Logic Synthesis of Concurrent Control Specifications

Pavlos M. Mattheakis

University of Crete
Department of Computer Science

Dissertation Submitted in partial fulfillment
of the requirements of the degree
of Doctor of Philosophy

Doctoral Thesis Committee:
University of Crete: Associate Professor Christos Sotiriou (supervisor), Professor Manolis Katevenis
Technical University of Crete: Associate Professor Ioannis Papaefstathiou, Professor Dionisios Pnevmatikatos, Professor Apostolos Dollas
Democritus University of Thrace: Lecturer Giorgos Dimitrakopoulos
University of Thessaly: Associate Professor Ioannis Moondanos

July 2013

© Copyright by Pavlos M. Mattheakis, 2013.

All rights reserved.

UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE

Logic Synthesis of Concurrent Control Specifications

Dissertation submitted by
Pavlos Matthaiakis

in partial fulfillment of the requirements for the
Doctor of Philosophy degree in Computer Science

Author: [illegible]

Pavlos Matthaiakis, University of Crete

Examination Committee: [illegible]

Christos Sotiriou, Associate Professor, University of Crete, *Supervisor*

[illegible]

Manolis Katevenis, Professor, University of Crete

[illegible]

Apostolos Dollas, Professor, Technical University of Crete

[illegible]

Dionisios Pnevmatikatos, Professor, Technical University of Crete

[illegible]

Ioannis Papaefstathiou, Associate Professor, Technical University of Crete

[illegible]

Ioannis Moondanos, Associate Professor, University of Thessaly

[illegible]

Giorgos Dimitrakopoulos, Lecturer, Democritus University of Thrace

Approved by: [illegible]

Panagiotis Trahánias, Chairman of the Department

Heraklion, July 2013

Abstract

The number of transistors per chip increases by 58% per year. At the same time, the designer productivity increases by 21% per year. Thus, an increasing number of design and verification engineers is required to tape-out a chip in the same amount of time. In order to close the design productivity gap the abstraction layer should be raised to boost the design productivity more than the above percentage. For instance, the productivity was increased by 10x in 80s when the state of the art design practice changed from stick diagrams to gate level design. Later on, during the 90s, the productivity increased further by 10x by moving to the RTL level design. Behavioral modeling, lately extended the productivity further by 5x.

In behavioral modeling, the control is decoupled from the datapath. It is separately described by HDL structures which correspond to monolithic FSMs, increasing thus the abstraction layer from RTL to FSMs. The underlying EDA tools extract, synthesize and verify monolithic FSMs with algorithms performing at this higher level of abstraction. For instance, state minimization which was originally handled by the engineers themselves, is automatically performed by the EDA tools increasing the quality of results, the design time and the verifiability.

Although a monolithic FSM is an adequately powerful formalism to describe sequential circuits, it fails to model concurrency without state explosion. Interacting FSM models have so far lacked the formal rigor for expressing the synchronizing interactions between different FSMs. The event based, PTnet model is able to model both concurrency and choice within the same model, however lacks a polynomial time flow to implementation, as current methods of exposing the event state space require a potentially exponential number of states.

In this work, a novel formalism for interacting FSMs is introduced i.e Multiple, Synchronized FSMs (MSFSMs), a compact Interacting FSMs model, potentially implementable using any existing monolithic FSM implementation method. MSFSMs efficiently describe concurrent control systems whilst also acting as an intermediate representation for synthesizing existing specifications described as PTNets with FSM-based flows or for verifying concurrency related properties for systems described as a FSMs with PTNet-based algorithms. PTNet to MSFSMs and MSFSMs to interacting FSMs transformation algorithms are proved in this work to be tractable. Thus, efficient PTNet synthesis and interacting FSMs verification flows are introduced which exploit MSFSMs and which do not exhibit state explosion. Furthermore, novel efficient algorithms introduced at the MSFSM level optimize the control specifications by exploiting the inter-FSM communication.

Experimental results indicate that PTNets can indeed be transformed to synthesizable FSMs through transformation to MSFSMs without exhibiting state explosion. A large set of concurrent specifications was transformed to MSFSMs in less than one second each, whereas tools generating the full state space needed days of execution time just to generate specification's state graph. The logic synthesis framework developed in this work, Expose, approaches the quality of results of logic synthesis tools which generate the exponentially large state space of the specifications, whilst approaching the execution time of the direct-mapping methodologies. Concurrent specifications which could only be implemented through direct mapping, as the execution time for full state space exploration is prohibitive, can now be synthesized using Expose. Our results also show that the MSFSM-based heuristic optimization algorithms drastically and predictably improve the implementation metrics of area and performance as they benefit from the confluence between MSFSMs and state space. By assembling a synthesis flow out of heuristic optimizations, an overall area and performance gain of 80% and 35% respectively was obtained.

Acknowledgements

This work has been partly funded by FORTH-ICS, Greece, which the author would like to acknowledge.

I am especially grateful to my supervisor, Associate Professor Christos Sotiriou for his guidance and support. Many thanks to Professor Peter Beerel for his time and his valuable input. I thank my committee members Lecturer Giorgos Dimitrakopoulos, Associate Professor Ioannis Papaefstathiou, Professor Manolis Katevenis, Professor Dionisios Pnevmatikatos, Professor Apostolos Dollas and Associate Professor Ioannis Moondanos for thoroughly studying this thesis and for their extremely helpful comments and suggestions. I am grateful to Dr. Nikolaos Andrikos, senior EDA Engineer at Atrenta for his comments on the thesis. Special thanks to Despoina Malliwtakh, English Language and Literature specialist for the many hours she spent polishing my writing.

During the past 5 years i had the privilege to cooperate with many exceptional scientists, engineers and managers. Dr. Spyros Lyberis (ARM), Dr. Evriklis Kounalakis (Synopsys), Vasilis Zebilis (Mentor Graphics), Manolis Pantelias (Synopsys), Valia Kassapaki (DTU), Brian Hamel (Nuvolex), Michalis Christofilopoulos (Imagination) and Labros Gkabogiannis (Synergic) were much more than colleagues during the Nanochronous Logic era (2007-2010). After Nanochronous shutdown, Associate Professor Ioannis Papaefstathiou hosted me at the Telecommunication Systems Institute of Technical University of Crete for 15 months (2011-2012). These 15 months were very crucial for the completion of this thesis as Ioannis provided an excellent environment to complete most of my research work. Andreas Brokalakis and Dr. Kostas Georgopoulos where not only my research companions but also friends and made me feel like home. During the spring of 2012 I accepted an offer from the industry and moved to France to work for the Place and Route Division of Mentor Graphics. Laurent Masse-Navette, Patrick Richier, Dr. Pierre-Olivier Ribet, Dr. Hamid

Bouzouzou and Vassilis welcomed me in the CTS group and generously share their extensive industrial experience with me since then (2012-...).

Above all I am grateful to my family. Especially my parents Triantafyllia and Manolis and my sisters Maria, Erasmia and Despoina have always been there for me and for thus I thank them infinitely. Maria has always been willing to share my fears and my dreams and has made me a better person. I would finally like to thank my friends. Giwrgos K., Giannis, Andreas, Vassilis, Ritsa, Natasa, Giwrgos M., Stavros, Manos, Dionisia just to name a few of the closest.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Electronic systems are widespread in everyday life. From appliances to military equipment, devices are controlled by microchips. A microchip receives data, processes it, and presents the result back to its environment. Data processing is performed by microchip components, which are categorized into control and datapath. The latter performs the actual data computation, whereas the former orchestrates the result flow through the components. Although both control and datapath are comprised of the same components, typically logic gates, memory and wires, the formalisms which are used to model them are different.

Datapath is usually represented with two-level or multilevel Boolean algebra, which are variants of the ordinary elementary algebra with restricted value set 0 and 1 modeling the low and high voltage of circuit components respectively. Control is typically specified in more stateful formalisms, as the flow of data depends on both the current and previous inputs which are thus stored in memory. These stateful formalisms span from models introduced to naturally describe state and causality, *e.g.* FSMs [61], to models more suitable to describe concurrency. Concurrency is the property that describes the parallel execution of events. PTNets [84] are a model which inherently expresses concurrency, as it has structures explicitly describing it, *e.g.*

fork and join transitions. The models which are not inherently concurrent, describe concurrency implicitly. For instance, an FSM describes concurrency by interleaving transitions which typically describe causality [24].

Although all specifications from the aforementioned formalisms spectrum, are adequate modeling concurrency either implicitly or explicitly, they suffer from the state explosion problem, either when representing the system or when getting synthesized to actual hardware. The state space explosion problem arises in concurrent systems in the cases where the complete set of states (state space) has to be generated, as the size of the latter is exponential when compared to the initial specification. For instance, FSMs poorly model concurrent systems due to the exponential number of states required to specify the system. Nevertheless FSMs enjoy a rich background of heuristic algorithms which efficiently synthesize them to hardware [49]. PTNets' on the other side, being an inherently concurrent formalism, describe concurrent systems with compact, *i.e.* non-exponential in size, specifications [76]. However, PTNet synthesis algorithms are exponential, as the state space of the model is implicit, and thus during synthesis it has to be generated. For this reason, PTNet synthesis typically begin by transforming the original model to an equivalent FSM [77] as the latter describes model's state explicitly.

Between PTNets which are very expressive but lack synthesis algorithms and the FSMs which enjoy mature synthesis algorithms but lack expressiveness, there exist various control circuit specification which trade expressiveness for implementability. For instance, interacting FSMs enhance the monolithic FSM model with the ability to explicitly describe concurrency. However, interacting FSMs lack the rigor to explicitly represent synchronization of the concurrent events. State machines with forks-joins [63] (SFJs), is a model which stems from PTNets and which enjoys a low complexity synthesis path. However, SFJ's expressiveness suffices to just model the simplest of the PTNet classes, *i.e.* Marked Graphs and State Machines.

1.1 Motivation

Specifying concurrent control systems with inherently concurrent specifications such as PTNets has the advantage that specifications are compact. However, the fact that changes at the specification level have unpredictable results at the final implementation, makes is cumbersome even to manually describe small designs. On the other hand, using models with less expressibility, *e.g.* FSMs, has the advantage that changes at the specification level are predictable but the size of the specification itself grows exponentially with the concurrency. A novel formalism which not only explicitly models concurrency but also enjoys an efficient implementation path could solve the deficiencies of the two aforementioned models and it would pose itself as an alternative solution for specifying and implementing hardware control systems with concurrency. As this work mainly focuses on the implementation of concurrent specifications, the novel formalism will be evaluated on the extend it solves three major issues which are faced by the concurrent systems synthesis approaches: state space explosion, specification and state confluence and decomposability. Throughout this work, PTNets will be used as the baseline formalism for describing concurrent formalisms due to their expressiveness and the rich background of algorithms operating at the PTNet level.

1.1.1 State Space Explosion

The state space explosion problem refers to the need to generate the complete state space described by a concurrent specification in order to synthesize it.

The conventional PTNet implementation flows are presented in Figure 1.1. In particular, PTNets are implemented to digital logic by either direct-mapping places and transitions to circuit structures [94] or by synthesizing the specification [25].

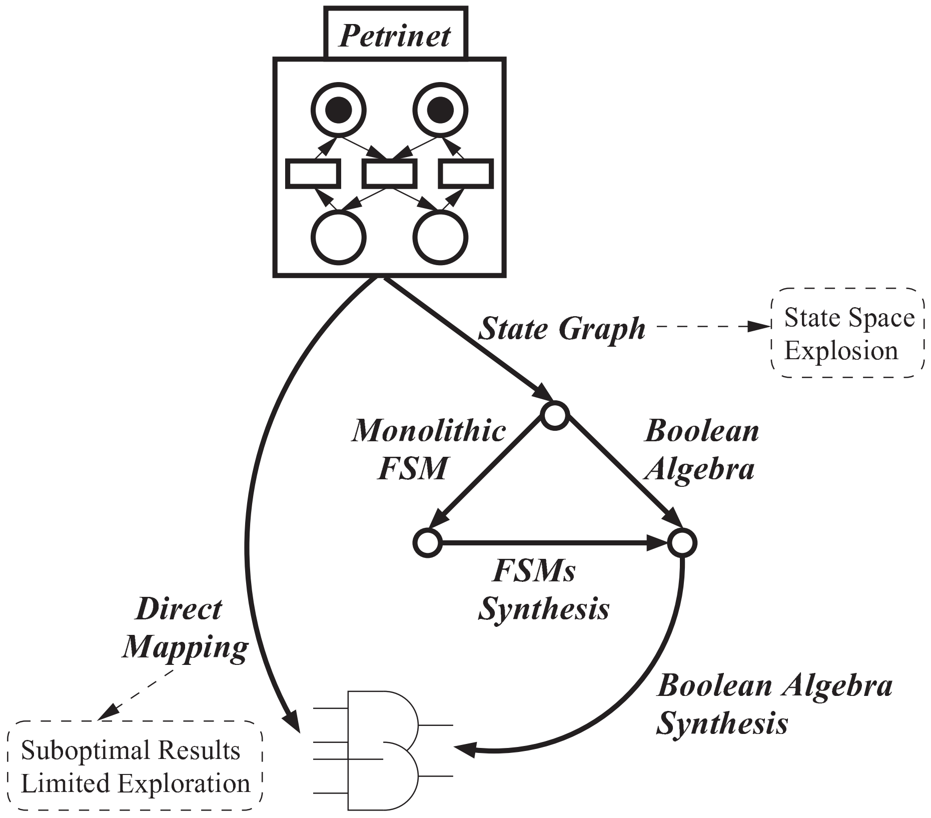
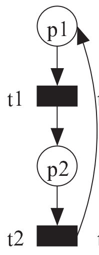


Figure 1.1: Petrinet Implementation Flows

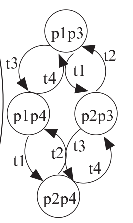
Direct mapping techniques transform the specification structures *i.e.* places and transitions to circuit components, so that a 1-1 correspondence between PTNet and digital logic is achieved. The drawbacks of direct mapping are the suboptimal implementations and the significantly limited exploration of the solution space. Nevertheless, direct-mapping presents a non-exponential flow, in which changes at the PTNet level have a predictable impact on both area and performance of the implementation.

Synthesis techniques, on the other side, start by transforming the PTNet to either a monolithic FSM [67] or to Boolean Algebra [28]. Afterwards, the corresponding synthesis algorithms are applied to the intermediate specification to optimize and map the design to digital logic. The main drawback of both synthesis approaches is the state space explosion which is due to the significant gap between higher level specifications, *e.g.* PTNets, and the lower level ones, *i.e.* monolithic FSMs and boolean algebra. The aforementioned gap can be realized as an inefficiency by the lower level control models to describe higher level properties, *e.g.* concurrency.

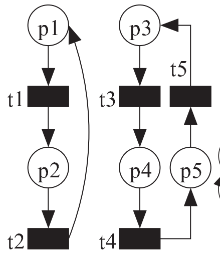
1.1.2 Specification and State Confluence



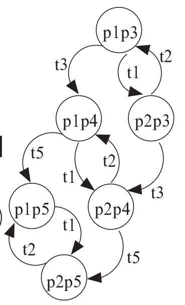
(a) Original Petrinet



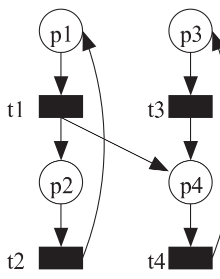
(b) Original Petrinet's State Graph



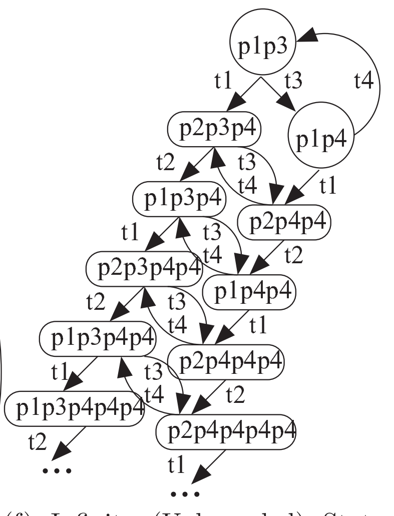
(c) Adding a Place after t_4



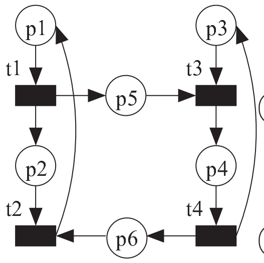
(d) State Graph Increases by 2 States and 4 Transitions



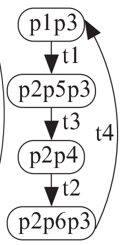
(e) Adding an Arc from t_1 to p_4



(f) Infinite (Unbounded) State Graph



(g) Adding two places



(h) Smaller State Graph

Figure 1.2: Confluence between Petrinet and the underlying State Graph

One of the advantages of the FSM model is the significant number of available synthesis heuristics [49]. These heuristics predict the impact on the final implementation, caused by operations on the model. As an example, the first step of the FSM synthesis flow, minimizes the number of states. Afterwards, the resultant model is assumed adequately optimal and hence the state minimization is not re-run. The above is due to a heuristic, which states that in most cases fewer states result to more efficient final implementations of the FSM. Analogous heuristics exist for the state encoding steps when targeting area, performance and power metrics. PTNets on the other side, lack relevant heuristics due to the unpredictable way that the state space is modified after processing the specification. Hence, many PTNet synthesis algorithms require returning to the initial specification, performing changes and then examining the resultant state graph, as there are no intermediate points (as the state minimized model for the FSM model) of the model in the flow, which can be assumed adequately optimized to freeze the model there. Besides that, in specific cases it is required to generate the complete state space after each operation on the PTNet, as it can be the case that the model is no longer synthesizable.

Figure 1.2 demonstrates that slight changes in the PTNet shown in Figure 1.2a, result to equivalent monolithic FSMs, which may significantly differ from initial PT-Net's original FSM shown in Figure 1.2b. In particular, Figure 1.2c illustrates a PTNet, which slightly differs from the original one as a place and transition pair is inserted after transition t_4 and before place p_3. The resultant monolithic FSM is very close to the initial one, as the pair of states p_1p_5 and p_2p_5 is added to capture the behavior of the system when the PTNet is at place p_5 and at places $p1$ or $p2$. The above can be assumed a predictable change in the state space, as adding a pair and a transition adds two states and two transitions in the monolithic FSM. On the contrary, Figure 1.2e shows the original PTNet with the addition of a single arc connecting transition t_1 with place p_4. The equivalent monolithic FSM shown in the Figure 1.2f has infinite number of reachable states, as the number of tokens of places p_3 and p_4 forever increase as transition t_1 fires. As the state space is infinite, an implementation is infeasible for this specification. Finally, in Figure 1.2g, the initial

PTNet is enhanced with two places, *i.e.* p_5 and p_6 and four arcs and the resultant state graph, shown in Figure 1.2h, has the same number of states and half the number of transitions when compared to initial PTNet's SG. Hence, there can be cases where introducing places and transitions may significantly reduce the initial state space, instead of increasing it.

1.1.3 Decomposition

PTNet clustering and decomposition have been suggested as solutions to the problem of state space explosion by transforming a PTNet to a set of smaller PTNets of which the total state space is significantly smaller than the one of the original specification.

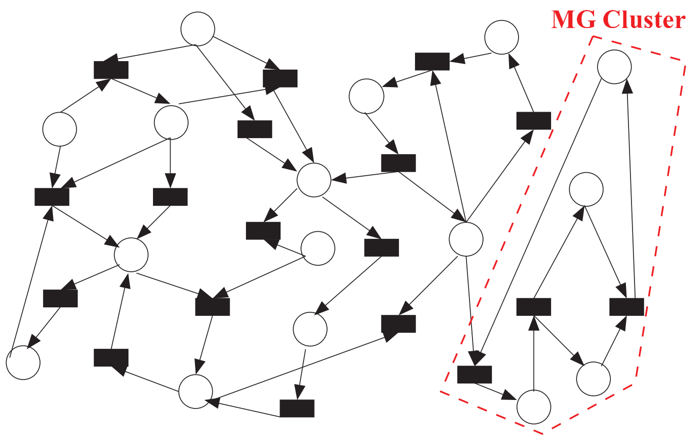


Figure 1.3: PTNet Clustering

Clustering [43] partitions the PTNet graph to subgraphs which fall to simpler PTNet classes *e.g.* Marked Graphs, and applies synthesis techniques only to them, leaving the rest of the network as is, assuming that low complexity direct translation flows will be applied. Besides the greedy algorithms identifying the optimizable portions of the PTNet, clustering can only be used as a local optimization, surgically applied to a restricted parts of the specification. There is no guarantee that the optimizable clusters represent a significant part of the complete state space and although it delivers better results than a pure direct mapping method, its results are not close to a full fledged synthesis approach. An example is shown in Figure 1.3 where a marked graph subgraph is identified out of the original PTNet which falls in the general PTNet class.

PTNet decomposition [102, 23] focuses on system's outputs, generating a component for each one of them. The parts of the original PTNet which are attached in each output's component are called component support. The aim of the decomposition is to yield a set of components with corresponding state graphs which are individually smaller than the original state graph. Hence, if the critical component *i.e.* the component with the largest state space, is significantly smaller than the original state graph, then it is assumed that the solution is towards solving the state space explosion problem. However, the efficiency of the components cannot be quantified unless their actual state graphs are generated which is exponential as well. Additionally, there is no guarantee that by partitioning the state space according to the primary outputs, the state graph will be decomposed to significantly smaller components. Finally, in the context of asynchronous circuits synthesis, not each PTNet decomposition is valid for synthesis. The above is due to unresolvable coding conflicts which were not present in the initial specification but were introduced during decomposing it. As with the quantification of the decomposed specification, identifying whether a PTNet can be encoded requires the complete state space of the system.

1.2 Aims

With respect to state space explosion, the aim of this work is a polynomial complexity flow for transforming an event-driven FCPTnet into a state-based Interacting FSMs model. This flow tackles the deficiencies of the PTnet and monolithic FSMs models, *i.e.* state explosion and efficient representation of concurrency, acting as bridge between these two models. The key to the flow is the definition of a new formalism which exposes the FSM interactions.

The above formalism results from a novel PTNet decomposition method which generates a set of components with explicit state which do not suffer themselves from the state space explosion problem.

Furthermore, this work aims to solve the PTNet-implementation confluence by introducing a set of operations acting on the introduced intermediate model and which have predictable and monotonous impact on the final implementation. As the transition from the PTNet model to the MSFSMs model is polynomial, several operations can be examined on the specification iteratively without significant cost with respect to execution time.

Finally, two control logic synthesis flows are developed targeting synchronous and asynchronous circuit implementations. The monotonous impact of the aforementioned operations, allows estimating the outcome of each step and hence building sophisticated synthesis scripts issuing mixes of the MSFSMs operations according to the desirable target *e.g.* area. The synthesis infrastructure is realized with a new logic synthesis tool.

1.3 Structure of the Work

This work is organized as follows:

- Chapter 2 briefly describes the most popular formalisms used to model concurrent and sequential control systems. Their formal definitions, basic properties and state of the art implementation flows are described as well.

