicate of $p_3$ and will be removed.

DESI is suggested to be applied in modular [3] design [KWVB03]. The modular design starts from an STG as specification. The STG is decomposed into components based on the output partition. Each component is synthesized separately,

[3] A Module is a self-contained component of a system, which has a well-defined interface to the other components

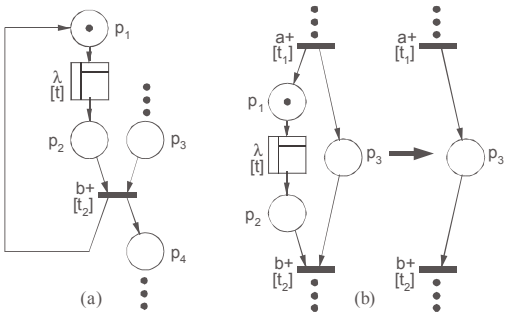


Figure 7.18. After contracting $t$, the STG has (a) a loop-only place, (b) a duplicate place

then the resulting modules are interconnected to form a modular circuit. The overall design flow is shown in Fig. 7.19.

DESI is designed as part of CASCADE, which can forward results to other synthesis tools such as *petrify* [CKK+96] for SI circuit and $3D$ [YDN92] for 3D circuit. Based on the modular design flow, DESI together with other tools completes the tool chain for a modular design (see Fig. 7.20).

DESI is an academic tool which may be downloaded from AG-Beister website. Version 2 uses the algorithm from [VW02]. This version was presented in [KWVB03]. An option for a risky strategy is also included in version 3.

### 7.4.1 Experimental Results for SI Circuits

The examples in the benchmark table are taken from a collection of benchmark examples that circulate in the STG community. In the experiment, *petrify* was used to synthesize the components. A plausible output partition [4] is given as input for DESI. For each example in the table, we have the number $r$ of reachable states, the area $a$ resulting from synthesis and the computation time $t$(in second) needed for an Intel Xeon 2.2 GHz with 1 GB memory. The lower-case literals are used for the original specification, the upper-case literals for the sum of all components resulting from decomposition. $T_d$ is the time taken by DESI for decomposition, $T_p$ is the time taken by petrify for synthesizing all the components. In the arbiter example, the ME-element found by DESI could not be synthesized by petrify (neither could the original specification) due to the output persistency requirement for SI circuits.

From the table, one can see that DESI produces the best results if $r$ is large. We obtained fewer reachable states, less area, and less computation time. Even for small examples, we mostly need less computation time. DESI overhead is small in most cases compared to the computation time needed for synthesis. Less computation time for synthesis is achieved because of fewer reachable states and

[4] more about finding effective output partition [WVW11]