- Chapter 3 introduces the MSFSMs intermediate model of computation developed in this work to link the PTNet and FSMs worlds. Its basic structures are defined, and its equivalence with the PTNet model is proved.

- Chapter 4 presents two polynomial flows transforming a PTNet to the equivalent MSFSMs model and the opposite.
- Chapters 5 and 6 introduce synchronous and asynchronous implementation flows respectively, based on the MSFSMs formalism.
- Chapter 7 presents a set of optimization operations applied to both synchronous and asynchronous flows.
- Chapter 8 shows the results of the introduced synthesis flows.
- Finally, Chapter 9 concludes this work and presents several directions for future work.

Chapter 2

Control Models

Control systems are modeled with formalisms, sufficient to express system characteristics such as state, choice, concurrency and synchronization. Monolithic FSMs constitute the most popular control model, and are supported by the majority of industrial scale logic synthesis tools [32, 33]. These logic synthesis tools exploit the rich background of heuristic algorithms developed to efficiently synthesize FSM specifications. Nevertheless, FSMs suffer from the state explosion problem when expressing concurrency through state interleaving. PTNets on the other hand, compactly represent concurrent specifications. However, current PTNet synthesis techniques, require the generation of the complete state space of the system, as at the PTNet level the state is implicitly described.

This chapter introduces the two aforementioned formalisms and presents their basic properties and their most popular implementation flows. In addition, it presents formalism variants of the above which were introduced to extend the expressibility of the original model *e.g.* interacting FSMs or to enhance it with timing information *e.g.* burst-mode FSMs.

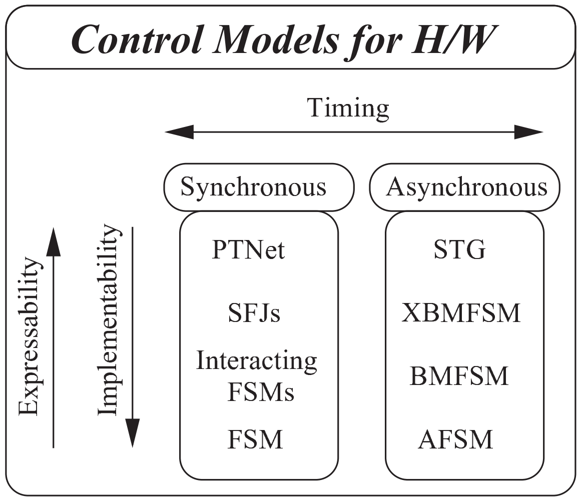


Figure 2.1: Hierarchy of the Control Models describing H/W with respect to Timing, Implementability and Expressiveness

2.1 Categorization

Figure 2.1 illustrates a set of control models and it categorizes them according to their timing, expressibility and implementability. Timing refers to whether the control model poses the assumption of a global synchronizing scheme or not, whereas expressibility and implementability denote the efficiency of a model to describe high level system properties and whether it enjoys heuristic synthesis algorithms respectively.

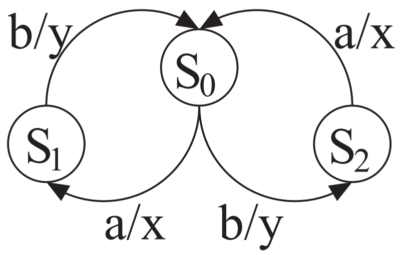
2.1.1 Expressiveness

Boolean Algebra, Monolithic FSMs, Interacting FSMs and PTNets are popular formalisms used to describe control systems. This section compares the above with respect to their efficiency on expressing control system properties *i.e.* State, Choice, Concurrency and Synchronization.

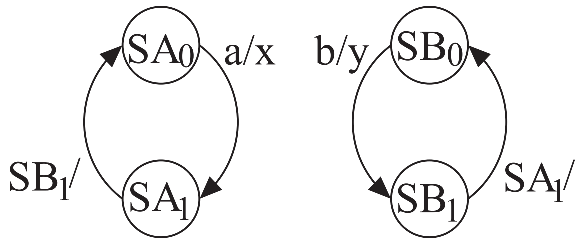
Figure 2.2 exploits the aforementioned formalisms to specify the control logic of a system. The latter has two inputs, a and b, and two outputs x and y. x and y are asserted whenever a and b are asserted respectively. The environment re-asserts a and

$$\boxed{\begin{array}{l} S_0 = S_1 b + S_2 a + S_0\overline{(a+b)} \\ S_1 = S_0 a + S_1\overline{b} \\ S_2 = S_0 b + S_2\overline{a} \\ x = (S_0 + S_2)a \\ y = (S_0 + S_1)b \end{array}}$$

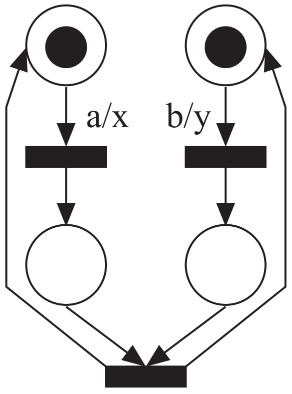
(a) Boolean Algebra



(b) Monolithic FSM



(c) Interacting FSMs



(d) PTNet

Figure 2.2: Models of Computation

b only after x and y are both asserted, and until then, the system is not allowed to reply to a and b assertions. Three states can be used to capture the above behavior, S_0, S_1 and S_2. The system is at S_0 when both x and y have been asserted, whereas it is S_1 and S_2 when only a and only b are asserted and outputs x and y are generated respectively.

Figure 2.2a shows the system if Boolean Algebra is used. Boolean Algebra is a low level formalism used for implementation purposes and thus modeling high level system properties, *e.g.* state, can only be implicitly described using operations which naturally describe another property *e.g.* the '+' operator which realizes the Boolean OR operation can also be used to implicitly describe choice. Furthermore, identifying such properties in a given model, is only possible after a detailed analysis. In particular, the system is assumed to be stateful, if functions have the same variable on both sides, *i.e.* there is feedback. State choice can be hypothesized by checking whether two state variables are set to true when the same state variable is active, but different signals should be satisfied as well, *e.g.* S_1 and S_2 are both set to true if S_0 is true, but S_1 requires a whereas S_2 requires b. The above presumes that a and b are mutually exclusive, otherwise both S_1 and S_2 are activated. Even higher level properties like concurrency can also be deduced if the functions are simulated in time, *e.g.* either activating S_0, S_1 and then S_0 or S_0, S_2 and then S_0 yields the same result *i.e.* x and y are generated after a and b are generated independently from their order. Furthermore, as the system in both cases returns to state S_0, it is assumed that a synchronization is performed.

Monolithic FSMs is probably the most popular formalism for specifying control systems by humans. Figure 2.2b shows the test-case described as a monolithic FSM. State is explicitly modeled using formalism semantics, whereas, choice is explicitly modeled using more than one edges departing from a single vertex. However the monolithic FSM formalism fails to explicitly express higher level properties, such as concurrency and synchronization. The concurrency is deduced by analyzing all state transition sequences which also reveals that the state branch and merge at state S_0, are used to ultimately model concurrency and synchronization respectively.

Interacting FSMs is a formalism which succeeds in explicitly modeling concurrency, as each separate FSM independently identifies inputs and generates outputs. However, synchronization cannot be expressed using model semantics. In the test-case, states SA_1 and SB_1 are added in the input and output set of the FSMs, as I/O signals SA_1 and SB_1 respectively. These signals allow modeling the information that FSMs FSM_1 and FSM_2 are at states SA_1 and SB_1 respectively and hence realize a synchronization scheme among the two FSMs.

PTNets are event driven systems, which naturally model concurrency and synchronization. Their basic structures are transitions and places. The former are similar to FSM transitions whereas places store system's state. A place may carry an arbitrary number of tokens. If at least one token resides in a place then this place is marked likewise an FSM is active. Synchronization is explicitly described with a join transition which consumes tokens from more than one predecessor places thus synchronizing the corresponding flows.

Model	**State**	**Choice**	**Concurrency**	**Synchronization**
Algebra	Implicit	Implicit	Implicit	Implicit
Monolithic FSM	Explicit	Explicit	Implicit	Implicit
Interacting FSMs	Implicit	Explicit	Explicit	Implicit
PTNet	Implicit	Explicit	Explicit	Explicit

Table 2.1: Control Models Expressiveness

The above analysis is summarized in Table 2.1. Boolean Algebra cannot explicitly express properties characterizing control models. Monolithic FSMs succeed in modeling state and choice, and their extension, interacting FSMs, enable modeling concurrency explicitly. PTNets, being the most expressive model, have inherent structures capturing synchronization as well, but the state space they model is encoded in the model itself.

2.1.2 Implementability

The implementability of a formalism refers to the efficiency of its implementation algorithms. For instance, as shown in Chapter 1, PTNets lack confluence between changes at the specification level and impact of the last on the implementation. Hence, PTNet synthesis algorithms are typically based on translation of the initial specification to a monolithic FSM which enjoys a rich background of heuristic algorithms targeting specific implementation metrics *e.g.* performance.

Control models described in Boolean Algebra are synthesized to digital logic using boolean logic synthesis tools [14]. These tools initially optimize the given specification by applying multilevel and two-level synthesis techniques and then map the optimized model to a given technology library. The metrics which estimate area, performance and power at this level are the literal count, the logic depth and the boolean node activity respectively [91]. The literal count is the sum of the literals in all boolean functions. Logic depth is defined for each Boolean network node and represents the maximum number of nodes (or levels) from the inputs to this node. It estimates the arrival time of the gates resulting from this node. Dynamic power is estimated by applying transition probabilities, also known as activity factors, at circuit's inputs. These probabilities are propagated through the boolean network measuring the probability for each node to change value and hence consume dynamic power.

FSM implementation flows [49] are comprised of three steps: state minimization, state encoding and logic optimization. State minimization reduces the number of states by collapsing states which show the same behavior. In general, an FSM with fewer states results to less area as the state holding elements realizing the FSM states are reduced. State encoding [37] assigns for each state a separate binary code. State encodings are realized by state holding elements, *e.g.* FFs. A change in the FSM state occurs when the differing bits from one state to the other have all changed value. The logic optimization step links the FSM and Boolean Algebra synthesis flows. The transition and output functions of the state-minimized and encoded FSM are transformed to a Boolean Network and the aforementioned Boolean Algebra synthesis techniques are applied.

Interacting FSM synthesis is based on operations between FSMs *e.g.* composition, and on the FSM communication. FSM composition collapses a set of FSMs into a single monolithic FSM whereas FSM decomposition [38] splits a monolithic FSM to a set of interacting FSMs. Besides operations between FSMs, the communication can also be exploited to optimize the interacting FSMs system [36]. In particular, if words in an FSM input alphabet are never generated by its communicating counterparts (sequential input don't cares), then these inputs can be assumed don't cares, exploited by the monolithic FSM minimization algorithms.

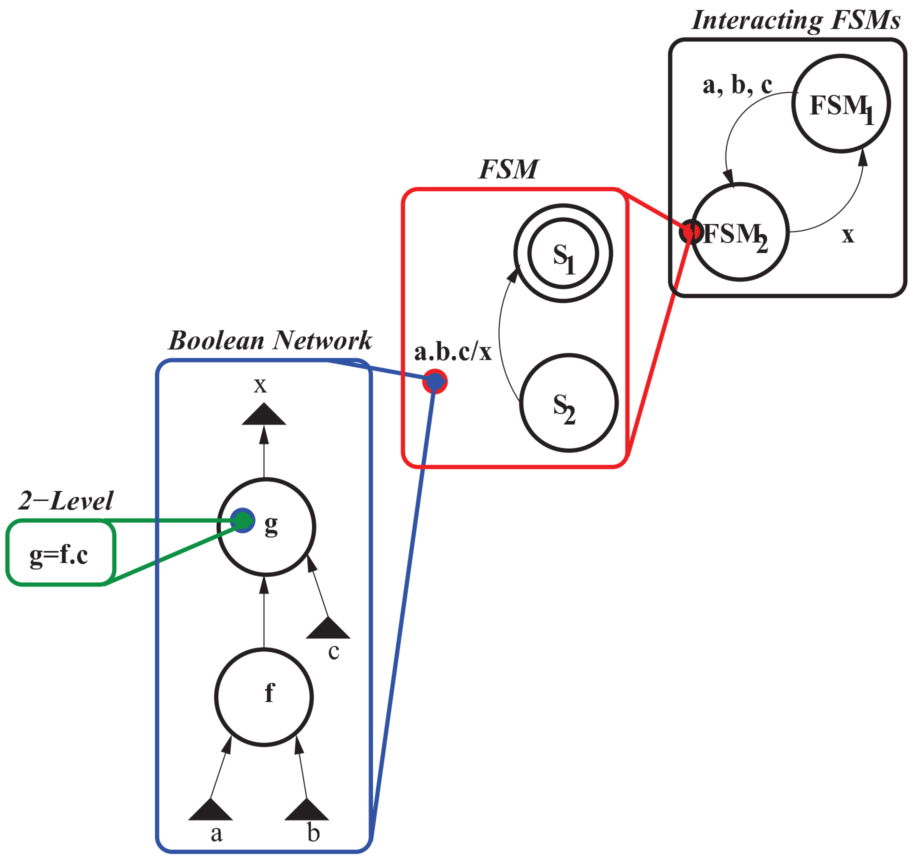


Figure 2.3: Hierarchical view of the Interacting FSMs/FSM/Boolean Network/2-Level Boolean Algebra Models

The above analysis is depicted in Figure 2.3. Moving from the interacting FSMs model to the 2-level boolean algebra, the hierarchy of the control models is revealed. In particular, each node at the interacting FSMs representation corresponds to a monolithic FSM, which is turn comprised of output and next state boolean functions. Each of the above functions maps to a multilevel boolean network, in which the nodes correspond to 2-level boolean functions.

2.1.3 Timing

In synchronous circuits, the control logic is evaluated at each clock cycle to dynamically define the data flow, unconditionally consuming power. This continuous polling of the digital logic is not the only solution for designing control circuits. Asynchronous circuits design represents an alternative design approach, where logic is evaluated and, hence, consumes power, only when inputs change.

Besides power savings, asynchronous circuits have several other advantages compared to their synchronous counterparts [77]:

1. **Elimination of the Clock Skew Problems**

   In clocked circuits, the clock signal must arrive at all FFs at exactly the same time. This task is not trivial, as it requires building balanced clock trees with the lowest possible latency. Balancing is achieved with the insertion of buffers which consume power at every clock cycle. Asynchronous circuits do not require driving any global signals throughout the whole chip, hence most of the problems related to skew are insignificant. Between synchronous and asynchronous there exist clocking techniques where the global clock is substituted by local clocks *e.g.* Desynchronization [29], where local clock skew is an issue.

2. **Average-Case Performance**

   Synchronous circuits must operate at the worst case scenario, *i.e.* a sum of the critical path delay, which is the greatest combinational delay in a chip when the operating conditions (Temperature, Voltage) are the worst possible and a set of margins added to compensate for clocking imperfections and variability which describes the varying behavior of the same circuit components (i) in the same chip (intra-die variability) and (ii) in different chips stemming from the same fabrication process (inter-die variability). On the other hand, asynchronous circuit performance is dynamic and dictated by the actual exercised critical path.

3. **Adaptivity to Variations**

   Clock period of synchronous circuits is over-constrained, compensating for the worst scenario of processing and environmental variations. Asynchronous circuits automatically adapt to any variations, speeding up or slowing down as necessary. For instance a clocked circuit operates at the frequency which is adequate for correct operation under the worst conditions of Voltage and Temperature whereas its asynchronous counterpart dynamically adapts its performance according to the actual operating conditions.

4. **Component Modularity and Re-use**

   Interfacing synchronous circuits from different clock domains, requires synchronization circuitry. Implementing and verifying synchronizers imposes extra effort to the design process, as the currently available tools operate at each separate IP without addressing all the issues stemming from the IP communication. On the other side, asynchronous circuits glue with less effort given that they follow the same communication protocol when exchanging data.

5. **Reduced EMI**

   In synchronous circuits the signaling activity happens at specific frequencies which are related to the clock domains frequencies. Hence, the energy is concentrated on specific points throughout the frequency spectrum resulting to high peaks and hence electrical noise. Asynchronous circuits do not suffer from high energy peaks in the frequency spectrum, as there are no specific frequencies gathering all the energy activity.

On the other side, the most significant disadvantages of asynchronous circuits are the following:

1. **Lack of mature EDA tools**

   Synchronous circuit design techniques enjoy two decades of continuous development of industrial scale EDA tools supporting them. On the contrary, the EDA tools supporting asynchronous circuit design, are often merely suited for academic purposes. Therefore, they lack the infrastructure to handle real life designs and hence to illustrate their advantages over the synchronous ones in an industrial oriented environment. For instance, significant effort has been put on synchronous EDA tools throughout the last decades in order to handle the multi-billion instance designs currently fabricated.

2. **Lack of Unified Specification Model**

   Synchronous circuits specifications are usually Hardware Description Languages (HDLs) such as Verilog and VHDL. Recently, High Level Synthesis (HLS) has been evolving, allowing specifying designs at a much higher abstraction level, closer to the most popular programming languages, *e.g.* C/C++. On the other side, asynchronous design requires familiarizing with new HDLs *e.g.* Balsa [9], and formalisms, *e.g.* PTNets, which are usually more complex than their synchronous counterparts, as they have to explicitly model the concurrency and synchronization among the hardware components. In synchronous designs, the presence of global clocks provides a trivial synchronization mechanism, encapsulated in the 'always' blocks of HDLs such as Verilog or in the 'wait' statements of higher level languages such as SystemC. Designs with multiple clocks and clock domain crossings, require extra effort to describe the communication between clock domains. However, this effort is almost transparent from the designer standpoint, as it is automatically handled by EDA tools with limited user input.

3. **Several Modes of Operation**

   The constraints between an asynchronous circuit and its environment define its operation mode. If there are no restrictions and hence the environment is allowed to reply at any time after it identifies circuit outputs changes, then the circuit operates in input-output mode. Alternatively, if the environment is allowed to respond to circuit outputs only after the circuit signals are all stabilized in specific values, then the circuit operates in fundamental mode. The fundamental mode can be further categorized into single input change (SIC) and multiple input change (MIC) modes of operation depending on the number of circuit inputs that the environment is allowed to change at each separate interaction with the circuit. Synchronous circuits just require inputs from the environment to satisfy the setup and hold violations. Furthermore, asynchronous circuits are divided into classes according to the delay models assumed for gates and wires. Circuits designed under the fundamental mode operation usually follow the same delay model as synchronous circuits which assumes bounded delays for both gates and wires. Delay insensitive (DI) circuits constitute another class which assumes unbounded delays for both gates and wires. The Speed Independent (SI) class lies between the two aforementioned classes, assuming unbounded gate delays and bounded wire delays. On the other hand, synchronous circuits have no such categorization and their correct operation only relies on timing analysis tools.

Summarizing the above remarks, asynchronous circuits potentially present significant advantages over their synchronous counterparts. However, the lack of mature

EDA tools combined with with the numerous operation modes and circuit classes have been obstacles towards the wide adoption of asynchronous circuits by the industry.

Besides their differences, synchronous and asynchronous design methodologies also share several similarities. There are many formalisms used to model and optimize sequential specifications and they are adequately abstract to not be explicitly synchronous or asynchronous. For instance, FSMs with specific enhancements for synchronous and asynchronous operation, are used to model control circuits of both, as they have no assumptions on the timing of the system. These formalisms carry a rich theoretical background which is can be exploited to optimize specification independent from their timing semantics.

2.2 FSMs

The monolithic Finite State Machine (FSM) model is the protagonist of contemporary formal control models. The monolithic FSM model's simplicity, its seemingly vast expressive power, combined with its implementability, include some of the key reasons for its popularity in the field of hardware design, where it represents the atomic unit for specifying and implementing sequential circuits. In the FSM circuit implementation literature, output signal timing, state changes and input sampling are usually synchronized to a clock. However, asynchronous FSM implementations have also been proposed, which typically impose assumptions on input arrival, output departure, and state changes.

2.2.1 Definitions

Definition 2.2.1. *An FSM, M, is a five-tuple, $M = (I, O, S, s0, \delta, \lambda)$, where I is a finite, nonempty set of inputs, O is a finite, nonempty set of outputs, S is a finite nonempty set of states, $s0$ is the initially active state, $\delta : I \times S \to S$ is the next state* *function, and $\lambda : I \times S \rightarrow O$ (for a Mealy machine), or $\lambda : S \rightarrow O$ (for a Moore machine) is the output function.*

All practical FSMs, possess an initial state S_0, assumed during system initialization. If, the next state and output functions, δ and λ, are specified for all possible inputs, then the FSM is completely specified, otherwise the FSM is incompletely specified with next state and output Don't Cares (DCs) respectively. FSMs are typically visualized using State Graphs, Flow Tables or Cube Tables [49].

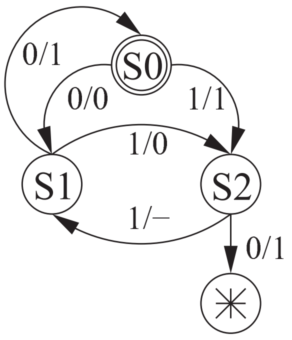


Figure 2.4: FSM Example

	0	**1**
S0	S1,0	S2,1
S1	S0,1	S2,0
S2	-,1	S1,-

Table 2.2: Flow Table of FSM in Figure 2.4

Figure 2.4 shows the following Mealy style FSM :

- $I = \{0, 1\}$
- $O = \{0, 1\}$
- $S = \{S0, S1, S2\}$
- $s0 = S0$
- $\delta = \{(S0, 0, S1), (S0, 1, S2), (S1, 0, S0), (S1, 1, S2), (S2, 0, -), (S2, 1, S1)\}$

- $\lambda = \{(S0, 0, 0), (S0, 1, 1), (S1, 0, 1), (S1, 1, 0), (S2, 0, 1), (S2, 1, -)\}$

The transition marked as * denotes the DC next state, *i.e.* when FSM is at state $S2$ and the input is 0 the FSM next state can be $S0$, $S1$ or $S2$. Accordingly, if the active state is $S2$ and the input is 1 then the output can be either 0 or 1 whilst moving to state $S1$.

Figure 2.2 shows the flow table of the FSM in Figure 2.4. The first row includes the input combinations (or input words). In the following rows, the first column has the active state and the following ones the (next state, output) pairs which correspond to the input word of the same column. For instance, the cell in the third row and second column describes that if the current state is $S1$ and the input is equal to the input word 0 then the next state will be $S0$ and the output will become 1.

2.2.2 FSM Synthesis and Implementation

The aim of conventional FSM synthesis is the generation of an efficient circuit in terms of area, power and speed which abides by the initial FSM specification. The FSM synthesis problem generally belongs to the NP class [49], however effective algorithms exist [89], after more than four decades of research, which can handle from small to large FSMs (with hundreds or thousands of states). To manage its complexity, FSM synthesis is broken down into three consecutive steps [49], *i.e.* (i) state minimization, (ii) state encoding and (iii) logic minimization. These steps are formalized with the appropriate, widely-used models *e.g.* State Minimization as a Binate Covering Problem [30] (BCP), and corresponding mature nowadays, and efficient algorithms. FSM algorithms fall into both the exact and heuristic categories and are based on a considerable amount of theory for existing mathematic formalisms. Specifically, State Minimization is based on Finite Automata Theory, State Encoding to Algebraic Partitions [51] theory and Logic Minimization on two and multi-level boolean level minimization [100].