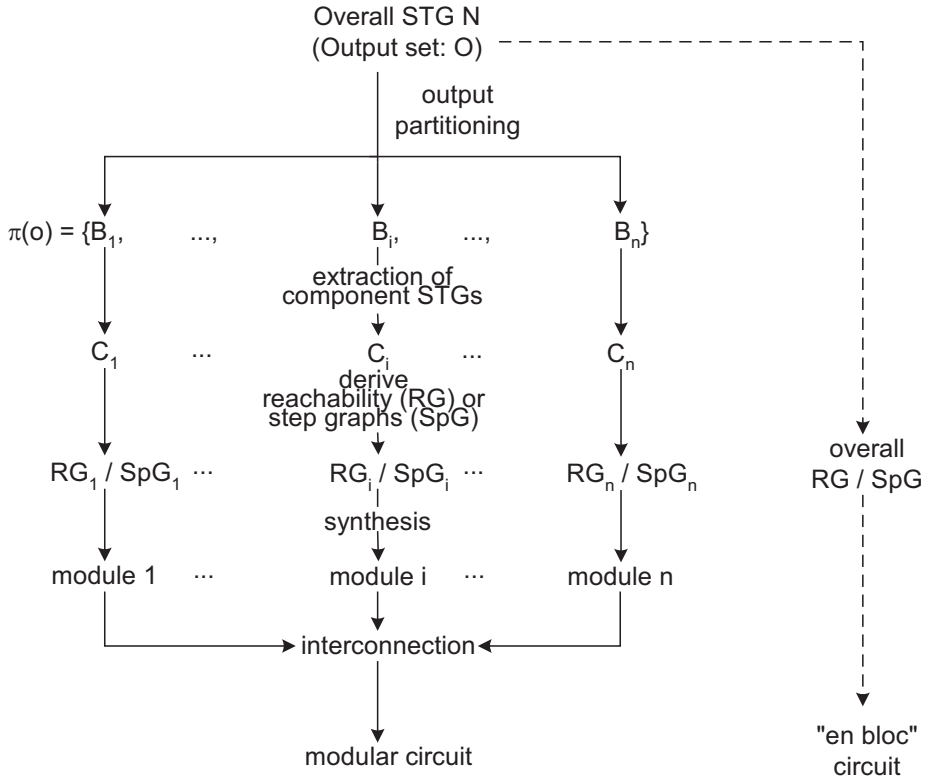


Figure 7.19. Modular design flow

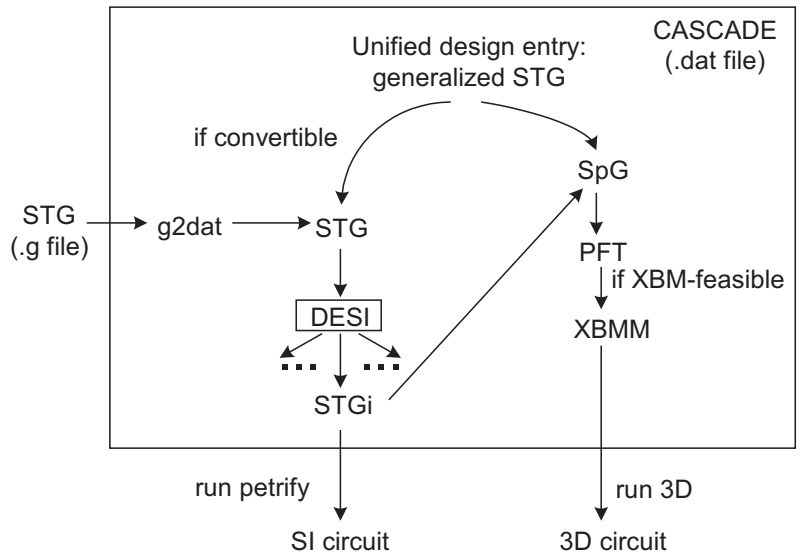


Figure 7.20. Modular design tool chain

| | name | original | | | all components together | | | | | ratio (%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | r | a | t | R | A | T | $T_d$ | $T_p$ | R/r | A/a | T/t | $T_d/T_p$ |
| 1 | mread | 8932 | 67 | 231.18 | 409 | 59* | 6.31 | 0.15 | 6.16 | 4.57 | 88.06 | 2.73 | 2.44 |
| 2 | stg-blunno' | 1241 | 54 | 401.35 | 248 | 36* | 2.95 | 1.27 | 1.68 | 19.98 | 66.67 | 0.74 | 75.6 |
| 3 | FIFO | 832 | 49 | 140.88 | 126 | 41* | 3.12 | 0.45 | 2.67 | 15.14 | 83.67 | 0.32 | 16.85 |
| 4 | locked2' | 168 | 23* | 6.91 | 82 | 29* | 2.36 | 0.39 | 1.97 | 48.81 | 126.09 | 34.15 | 19.8 |
| | (risky) | | | | 44 | -- | -- | 0.35 | -- | 26.19 | -- | -- | -- |
| 5 | pe-send-ifc' | 117 | 50 | 2.07 | 100 | 62* | 4.82 | 0.43 | 4.39 | 85.47 | 124 | 232.85 | 9.79 |
| 6 | mux2' | 101 | 68* | 255.42 | 93 | 109* | 50.26 | 0.47 | 49.75 | 92.08 | 160 | 19.68 | 0.94 |
| 7 | LL Arbiter' | 64 | -- | -- | 82 | ^ | -- | 0.1 | -- | 128.13 | -- | -- | -- |
| 8 | post-office' | 62 | 28 | 22.3 | 72 | 32* | 17.72 | 0.49 | 17.23 | 116.13 | 114.29 | 79.46 | 2.84 |
| | (risky) | | | | 67 | -- | -- | 0.43 | -- | 108.06 | -- | -- | -- |
| 9 | nak-pa | 58 | 18 | 0.19 | 54 | 18 | 0.38 | 0.26 | 0.12 | 93.1 | 100 | 200 | 216.67 |
| 10 | adfast | 44 | 17 | 1,34 | 38 | 15* | 0.91 | 0.15 | 0.76 | 86.36 | 88.24 | 67.91 | 19.74 |
| 11 | adc-yak | 44 | 15 | 1.29 | 39 | 16* | 0.64 | 0.14 | 0.50 | 88.64 | 106.67 | 49.61 | 28 |
| 12 | NEI Arbiter' | 42 | -- | -- | 33 | ^ | -- | 0.09 | -- | 78.57 | -- | -- | -- |
| 13 | tsend-csm' | 36 | 38 | 9.43 | 41 | 35* | 1.4 | 0.24 | 1.16 | 113.89 | 92.11 | 14.85 | 20.69 |
| 14 | vmecon' | 24 | 19 | 1.72 | 27 | 22* | 0.57 | 0.13 | 0.44 | 112.5 | 115.79 | 33.14 | 29.55 |

' could be handled only by improved algorithm; i.e. DESI version 3 or above
Example 2,4,6,8 and 13 have dummy transition
Example 2,5,7,12 and 14 have structural auto conflicts
Example 2 has an io-conflict
* there is module which is not speed independent (petrify synthesis with "slow environment")
^ ME-element plus synthesizable component(s)

Figure 7.21. Benchmark table. The times are given in seconds [VK07].

| Original | Components | | | | Remarks |
|---|---|---|---|---|---|
| STG-name/Type<br>origin<br>markings / area<br>CPU time (sec)<br>(petrify&write_sg) | ∑ markings / ∑ area<br>∑ CPU times (sec)<br>(desi+&petrify&write_sg / desi+) | output signals<br>input signals<br>marking / area<br>CPU time<br>(petrify&write_sg) | | | |
| stg_blunno#%! / NFC<br>Blunno<br>1241 / 54<br>401.35 | 248 / 36<br>2.95 / 1.27 | bo*<br>bi,co!!,op§,strt,ci&,cmdc0i!!<br>102 / 8<br>0.59 | co,cmdc0o,stop*<br>op,strt,ci,cmdc0i<br>44 / 20<br>0.5 | ao*<br>co!!,ai,op§,strt,ci&,cmdc0i!!<br>102 / 8<br>0.59 | |
| pe-send-ifc! / NFC<br>SIS Example<br>117 / 50<br>2.07 | 100 / 62<br>4.82 / 0.43 | y0_pesendifc,y1_pesendifc,tack<br>adbldout,rdiq,treq,ackpkt,reqsend<br>68 / 38<br>1.04 | | peack,adbld*<br>ackpkt,y0_pesendifc,y1_pesendifc<br>32 / 24<br>3.35 | |
| vmecon! / NFC<br>petrify Example<br>24 / 19<br>1.72 | 27 / 22<br>0.57 / 0.13 | lds,ds*<br>ldtack,dsr,dsw<br>19 / 16<br>0.37 | | dtack*<br>ds,dsw<br>8 / 6<br>0.07 | |
| NEI Arbiter! / NFC<br>Wollowski/AG DIGI<br>42 / --<br>0.03 | 33 / 12<br>0.24 / 0.09 | A1,A2<br>R1,R2<br>12 / --<br>0.05 | | G1,G2,R<br>G,A1,A2<br>21 / 12<br>0.1 | Component {A1,A2} is a ME Element |