2.2.2.1 FSM Minimization

FSM minimization is the process of minimizing the FSM's states. The heuristic metric for FSM minimization algorithms is the number of states of the resultant machine. FSM minimization algorithms can be categorized to the ones operating on completely or incompletely specified FSM. The former describe FSMs where at each state or transition the inputs and outputs have specific values and the next state is specified whereas in the latter inputs and outputs can be dont cares *i.e.* either 0 or 1, and the next state can be an arbitrary one.

The complexity of the completely specified FSM minimization has proven to be $nlogn$ where n is the initial number of states [53]. Therefore, an exact solution can be found in short time, even for huge FSM specifications.

However, Completely specified FSMs rarely appear in practical designs. A designer never specifies the next state function for the whole set of input vectors or the outputs' function, as a significant portion of input vectors are don't cares for each state. This implies that these input vectors will never appear according to the specification at that state. Exact algorithms for minimizing an incompletely specified FSM have been defined since the late 1970s [61]. For Incompletely-specified FSMs, the state minimization problem is broken to a number of subproblems, namely prime compatible state extraction [72], BCP formulation, BCP covering and FSM mapping of the BCP solution.

The prime compatibles extraction procedure identifies sets of states which may be collapsed to a single state. For a set of states to be classed as compatible, the following must hold.

1. their outputs, when specified, should be identical,

2. their next states, when specified, should also be compatible.

The compatibles which are not a subset of other compatibles and the compatible sets they imply are relatively small, are good candidates for state collapsing. This set of candidate compatibles grows exponentially with the number of don't cares. A good approximation is the extraction of the maximum compatibles [72], that is the compatibles which are just not included in another compatible without taking into account the implying sets of compatibles.

BCP covering selects among all the prime compatibles the best solution which covers all the states of the initial specification using the minimum number of prime compatibles. The complexity of the BCP problem is NP. The size of the BCP problem grows with the number of the compatibles which are exponential in size. However, methods exist which prune the solution space of the BCP problem by reducing its input [88].

If a state of the original specification is included into more than one of the compatibles of the final solution, it should be decided which of them should take its place in the final flow table. The latter is the FSM state mapping step. This decision is very crucial, as a wrong decision at that step may cancel the whole minimization procedure.

Industrial synthesis programs, such as Synopsys DC [4], are able to perform state minimization.

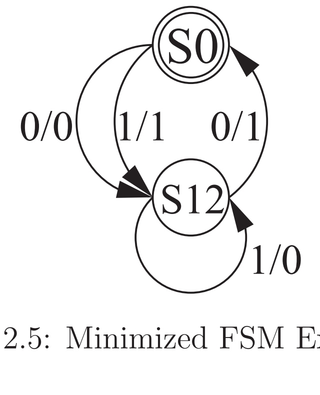


Figure 2.5: Minimized FSM Example

In the FSM shown in Figure 2.4 state $S0$ is neither compatible to state $S1$ nor to state $S2$. In the former case, for both input combinations the output value is different whereas in the latter case the outputs dont match only for input 0. On the contrary, states $S1$ and $S2$ are compatibles as not only their outputs match for each input but also the next states are compatibles. Collapsing the compatible states of the above FSM results to a the minimized specification shown in Figure 2.5.

2.2.2.2 State Encoding

State encoding is the process of assigning a single binary encoding to each FSM state. As the next state functions and the output functions depend on the encoding, different encodings result to different logic equations which in turn produce different circuit implementations.

Algorithms exist [37] which target next state and output functions with the fewest possible number of literals. The heuristic in these approaches is to assign encodings which differ to as possible less number of bits (neighbor encodings) to states which share common information. For example, one of the heuristics is to assign neighbor encodings to states which share their successor states. This enables the simplification of the next state function logic, as the next state function will be factored with the common part of the neighbor encodings. The same heuristics can be followed for the output functions and the states which act on the same input vectors.

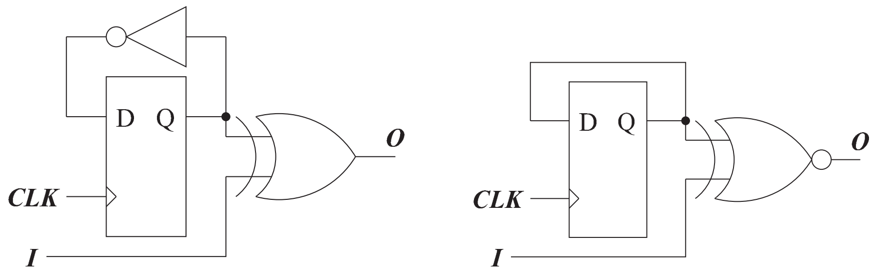
Other approaches try to partition states according to output functions support [52] [61]. The target there is to obtain groups of states which specify different output signals. As a result, the support of the output functions is reduced which is a good indicator that simpler logic will be derived. The same algorithms can be extended for the next state functions. Thus, if the set of states is decomposed to a number of simpler FSMs which have no state transitions between them then a parallel decomposition is derived which minimized the support for all the next state functions. This happens because for each decomposed segment of the initial FSM, separate state variables are used.

Although state encoding is a synthesis step which can significantly optimize the FSM's implementation, many industrial designers constrain EDA tools to encode the state minimized FSMs using a specified encoding [47]. Surprising as this may seem academically, it is now common industrial practice to (almost blindly) use one-hot encoding, if the synthesis goal is speed. Alternatively, if the synthesis goal is area, binary encoding is typically the preferred choice, as it generates encodings with $logn$ state vectors, where n is the number of states after minimization. Gray encoding is the solution adopted by engineers if the design is power constrained, as a single bit will change value during a state transition.

2.2.2.3 Logic Minimization

During the last step, logic minimization is performed. All the previous algorithms target to providing this step with a set of Boolean equations amenable to logic minimization. For instance, a fanout oriented encoding algorithm [37] produced output functions with common factors, as it is expected that multilevel optimization procedure will be favored by this characteristic.

State of the art exact and heuristic algorithms alike Espresso [15] are employed to optimize the design at this synthesis step.



(a) S0=0, S12=1 (b) S0=1, S12=0

Figure 2.6: Logic Implementation of FSM in Figure 2.4 for two possible state encodings

Figure 2.6 shows the circuit implementations of the FSM of Figure 2.4 using two possible state encodings. The reset circuitry is omitted. The implementation of

Figure 2.6a requires a D-FF, an inverter and a XOR whereas the implementation of Figure of Figure 2.6b is comprised of a D-FF and an XNOR.

2.2.3 Control Flow Graphs

High Level Synthesis (HLS) [31] flows compile hardware described in software programming languages into hardware. Many of the semantics and algorithms originally developed for software optimization are exploited in the context of hardware optimization as well. The formalism used to abstract the control flow of the compiled programs is the Control Flow Graph (CFG) [6].

CFGs and FSMs stem from finite automata and hence they share the same properties. For that reason, algorithms developed for FSM analysis and synthesis can be used for CFGs and vice versa. For instance, dominator trees, which are data structures exposing the control dependence in CFGs, were introduced in software compilers to minimize the number of states by removing unnecessary operations without performing reachability analysis in a CFG. Later on, in the context of HLS, dominator trees are exploited to optimize the delay in large Boolean networks [8] or to identify re-converging paths in circuits [96].

2.2.3.1 Definitions

Dominator trees graphically represent the immediate dominator relation between states in a CFG. The definitions of dominators, immediate dominators and dominator trees are as follows.

Definition 2.2.2. *A node d (strictly) dominates a node n if every path from the start node to n must go through d (and d does not equal n).*

Definition 2.2.3. *A node d immediately dominates n if it strictly dominates n but does not strictly dominate any other node that strictly dominates n.*

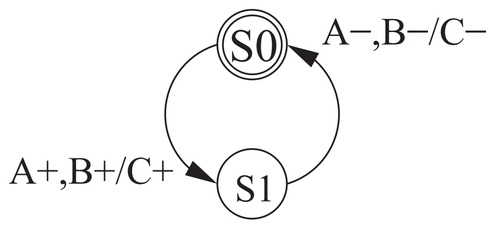
Definition 2.2.4. *In a dominator tree each node's children are those nodes it immediately dominates.*

2.2.4 Asynchronous and Burst-Mode FSMs

Asynchronous FSMs (AFSM) were initially formalized by Huffman, Unger [98] to follow a Single Input Change (SIC) operation mode where at every state transition, a single input event can be processed and a single output event may be generated. The environment cannot feed the AFSM with a new signal event, until the AFSM has stabilized, i.e. no wires or gates are changing value. This one-sided constraint for the environment has characterized the operation mode of AFSMs as Fundamental Mode.

In the case of interacting AFSMs, a part of the environment is modeled by AFSMs as well. As a result, for the fundamental assumption to hold, significant delays will be needed for both interacting components forming a two-sided constraint. Nowick [79], extended the SIC to Multiple Input Change (MIC) mode, where during every FSM state transition, a set of input events may be processed and a set of output events may be generated. Input and output event sets are called bursts and FSM operation based on such bursts is called Burst-Mode (BM). BM circuits were used in industrial test chips, *e.g.* Davis Post Office Chip [34], even before their formalization.

Definition 2.2.5. *A BM-FSM is a 4-tuple (X, Y, Z, δ), where X is a non-empty set of PIs, Y is a non-empty set of POs, Z is a possible empty set of internal state variable symbols and $\delta : X \times Y \times Z \rightarrow Y \times Z$ is a next state function. BM-FSMs can also been represented by a directed graph $G = (V, E, I, O, v_0)$ where V is a finite set of states, E subset $V \times V$ is a set of finite transitions, I is the set of inputs, O is the set of outputs and v_0 is the initial state.*



	A	B	C
S0	0	0	0
S1	1	1	1

Figure 2.7: BM-FSM Example

Figure 2.7 shows a BM-FSM describing the operation of a C-Element [95] which is a sequential element with a single output which is (de)asserted when all its inputs are (de)asserted. The sets which define the BM-FSM are the following:

- X = {A, B}
- Y = {C}
- Z = {S0, S1}
- δ : (A+,B+,S0) $\rightarrow$ (C+,S1), (A-,B-,S1) $\rightarrow$ (C-,S0)

Output transitions depend on the current FSM state and a subset of input transitions, similarly with synchronous Mealy FSM specifications. However, there is a number of restrictions which differentiate BM-FSMs from synchronous Mealy FSMs.

1. **Asynchronous Operation**

   BM-FSMs are asynchronous, thus if the BM-FSM is in a given state, and all input burst transitions have fired, then the machine will transition immediately to the corresponding next state. Hence, the performance metric for this type of circuit is not the slower register-to-register path, but latency *i.e.* the delay from the input events to their related output events .

2. **Unique Entry Condition**

   If a state may be reached from the initial state through different paths, then the input and output values should be the same regardless of the path followed. If this condition is not met, then the state must be split to a set of states equal to the number of states which precede it and generate different values for a subset of inputs and outputs

3. **Monotonic Signal Transitions**
   An FSM only accepts monotonic transitions. The behavior of the system will be erroneous if a glitching signal is perceived or a hazard occurs. The same holds for the produced outputs. All produced output events are monotonic and can only occur during the specified FSM transitions. With respect to the BM-FSMs state signals, a hazard may cause the FSM to enter the wrong next state.

4. **Signal Events rather than Levels**
   All FSM transitions are based on signal events. It is illegal to use a signal level at the input conditions.

2.2.4.1 Synthesis

Although the BM-FSM synthesis steps resemble that of the synchronous FSM synthesis flow, a number of modifications and enhancements have been introduced to support a completely asynchronous flow. During conventional state minimization, states which can be collapsed are grouped to compatible sets, and the multi-set combination, which generates the least number of states is identified. In general, a set of states is compatible, if all of its states are output compatible, *i.e.* produce the same output values, and their next states are also compatible, *i.e.* the next states for each input combination are members of a compatible set. In BM-FSMs, an additional restriction is imposed, *i.e.* that no hazard is introduced by merging the compatible states, are so-called DHF-compatible [44]. State assignment is also more complicated than in the synchronous case, as the encodings selected for two possibly consecutive states must not generate ambiguous transitions, *i.e.* the final next state must not depend on the order that the state bits change value [45]. With respect to logic minimization, conventional exact and heuristic minimization algorithms [97] have been transformed so that synthesised functions are monotonic, for the specified set of valid transitions. Finally, two-level circuit decomposition is accomplished using rules which do not introduce hazards such as the associative, distributive and De-Morgan's laws.

2.2.5 Advantages and Disadvantages

The ubiquitously used monolithic FSM represents the maturest control model of behavior and direct implementation in hardware design and verification. FSMs are intimately related to Finite Automata, their theoretical counterparts. In the monolithic FSM model, each state corresponds to a set of memory cells, depending on state encoding, and the FSM's next state and output logic functions are straightforwardly implementable using digital logic and may be manipulated using Boolean Algebra.

The primary advantage of the FSM model is its direct correlation to its logic circuit implementation, which, in contrast to higher-order models studied in this work, *e.g.* PTnets, implies that an FSM specification possesses a viable logic synthesis path. Further on, specification optimizations imply predictable and deterministic changes to hardware cost and resources, *i.e.* the FSM model is confluent with its boolean logic implementation.

The drawbacks of the monolithic FSM model include its inefficiency to model concurrency [24], and the fact that it is not scalable, *i.e.* composing multiple FSMs into a single one is generally intractable [71].

2.3 Place Transition Nets

A PTNet (or PTNet) is a formalism inherently describing concurrent systems. Carl Adam Petri informally introduced them in 1939 to describe chemical processes [85]. In the context of control systems for hardware design, PTNets were introduced in [23] to describe and synthesize asynchronous circuits. Besides asynchronous circuits, PTNets were also suggested as a vehicle to model concurrent synchronous systems [48].

2.3.1 Definitions

PTnets are divided into classes, depending on structural interactions between choice and concurrency. Below, we present the fundamental PTnet definitions.

Definition 2.3.1 (Net)**.** *A net N is a triple (S, T, F) where S, T are two finite disjoint sets, corresponding to the Places and Transitions of the net respectively, and F is a flow relation on $S \cup T$, such that $F \cap (S \times S) = F \cap (T \times T) = \emptyset$.*

Given a net N, the set ${}^\bullet x = \{y|(y, x) \in F\}$ is the pre-set of node x and the set $x^\bullet = \{y|(x, y) \in F\}$ is its post-set. A triple $N' = (S', T', F')$ is a Net's $N = (S, T, F)$ subnet if S', T' are subsets of S, T respectively and F' consists of the subset of $(S' \times T') \cup (T' \times S')$ which is in F. A Net's marking assigns to every Place a non-negative integer. This number is the marking of the specified Place. The visualization of a Net's marking is a number of tokens in each Place, equal to Place's marking. A marking M enables a Transition t_i, if every Place in ${}^\bullet t$ is marked. If a Transition t is enabled in a marking M_j, then it fires reducing the marking of ${}^\bullet t$ by 1 and increasing the marking of $t^\bullet$ by 1. The set of all markings reachable from M is denoted as $[M >$.

Definition 2.3.2 (PTnet System, Reachable Markings)**.** *A Place, Transition Net system is a pair (N, M_0), where N is a connected net with at least one Place and one Transition, and M_0 is a marking of N, called the initial marking. A marking is called reachable in a system, if it is in $[M_0 >$.*

The fundamental difference between a net and a PTnet system is that the former is merely a set of static relations, whereas the latter is a dynamic system able to model concurrency, conflict (choice) and complex interactions of the two.

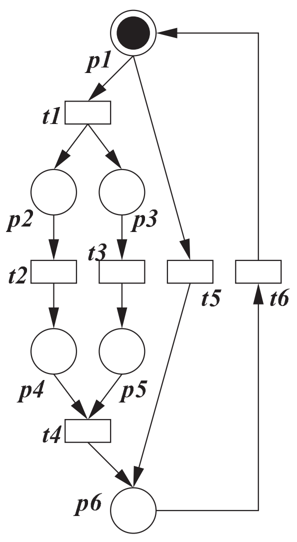


Figure 2.8: PTNet example

Figure 2.8 shows a PTNet system with six places and five transitions. The initial marking of the system is $\{p1\}$ and the set of the rest reachable markings is the following: $\{p2, p3\}$, $\{p2, p5\}$, $\{p3, p4\}$, $\{p4, p5\}$, $\{p6\}$.

We now review fundamental PTnet properties.

Definition 2.3.3 (Liveness and Deadlock Freedom). *A system is live if, for every reachable marking M and every Transition t, there exists a marking $M' \in [M >$ enabling t. If $(N,\ M_0)$ is a live system, then M_0 is denoted as a live marking of N. A system is deadlock free, if every reachable marking enables at least one Transition, i.e. no dead marking is reachable from the initial marking.*

Liveness states that any transition can be activated from any reachable marking. It is a crucial property for concurrent systems as it guarantees that the system will be never deadlocked. The PTNet shown in Figure 2.8 is live as all transitions can be enables from any reachable marking.

Definition 2.3.4 (Place Bound, Bounded Systems). *A system is bounded, if for every Place s, there exists a natural number b, such that $M(s) \leq b$, for every reachable marking M. The bound of place s is $max\{M(s)|M \in [M_0\}$. A system is b-bounded if every Place is bounded by b.*

PTNet boundedness guarantees that during system operation the number of tokens residing in a place is bounded. If this property does not hold then an implementation following the specification behavior cannot be derived, as it would require an infinite number of resources modeling PTNet tokens. The PTNet shown in Figure 2.8 is 1-bounded (also known as safe-PTNets) as at most one token can reside at its places during token move.

Definition 2.3.5 (Well-formed nets)**.** *A net N is well-formed, if there exists a marking M_0 such that (N, M_0) is a live and bounded system.*

An important predicate of a net, used by both analysis and covering algorithms, is the characterization of a set of Places as a siphon.

Definition 2.3.6 (Siphons)**.** *A set R of Places of a net N is a siphon, iff $R \neq \emptyset \wedge^{\bullet} R \subseteq R^{\bullet}$.*

In other words, a siphon is a set of places which cannot receive tokens from its neighboring places. Hence, if any of the siphons loses its tokens during operation, it is guaranteed that the system is not live. The PTNet shown in Figure 2.8 includes three siphons: $\{p1, p2, p4, p6\}$, $\{p1, p3, p5, p6\}$, and $\{p1, p2, p3, p4, p5, p6\}$.

Definition 2.3.7 (S-Component)**.** *A net N' is an S-Component of net N, generated by a nonempty set of nodes X, where for each Place s of X, ${}^{\bullet}s \cup s^{\bullet} \subseteq X$, $|{}^{\bullet}t| = 1 = |t^{\bullet}|$, for every Transition of N', N' is strongly connected and N' is a subnet of N.*

An S-Component is similar to a state machine as it cannot express concurrency.

Definition 2.3.8 (S-Cover)**.** *A set of S-Components, C, is an S-Cover, if every Place of a net belongs to an S-Component of C.*

The existence of an S-Cover guarantees that the system can be covered by a set of S-Components. The PTNet shown in Figure 2.8 is covered by the two S-Components shown in Figure 2.9.

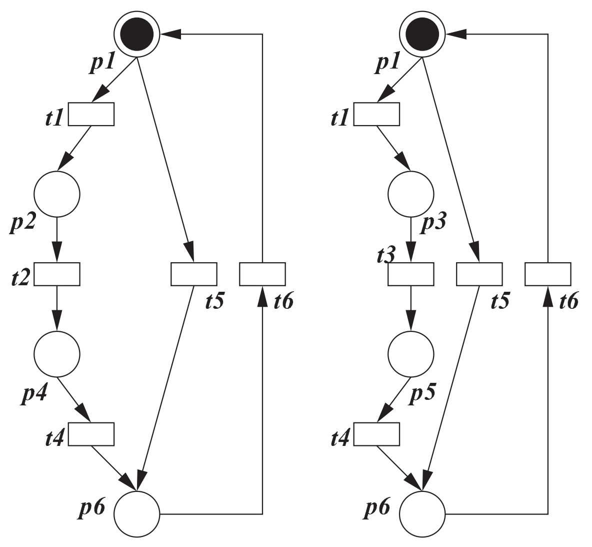


Figure 2.9: SCover of PTNet in Figure 2.8

Definition 2.3.9 (Free-Choice PTnet System, Net [35]). *A net $N = (S, T, F)$ is free-choice, if $(s, t) \in F$ implies ${}^{\bullet}t \times s^{\bullet} \subseteq F$ for every Place s and Transition t. A system (N, M_0) is free-choice if its underlying net is free-choice.*

FCPTnets represent the most ubiquitous form of PTnets, due to the following two Theorems.

Lemma 2.3.1 (S-Coverability [35]). *Well-formed, connected free-choice nets are covered by S-Components.*

S-Coverability has been shown to be achievable in polynomial time for Free-choice [55] and Extended Free-choice (EFC) [56] Nets. A practical algorithm for the minimization of the obtained S-Cover is presented in [68]. The inverse of the previous theorem, *i.e.* that an FCPTnet which is covered by S-Components is well-formed, also holds.

Lemma 2.3.2 (Well-formedness of FCPTnet[57]). *For a live and bounded FCPTNet, N, it holds that:*

- *N is strongly connected,*

- *each Place belongs to a minimal siphon R,*
- *each N's subnet $(R,\ R^\bullet \cup^\bullet R,\ (R \times R^\bullet) \cup (^\bullet R \times R))$ is an S-Component,*
- *each S-Component is initially marked.*

2.3.1.1 Higher-Order PTnet Classes

Definition 2.3.10 (Asymmetric-Choice PTnet System, Net [35])**.** *A net is asymmetric-choice if, for every two Places s and r, either $s^\bullet \cap r^\bullet = \emptyset$, or $s^\bullet \subseteq r^\bullet$, or $r^\bullet \subseteq s^\bullet$. A system $(N,\ M_0)$ is asymmetric-choice if N is asymmetric-choice.*

Definition 2.3.11 (General PTnet System, Net)**.** *A net is general if, for every two Places s and r such that $s^\bullet \cap r^\bullet \neq \emptyset$, it holds that $(s^\bullet - r^\bullet \neq \emptyset) \wedge (r^\bullet - s^\bullet \neq \emptyset)$.*

Higher-order PTnet classes, *e.g.* Asymmetric-Choice (AC-PTnets) and General (G-PTnets) PTnets, exhibit confusion, *i.e.* complex interactions between concurrency and choice. In practice, algorithms which, for FCPTnets, exhibit polynomial complexity, such as S-Covering, have been shown to be intractable for the AC-PTnets and G-PTnets classes.

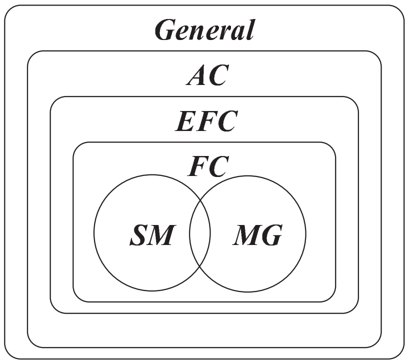


Figure 2.10: PTNet Classes Hierarchy

Figure 2.10 illustrates the hierarchy of the PTNet classes. For instance a EFC-PTNet is also AC-PTNet and G-PTNet but not FC-PTNet, SM-PTNet or MG-

PTNet. It should be noted that there are PTNets which belong to all the aforementioned classes. These PTNets lie in the SM-PTNet and GM-PTNet sets intersection.

2.3.2 Signal Transition Graphs

This section introduces STGs [23], *i.e.* PTNets enriched with the notion of signal events, capable of describing asynchronous circuits behavior.

Definition 2.3.12. *An STG is a tuple $N = (P, T, W, MN\ , In, Out, l)$ where $(P, T, W, MN\ )$ is a Petri net and In and Out are disjoint sets of input and output signals. For $Sig := In \cup Out$ being the set of all signals, $l : T \rightarrow Sig \times \{ +, - \} \cup \{ \lambda \}$ is the labeling function.*

A plus sign in a transition's label denotes that a signal value changes from logical low (written as 0) to logical high (written as 1), and a minus sign denotes the other way around. The function λ must consistently encode the STG markings, that is, no marking M can have an enabled rising (falling) transition $M\rangle a+$ $(M\rangle a-)$ if $\lambda(M)_a = 1$ $(\lambda(M)_a = 0)$.

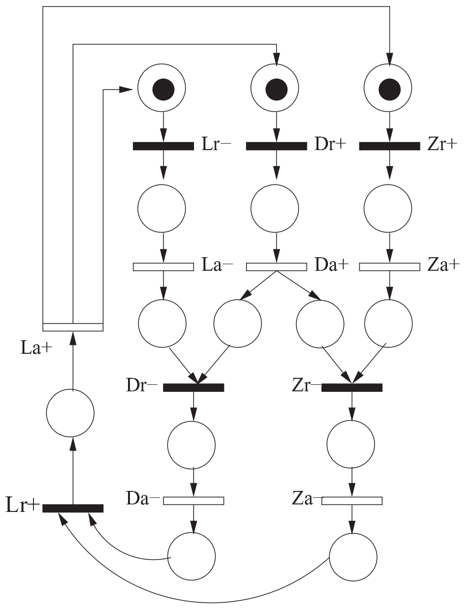


Figure 2.11: STG Example: adfast asynchronous circuit benchmark

Live and safe free-choice STGs satisfy the consistency condition iff they do not contain autoconcurrent transitions, and every sequence of signal transitions is switchover correct. In order to avoid autoconcurrent transitions, no pair of transitions a_i* and a_j* of the same signal can be concurrently enabled at any marking.

Figure 2.11 shows an STG describing the $adfast$ asynchronous control of an A/D converter.

2.3.3 Structural Properties

This section presents the PTNet structural properties which are used to efficiently examine significant characteristics of the specification *e.g.* consistency. Concurrency and Interleave relations are two such properties defined between places, transitions and signals of the PTNet and are mainly used to prove PTNet's implementability criteria. Marked Regions and Cover Cubes are structures approximating the ON-set of output signals boolean functions, exploiting specification's structural properties.

The Concurrency Relations [82] between pairs of nodes $(T \cup P)$ of an STG are defined as a binary relation CR.

Definition 2.3.13. *Given places p_i, p_j and transitions t_i, t_j*

- $(t_i,\, t_j\ ) \in CR \Leftrightarrow \exists\, M\, :\, M[t_i t_j\rangle \wedge M[t_j t_i\rangle$
- $(p,\, t_i) \in CR \Leftrightarrow \exists\, M,\, M'\, :\, [M[t_i\rangle M' \wedge M(p) = M'(p) = 1]$
- $(p_i,\, p_j) \in CR \Leftrightarrow \exists\, M\, :\, [M\rangle t_i M' \wedge M(p_1) = M(p_2) = M'\,(p_1) = 1 \wedge M'(p_2) = 0]$

The Concurrency Relations between a pair node $x_j \in P \cup T$ (either place or transition) and a signal $a \in A$ are defined as a Signal Concurrency Relation (SCR), such that $(x_j,\, a) \in SCR \Leftrightarrow \exists\, a_i* : (x_j,\, a_i*) \in CR$.

Besides the CR which characterizes the parallelism between PTNet structures, the predecessor and successor functions and the Interleave Relation are introduced to define the transitions ordering.

Definition 2.3.14. *A transition a_i is a predecessor of a_j if there exists an allowed sequence $a_i\ \sigma\ a_j$ that does not include other transitions of signal a.*

Conversely, a_j is a successor of a_i. We will also say that the pair $(a_i,\ a_j\ )$ is adjacent. The set of successors (predecessors) of a_i is denoted by $next(a_i)$ $(prev$ $(a_i))$.

Definition 2.3.15. *The Interleave Relation is a binary relation IR between nodes in $P \cup T$ and pairs of adjacent transitions a_i and a_j of a signal a such that, a node u_j is interleaved with $(a_i,\ a_j\ )$ $(u_j \in IR(a_i,\ a_j\ ))$, if there exists a simple path from a_i to a_j containing u_j and not containing any node concurrent to a.*

The IR, besides the switchover correctness verification, is also used to approximate the function covers of the specification outputs, realised through Marked Regions (MR).

Definition 2.3.16. *Given a place p in an STG, its marked region is the set of markings in which p has at least one token and it is denoted $MR(p)$, i.e. $MR(p) = \{\ M \in M[M_0\rangle : M(p) > 0\}$.*

The cover cube for a marked region $MR(p)$ is the smallest cube, i.e. with the greatest number of literals, which covers $MR(p)$, and is defined to be the cube C_p such that for every literal corresponding to a signal b:

1. non-concurrent to p then: $C_p = b$ if $b = 1$ in $MR(p)$, $C_p = b'$ if $b = 0$ in $MR(p)$.

2. concurrent to p then: Cp $= -$.

The Interleave Relation gives a polynomial-time algorithm (for FC STGs) to determine the value of each literal C_p^a in the cover cube of place p. $C_p^a =$

- 1 if $\exists$ adjacent $(a_i+,\ a_i-)$: $p \in IR(a_i+, a_i-)$

- 0 if $\exists$ adjacent $(a_i-,\ a_i+)$: $p \in IR(a_i-, a_i+)$

- $-$ if $(a,\ p \in SCR)$.

	Places		**Transitions**		**Signals**	
	p7	**p11**	**Zr+**	**Za-**	**Zr**	**Za**
p0	cr	cr	cr	cr	cr	cr
p1			cr			
Lr-	cr	cr	cr	cr	cr	cr
Dr+			cr			
Lr					-	-
Dr			cr		-	-

Table 2.3: Concurrency Relations of $adfast$'s components

	Adjacent Transitions		
	Lr-,Lr+	**Dr-,Dr+**	**Zr+,Zr-**
p3	ir		
p1		ir	
Lr+		ir	
Za+			ir

Table 2.4: IR Relations of $adfast$'s components

Places	**Signals**					
	Inputs			Outputs		
	La	Da	Za	Lr	Dr	Zr
p0	1	-	-	0	-	-
p1	-	0	-	-	0	-
p2	-	-	0	-	-	0
p12	1	0	-	0	0	-
p13	-	-	0	0	-	0
p14	0	0	0	1	0	0

Table 2.5: Cover cubes of $adfast$'s places

Tables 2.3, 2.4 and 2.5 present the structural properties of $adfast$'s components with respect to concurrency relations, interleave relations and cover cubes respectively. The cr symbol in Table 2.3 indicates that of the corresponding row is in the

CR with the element in the corresponding column *e.g.* $(p_0,\, p_7) \in CR$. Accordingly, the *ir* symbol in Table 2.4 indicates that the place or transition in the corresponding row is interleaved between the two transitions in the transition pair of the corresponding column *e.g.* $Lr+ \in \mathrm{IR}(Dr-,\, Dr+)$. In Table 2.5 the first column of each row includes a place and the rest columns show the place's cover cube with respect to input and output signals. For instance, when p_4 is marked, all the signals have value 0 except the Lr output which has value 1. A '-' in array cells denotes a dont care, *i.e.* the corresponding signal may either be 0 or 1.

2.3.4 Synthesis

FCPTnets are a popular formalism for asynchronous circuit specifications [17], as they can model concurrency between asynchronous signals. An FCPTnet (or its STG representation), are implementable, provided that it is well-formed and consistent, *i.e.* signal transitions alternate rising to falling and vice versa. Multiple asynchronous control circuit synthesis methodologies from PTnets exist, and these may be categorised into state-based, structural and direct-mapped.

The state-based flow [26] requires the generation of the global signal state space, known as the State Graph (SG) which is worst-case exponential in state size. The SG must have the Complete State Coding (CSC) property [26], *i.e.* each SG state must either possess a unique code or, if multiple states share the same code, they should be distinguishable by the circuit's PIs. If the CSC property is not satisfied, internal signals, *i.e.* additional state, are added in order to differentiate the ambiguous states [66]. However, current CSC satisfaction algorithms are heuristic and do not guarantee the solution of the CSC problem. A significant additional drawback is that the FCPTnet model is not confluent to its resultant circuit implementation, *i.e.* even small changes to the FCPTnet can produce unpredictable results with respect to the produced circuit. Last, but not least, the state based flow is unable to explore area, delay or power tradeoffs. Petrify [25] is currently the most popular asynchronous control circuit synthesis tool, based on the state-based PTnet implementation flow. In [102] a decomposition based flow is presented which reduces the synthesis execution time by decomposing the initial PTnet and using the Petrify tool as a backend tool to synthesize each PTnet component. CSC conflicts introduced during the above decomposition are solved if they fall in one of the supported conflict structures, *e.g.* structural self-triggers.

The structural approach represents an effort to tackle the exponential complexity of the SG, whereby the SG generation is replaced by structural analysis of the FCPTnet specification. However, for concurrent sections of the FCPTnet, it has been shown that the structural analysis implies worst-case exponential complexity, while analyzing signal orders by S-nets [82]. Direct-mapping represents an alternative view to FCPTnet implementation, whereby logic synthesis is completely bypassed, and the FCPTnet is directly translated into a circuit by splitting it into a circuit specification and its environment model and represents Places at the circuit level using one-hot encoding [94].

Direct-mapping is polynomial in complexity, albeit generates circuits where the number of sequential elements are linear with respect to the number of Places, and is also unable to explore area, delay or power tradeoffs.

In VLSI Programming flows, higher level programming languages like Balsa[41] and Tangram/Haste[60] are translated into predefined circuit structures. Implementations stemming from such flows are less efficient than the ones obtained from synthesis procedures although there have been efforts in using synthesis flows as a backend engine after program translation [18].

2.3.5 Advantages and Disadvantages

PTnets represent an graph-based, event-oriented alternative control model, in contrast to the state-oriented FSM models. PTnets exhibit certain specific benefits, including the combination of concurrency and choice within the same model, as well as, more importantly, static analysis algorithms for verifying important control system properties, such as deadlock-freedom and boundedness. The PTnet model, being event-based, is typically considered asynchronous in nature. However, with the introduction of assumptions similar to those of synchronous FSMs, *i.e.* when input and output events are fired with respect to a clock signal, the PTnet model may also be used to specify or implement synchronous control systems [12]. The sole,yet significant, drawback of PTnets is that they do not currently possess a low-complexity path to implementation, similar to that of state-based models.

Hence, whether under synchronous or asynchronous timing, the monolithic FSM model possesses a low-complexity implementation path, whereas PTnets do not. Their intermediate counterpart, interacting FSMs, lack the formal expressiveness to express synchronization, this is why they are used the least. A number of prior works [65, 26] and [12] have attempted to transform PTnets into a monolithic or multiple FSMs respectively, however they either suffer from state explosion or cannot guarantee the implementability of any live and bounded PTnet.

2.4 FSM-PTNet Intermediate Control Models

Besides PTNets and FSMs, there are models which have been introduced and lie between them with respect to expressibility and implementability. Interacting FSMs, state machines with forks and joins and extended burst mode FSMs are three such examples.

2.4.1 Interacting FSMs

Synchronous control circuits are modeled by FSMs or networks of FSMs in the same manner as combinational logic is modeled by two-level or multilevel logic [49]. As with multilevel synthesis, FSM-network synthesis enables handling large FSM specifications which were unmanageable by single FSM minimization algorithms.

Networks of FSMs are extracted from High Level Specifications, such as Programming Languages or Behavioral HDLs. In Behavioral HDL specifications, the set of FSMs declared, along with their dependencies extracted from their common signals form a Network. At the high level synthesis context, structures like loops, and functions are divided to control and data-path. Except the FSM minimization algorithms analyzed in the previous section, compiler-level optimization algorithms, like loop-unrolling or inline function expansion can be applied at the FSM context [103].

Networks of FSMs, being a formalism close to the Boolean level, provide better area, power and delay estimations, compared to other intermediate representations of control circuits like the control-data flow graphs. Moreover, as they stem from Boolean Algebra and Automata theory, notions like composition, decomposition or minimality are clearly defined. However, not all these operations can be applied at an acceptable execution time. For example, collapsing a network of FSMs, using FSM compositions reveals the total state space of the control system. Therefore its exact minimization would produce the needed state space to be covered by the next steps. Nevertheless, generalized FSM composition, performed using the cartesian product, results to huge FSMs with many redundant states. Exact minimization of such FSM structures is clearly defined but its complexity is unmanageable. Partial composition, where a portion of the connected FSMs are composed is preferred at this level of abstraction. An example is the loop unrolling optimization of a while structure iterating until an external counter reaches an upper bound. Initially a sequencer FSM forming the body of the loop and a counter checking FSM are connected. These two machines can be composed to a single FSM, where the total state space of the loop only is unveiled and an exact or heuristic optimization step could result to a better solution.

FSM decomposition of Incompletely specified FSMs is still an open issue. Parallel decomposition is clearly defined, but only completely specified FSMs can be decomposed this way. Cascaded FSM decomposition can be performed for an FSM, but one of the two decomposed components has to be independent and extracted before the decomposition [39]. The independent component can be thought of like the divisor of the Boolean division. There are no algorithms which indicate good algebraic divisors for an FSM decomposition. However, there are methods [75, 11, 22] which partition the original FSM structurally, having as metric the power dissipation of each component.

2.4.2 State Machines with Forks-Joins

Monolithic FSMs exhibit parallelism between input or between output signals. State Machines with Forks and Joins (SFJs) [64] are an extension of the monolithic FSM model which allows parallelism between input and output signals. Communication among parallel signals is achieved through fork and join transitions.

Definition 2.4.1. *An SFJ graph is a triplet (P, T, F) where P is a set of places, T is a set of transitions, and F is a flow relation on $P \cup T$, such that $F \cap (P \times P) = F \cap (T \times T) = \emptyset$.*

T consists of fork transitions (T_f), join transitions (T_j), and sequential transitions (T_s). A fork transition has in-degree equal to one and an arbitrary out-degree greater than one, a join transition has in-degree greater than one and out-degree equal to one and a sequential transition has a single place at its input side and at its output side respectively. Additionally, each transition $t_f \in T_f$ is paired with a transition $t_j \in T_j$, such that the in-degree of t_j matches the out-degree of t_f. A pair of a places (p_{f_i}, p_{j_i}), such that the former follows a fork transition and the latter precedes the matching join transition of a fork, define a single threaded subgraph (STS) or the i-th thread defined by this pair of fork and join transitions.

An STS is a triplet (P_{st}, T_{st}, F_{st}) which represents a subgraph in the SFJ in which the F_{st} is restricted to places and transitions of the STS. An STS is disjoint from the other STSs and it may only contain choice places. Therefore nested threads are not allowed by the model. Each STS has a single entry place and a single exit place as well, called input and output places respectively.

SFJs allow thread level parallelism in which a place and a transition are concurrent with places and transitions of other threads with the same join and fork transitions. This concurrency model is significantly limited, when compared with even the simplest classes of PTNets, *e.g.* MGs. Nevertheless, it is the dominant parallel programming model for software design [5].

2.4.3 Extended Burst-Mode FSMs

Extended Burst-Mode FSMs (XBM-FSMs) [105, 104] is a formalism which enriches the BM FSM model with a limited form of input/output concurrency and constrained handling of glitching signals.

BM-FSMs allow only input/input concurrency and output/output concurrency. During an input burst, the signal events which form the burst may fire at any order. The same holds for the BM-FSMs outputs, as the FSM should operate correctly independently from the order of output signal events during an output burst. Input/Output concurrency is not allowed by BM-FSM specifications. However, XBM specifications do allow for limited input/output concurrency, whereby an input signal si may be declared as a Directed Dont Care. This means that it is allowed to change its value during a set of consecutive transitions, rather than only during a single transition. Such as signal is concurrent not only with input bursts signals but also with output bursts. This feature comes with a set of constraints that should be satisfied, the most significant being that every such dont care must be paired with a non empty set of signal events, and that a set of consecutive directed dont cares for a signal must be terminated with a compulsory signal event for that signal.

BM-FSMs are able to express choice. The only constraint is distinguishability, i.e. ambiguity between input bursts emanating from the same state is not allowed. XBM machines extend the expresiveness of BM machines by letting glitching inputs to be present in specification. Glitching inputs must be declared separately from the input signals in the conditional signal set. In XBM specifications, if an input signal is not present at an input burst, then it is not allowed to fire. Conditional signals are able to change value if they are not present in an input burst. However, if they are present in an input burst along with a non-empty set of signal events, they should have been stabilized as soon as any of the input events changes value, and should remain stable until after all of the input signal events have fired. The minimum time for which a conditional signal must have been stabilized, before any input signal event changes value is the setup time for that conditional. Moreover, the time that the conditional signal keeps its value stable after the firing of all the input signal events, is the hold time for that conditional. This means that the input signals which pair the conditionals have the same role as the clock edges in the synchronous counterparts, i.e. whenever the clock edge arrives the sampled signals must have stabilized a setup time before the edge and they must remain stable a hold time after the clock edge.

2.4.4 Concurrent Hierarchical FSMs

In a Hierarchical FSM (HFSM) there are two types of states, simple and composite. A simple state is the same as an FSM state, whereas a composite state is comprised of a set of states. The above defines a hierarchy between states, which can be modeled by a tree.

Statecharts [50] is a formalism for visualizing HFSMs. Besides the typical FSM structures, Statecharts allow state concurrency by introducing and AND operator between two composite states which means that states belonging to the two composite states should be concurrently activated. The above is the dual of the OR state composition found in FSMs as well, which allows only of the states to be be active during operation. Synchronization between the above concurrent composite states is achieved by exchanging information between them about the underlying active states. In [40] a hardware synthesis algorithm was introduced which decomposed the Statechart to non-trivial and trivial FSMs implementing the FSM hierarchy and the system functionality respectively.

Parallel HFSMs (PHFSMs) [92] exhibit concurrency at a coarser level when compared to the AND composition of Statecharts. The hierarchy in PHFSMs is modeled with a module stack whereas the underlying FSMs are modeled with an FSM stack. Modules realize the hierarchy in the same manner as the non trivial FSMs in Statecharts. The parallelism is realized with multiple FSM and module stacks paired with a special block comprised of semaphores. The semaphores implement the inter-HFSM synchronization. A synthesis flow is introduced in [92] which directly maps modules, FSMs and semaphores to predefined synthesizable HDL structures.

This chapter introduced the monolithic FSM which is the most popular model for specifying sequential control and PTNets which represent one of the most widespread formalisms for modeling concurrent systems. FSMs and PTNets suffer from the state explosion problem when specifying and when synthesizing a control system respectively. Hence, numerous formalisms are introduced in the literature to alleviate the drawbacks of both models. IFSMs, SFJs, XBMFSMs and Statecharts are three such formalisms. IFSMs lack the formal rigor for expressing synchronisation, as well as the analysis capabilities of PTnets. XBMFSMs, SFJs and Statecharts on the other side allow a limited form of parallelism when compared to PTNets. The next Chapter introduces a novel formalism for describing concurrent systems, which not only is as expressive as PTNets bus also enjoys a viable synthesis path through transformation to FSMs.

Chapter 3

The MSFSM Control Model

The predominant control models for describing concurrent systems are monolithic FSMs, interacting FSMs and PTNets. In Chapter 1, the advantages and disadvantages of the above models were analyzed. This Chapter presents MSFSMs as a new way for designing concurrent specifications and argues on its efficiency when compared to the above models.

The multiple Synchronized FSM (MSFSM) model is a compact representation of a set of Interacting FSMs, which explicitly models inter-FSM synchronization, using a set of two basic synchronization primitives, Wait States and Transition Barriers. In this chapter, we introduce the MSFSMs model and analyze its properties.

3.1 Definitions

Definition 3.1.1 (MSFSM Set). *An MSFSM set, MS, is a five-tuple $(I, O, S, \Delta, \Lambda)$, where I is a finite, nonempty set of global inputs, O is a finite, nonempty set of global outputs, S is a finite nonempty set of N FSMs, with state sets S_i and corresponding local output sets, $\lambda_i : I \times S_i \to O_i$ (if S_i is a Mealy machine), or $\lambda_i : S_i \to O_i$ (if S_i is a Moore machine), Δ is a set of next state functions, one per FSM i, where*

$\Delta_i : I \times O_1 \times O_2 \times \ldots \times S_i \times \ldots \times O_N \rightarrow S_i$, *and* $\Lambda : I \times O_1 \times O_2 \times \ldots \times O_N \rightarrow O$ *is the global output generation function.*

The MSFSM is a set of shared input support interacting FSMs, where each FSM changes state, or produces its local outputs (in the usual Mealy or Moore fashion), based on both the global inputs, I, and the state-dependent outputs of other FSMs of the set. Local FSM outputs can then be combined, combinationally, to generate global outputs. The simplest case of a local output, O_i, is when it is directly dependent upon a local state, *i.e.* $O_i = S_i$, thus each FSM's state change or local output generation may directly depend to the state of other FSMs. The initial state of the MSFSM set corresponds to the set of initial states of its member machines.

An MSFSM set may be represented as a set of flow tables (or state transition graphs), one per FSM in the set, and a set of combinational logic equations. Each flow table, i, represents the next state function, Δ_i, and the local outputs, O_i of FSM i, whereas the set of combinational logic equations generate the global outputs, *i.e.* global output function Λ. An MSFSM set corresponds to a potentially implementable specification, as each FSM of the set is indeed implementable, using existing monolithic FSM implementation techniques.

3.2 Synchronization Primitives

The fact that Δ_i functions of FSMs, implicitly include the current state of other FSMs in the set, allows for the inter-FSM communication, and more importantly, synchronization to be exposed. We define two synchronization primitives, Wait States and Transition Barriers, which stem from the Δ_i functions interaction, and prove that these are sufficient when only conjunctive (AND) synchronization is allowed between Synchronized FSMs.