* not SI (petrify synthesis with parameter "slow environment")
^ Component STG is the same with original STG; only exchange input and output signal
name§ : contracting transition with signal "name" is not allowed -> backtracking
name& : contracting transition with signal "name" is not secure contraction -> backtracking
name!! : structural auto-conflict after contracting transition with signal "name" -> backtracking
name!: original STG "name" has structural auto-conflict
name#: original STG "name" has dummy transition
name%: original STG "name" has structural input/output conflict

CPU: Intel Xeon 2.2 GHz
Memory: 1 Gbyte

Figure 7.22. Detail of some example described in table Fig. 7.21

fewer output signals which cause less time needed to find complete state coding (CSC). The decomposition equation results are small in most of the cases that can be implemented with only simple gate instead of complex gate. With the risky strategy, we obtained smaller numbers of $R$, but with the penalty that the components could not be synthesized by petrify if they involve dynamic auto conflicts. In the arbiter example, the ME-element found by DESI could not be synthesized by petrify (neither could the original specification) due to the output persistency requirement for SI circuits.

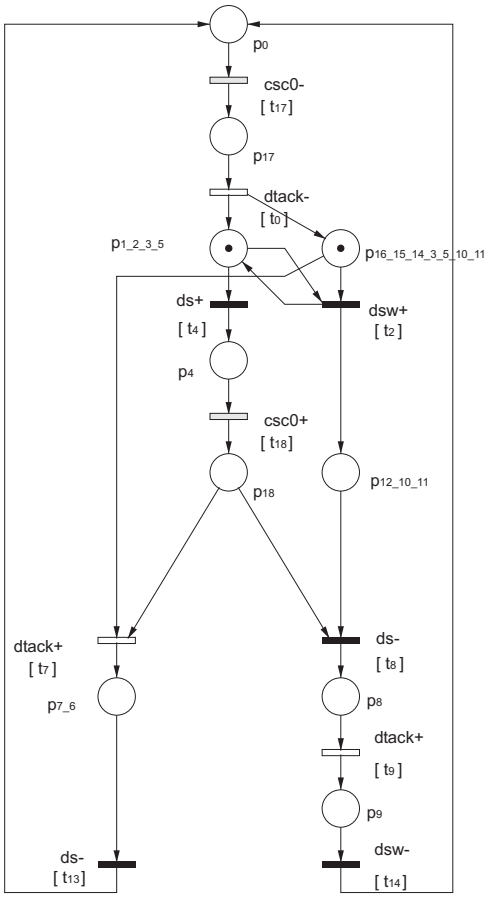


Figure 7.23. *dtack* component from Fig. 7.3 after CSC solving

Due to decomposition, each circuit should have its internal state signals. The insertion of these internal signals into the specification without changing the interface is sometimes impossible, because the specification after decomposition has fewer output signals. Therefore, sometimes it is needed to ensure that some internal event has happened before some input event. If this cannot be achieved, then it is needed to slow the environment.

An example is the VME controller vmecon (last line in table Fig. 7.21). It is decomposed into 2 components $\{lds, ds\}$ and $\{dtack\}$ (see table in Fig. 7.22). Its

STG is shown in Fig. 2.8, that of its *dtack*-component in Fig. 7.3. This component has an irreducible CSC. Therefore, the circuit derived from the *dtack*-component after inserting the internal signal *csc*0 (see Fig. 7.23), requires the event *csc*0+ to happen before *ds*−. Only if this is fulfilled, can the derived circuit be guaranteed to work correctly. More about solving CSC for STG decomposition can be found in [SV07].