Definition 3.2.1 (MSFSM Wait State)**.** *In an MSFSM set, MS, a Wait State W is a state of a machine M, which belongs to MS, where the next state function for state W, $\Delta_i(W)$, depends on a combinational function f of the global inputs I, and on a product of local outputs of a set j of the FSMs of MS, i.e. is of the form: $\Delta_i(W) = f(I) . \prod_{j \in N} O_j$*

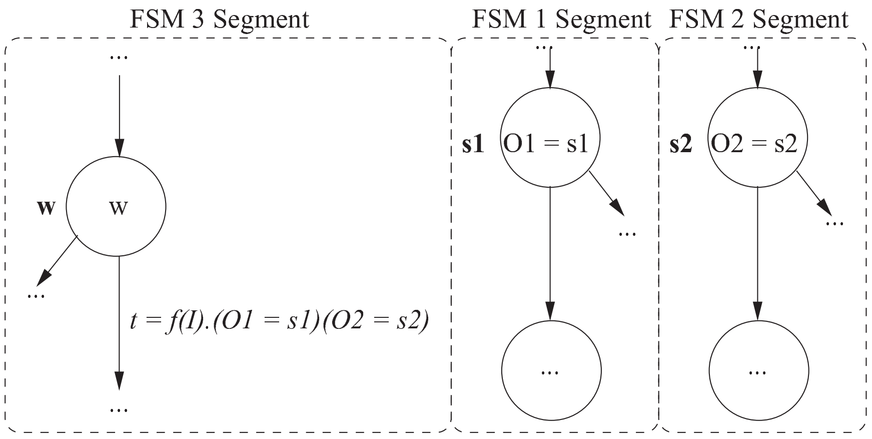


Figure 3.1: Wait State Example

The next state logic of a Wait state W of M, will be activated, not only when the corresponding inputs function, $f(I)$ is activated, but also when a set of local FSM outputs, O_j, are activated as well. These, in turn, depend on local states of their corresponding FSMs, *i.e.* a conjunction of outputs. Thus, transitively, a Wait state W awaits for a set of FSM states to be reached and potentially a set of global inputs as well. This form of synchronization is unidirectional and represents conjunctive (AND) causality. A Wait State synchronization may be represented by a tuple (W, t, $s1$, $s2$, …, sm), where W, t and $s1$ to sm correspond to state W, the relevant transition function, $f(I)$, and the relevant FSM states of MS generating O_j respectively. Figure 3.1 illustrates a simple wait state dependency, where transition t of a state w of FSM 3, is activated by states $s1$ and $s2$, of FSMs 1 and 2 respectively.

Another synchronization primitive can be defined by considering the special case of two, or more, mutually dependent Wait States between two or more FSMs of MS.

For the two state case, a Transition Barrier (named after barrier synchronization) corresponds to two Wait States mutually dependent upon each other, and is generalizable to any number of Wait States.

Definition 3.2.2 (MSFSM Transition Barrier)**.** *In an MSFSM set, MS, a Transition Barrier T, is a set of Wait State transitions of different FSMs of MS, with identical combinational function $f(I)$, and an equivalent output product in the respective Δ_i's, i.e. each transition of the synchronization barrier T and corresponding wait state local output product, $\prod_{j \in N} O_j$, includes all other wait states of T.*

Thus, the transitions of a Barrier may only be activated simultaneously, when all of the relevant Wait States, of the respective FSMs are reached. As each Wait State represents conjunctive (AND) causality, the barrier also represents the same.

A Transition Barrier may be represented by an unordered transition set, *i.e.* {*t*1, *t*2, . . ., *tm*}, where *t*1 to *tm* correspond to the set of Wait State transitions of the barrier. Figure 3.2 illustrates the simplest form of a Transition Barrier, {*t*1, *t*2}, whereby transitions *t*1 and *t*2, which belong to FSMs 1 and 2 respectively, possess the same combinational function g and are mutually dependent, as *t*1 is activated by $O2$, which in turn is set by $s2$ and $t2$ by $O1$, which is set by $s1$. Thus, one awaits for the other, and they may only be activated simultaneously.

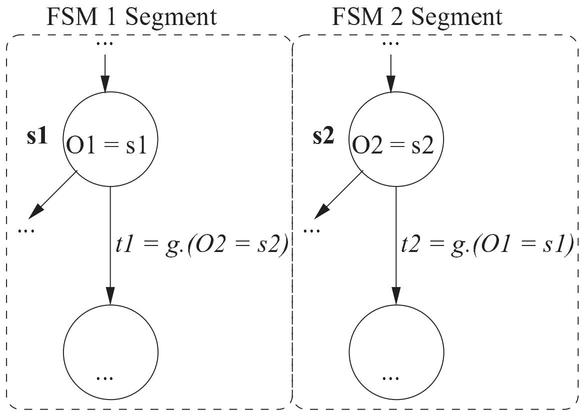


Figure 3.2: Transition Barrier Example

3.3 Manually Describing Concurrent Systems with MSFSMs

This Section presents MSFSMs as a new formalism for manually describing concurrent control systems. A handshake controller is used throughout this section as a means of demonstrating MSFSMs effectiveness for describing concurrency and synchronization.

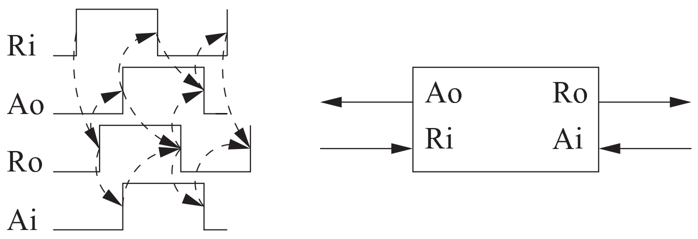


Figure 3.3: Half Handshake Controller Signal Transitions Dependencies

Figure 3.3 shows the timing diagram of the aforementioned handshake controller. Signals Ri and Ai are controlled by the environment, whereas the controller drives signals Ao and Ro. Signal pairs (Ao, Ri) and (Ro, Ai) will from now on be denoted as the left and right handshake respectively. With respect to controller's operation, initially all signals are low. The environment initiates the left handshake by rising Ri. Then the controller replies by rising Ro, initiating this way the right handshake. Afterwards, the controller rises Ao (acknowledging the rising of Ri at the left handshake) and concurrently the environment rises Ai (acknowledging the rising of Ro at the right handshake). Following the timing diagram, it can be deduced that the rising of Ao causes the concurrent fall of signals Ri and Ro, the falling of Ro causes the concurrent fall of Ao and Ai, etc.

Figure 3.4 and Table 3.1 present the FSM set and the Wait states set of an arbitrary specification of the handshake controller respectively, using the MSFSMs control model. Each interface signal is controlled by a single FSM, although different FSM schemes with arbitrary number of signals per FSM could have been chosen. All states participate in wait state synchronization primitives as shown in Table 3.1. The

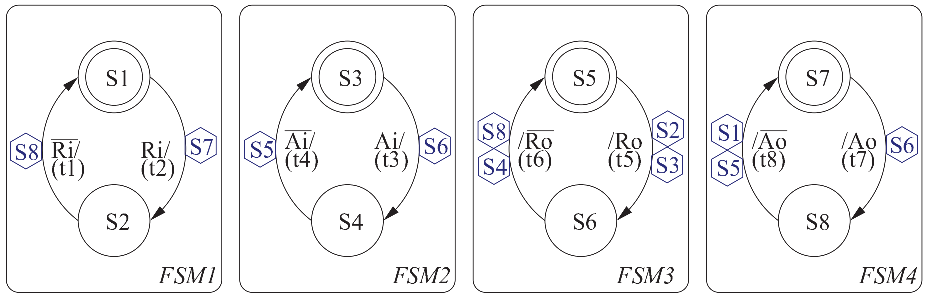


Figure 3.4: Half Handshake Controller FSMs

W	Trans.	f(I)	Dependent States
S1	t2	Ri	S7
S2	t1	$\overline{Ri}$	S8
S3	t3	Ai	S6
S4	t4	$\overline{Ai}$	S5
S5	t5	ϵ	S2, S3
S6	t6	ϵ	S4, S8
S7	t7	ϵ	S6
S8	t8	ϵ	S1, S5

Table 3.1: Half Handshake Controller Wait States

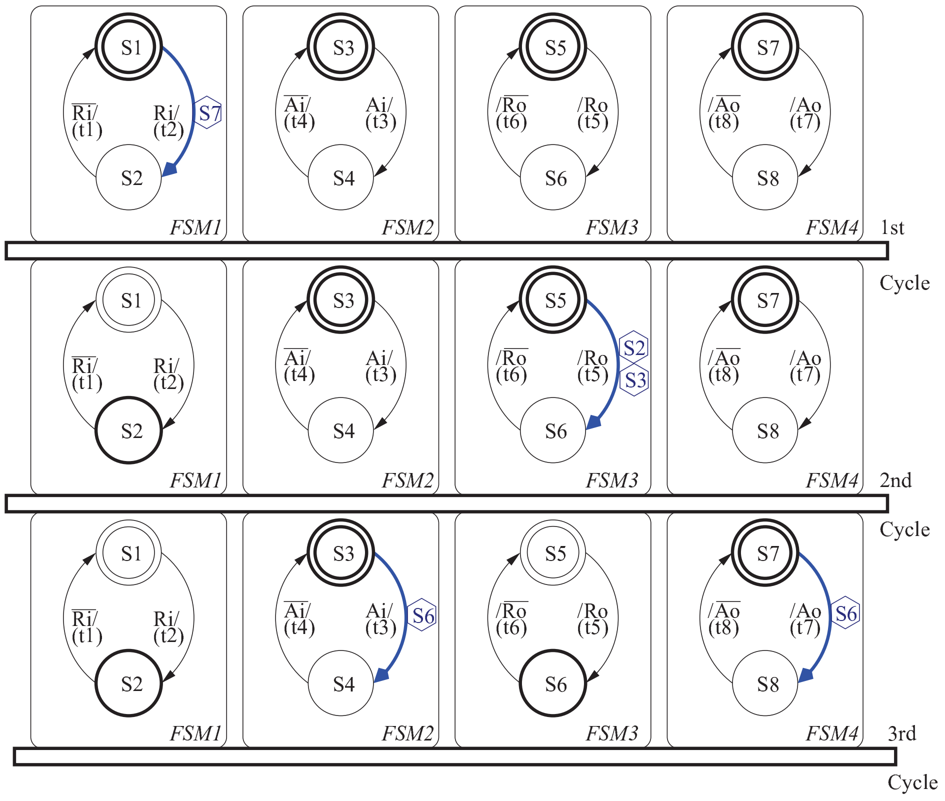


Figure 3.5: Simulation of the MSFSMs system for 3 clock cycles

activation of the system's transitions depends on more than one FSMs, each. For instance, the activation of transition t2, apart from the activation of its source state, S1, also requires the activation of S7. The activation of S7 state also shows that the output signal Ao is low.

Figure 3.5 illustrates the simulation of the MSFSMs, given that the underlying implementation is synchronous *i.e.* state changes happen on clock edges. Initially, only transition t2 is enabled as the wait state it depends on, S1, is activated and its transition function Ri is satisfied. The other transitions following active states, *e.g.* t3 are not enabled. This is because either the transition function, denoted as $f(I)$ in Table 3.1 is not satisfied *e.g.* Ai, or the dependent states of the corresponding state are not all active *e.g.* S5. After the first cycle, both states S2 and S3 are active and thus the Wait state S5 is enabled (ϵ transition function are always satisfied), causing the firing of transition t5. During the third cycle, transition t7 unconditionally fires, and in the case that the environment has asserted Ai, t3 is triggered as well. Otherwise, t3 fires in an arbitrary number of cycles, depending on environment's availability on replying to Ro assertion.

Another option for describing the concurrent controller would have been the monolithic FSM control model. However, besides the exponential number of states required to describe concurrency, monolithic FSMs come with a latency penalty when describing concurrency between input and output signals. The above is due to the interleaving mechanism which is used to express concurrency with a single FSM. Transition interleaving, being an implicit mechanism, cannot describe systems where a set of concurrent transitions are triggered in the same cycle. The above is shown in Figure 3.6 which illustrates a possible monolithic FSM description of the half handshake controller. In particular, moving from state S0 to state S8 takes 2 clock cycles, although input signal Ai and output signal Ao can be both asserted during the same clock cycle.

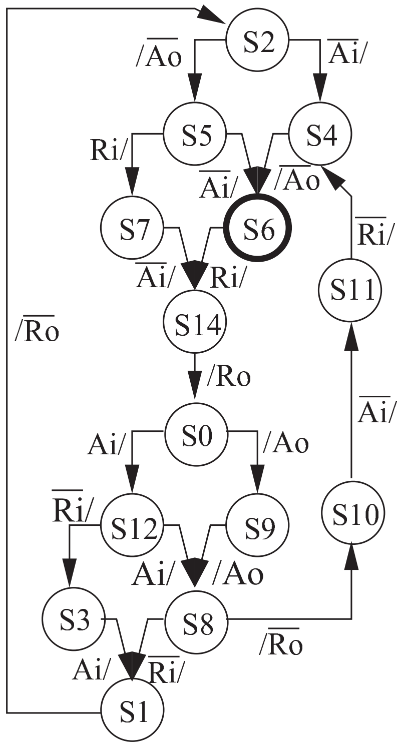


Figure 3.6: Monolithic FSM Specification of the half handshake controller

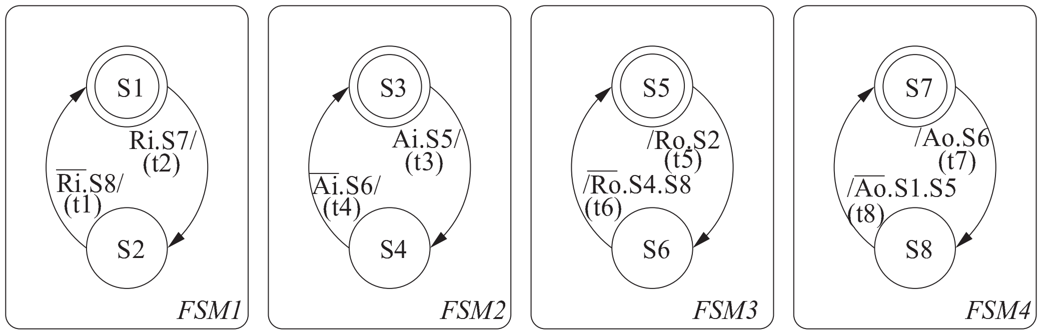

S1
Ri.S7/
(t2)
$\overline{\text{Ri}}$.S8/
(t1)
S2
FSM1
S3
Ai.S5/
(t3)
$\overline{\text{Ai}}$.S6/
(t4)
S4
FSM2
S5
/Ro.S2
(t5)
/$\overline{\text{Ro}}$.S4.S8
(t6)
S6
FSM3
S7
/Ao.S6
(t7)
/$\overline{\text{Ao}}$.S1.S5
(t8)
S8
FSM4

Figure 3.7: Interacting FSMs Specification of the Half Handshake Controller

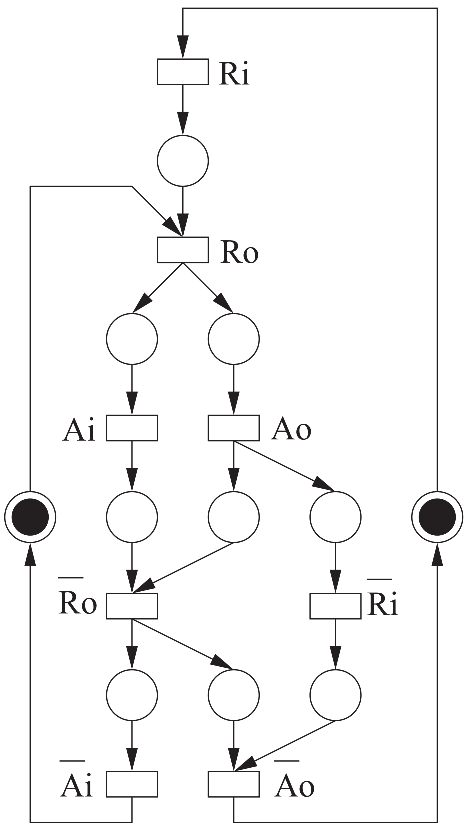
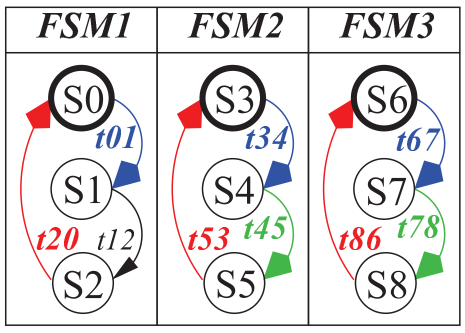


Figure 3.8: Petrinet Specification of the Half Handshake Controller

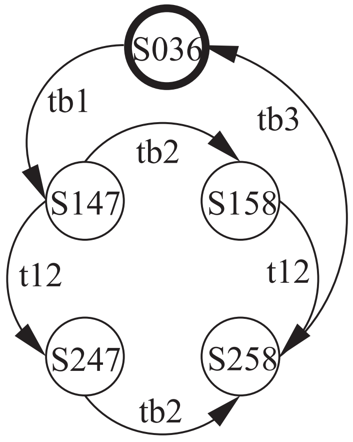
Half handshake controller interacting FSM specification, shown in Figure 3.7, expresses concurrency in the same way as the MSFSMs model does. However, the inter-FSM synchronization is encoded in the transition functions and as explained in Chapter 1, it is not always trivial to statically and efficiently identify its form. Hence, verifying high level system properties *e.g.* deadlock freedom is intractable for the interacting FSM model [86]. MSFSMs on the other hand, allow such a restricted inter-FSM communication mechanism, which enables transforming the system to higher level abstractions *e.g.* PTNets for which efficient verification algorithms exist.

Finally, the description of the half handshake controller with the PTNet control model is shown in Figure 3.8. Multiple places are concurrently active during operation with fork and join transitions synchronizing the parallel corresponding flows. As analyzed in the two previous Chapters, PTNets model concurrent systems compactly and enjoy a rich background of efficient algorithms analyzing high level system properties [20], *e.g.* concurrency relations. However, manually designing PTNets is not a trivial task, as designer's changes at this level has unpredictable impact to system properties and implementation. MSFSMs on the other hand, can be not only transformed to equivalent PTNets and thus enjoy the same higher level algorithms but also benefit from their FSM nature, which allows predictable exploration of the implementation state space.

3.4 MSFSMs Product



(a) Initial MSFSM



(b) MSFSM Product

Figure 3.9: MSFSM Product

Transition Barrier	Dependent Transitions
tb1	t01, t34, t67
tb2	t45, t78
tb3	t20, t53, t86

Table 3.2: Transition Barriers of System in Figure 3.9a

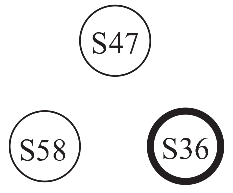
The MSFSM product is an operation which generates a monolithic FSM with the same behavior as a set of synchronized FSMs in the MSFSM. The resultant FSM has exponential number of states and transitions.

Its conventional counterpart, FSM product, is usually generated during system's verification to enumerate all the reachable states of the system. In the example shown in Figure 3.9a the product FSM would have 27 states unless a subset of the transition functions were equivalent in which case the total number would be less.

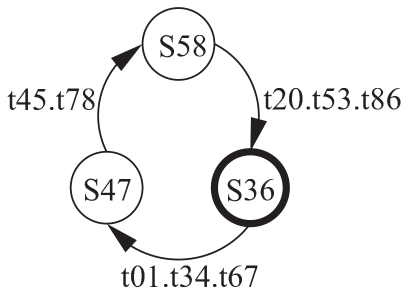
In the MSFSM context, besides verification, the product operation leverages the explicit synchronization to also reduce the number of generated states. As an example, substituting the MSFSM product of the system in Figure 3.9a, with its MSFSM product shown in Figure 3.9b, reduces the total number of states from nine to five. The MSFSM product has less states than the conventional FSM product due to state combinations which cannot occur due to synchronization. For example state S_{037} which represents the combination of states S_0, S_3 and S_7 cannot occur, as the transition barrier tb_1 forces the MSFSM to move from state S_{036} (FSM1 at S_0, FSM2 at S_3 and FSM3 at S_6) directly to state S_{147} (FSM1 at S_1, FSM2 at S_4 and FSM3 at S_7).

```
Algorithm 1 - MSFSM_product
Require: Two MSFSMs, $FSM_i$ and $FSM_j$
Ensure: A product MSFSM
1: Disconnect $FSM_i$ and $FSM_j$ at their transition barriers $tb_{ij}$
2: for all $tb \in tb_{ij}$ do
3:     Generate the FSM products starting from $tb$'s next states
4: Connect the FSM products at $tb_{ij}$'s, previous and next state pairs
```

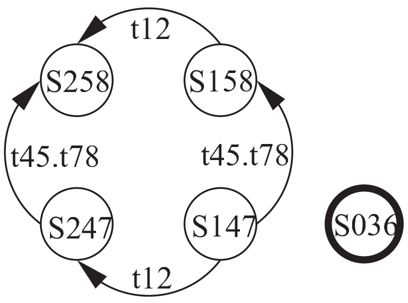
Algorithm 1 describes the procedure of generating an MSFSM product of two FSMs in an MSFSM. Initially, the two FSMs are disconnected at their common transition barriers, which means that the arcs representing these transitions are removed. Then, for each such barrier, the conventional FSM composition is applied starting from the states which follow it. The set of FSMs created, is connected at the states which precede and succeed the aforementioned barriers. The MSFSM product operation on a set of FSMs which are not synchronized reduces to the conventional FSM product operation.



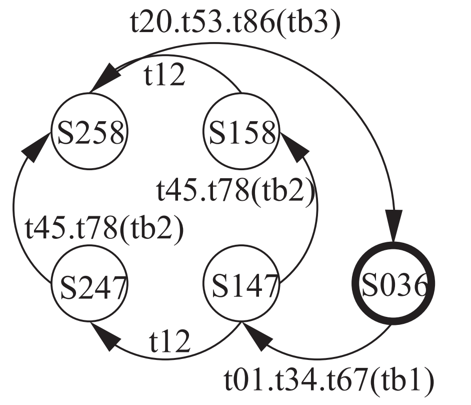
(a) FSM_2 and FSM_3 Products



(b) FSM_2 and FSM_3 Product MS-FSM, FSM_{23}



(c) FSM_{23} and FSM_1 Products



(d) FSM_{23} and FSM_1 Product MS-FSM, FSM_{123}

Figure 3.10: MSFSM Product Algorithm Steps

Applying Algorithm 1 on FSMs FSM_2 and FSM_3 of Figure 3.9a results to the following steps:

- The common transition barriers of the two FSMs are extracted, which are tb_1 (FSM2 and FSM3 contribute with t_{34} and t_{67} respectively), tb_2 (FSM2 and FSM3 contribute with t_{45} and t_{78} respectively) and tb_3 (FSM2 and FSM3 contribute with t_{53} and t_{86} respectively).

- Each FSM is disconnected at the arcs which represent transitions in the common transition barriers. For instance, FSM2 is disconnected at the arcs which represent transitions t_{34}, t_{45} and t_{53} as they all contribute to at least one common transition barrier.

- The conventional FSM product algorithm is applied on the states which follow the common transition barriers. Thus, state S_4 is multiplied with state S_7 as they follow transitions $t_3 4$ and $t_6 7$ respectively. The three resultant FSM products are shown in Figure 3.10a.

- The FSM products derived in the previous step are interconnected at the states which preceded and succeeded common transition barriers. For instance state S_{47} is interconnected with state S_{58} as transition barrier tb_2 succeeded states S_4 and S_7 and preceded states S_5 and S_8. The resultant MSFSM product of FSMs FSM2 and FSM3 is shown in Figure 3.10b.

Re-applying the same algorithm on synchronized FSMs FSM_1 and FSM_{23} yields two intermediate MSFSM products shown in Figure 3.10c which in turn if connected at their common synchronization points produce the MSFSM product of the MSFSM system, shown in Figure 3.10d.

3.4.1 Transition Barrier Transformation

The MSFSM product operation can transform the transition barriers of the MSFSM system.

Lets assume that in an MSFSM we have an FSM pair FSM_i and FSM_j which is substituted by its MSFSM product FSM_{ij}. Then the following two rules are applied to the MSFSM for each transition barrier $tb = \{t_1, t_2, ..., t_i, ..., t_j, ..., t_n\}$ in which t_i and t_j belong to FSMs FSM_i and FSM_j respectively:

- If $tb = \{t_i, t_j\}$ then it is removed from the MSFSM.

- If $tb \neq \{ t_i, t_j \}$ then a new transition barrier is inserted for each instance of the t_i and t_j transitions in the MSFSM product.

In the example shown in Figure 3.9a, generating the MSFSM product of FSMs FSM_2 and FSM_3 and using it to substitute both FSMs, results to removing transition barrier tb_2 as it only synchronizes these two FSMs. The single transition substituting the transition barrier has its input and output functions formed by the complement of the corresponding functions of the two transitions contributing to the transition barrier.

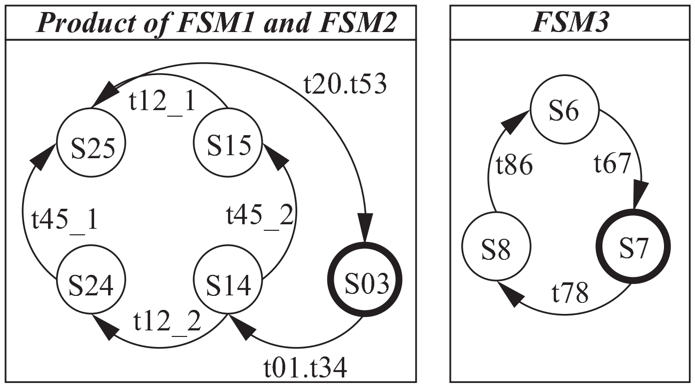


Figure 3.11: MSFSM after substituting FSM_1 and FSM_2 by their product

Transition Barrier	Dependent Transitions
tb1	t01.t34, t67
tb2	t45, t78
tb3	t20.t53, t86

Table 3.3: Transition Barriers of System in Figure 3.11

If the MSFSM product of FSMs FSM_1 and FSM_2 was formed first and it was used to substitute them in the MSFSM system, then the result would be the one shown in Figure 3.11. None of the transition barriers is removed, as they also synchronize FSMs other than just FSM_1 and FSM_2. Transition barrier tb_2 initially synchronized t_{45} and t_{78}. As t_{45} is instantiated twice during the FSM composition, two instances of tb_2 are generated, tb_{2_1} and tb_{2_2}.

Using the transition barriers formulation as a Boolean function, the MSFSM product operation translates to substituting a transition with a set of instances generated during the FSM composition procedure. Initially the transition barriers Boolean function was $t_{01}.t_{34}.t_{67} + t_{45}.t_{78} + t_{20}.t_{53}.t_{86}$ and after the substitution of FSM_1 and FSM_2

by their product FSM_{12} the Boolean function changed to $t_{01}.t_{34}.t_{67} + (t_{45\_1}+t_{45\_2}).t_{78} + t_{20}.t_{53}.t_{86} = t_{01}.t_{34}.t_{67} + t_{45\_1}.t_{78} + t_{45\_2}.t_{78} + t_{20}.t_{53}.t_{86}$.

This Chapter introduced MSFSMs, a novel control model for describing concurrent systems. MSFSMs is a compact interacting FSMs model, capable of exposing state and expressing inter-FSM synchronization, while being potentially implementable with any existing monolithic FSM implementation technique. MSFSMs can be used to efficiently describe concurrent specifications by decoupling the functionality of the concurrent interfaces from their synchronization. The following Chapters introduces MSFSMs as an intermediate control model for synthesizing FCPTNets and for verifying properties of systems described by interacting FSMs. In the former case, MSFSMs allow mapping FCPTNets to a set of synthesizable FSMs, whereas in the latter MSFSMs capture the inter-FSM communication and map it to a PTNet for which efficient verification algorithms exist.

Chapter 4

Petrinet to MSFSM Transformation

In this section, we introduce a polynomial complexity flow for transforming a FCPT-net into an MSFSM set. This flow bridges the PTnet and monolithic FSM models, tackles the state explosion problem associated with existing FCPTnet implementation approaches, as well as guarantees existence for any FCPTnet implementation.

Free-choice PTnets have been shown to be decomposable not to the selfsame model, but to a set of S-Components (or T-components). As illustrated in Section 2.3.1, an FCPTnet is decomposable into an S-Cover, where each S-Component is an FSM-like graph, *c.f.* Definitions 2.3.7, 2.3.8, Theorems 2.3.1 and 2.3.2. In fact, S-Coverability is achievable *in polynomial time* [56], and it has been shown that a non-exponential, practical algorithm may be used to derive a minimal S-Cover [68]. Hence, this path represents a very viable and practical path for the PTnet to MSFSM transformation. Our contribution in the transformation step lies in the conversion of S-Covers to "proper" FSMs, and MSFSMs, whereby (i) input transition relevant PTnet Places are eliminated, as they don't represent state in the FSM sense, and (ii) FSMs include all the necessary interaction signals for the purposes of synchronization, and are thus behaviorally equivalent, to the original PTnet. The latter is performed through extraction of the aforementioned synchronization primitives.

The polynomial time FCPTnet to MSFSM set transformation is comprised of the following five polynomial complexity steps, and respective complexities.

1. *FCPTnet S-Covering [35, 56, 68]*: $O(PT + P^2)$

2. *S-Component to Non-Interactive FSM mapping*: $O(P^2T^2)$

3. *FSM Collapsing*: $O(P^2T^4)$

4. *Synchronization Primitive Extraction*: $O(P^3T^2)$

5. *Inter-MSFSM Synchronization Integration*: $O(P^2T)$

In the following sections, these transformation steps are presented in detail, with the aid of a FCPTnet specification example borrowed from the field of asynchronous circuit design, $adfast$, an A/D converter controller [16]. In the relevant diagrams, solid black and white transitions are used represent output and input events respectively.

4.1 FCPTnet S-Covering

Extracting all possible S-Components of an FCPTnet is known to be exponential in complexity. However, prior work has shown that extracting a single S-Cover is achievable in $O(PT)$ time [56]. Further on, an extra step of $O(P^2)$ complexity can be used to extract a minimal (not the minimum) S-Cover, as shown in [68]. Hence, prior work has indeed established that covering an FCPTnet with a minimal S-Cover is achievable in polynomial time. It should be noted that the total number of places included in the S-Cover will typically be larger than that of the original FCPTnet. This is because the S-Covering process distributes and replicates transitions with multiple input or output places to different S-Components. Hence, some of the input and output places of such transitions are thus instantiated multiple times in the derived S-Components.

Provided that the FCPTnet is well-formed, *i.e.* live and bounded, *c.f.* 2.3.2, each S-Component includes an initially marked place. This initially marked place corresponds to the initial state of the subsequently derived FSM.

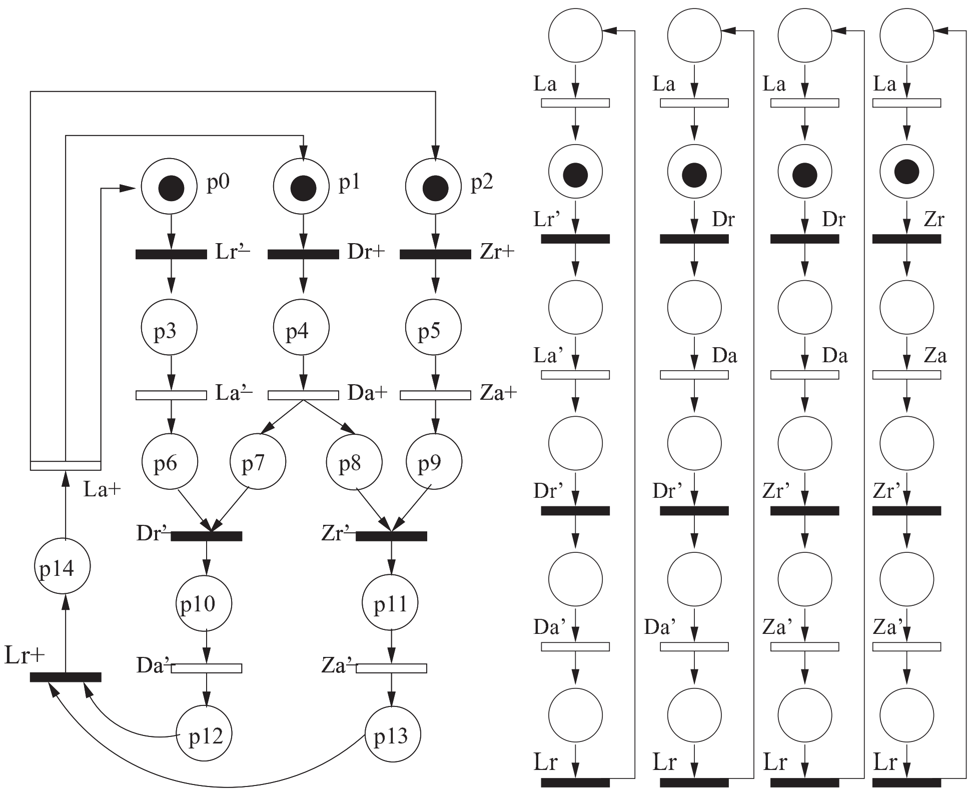


Figure 4.1: $adfast$ - FCPTnet S-Covering

Figure 4.1 illustrates the S-Covering process, with the LHS and RHS of Figure 4.1 illustrating the original FCPTnet specification, and a corresponding, minimal S-Cover of $adfast$ respectively (black transitions correspond to output, white to input signals). It is evident that while the original net contains a total of 15 Places and 12 Transitions, the S-Cover contains 24 Places and 24 Transitions, due to Place (state) and Transition replication (*e.g.* transition Lr).

4.2 S-Component to Non-Interactive FSM mapping

A one-to-one and onto mapping (bijection relation) exists between each S-Component of the FCPTnet and a Mealy FSM, which can be implemented as a sequence of three steps.

In the first step, each S-Component transition is mapped to a corresponding FSM transition, and accordingly each S-Component place is mapped to a corresponding FSM state. In the second step, the FSM's input and output sets, I and O, are extracted from the corresponding S-Component's T set, depending on whether T is labeled as an input or output.

Finally, in the third and last step, each FSM's δ and λ functions are constructed, based on the corresponding S-Component's flow relation F. The next state function, δ, at this point assumes its monolithic FSM form, *i.e.* $\delta : I \times S \rightarrow S$, as no explicit inter-FSM interaction is yet expressed. In the final step of the flow, Section 4.5, δ is promoted to Δ, *i.e.* includes inter-FSM synchronizations.

Hence, for each couple of an S-Component's flow relation pairs, *i.e.* (p, t), (t', p'), whereby $t = t'$, state transition (s, t_f, s') is extracted, for s, s', the respective states which correspond to the places of F. Such a state transition is appropriately added to the respective FSM's δ or λ functions, depending on whether t_f is an input or output. The corresponding complexities of the three steps are $O(T + P)$, $O(T)$ and $O(P^2T^2)$ respectively. Figure 4.3 illustrates the mapping of the first S-Component of $adfast$ to an FSM.

The set of Non-Interactive FSMs, which stem from this second step of the transformation flow, are implicitly synchronized, as per the original FCPTnet semantics. This implicit synchronization is implied between states and transitions of the Non-Interactive FSMs which stem from, and map to the same original place or transition of the covered FCPTnet. Exposing this implicit synchronization is necessary, so as to produce an Interactive FSMs model. The fourth step of the transformation flow, extracts the synchronization primitives for this particular purpose.

4.3 FSM Collapsing

Each element $(s,\, i,\, s')$ of an FSM's δ set is comprised of two states s and s' and an input transition i. Up to this point, the PTNet to FSM transformation assumed that each input corresponds to a single signal. However, FSM semantics dictate that each transition corresponds to a Boolean Function of input signals. For example, if $f_i = in_i.in_j$ then transition $(s,\, f_i,\, s')$ is activated when both in_i and in_j are activated, independently of their activation order. Effectively, in_i and in_j are concurrent in f_i. This type of concurrency may also be exploited for the outputs in the elements of the λ set.

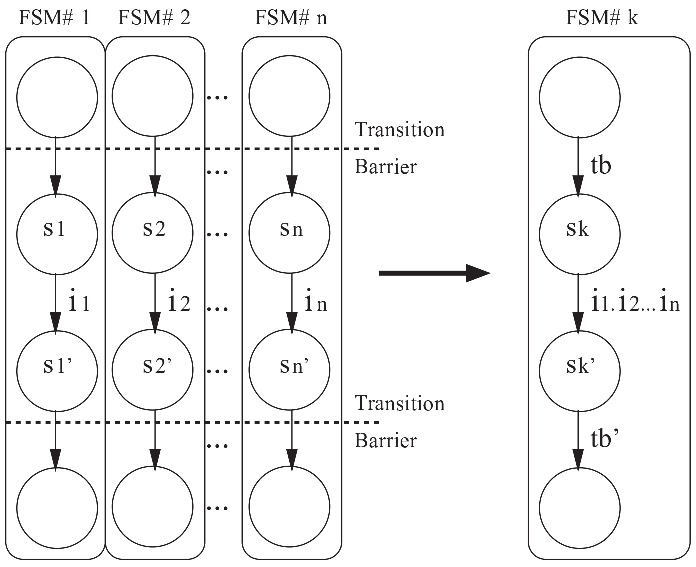


Figure 4.2: FSM Collapsing - Example

Thus, in the case where multiple FSMs exhibit concurrency only between inputs or outputs, these may be collapsed into a single FSM. An example is shown in Figure

4.2, where n concurrent input events are distributed to n synchronized FSMs which only exhibit concurrency at the aforementioned input events.

The complexity for collapsing is $O(P^2T^4)$, as each FSM's transitions are compared to the transitions of all other FSMs, so as to ascertain whether the latter, as well as their predecessor and successor transitions respectively stem from the same corresponding PTnet transitions (the number of FSMs is in worst case $|P|$ and the number of transitions in each FSM is in worst case $|T|$).

4.4 Synchronization Primitive Extraction

This fourth step of the transformation flow identifies the complete set of implicit synchronization primitives, *i.e.* Wait States and Transition Barriers (*c.f* Section 3.2), as expressed by the S-Cover and S-Components of the FCPTnet. The latter are extracted by analyzing the S-Component's flow relations. This is achieved by transforming each flow-relation into the form $F' : S \times T \rightarrow S$. Common transitions in T of different F''s, *i.e.* shared between the S-Components, produce Wait States, as they implicitly describe a state synchronization dependence for entering a state of the S-Component. Similarly, common States in S, *i.e.* originating from the same PTnet place, produce Transition Barriers, as they implicitly describe states of multiple S-Components which must be simultaneously entered or left.

Figure 4.3 illustrates the Non-Interacting FSMs of $adfast$. Synchronization primitive analysis of the original FCPTnet derives 8 Transition Barriers, two of which, *i.e.* $\{t_0, t_6, t_{12}, t_{18}\}$, $\{t_5, t_{11}, t_{17}, t_{23}\}$ synchronism all four FSMs, whereas the remaining six synchronise two FSMs each, *i.e.* $\{t_7, t_{13}\}$, $\{t_8, t_{14}\}$ synchronism FSM_1, FSM_2, $\{t_3, t_9\}$, $\{t_4, t_{10}\}$ synchronism FSM_0, FSM_1 and $\{t_{15}, t_{21}\}$, $\{t_{16}, t_{22}\}$ synchronism FSM_2, FSM_3.

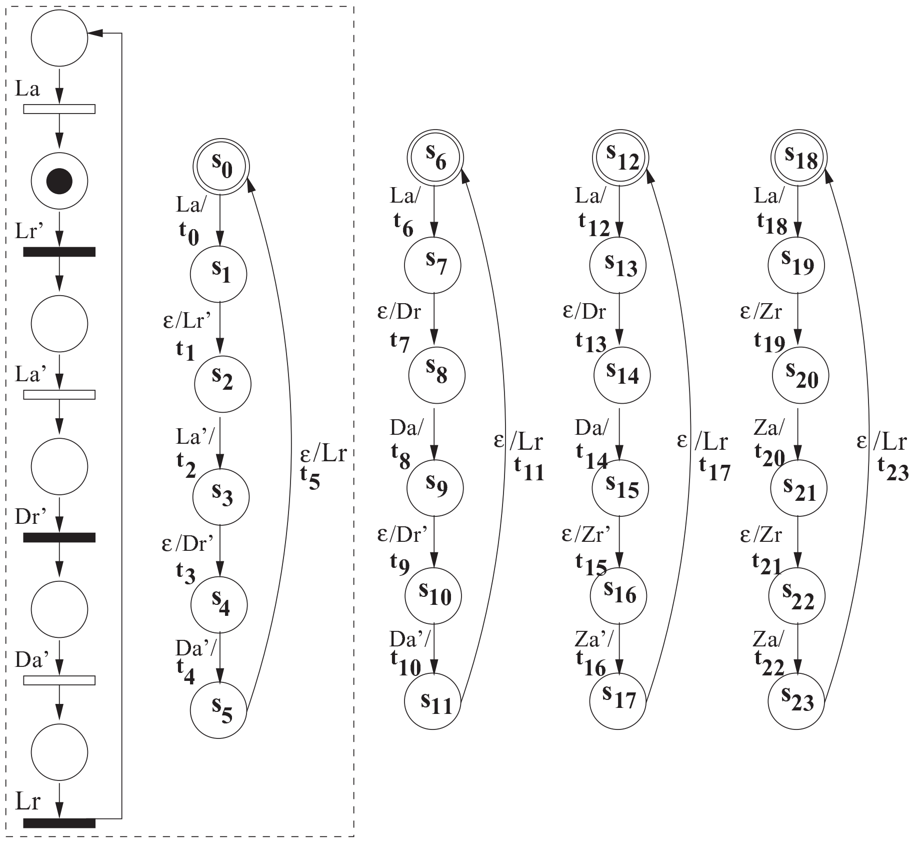


Figure 4.3: $adfast$ - S-Component to FSM Conversion

The complexity for extracting the synchronization primitives is $O(P^3T^2)$, as each place, transition pair of the original PTNet is analysed with respect to the states and transitions sets of the $|P|$, in the worst case, S-Components.

4.5 Inter-MSFSM Synchronization Integration

With the synchronization primitives identified, this step completes the transformation process by promoting the monolithic δ_i functions of each FSM to Δ_i, so as to explicitly include the inter-MSFSM synchronizations dictated by the synchronization primitives, and it also implements the global output generation function, Λ.

The finest grain of synchronization, according to Definition 3.2.1, is the Wait State. A Transition Barrier, Definition 3.2.2 is indeed a set of interlocked Wait

States. Thus, with respect to integrating inter-MSFSM synchronization, it is both necessary and sufficient to integrate Wait State synchronization and express each Δ_i as $\Delta_i : I \times O_1 \times O_2 \times \ldots \times S_i \times \ldots \times O_N \rightarrow S_i$ functions, *i.e.* determine the dependent intra-FSM outputs for a state change.

We now illustrate specifically how the synchronization is integrated. Each Wait State (W, t, s_1, s_2, ..., s_m), which may indeed be a member of a Transition Barrier, dictates that states (s_1, s_2, ..., s_m) form corresponding state-generated outputs, ($o(s_1)$, $o(s_2)$, ..., $o(s_m)$), which, along with transition t form the transition function for entering State W. With respect to the state-generated outputs, $o(s_i)$ will evaluate to logic 1, if the FSM is currently in this corresponding state, or logic 0 otherwise. Thus, the transition function into W will be of the form $W_{enter} = (t\ .\ o(s_1).\ o(s_2)\ .\ \ldots\ .\ o(s_m))$, *i.e.* the Boolean conjunction of t and the state-generated outputs. In this way, each Wait State contributes to the generation of local state-generated outputs, and subsequently the formation of the Δ_i next state functions.

The global output generation function, Λ, where $\Lambda : I \times O_1 \times O_2 \times \ldots \times O_N \rightarrow O$, is expressed by forming the consensus of the corresponding local outputs. Hence, output transitions which exist in multiple FSM's λ_i functions must be combined, again through Boolean conjunction, so as to render the consensus of the local outputs.

The complexity of this step is equal to the worst case total number of wait states *i.e.* $O(P^2T)$, as each place-transition pair in the original PTNet may have been cloned to all $|P|$ FSMs.

4.6 Transformation Completeness and Equivalence

In this section, we formally prove that the presented transformation flow is complete, *i.e.* any FCPTnet is transformable to an equivalent MSFSM set. Prior to the proof itself, we formally define the notions of global state for the MSFSM set, and input, output trace equivalence.

Definition 4.6.1 (FCPTnet, MSFSM Global State)**.** *The global state of a safe FCPTnet System (N, M_o), is its current Marking, M_C, i.e. the set of currently Marked Places.*

The global current state of an MSFSM set M, composed of n FSMs, is the n-tuple, (CS_1, CS_2, ..., CS_n), formed by the current states of the n MSFSMs.

The presented transformation indeed forms a bijection between the global states of the FCPTnet and MSFSM set, as we prove in the following Lemma.