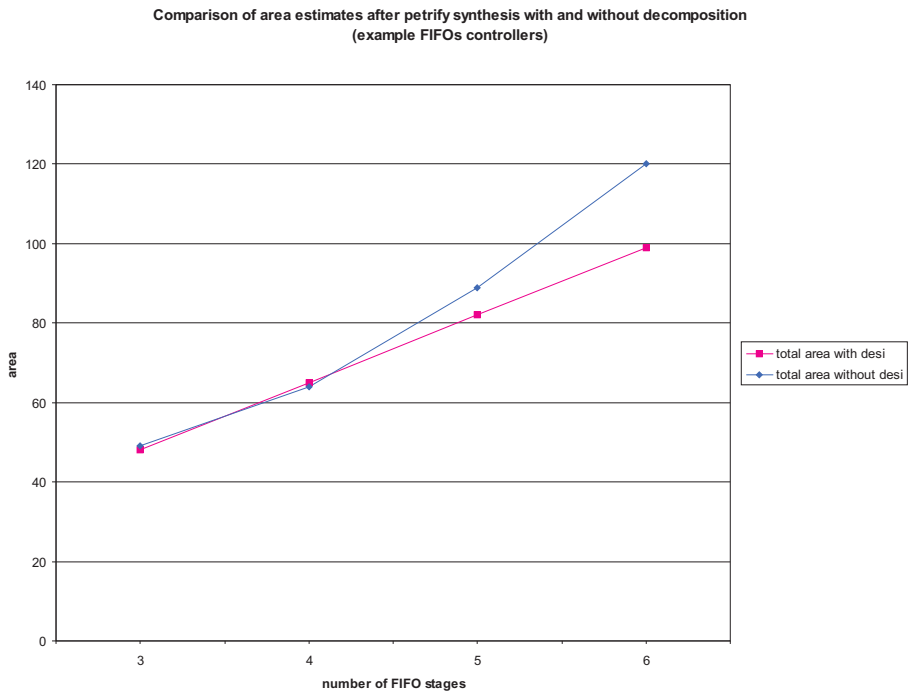


Figure 7.24. Area comparison for FIFO example with and without decomposition

Decomposition is especially effective for iterative specifications. An example for this is the FIFO controller specification in Fig. 5.10. It has 3 stages in the specification and can be enlarged to any finite number of stages. Experimental results for FIFO specifications with 3 to 6 stages are shown in Fig. 7.24 and Fig. 7.25. From Fig. 7.24, one can see that with increasing number of stages, the decomposed approach yields less implementation area. This is because the synthesis tool cannot give an optimal result if the number of reachable markings is large, as in the case of FIFO specification with 5 and 6 stages, without decomposition. Also without decomposition, the time needed to synthesize the specification exponentially increases as shown in Fig. 7.25. Without decomposition, the FIFO specification with 7 stages could not be synthesized due to lack of memory resources.

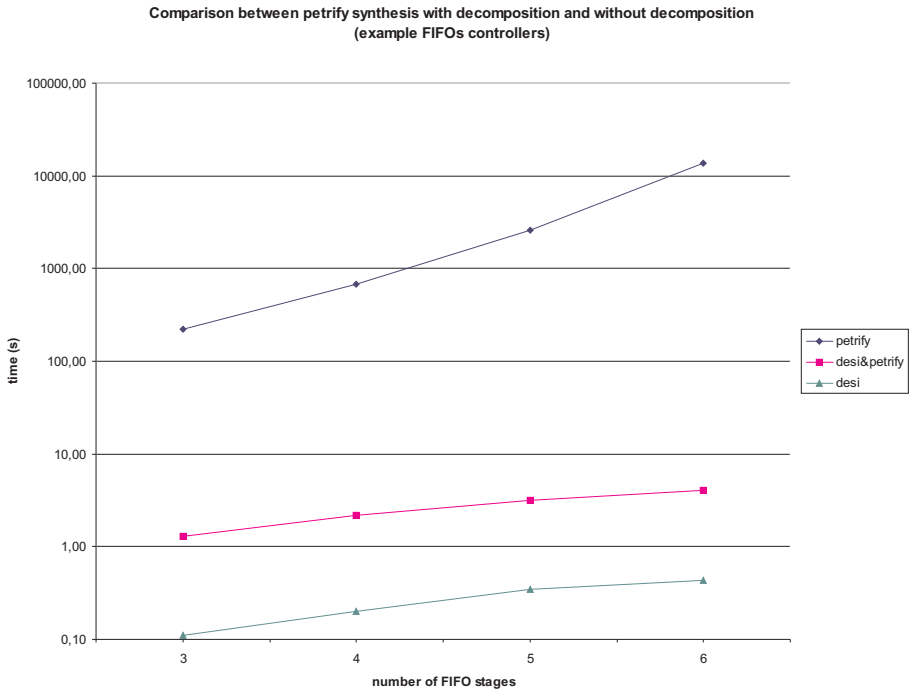


Figure 7.25. Time comparison for FIFO example with and without decomposition (the *desi* graph is for decomposition time only).

# Chapter 8

# Conclusion and Future Work

Improvements of the [VW02] STG decomposition algorithm are in chapter 4 presented. With this improvements not only STGs from real applications can be decomposed, but also better decomposition results can be obtained. Dummy transitions and structural auto-conflicts are allowed. Dummy transitions are often introduced by translating from hardware description language into STG [BL00]. Structural auto-conflicts with control places often occur in an indeterministic specification (e.g for an VME bus controller) or an arbiter specification [Wol97] [YKKL94]. The problem with structural auto-conflict is solved by introducing transition fusion. Problems with non-secure dummy transitions are solved by transforming them into secures ones. This securing transformations and the deletion of loop-only dummy transitions reduce the frequency of backtracking in the algorithm, thereby yielding better decomposition results. Algorithm efficiency is also increased by contracting globally irrelevant signals before decomposition and by reordering the transitions to be contracted.

In chapter 5 the new structural approach of STG decomposition is suggested. A component for an output block is found by synchronizing SCSMs that is needed to produce the output for current component and preserving the synthesizeable property of the initial net. By using this approach, the limitation of net reduction operation is overcomed resulting in smaller end component than [VW02]; also the method is easier to be applied in practice. Comparing it to the [VW02] method, one may think that the SMD-subnet method has an overhead for finding the initial SMD-subnets. But, it is not the case. Finding the SCSMs should be done anyway to decide liveness and safeness of FC nets [KB92] before synthesis. The same argument applies to finding extended structures of the FC net (line 2 - line 6 of algorithm *SMD-Subnet*).

The only overhead of the SMD-subnet method is for synchronizing relevant SCSMs, but this is just a net union. This small overhead pays off by fewer transitions to be contracted, which may yield smaller results than the [VW02] method, as shown by the previous examples.

Hence, the conclusions are as follows. The SMD-subnet method is in most cases more efficient than the [VW02] method. In the case of nets with level SCSMs or regulation circle paths, the SMD-subnet method yields smaller components. Also, it is easier to preserve the layout of the net by the SMD-subnet. The only drawback is that SMD-subnet can only be applied to a restricted class of P/T nets. Despite its current limitations, in most cases, the SMD-subnet method can be applied to practical STG benchmarks. It should be possible to remove this restriction in the future – i.e. by using [VW02] [VK07] method to find SCSMs. This can be done by contracting all the transitions in the net(total contraction) and taking loop-only places as SCSMs.

The explicit consistency requirement is too strict and needs to be improved in the future. A possible improvement could be by finding a consistent SCSM (or even a consistent SMD), such that the net in Fig. 5.4 and 5.5 could be handled by the SMD-subnet method. This could be done e.g. when checking consistency for FC STGs, there already is a polynomial algorithm to check consistency suggested by [Esp03].

Using implicitly consistent SCSMs in addition to explicitly consistent could not only extend the class of nets that be handled by the SMD-subnet method, but could also increase algorithm efficiency. An example is the FIFO net in Fig. 5.10. From the safe cover in Fig. 5.11, the initial *Ain*-component is shown in Fig. 5.12a. $N_2$ is the relevant SCSM for the *Ain*-component which is also a consistent SCSM for *Ain*. Contracting $t_0$ and $t_{23}$ is trivial, because both divining transitions have only one pre and one post place. If the FIFO net is covered by $N_1', N_2'$, instead of $N_1, N_2$ (see Fig. 8.1), then contracting $t_0, t_{23}$ is no longer trivial.

In addition to consistent SCSMs, the SMD-subnet method still could be extended by finding another SCSM structure that, when removed, leaves a live and safe net.For example, generalize the regulation circle path such that the net in Fig. 3.4a also could be handled by SMD-subnet method.

In the future, using the SMD-subnet method, decomposition could be better embedded in a synthesis process. As in synthesis, some conditions for a net to be synthesizable such as liveness, safeness and consistency should be checked. The SMD-subnet method could use parts of the result of this test such as: an SCSM-cover found by liveness and safeness test of FC nets [KB92], an consistent SCSMs (or even consistent SMDs) found by a consistency check. Also, using the so-called correct SCSM-cover [PCKR98], decomposition could be done such that the transition with the signal needed for solving coding conflicts is preserved.

As there are many possible safe covers for a net $N$ due to many possible SCSM subnets of $N$, finding a good cover should be considered in the future. For, the efficiency of the SMD-subnet method may be decreased if the cover is not good, as described above with the *Ain*-component of the FIFO net as the example. Another example of bad covers are the covers with redundant SCSMs. The safe cover $\chi_3 = \{N_1, N_3, N_4, N_5\}$ of $N$ in Fig. 5.2 includes the redundant SCSM $N_1$. $N_1$ and $N_3$ are relevant SCSM for the $x$-component. Without the redundant