Lemma 4.6.1 (Global State Change Bijection)**.** *A bijection relation exists between the Global State Change of the FCPTnet and the Global State Change of the MSFSM set.*

Proof. A global state change of the FCPTnet represents a change in the marking of the net. The latter is based on token movement, in accordance to the firing of input and output transitions. According to Lemmas 2.3.1 and 2.3.2, S-Cover's global state change is indeed a bijection of the global state of the FCPTnet, as any token movement in the latter has a direct correspondence to token movement in the former. Now, the MSFSM set is generated directly from the S-Cover, and as described in Sections 4.2-4.5, both a unique and bi-directional structural pairing exists between places of the S-Covers and states of the MSFSM set, and a unique and bi-directional correspondence, *i.e.* a matching by name, exists between input and output transitions of the S-Covers and inputs and outputs of the MSFSM set. Further on, the next state transition functions, Δ_i, of the MSFSMs are formed precisely to correspond to, and implement the allowed token movement of the S-Cover. Thus, any input transition, t, simultaneous between S-Covers corresponds to a Transition Barrier, and in turn corresponds to Wait State Boolean transition functions $W_{enter} = (t \, . \, o(s_1). \; o(s_2) \; . \; \ldots$

. $o(s_m)$). In addition, any output transition of the S-Cover, corresponds directly to a local output generation in the MSFSM set. From this bi-directional pairing between places, states, inputs and outputs and the formation of the transition functions to implement the allowed token movement of the S-Covers, it follows that any global state change of the S-Covers, through an allowed token movement, corresponds to an exactly matched, 1-1 and onto global state change in the MSFSM set, as dictated by the input and output transitions. As the opposite also holds, based on the above, the global state change relation is a bijection. □

Now, proving input, output trace equivalence between an FCPTnet and its corresponding MSFSM set, requires defining input and output trace equivalence with respect to the global states of the two models.

Definition 4.6.2 (k-Distinguishable Global States)**.** *The global states of an FCPTnet System, (N, M_o), M_c, and the MSFSM set, M, $(CS_1, CS_2, \ldots, CS_n)$, are distinguishable if and only if there exists at least one finite, allowed input sequence of length k, which, when applied to both N and M, with N and M which reside in their corresponding initial global states, it causes different output sequences.*

It thus follows that global states which are not k-distinguishable are k-equivalent. Thus, equivalence may be defined as follows.

Definition 4.6.3 (FCPTnet, MSFSM Equivalence)**.** *An FCPTnet System, (N, M_o) is equivalent to an MSFSM set, M, if and only if, for every allowed input sequence, the same output sequence will be produced from their corresponding initial global states of the FCPTnet N and MSFSM set M.*

The allowed input sequences are the valid input transition sequences described in the FCPTnet specification. Now, equivalence between the original FCPTnet and resultant MSFSM set must be proved with respect to Definition 4.6.3.

Theorem 4.6.1 (Transformation Equivalence, Completeness)**.** *For any well-formed, STG-labeled FCPTnet System, $(N,\, M_o)$, an equivalent MSFSM set, M, is derived by the presented flow.*

Proof. (Induction) Assume the FCPTnet System (N, M_o), and the MSFSM set, M, *are not k-distinguishable*, according to Definition 4.6.2, and are thus k-equivalent, *i.e.* for all input sequences of length k (or less), output sequences match. We must subsequently prove that they are not (k + 1)-distinguishable and thus (k + 1)-equivalent.

According to our induction hypothesis, the global states of the FCPTnet System (N, M_o), and the MSFSM set, are not k-distinguishable, *i.e.* after k input transitions, the global states of the two are indeed equivalent, as they produce the same output sequences. Thus, starting from equivalent global states, we consider the arrival of an additional, allowed input; the next observable output will determine (k + 1)-distinguishability. Now, if an output is not generated for the (k + 1) input, it follows straightforwardly that (k + 1)-distinguishability does not hold. Thus, we consider the case that an observable output is indeed generated. According to Lemma 4.6.1, the global state change of the FCPTnet and that of the MSFSM set is a bijection, thus the global state change of the FCPTnet, which will produce the next observable output, will possess a corresponding global state change for the MSFSM set, and both will be reachable from the two equivalent global states of the k-input sequence. It thus follows that the (k + 1) input must also lead to equivalent global states for the FCPTnet and the MSFSM set, hence the (k + 1) allowed input sequence is not (k + 1)-distinguishable. □

Chapter 5

MSFSM Set to PTnet Transformation

MSFSMs were the key for transforming PTNets to interacting FSMs without exhibiting the state explosion problem. This flow allowed synthesizing PTNet specifications for which the only viable path to implementation was so far direct mapping. At the interacting FSMs abstraction level, each monolithic FSM enjoys a rich set of both exact and heuristic synthesis algorithms. However, verifying a control specification comprised of multiple FSMs requires the generation of the FSM product, the size of which is exponential with respect to the total number of original states. The aforementioned verification process can be efficiently performed at the PTNet level, where polynomial complexity algorithms exist for verifying the same properties.

This Chapter introduces a practical and novel transformation flow for the conversion of an FSM set into a PTnet and classify the resultant PTnet class according to the MSFSM set structure. As with the reverse transformation, the key is the MSFSM model which models the inter-FSM synchronization with its underlying synchronization primitives. The combination of the transformations of the previous and the current Chapter illustrates a duality between interacting FSMs and PTNets. We are aware of no prior contribution to the solution of this problem.

The interacting FSMs to PTnet transformation requires the following four steps

1. *Synchronization Primitive Extraction*

2. *Interacting FSMs to MSFSM set Conversion*

3. *MSFSM set FSM to S-component mapping*

4. *MSFSM set S-component merging to PTnet*

5.1 Synchronisation Primitive Extraction

This step is similar to the third step (Section 4.4) of the FCPTnet to MSFSM set flow. Its purpose is identical, *i.e.* to identify, according to the interacting FSMs structure, the complete set of Wait states and Transition barriers. Extraction takes place in two separate steps. Wait state extraction takes place by analysing the next state, δ, and output, λ, functions of each FSM, and their interaction, whereas Transition barrier extraction takes place by analysing the Wait state sets.

Algorithm 2 - Wait State Extraction

```
Require: An MSFSM set
Ensure: A W set with all the Wait state tuples
 1: $W = \emptyset$;
 2: for all $fsm_i \in FSMSET$ do
 3:   for all $\delta_i = (s_i,\ f(\lambda_{(FSMSET-i)}),\ s_j)$ of $fsm_i$ do
 4:     $w_{\delta_i} = \emptyset$;
 5:     for all states $s_k \in$ other FSMs of $FSMSET$ do
 6:       if $f|_{s'_k} == 0$ then
 7:         $w_{\delta_i} = w_{\delta_i} \cup \{\ s_k\ \}$;
 8:     if $w_{\delta_i} \neq \emptyset$ then
 9:       $w_{\delta_i} = (s_i,\ f,\ w_{\delta_i})$;
10:       $W = W \cup \{\ w_{\delta_i}\ \}$;
```

Algorithm 2 assumes as input the set of interacting FSMS, $FSMSET$, and generates W, *i.e.* the set of Wait states. A Wait state tuple w_{δ_i} is defined for each element of the δ set, δ_i. Wait state tuples are generated by inspecting the next state function per FSM, δ_i, and extracting the combinational transition function, f, whereby $\delta_i = (s_i,\ f(\lambda_{(FSMSET-i)}),\ s_j)$, *i.e.* f is a boolean function which satisfies the transition from state s_i to state s_j, and depends on the output functions, λ of other FSMs, ($FSMSET$ - i) (line 3). The support of f will include state generated outputs of other FSMs, *i.e.* the outputs of other FSMs, which depend on their respective states, this is how the FSM interaction can be identified in the next state function. The algorithmic complexity of the above algorithm is $O(|T| * N * S)$ where T is initial PTNet's transition set, N is the resultant MSFSM's number of FSMs and S is the total number of FSM Set's states.

Checking that an FSM state s_i is a Wait state and depends on state s_k, of another FSM, may be performed by evaluating the negative boolean co-factor of f with respect to s_k, *i.e.* $f|'_{sk}$ (line 6). If the latter is false, it implies that s_k directly determines f, as f is in the form $f = f|_{sk}\ .\ s_k$. If this is the case, s_k is added to w_{δ_i}, to create an unordered set of dependent states (line 7). After the traversal of all other FSM's states, the complete Wait state tuple is created by appending w_{δ_i} with s_i and the transition function f (line 9). Thus, W is created as the set of all wait states, w_{δ_i} (line 10). After the extraction of Wait states, Transitions Barriers can be derived, by a pairwise comparison of the Wait state states. Two or more Wait states with identical states constitute, according to Definition 3.2.2, a Transition Barrier.

Algorithm 3, which generates the set of Transition barriers, TB, takes the Wait State set W, as input, and identifies which of them also constitute a transition barrier. All Wait state tuples are pairwise compared. Those comprised of the same states and transition functions, form a transition barrier (line 4). Then if w_i's transition is not present in any of TB's members, a new transition barrier is created with t_i as the only

Algorithm 3 - Transition Barrier Extraction

```
Require: A W set with all the Wait state tuples
Ensure: A TB set with all the transition barriers
 1: for all $w_i$, $w_j \in$ W do
 2:    if $i \neq j$ then
 3:       if $w_i \cap w_j \equiv w_i$ then
 4:          $FOUND$ = FALSE;
 5:          $t_i$ = TRANSITION($w_i$);
 6:          $t_j$ = TRANSITION($w_j$);
 7:          for all $tb$ in $TB$ do
 8:             if $t_i \in tb$ then
 9:                $FOUND$ = TRUE;
10:                $tb_i = tb$;
11:          if $FOUND$ == FALSE then
12:             $TB = TB \cup \{ t_i \}$;
13:          else
14:             $tb_i = tb_i \cup \{ t_j \}$;
```

element and is inserted in TB (line 14). Otherwise, t_j is inserted in the transition barrier where t_i is also present (line 12). The algorithmic complexity of the above algorithm is O($|P|^2 * |T|$) where P and T are original PTNet's place and transition sets respectively.

5.2 Interacting FSMs to MSFSM set Conversion

The second step of the MSFSM set to PTnet conversion flow constructs the corresponding MSFSM set from the original set of interacting FSMs, *i.e.* generates the corresponding five-tuple $(I, O, S, \Delta, \Lambda)$. Generating the latter is based on the synchronisation primitives, derived in the previous step, which enable distinction to be made between global and local inter-FSM inputs and outputs, and the formation of the Δ and Λ functions. The global input and output sets, I and O, are formed by: (i) creating the union of the inputs and outputs of all FSMs of the set respectively, and then (ii) removing the inter-FSM signals which are contained in the synchronisation primitives. The set of next state and output functions, Δ and Λ of the MSFSM set, are generated by converting the individual FSM next state functions, $\delta : I \times S \rightarrow S$, to $\Delta_i : I \times O_1 \times O_2 \times \ldots \times S_i \times \ldots \times O_N \rightarrow S_i$, and individual FSM output functions, $\lambda : I \times S \rightarrow O$, to $\Lambda : I \times O_1 \times O_2 \times \ldots \times S_i \times \ldots \times O_N \rightarrow O$. This conversion takes place by correlating each member state of the next state and output function to Wait states or Transition Barriers, and appending the relevant state-generated output signals to both functions. Thus, this step generates an MSFSM set, in accordance to Definition 3.1.1.

5.3 MSFSM set FSM to S-component mapping

The purpose of the fourth step is to convert the MSFSM set's FSMs to S-components, *i.e.* the reverse to the S-component to non-interactive FSM mapping described in Section 4.2. For this step, the synchronisation information, *i.e.* local inputs and outputs are ignored, as S-components do not require them. The transformation of each FSM to an S-component takes place as follows. Each state s of the FSM, is converted to a Place of the S-component, bearing the same name, and the Δ_i next state and the Λ_i output functions of each FSM, i, are used to generate Transitions. Each next state transition of the next state function, may be represented in the form ($s1$, t, $s2$). Hence t is converted to a transition of the S-cover. Similarly, each output transition of the output function, may be represented in the form ($s1$, t, o). Hence the latter is converted to a transition t of the S-cover, and a place which corresponds to the output state, s_o. The interconnection of the S-component takes place by correlating the next state and output function representations. Particularly, if for a next state transition, ($s1$, t, $s2$), no output transition of the form ($s1$, t, o) exists, then place $s1$ is connected to transition t, and transition t to place $s2$. Otherwise, $s1$ is again connected to transition t, however t is connected to s_o, the place corresponding to the output transition, s_o is connected to output transition o, and o to place $s2$. The aforementioned places and transitions are inserted into the S-components P and T sets, and the connections between places and transitions create the S-cover's flow relation, F.

Theorem 5.3.1. *An FSM, represented by a strongly-connected State Graph, is transformable to an S-Component.*

Proof. Each triple $(s1, t, s2)$ of FSM's δ is transformed to (s, t) and (t, s') tuples or (s, t), (t, s_o), (s_o, o) and (o, s') tuples if element with (s, t) does not exist in λ or exists respectively. In both cases, a path from place s to place s' exists, for each transition connecting them. Thus the PTnet's connectivity follows the FSM's connectivity and the PTnet's net is strongly connected. $\square$

5.4 MSFSM set S-component merging to PTnet

In the final step of the MSFSM set to PTnet flow, the individual S-components, each represented by their individual P, T and F sets, are merged into their resultant PTnet N. The net is formed in two stages. In the first, N is formed by creating N's P, T and F sets as the union of the corresponding sets of the S-components. In the second, the synchronisation primitives, derived in the first step of the flow, are used to finalise it, by merging identical transitions, or inserting places and connections to existing transitions.

For each Transition barrier of the MSFSM set, *i.e.* $tb = (t_1, t_2, \ldots, t_n)$, the relevant states are extracted from the Δ next state functions of the FSMs, per transition. Then, the corresponding S-component transitions are merged into the single transition represented by the Transition barrier, tb, and the individual transitions of F, which constitute the Transition barrier, are converted to tb. The Transition barrier S-component transformation is illustrated in Figure 5.1, where the original MSFSM, the

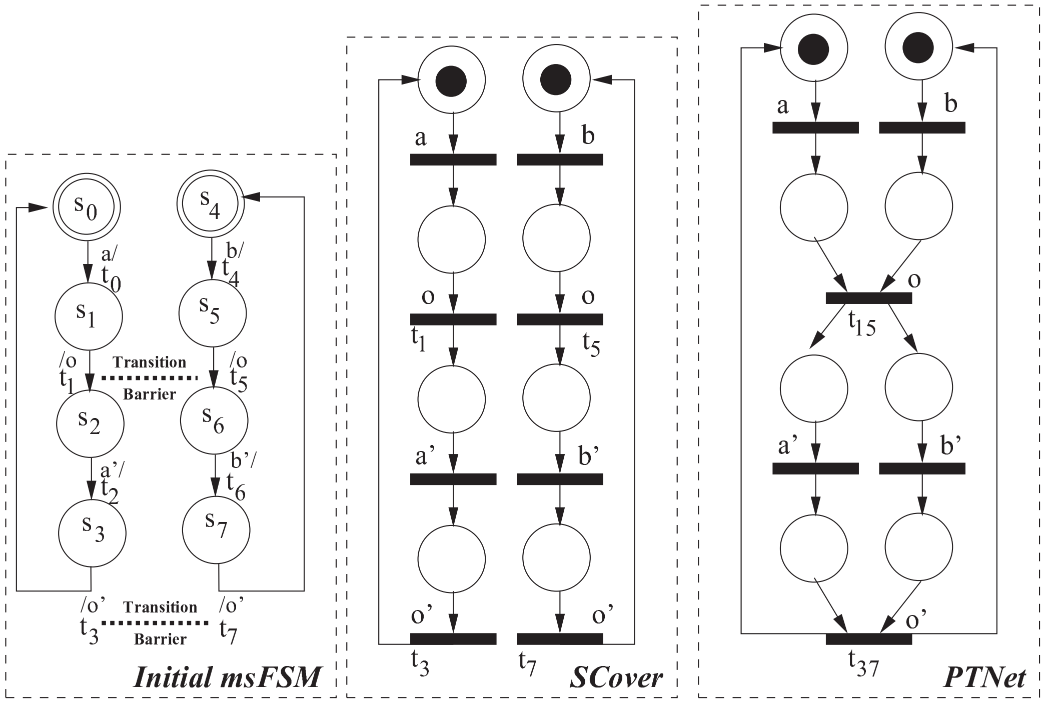


Figure 5.1: MSFSM Transformation to PTnet

S-components and the final PTnet of a C-gate are shown. Transitions o and o', which correspond to the two Transition barriers of the MSFSM, $\{t_1, t_5\}$ and $\{t_3, t_7\}$, are mapped to multiple transitions in the two S-components, t_1, t_5 and t_3, t_7 respectively. The Transition barrier transformation merges them into a single transition, t_{15}, t_{35}, yielding the C-gate's usual PTnet representation.

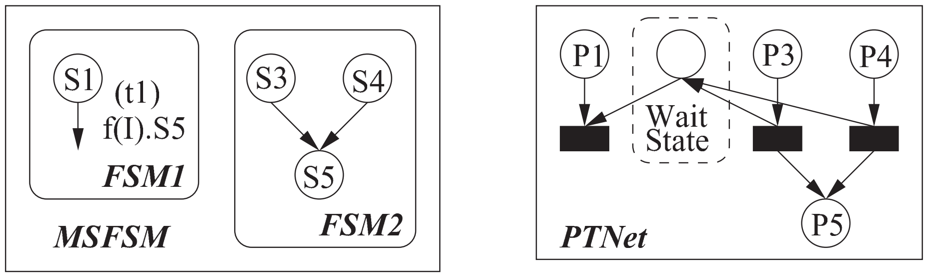


Figure 5.2: MSFSM with Wait state transformed to PTnet

Similarly, for each Wait state of the MSFSM set, $(w, t, s_1, s_2, \ldots, s_n)$, again the relevant states are extracted from the Δ next state functions of the FSMs, for the relevant transition t. Each of the n dependent states of the Wait state w, *i.e.* s_1,

s_2, …, s_n, corresponds to a place p_{si}. To implement the Wait state semantics in the PTnet, a new place, p_{wsi}, per dependent state is inserted, and F is appended with (p_{wsi}, t) and ($^\bullet p_i$, p_{wsi}), for all transitions $^\bullet p_i$. An illustrative example is shown in Figure 5.2, where the original MSFSM's Wait state (s_1, t_1, s_5) requires the addition of a place, labelled $WaitState$, and three arcs.

5.5 Resultant PTnet Classification

In the MSFSM set to PTnet flow, the resultant PTnet has not yet been classified. FCPTnets, as mentioned in previous sections, represent a very effective PTnet class, as FCPTnet algorithms are of polynomial complexity. The same algorithms for more complex PTnet classes, *i.e.* Asymmetric Choice (ACPTnet) or General (GPTnets), fall into the NP category. In practice, concurrent systems described using ACPTnets or GPTnets can, in many cases, be converted to FCPTnets using PTnet unfoldings.

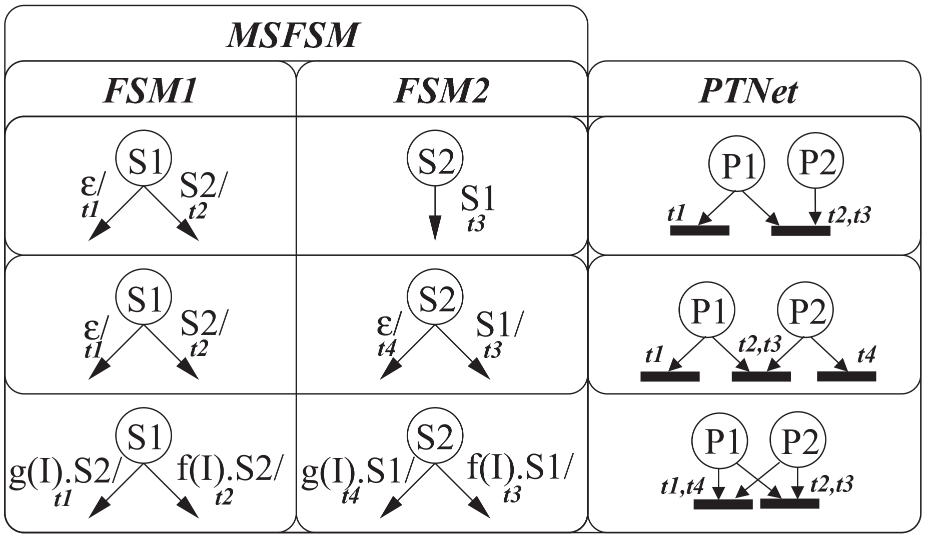


Figure 5.3: MSFSM Transformation to higher-level PTnet classes: ACNet, General PTNet and EFCPTNet

The PTnet, stemming from the MSFSM set to PTnet conversion, in fact may fall into one of the following classes, *i.e.* FCPTnet, ACPTnet or GPTnet, depending on the MSFSM interaction stemming from synchronisation primitives and the relevant MSFSM structure. Figure 5.3 graphically depicts the relevant PTnet structures which characterise the resultant class, and how these relate to MSFSM transitions and Transition barriers. The upper part of the Figure illustrates two MSFSMs with Transition barrier (t_1, t_2), where t_1's predecessor state is a choice state. The resultant PTnet structure corresponds to an ACPTnet. The middle part of the Figure illustrates again two MSFSMs with the same Transition barrier, whereby both transitions' predecessor states are choice states. The resultant PTnet structure corresponds to an GPTnet. The bottom part of the Figure illustrates two transition barriers, (t_1, t_3) and (t_2, t_4), between corresponding transitions of two choice states. In this case, the resultant PTnet structure corresponds to a FCPTnet. Wait states determine PTnet class in a similar manner.

5.6 Using PTNet Structural Properties to Verify System Behavior

PTNets have been extensively used to verify correct operation properties of concurrent software, such as deadlock freedom. Ada, which is a programming language supporting task level concurrency, is an example where Petrinet properties were successfully used to efficiently verify system properties related to concurrency, *e.g.* liveness [76]. In order to use Petrinet theory to verify Ada programs, a flow was developed which initially transformed software to Petrinets, also called Ada-nets. A similar approach was followed in [101] to verify C++ concurrent programs by initially transforming the program to a Petrinet (so called Gadara-Net) and then applying structural Petrinet techniques to verify the correct concurrent behavior of the system.

The flow introduced in this chapter follows a similar path to the ones of software verification. A significant difference, which also signifies the MSFSMs control model, is that concurrent software uses explicit synchronization between the concurrent tasks *e.g.* the `accept` keyword of Ada, whereas hardware description has to encode the synchronization in the transition functions of FSMs. Thus, initially the interacting FSMs are analyzed and the synchronization is made explicit as wait states and transition barriers and finally expressed as an MSFSM.

5.6.1 PTNet Classes with Polynomial Verification Algorithms

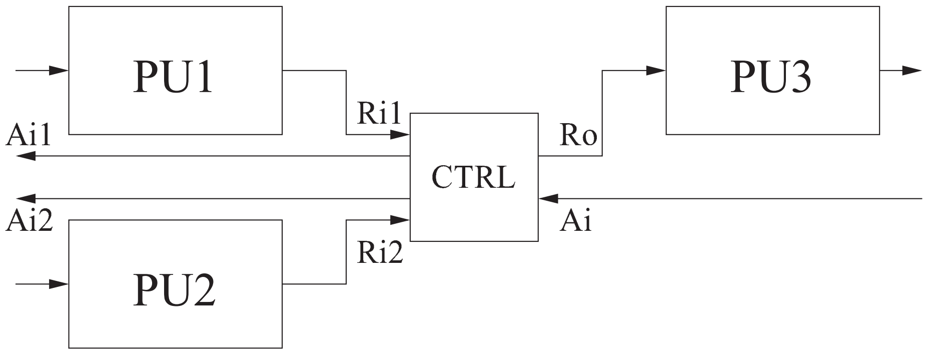


Figure 5.4: Join Handshake Controller Block Diagram

Figure 5.4 shows the block diagram of a circuit with three processing units PU1-3 and a join handshake controller. PU1 and PU2 assert Ri1 and Ri2 respectively when they have finished their operation. When both of the above signals are asserted, PU3 can be fed with valid data which is denoted by signal Ro. When the PU3 processing is finished, which is denoted by rising Ai, the modules driving PU1 and PU2 should be notified as well by the corresponding acknowledge signals. The under synthesis module is the controller noted as $CTRL$ which orchestrates the flow of data among the processing units.