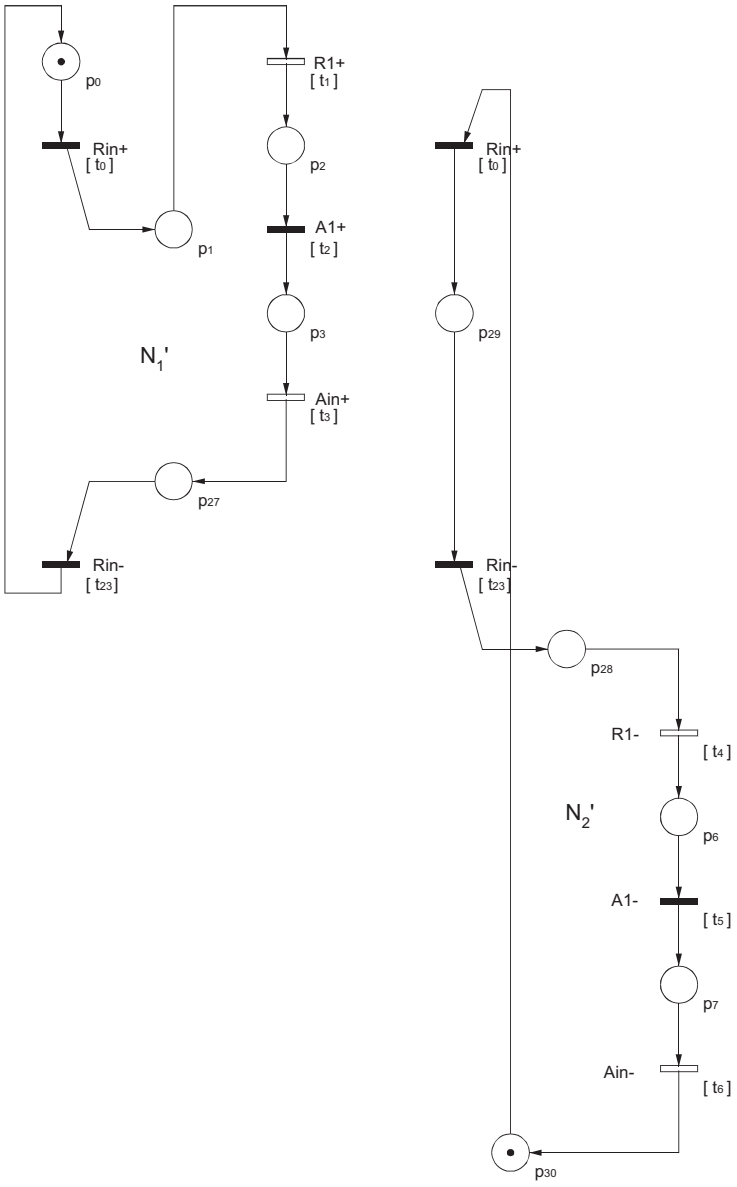


Figure 8.1. Non-consistent SCSM subnets of FIFO example in Fig. 5.10

SCSM $N_1$, only $N_3$ is relevant. The redundant SCSM $N_1$ in this example cause more transitions to be contracted ($t_1$), and non-trivial contraction. Note that, this does not mean that all redundant SCSMs should be avoided. Because, there are redundant SCSM that are needed in the component; e.g. redundant but consistent SCSM that are needed to ensure consistency as in the case of consistent SCSM $N_9$ in Fig. 5.5.

Furthermore, relevant SCSMs in the $\chi_c$ for an output block component could be more restricted to increase efficiency; e.g. a relevant SCSM that will become a loop only place like $N_3$ or $N_4$ for the *Aout*-component of the FIFO example (Fig. 5.12) could be removed directly from $\chi_c$.

Using the structure graph described in chapter 6 to find an SCSM subnet is more efficient in most cases than traversing the P/T net directly. Also, by contracting middle node transitions in one step, transitions contraction is done more efficiently than by successively contracting each transition.

The structure graph of an ordinary P/T net can be used not only for finding SCSM subnets or contracting middle node transitions; but also for other algorithms which traverse nodes in the P/T net. Therefore, it is suggested to use the structure graph as an abstract data structure for a P/T net when implementing such algorithms. Some applications of structure graphs in finding subnets of P/T nets and the experimental results can be found in [War05] [Taw04].

A super node with only divining transitions as middle node transitions can become a redundant place. In the future, this kind of super node can be deleted directly from the net without first contracting all the middle-node divining transitions. For example, the super node $N_2$ in Fig. 6.14 can be deleted directly from the net because it will become a redundant place.

As for asynchronous circuit, [VW02] algorithm is good for speed independent circuit implementation, because such implementation should not reduce the net too much. Too much reduction could remove input signal which is also has a role to achieve complete state coding. Thus, too much reduction could cause that complete state coding could not be found. SMD-subnet base algorithm instead is good for XBMMs implementation, because it could give smaller component and has less problem with loop on transition, which could be found often in XBM specification due to level transition.

The suggested direct translation from STG specification to XBM specification in chapter 7.3.1 improves efficiency of the translation suggested in [BEW99]. This efficiency is achieved by directly translate the specified concurrency instead of deriving the state machine from STG specification. However, not all STGs can be translated directly to XBM specifications because XBM specification allows only specific concurrency which is a subset of the concurrencies possible in STG specifications. The STG decomposition can be used to make an STG specification translatable to an XBM specification. But decomposition does not always solve the problem, because even after decomposition some components may not be XBM translatable.

As shown in DESI experimental results (chapter 7.4.1), decomposition is effective for specifications with a large number of reachable marking. For synthesis tool like petrify cannot synthesize effectively if the number of reachable marking is large, i.e. need more area. It also require plenty computing resources and time. This make it not feasible for that e.g. 7 stage of FIFO specification. However with decomposition there is no problem with computing resources and time. Not only that, the resulting area is smaller with decomposition. For a small number of reachable marking like NEI-arbiter one still can benefit from decomposition which extract module like ME-element from specification.

# Bibliography


[And83] C. André. Structural transformations giving B-equivalent PT-nets. In Pagnoni and Rozenberg, editors, *Applications and Theory of Petri Nets*, Informatik-Fachber. 66, 14–28. Springer, 1983.

[BC92] L. Bernardinello and F. Cindio. A Survey of Basic Net Models and Modular Net Classes. *Lecture Notes in Computer Science; Advances in Petri Nets 1992*, 609:304–351, 1992.

[Bei00] J. Beister. Vorlesung "Entwurf ungetakter Schaltwerke". Technical report, Universität Kaiserslautern, 2000.

[Ber87] G. Berthelot. Transformations and decompositions of nets. In W. Brauer et al., editors, *Petri Nets: Central Models and Their Properties*, Lect. Notes Comp. Sci. 254, 359–376. Springer, 1987.

[Bes87] E. Best. Structure Theory of Petri Nets: the Free Choice Hiatus. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Lecture Notes in Computer Science: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986*, volume 254, pages 168–205. Springer, 1987.

[BEW99] J. Beister, G. Eckstein, and R. Wollowski. From STG to Extended-Burst-Mode Machines. In *Proc. 5th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, 1999.

[BEW00] J. Beister, G. Eckstein, and R. Wollowski. CASCADE: a tool kernel supporting a comprehensive design method for asynchronous controllers. In M. Nielsen, editor, *Applications and Theory of Petri Nets 2000*, Lect. Notes Comp. Sci. 1825, 445–454. Springer, 2000.

[BL89] K. Barkaoui and B. Lemaire. An Effective Characterization of Minimal Deadlocks and Traps in Petri Nets Based on Graph Theory. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets, 1989, Bonn, Germany*, pages 1–21, 1989.

[BL00] I. Blunno and L. Lavagno. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 84–92. IEEE Computer Society Press, April 2000.

[BT87] E. Best and P. S. Thiagarajan. Some Classes of Live and Save Petri Nets. *Concurrency and Nets - Advances in Petri Nets*, pages 71–94, 1987. NewsletterInfo: 27.

[BW93] J. Beister and R. Wollowski. Controller implementation by communicating asynchronous sequential circuits generated from a Petri net specification of required behaviour. In G. Caucier and J. Trilhe, editors, *Synthesis for Control Dominated Circuits*, 103–115. Elsevier Sci. Pub. 1993.

[Chu86] T.-A. Chu. On the models for designing VLSI asynchronous digital systems. *Integration: the VLSI Journal*, 4:99–113, 1986.

[Chu87a] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT, 1987.

[Chu87b] T.-A. Chu. Synthesis of self-timed VLSI circuits from Graph-Theoretic Specifications. In *IEEE Int. Conf. Computer Design ICCD '87*, pages 220–223, 1987.

[CKK+96] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. Technical report, Universitat Politècnica de Catalunya, 1996.

[CKK+02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer, 2002.

[Com72] F. Commoner. *Deadlocks in Petri Nets.* Wakefield: Applied Data Research, Inc., CA-7206–2311, 1972.

[DE95] J. Desel and J. Esparza. *Free Choice Petri nets.* Cambridge University Press Cambridge Tracts in Theoretical Computer Science, 1995.

[Dil88] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent circuits*. MIT Press, Cambridge, 1988.

[DN95] A. Davis and S. Nowick. Asynchronous Circuit Design: Motivation, Background, and Methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer, 1995.

[Ebe92] J. Ebergen. Arbiters: an exercise in specifying and decomposing asynchronously communicating components. *Sci. of Computer Programming*, 18:223–245, 1992.

[EBS89] J. Esparza, E. Best, and M. Silva. *Minimal Deadlocks in Free Choice Nets.* Universität Hildesheim (Germany), Institut für Informatik Hildesheimer Informatik-Berichte 1/89, July 1989.

[ES89] J. Esparza and M. Silva. Circuits, Handles, Bridges and Nets. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets, 1989, Bonn, Germany*, pages 134–153, 1989.

[ES91] J. Esparza and M. Silva. *Handles in Petri Nets.* Universität Hildesheim (Germany), Institut für Informatik Hildesheimer Informatik-Berichte 3/91, April 1991.

[Esp90] J. Esparza. Synthesis Rules for Petri Nets, and How they Lead to New Results. In Baeten, J.C.M. et al., editors, *Lecture Notes in Computer Science; CONCUR'90, Theories of Concurrency: Unification and Extension. (Conference, 1990, Amsterdam, The Netherlands)*, volume 458, pages 182–198, Berlin, Germany, 1990. Springer.

[Esp03] J. Esparza. A Polynomial-Time Algorithm for Checking Consistency of Free-Choice Signal Transition Graphs. In *Third International Conference on Application of Concurrency to System Design (ACSD'03), Guimares, Portugal*, pages 61–70. IEEE, June 2003. InternalNote: Submitted by: hr.

[FN01] R. Fuhrer and S. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools.* Kluwer Academic Publishers, 2001.

[GF66] A. Grasselli and F.Luccio. A Method for the Combined Row Column Reduction of Flow Tables. In *Proceedings of the 7th Ann. Symp. Switching Theory*, 1966.

[Hac72] M. Hack. *Analysis of Production Schemata by Petri Nets.* Cambridge, Mass.: MIT, Dept. Electrical Engineering, MS Thesis., 1972.

[Hac74] M. Hack. *Extended State-Machine Allocatable Nets (ESMA), an Extension of Free Choice Petri Net Results.* MIT, Project MAC, Computation Structures Group, Memo 78–1, 1974.

[Huf64] D. A. Huffman. The Synthesis of Sequential Switching Circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison Wesley, 1964.

[JV80] M. Jantzen and R. Valk. Formal Properties of Place/Transition Nets. In Brauer, W., editor, *Lecture Notes in Computer Science: Net Theory and Applications, Proc. of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, 1979*, volume 84, pages 165–212, Berlin, Heidelberg, New York, 1980. Springer.

[Kan02] B. Kangsah. *Entwicklung und Implementierung eines Algorithmus zur parallelen Dekomposition von Automaten unter Entfernung irrelevanter Eingangsvariablen.* Master thesis, Technische Universität Kaiserslautern, FB Elektrotechnik und Informationstechnik., 2002.

[Kan03] B. Kangsah. Finding STG component with concessioner path. Technical report, Universität Kaiserslautern, 2003.

[KB92] P. Kemper and F. Bause. An Efficient Polynomial-Time Algorithm to Decide Liveness and Boundedness of Free Choice Nets. In Jensen, K., editor, *Lecture Notes in Computer Science; 13th International Conference on Application and Theory of Petri Nets 1992, Sheffield, UK*, volume 616, pages 263–278. Springer, June 1992.

[Kem93] P. Kemper. Linear Time Algorithm to Find a Minimal Deadlock in a Strongly Connected Free-Choice Net. In Ajmone Marsan, M., editor, *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1993, Proceedings 14th International Conference, Chicago, Illinois, USA*, volume 691, pages 319–338. Springer, 1993.

[KGJ96] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *33rd ACM/IEEE Design Automation Conf.*, pages 67–70, 1996.

[KKT93] A. Kondratyev, M. Kishinevsky, and A. Taubin. Synthesis Method in self-timed design. Decompositional approach. In *IEEE Int. Conf. VLSI and CAD*, pages 324–327, 1993.

[KVWB04] B. Kangsah, W. Vogler, R. Wollowski, and J. Beister. Improving STG decomposition. Technical report, Universität Kaiserslautern, 2004.

[KVWB05] B. Kangsah, W. Vogler, R. Wollowski, and J. Beister. DESI: A Tool for Decomposing Signal Transition Graphs. In *Tool demonstration on Application of Concurrency to System Design (ACSD'05)*, 2005.

[KWVB03] B. Kangsah, R. Wollowski, W. Vogler, and J. Beister. DESI: A Tool for Decomposing Signal Transition Graphs. In *3rd ACiD-WG Workshop*, 2003.

[LSV93] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits.* Kluwer Academic Publishers, 1993.

[MB59] D. Muller and W. S. Bartky. A Theory of Asynchronous Circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[Now93] S. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers.* PhD thesis, Stanford University, Department of Computer Science, 1993.

[PCKR98] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Structural Methods for the Synthesis of Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998.

[Pet66] C.A. Petri. Kommunikation mit Automaten. *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, 1:1–Suppl. 1, 1966. English translation.

[RY85] L. Rosenblum and A. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proc. Int. Work. Timed Petri Nets*, Torino, Italy, 1985.

[SF01] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001.

[Sta90] P. Starke. *Analyse von Petri-Netz-Modellen.* Stuttgart, Germany: Teubner, 1990.

[SV05] M. Schäfer and W. Vogler. Component Refinement and CSC Solving for STG Decomposition. In *Lecture Notes in Computer Science: Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005 / Vladimiro Sassone (Ed.)*, volume 3441, pages 348–363. Springer, 2005.

[SV07] Mark Schäfer and Walter Vogler. Component refinement and CSC-solving for STG decomposition. *Theor. Comput. Sci.*, 388(1-3):243–266, 2007.

[SVJ05] M. Schäfer, W. Vogler, and P. Jančar. Determinate STG Decomposition of Marked Graphs. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, Lect. Notes Comp. Sci. 3536, pages 365–384. Springer, 2005.

[Taw04] P. Tawdross. *Structural decomposition of STGs and transformation into XBM machines.* Master thesis, Technische Universität Kaiserslautern, FB Elektrotechnik und Informationstechnik., 2004.

[VK04] W. Vogler and B. Kangsah. Improved Decomposition of Signal Transition Graphs. Technical report, Universität Augsburg, 2004.

[VK05] W. Vogler and B. Kangsah. Improved Decomposition of Signal Transition Graphs. In Jörg Desel and Yosinori Watanabe, editors, *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 244–253. IEEE Computer Society Press, 2005.

[VK07] Walter Vogler and Ben Kangsah. Improved Decomposition of Signal Transition Graphs. *Fundam. Inform.*, 78(1):161–197, 2007.

[VW02] W. Vogler and R. Wollowski. Decomposition in Asynchronous Circuit Design. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design*, Lect. Notes Comp. Sci. 2549, pages 152–190. Springer, 2002.

[VYCLdM94] P. Vanbekbergen, C. Ykman-Couvreur, B. Lin, and Hugo de Man. A generalized signal transition graph model for specification of complex interfaces. In *Proc. European Design and Test Conference*, pages 378–384. IEEE Computer Society Press, 1994.

[War05] S. Warman. *Decomposition of the Structure Graph of a P/T net into siphon-trap Subnets.* Master thesis, Technische Universität Kaiserslautern, FB Elektrotechnik und Informationstechnik., 2005.

[WB00] R. Wollowski and J. Beister. Comprehensive Causal Specification of Asynchronous Controller and Arbiter Behaviour. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 3–32. Kluwer Academic Publishers, March 2000.

[Wen77] S. Wendt. Using Petri nets in the design process for interacting asynchronous sequential circuits. In *Proc. IFAC-Symp. on Discrete Systems, Vol.2*, Dresden, 130–138. 1977.

[Wol97] R. Wollowski. *Entwurfsorientierte Petrinetz-Modellierung des Schnittstellen-Sollverhaltens asynchroner Schaltwerksverbünde.* PhD thesis, Universität Kaiserslautern, FB Elektrotechnik, 1997.

[WVW11] Dominic Wist, Walter Vogler, and Ralf Wollowski. STG Decomposition: Partitioning Heuristics. In Benoît Caillaud, Josep Carmona, and Kunihiko Hiraishi, editors, *ACSD*, pages 141–150. IEEE, 2011.

[YD99] K. Yun and D. Dill. Automatic Synthesis of Extended Burst-Mode Circuits: Part I (Specification and Hazard-Free Implementation). *IEEE Transactions on Computer-Aided Design*, 18(2):101–117, February 1999.

[YDN92] K. Yun, D. Dill, and S. Nowick. Synthesis of 3D Asynchronous State Machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, October 1992.

[YKKL94] A. Yakovlev, M. Kishinevsky, A. Kondratyev, and L. Lavagno. OR Causality: Modelling and Hardware Implementation. In R. Valette, editor, *Applications and Theory of Petri Nets 1994*, Lect. Notes Comp. Sci. 815, 568–587. Springer, 1994.

[Yun94] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems.* PhD thesis, Stanford University, August 1994.

# Lebenslauf

## Akademische Laufbahn

*September 2001 - April 2006*
Wissenschaftliche Mitarbeiter
Lehrstuhl Digitaltechnik
Prof. Dr.-Ing. J. Beister

*August 1999 - August 2001*
Technische Universität Kaiserslautern
Fachbereich Elektro- und Informationstechnik
Abschluss: Master

*August 1992 - August 1996*
Trisakti University, Jakarta
Fachbereich Elektrotechnik
Abschluss: Bachelor