In the conventional PTNet based specification of asynchronous circuits, the functionality of each interface is mixed with the synchronization between them. This is shown in Figure 5.5 where two different synchronization styles of the interfaces

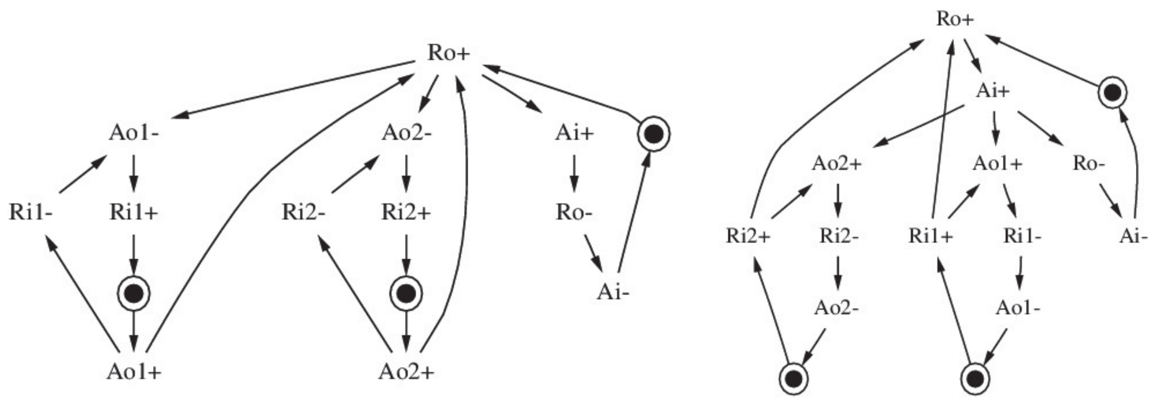


Figure 5.5: Join Handshake Controller Block Diagram

are described. In the leftmost specification, Ro is asserted after both of the joined handshakes have asserted their acknowledges. The above acknowledges can only be deasserted after the rising of Ro. In the rightmost specification, Ro is asserted after the two other requests are asserted. Furthermore, the acknowledges to the joined handshakes are not initiated unless the processing of PU3 is finished, which is denoted by rising Ai.

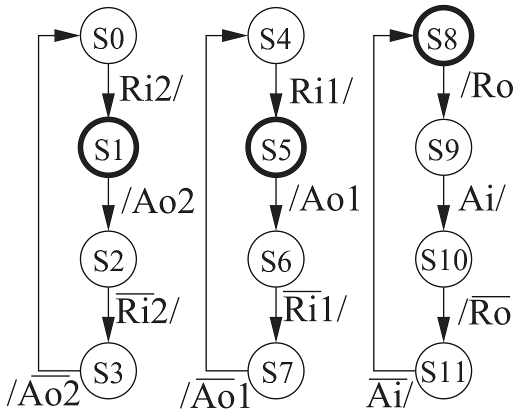


Figure 5.6: Join Handshake Controller FSMs

Figure 5.6 shows the FSMs describing the functionality of the three separate interfaces of the controller. As in the MSFSM model the functionality is decoupled from the synchronization, these FSMs remain unchanged whilst examining the different specifications resulting from different synchronization styles.

W	f(I)	Dep. States	W	f(I)	Dep. States
S8	ϵ	S2, S6	S8	ϵ	S1, S5
S3	ϵ	S9	S1	ϵ	S10
S7	ϵ	S9	S5	ϵ	S10

Table 5.1: Join Handshake Controller Wait States for the two different synchronization scenarios of Figure 5.6

Tables 5.1 includes the wait state tuples explicitly describing the two synchronization styles expressed by the two corresponding PTNets in Figure 5.5.

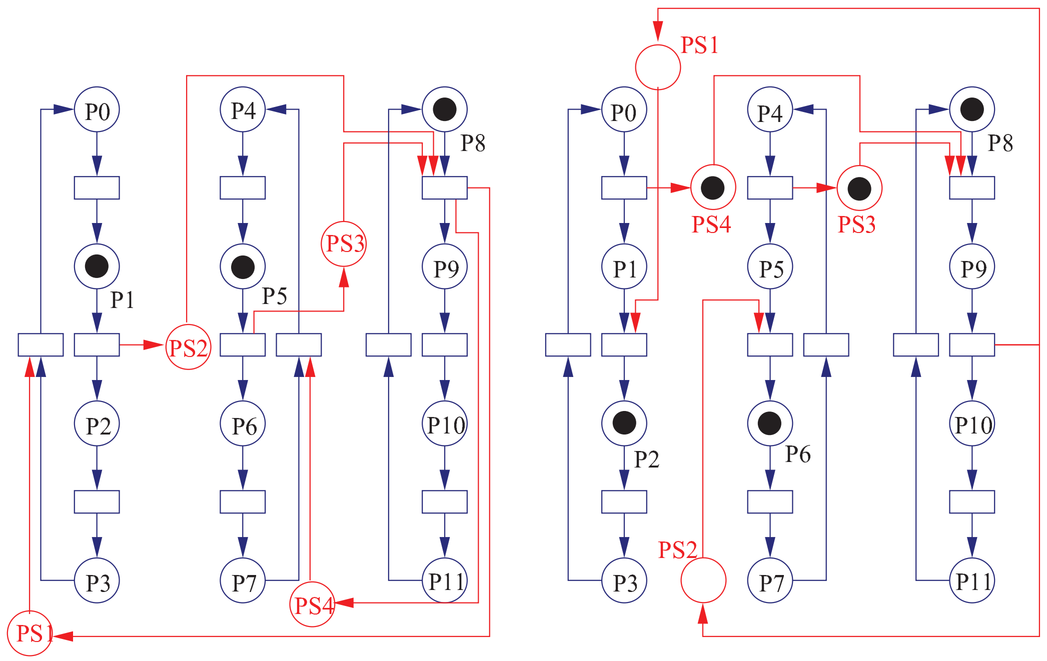


Figure 5.7: Join Handshake Controller PTNet

In order to verify the correct concurrent operation of the two MSFSM systems for the join handshake controller, the transformation introduced in this Chapter should be applied. The two corresponding resultant PTNets are shown in Figure 5.7. Both of them belong to the class of safe FCPTNets, hence there exist efficient algorithms verifying for properties such as liveness in polynomial time [21].

In particular, verifying PTNet well formedness *i.e.* liveness and boundedness, it suffices to prove that the PTNet is covered by a set of minimal siphons which are all marked in the initial marking. This is the case for both of the resultant PTNets as they both include the interface functionality minimal siphons $\{P0, P1, P2, P3\}$, $\{P4, P5, P6, P7\}$ and $\{P8, P9, P10, P11\}$, the leftmost is additionally covered the initially marked minimal siphons $\{P0, P1, PS2, PS1\}$ and $\{P4, P5, PS3, PS4\}$ and the rightmost by $\{PS4, P9, PS1\}$ and $\{PS3, P9, PS2\}$.

5.6.2 PTNet Classes with Intractable Verification Algorithms

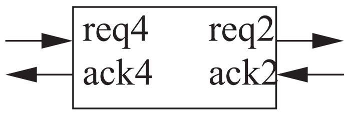


Figure 5.8: Four to Two Phase Handshake Transformation Controller Block Diagram

Figure 5.8 shows the block diagram of a controller transforming a four phase handshake to a two phase one. The four phase handshake is realized by the input $req4$ and the output $ack4$ and a complete handshake with the environment is comprised of the transition sequence $req4+ \rightarrow ack4+ \rightarrow req4- \rightarrow ack4-$. Accordingly, the two-phase handshake is realized by output $req2$ and input $ack2$ and a complete handshake adheres to the transition sequence $req2^{+(-)} \rightarrow ack2^{+(-)}$.

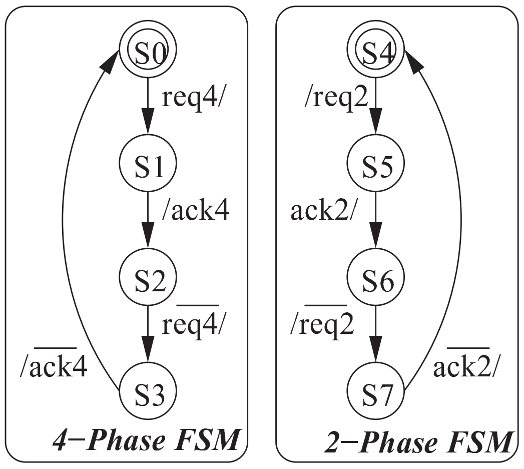


Figure 5.9: Four to two handshake controller MSFSM

The described functionality for each handshake is captured by the two FSMs shown in Figure 5.9. These two FSMs can be synchronized to implement the protocol transformation. According to the concurrency needed, different synchronization options can be examined.

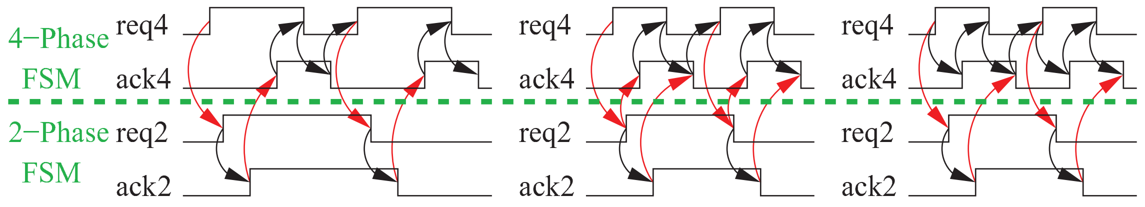


Figure 5.10: Four to two handshake controller different synchronization styles

Figure 5.10 shows three possible synchronization options. In the leftmost scenario the four phase handshake initiates the two phase one and it is not unless the rising of $ack2$ by the 2 phase environment that the four phase handshake can resume. In the middle synchronization style, the four phase handshake initiates the two phase one but it does not have to wait the acknowledge of the two phase environment to acknowledge its own one. Finally, in the rightmost synchronization style, the four phase handshake does not have to wait for the initiation of the two phase one before it acknowledges its environment, and it is only at the end of the handshake *i.e.* when the acknowledge falls where the four phase handshake waits for the two phase one. Hence, from the above, the rightmost is the most concurrent specification whereas the leftmost one is the least concurrent one.

The synchronization between the two FSMs can be deduced by following the dependency arrows, highlighted in red, which cross the green bound of the two FSMs. Implementing the above synchronization using the conventional monolithic FSM, interacting FSM or PTNet requires mixing the functionality and the synchronization between the controller interfaces. MSFSMs on the other side decouple the interfaces functionality from the interfaces synchronization and thus the three different synchronization styles can be realized by just changing the MSFSM synchronization primitives, as shown in Tables 5.2.

W	**f(I)**	**Dep. States**
S4	ϵ	S1
S6	ϵ	S1
S1	ϵ	S6
S1	ϵ	S4

W	**f(I)**	**Dep. States**
S4	ϵ	S1
S6	ϵ	S1
S1	ϵ	S5
S1	ϵ	S7
S3	ϵ	S4
S3	ϵ	S6

W	**f(I)**	**Dep. States**
S4	ϵ	S1
S6	ϵ	S1
S3	ϵ	S6
S3	ϵ	S4

Table 5.2: Half Handshake Controller Wait States for the three different synchronization scenarios of Figure 5.10

The description of the three different synchronization styles with the MSFSMs model allows verifying the properties of the specification using algorithms performing at the PTNet level. According to the PTNet class of the resultant PTNet the complexity of the above algorithms may change. Hence at the PTNet world, it is the class of the resulting PTNet and the PTNet itself which mainly define the complexity of the under verification property and not the size of the encapsulated state space.

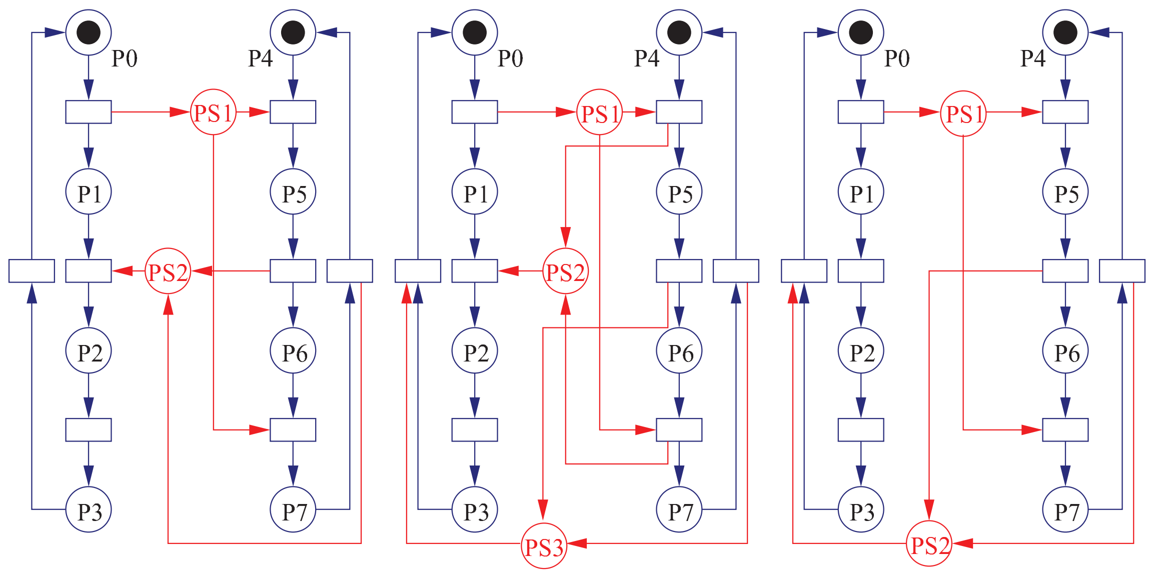


Figure 5.11: Four to two handshake controller PTNet

The PTNets which result from the transformation of the above MSFSM specifications are shown in Figure 5.11. All three PTNets belong to Asymmetric-Choice class. Proving properties which illustrate the correct concurrent operation of the system, such as liveness and boundedness is more complex than the case of Free Choice Petrinets [54]. This is due to the fact that in the general case, the total set of minimal siphons should be extracted and checked for the siphon trap property [10].

The leftmost PTNet has three minimal siphons, $\{P0, P1, P2, P3\}$, $\{P4, P5, P6, P7\}$ and $\{P0, PS1, P5, P6, P7, PS2, P2, P3\}$ and each siphon is also a trap. Hence the PTNet is free of deadlocks. The same can be proved for the other two PTNets.

It should be noted that efficient algorithms have been proposed to solve the problem of the minimal siphon set computation [78], although it has been proved that the total number of minimal siphons grows exponentially with PTNet's size. Nevertheless, there are AC PTNet subclasses for which polynomial time algorithms have been recently introduced [70, 99].

Chapter 6

Synchronous Synthesis

The previous chapters introduced the MSFSM model and highlighted its importance for the flow which transforms PTNets to interacting FSMs. The above flow can be used as a vehicle for PTNet synthesis to digital circuits. This chapter focuses on synchronous digital circuits and presents a methodology which generates RTL or Structural level Verilog [93] starting from PTNet descriptions.

The contribution of this Chapter is a PTNet synthesis flow which is compatible to the state of the art synthesis tools. This flow uses MSFSMs as an intermediate control model for mapping PTNets to a set of interacting FSMs for which mature synthesis algorithms exist. Furthermore, the synchronous synthesis flow will evaluate the novel heuristic algorithms introduced in Chapter 8 which perform at the MSFSM level.

6.1 Synchronous PTNets

The PTNet control model does not include any timing assumptions. As soon as a transition is enabled it can fire. However, conventional digital synchronous are synchronous, which means that sequential elements change state only at global clock edges. Synchronous FSMs which are the contemporary control model for digital

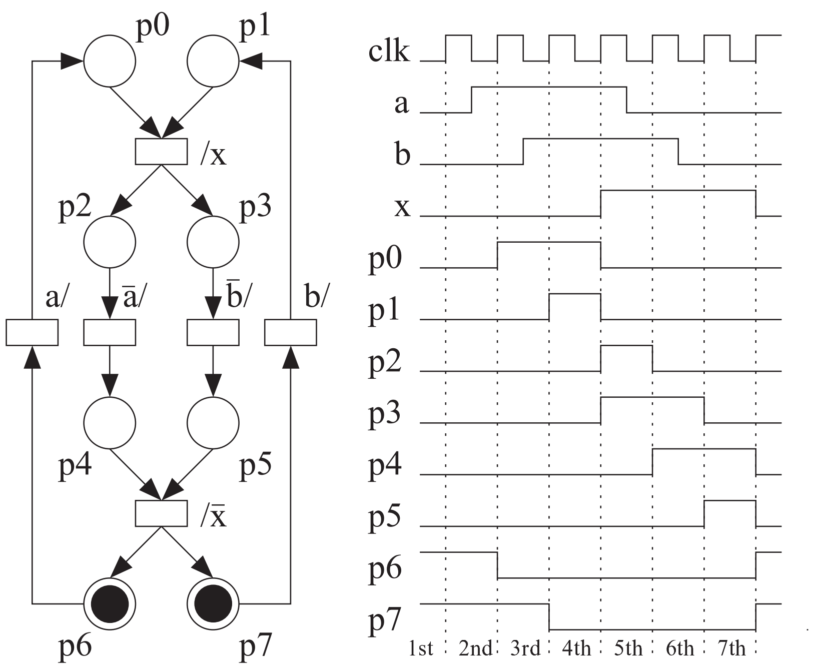


Figure 6.1: Synchronous PTNet Operation Example.

circuits, follow the assumption that a state can be activated only at a global clock edge. Synchronous PTNets follow the same assumption and hence their places, which hold the current state of the system, can only be activated at a global clock edge. The Synchronous PTNet can be thought of as an Asynchronous PTNet, in which the global clock synchronizes the firing of all the activated transitions. Thus, firing concurrency between the activated transitions can not occur.

Figure 6.1 illustrates the operation of a synchronous PTNet modeling a C-Element [95]. A C-Element is a sequential gate with M-inputs and one output. The output is asserted(de-asserted) when all M inputs are asserted(de-asserted). Each input is concurrent to the other inputs and hence the behavior of the C-Element can be accurately modeled by a PTNet. In the timing diagram shown in the same Figure, the first seven cycles of an arbitrary simulation is shown. Initially, all signals are low and the active places are p6 and p7. During the second cycle input a is asserted which causes places p0 and p6 to be activated and deactivated respectively at the beginning of the third cycle. Then, input b is asserted causing place p1 to be activated at the beginning of the fourth cycle. As output x only depends on the activation of the previous places, it also rises at the beginning of the fourth cycle, likewise a mealy-style FSM transition. As both p0 and p1 are both active during the fourth cycle and the input transition function is always true, at the beginning of the next cycle they get deactivated and places p2 and p3 are activated instead. The rest of the simulation follows the aforementioned pattern until the beginning of the eighth cycle during which the initial state is revisited.

6.2 Synchronous MSFSMs

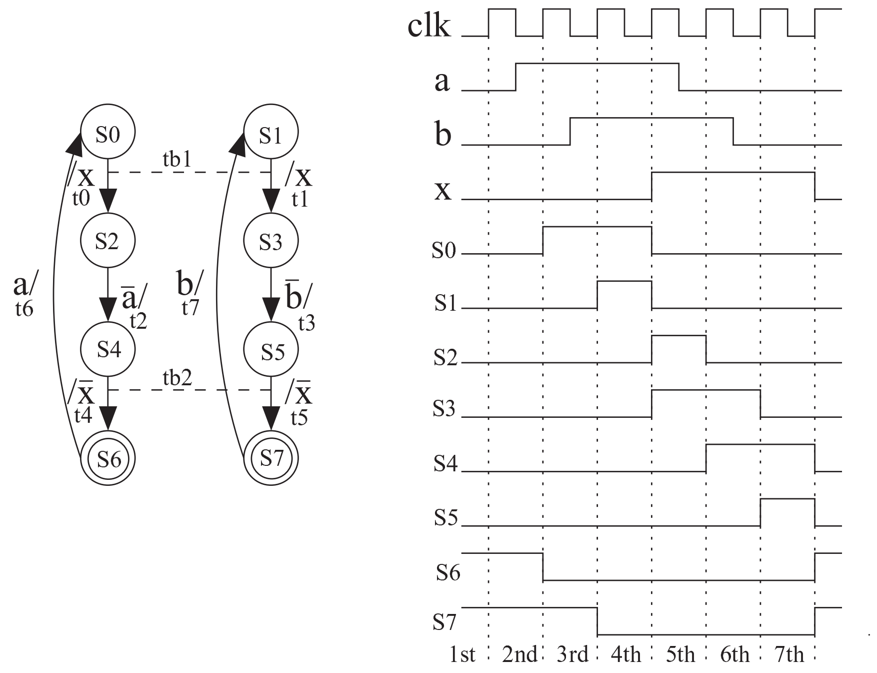


Figure 6.2: Synchronous MSFSM Operation Example.

The PTNet to MSFSM transformation introduced in Chapter 4 was proved to maintain the original behavior of the system. This section illustrates the above, when the timing assumption of global clock is introduced.

Figure 6.2 shows the two input C-Element of the previous section after the corresponding PTNet has been transformed to an equivalent MSFSM. The above MSFSM is comprised of two FSMs, one accepting the transitions of input signal a and one for the transitions of input signal b. Additionally, two transition barriers tb1 and tb2 synchronize the two FSMs at their transition pairs $\{t0, t1\}$ and $\{t4, t5\}$ respectively. The MSFSM operation, shown in the timing diagram of the same Figure produces the same signal transitions in the same clock cycles, illustrating the equivalence of the PTNet and the MSFSM model. It should be noted that the transition barriers are assumed to have no state and hence the synthesis tools should preserve this and not add extra cycles for transition barrier activation.

6.3 RTL HDL Description of Synchronous MSFSMs

The previous Section analyzed the operation of an MSFSM under the synchronous timing assumption. This Section presents a methodology for describing the building blocks of the MSFSM model in a industrial-standard format in order to guide the state-of-the-art industrial scale synthesis tools produce final implementations which follow the Synchronous MSFSM operation. The target HDL is Verilog, which along with VHDL are the most popular HDLs for describing digital circuits.

6.3.1 FSMs

MSFSM FSMs are composed of encoding parameters, state registers a state transition combinational block and the local output functions [33].

The encoding parameters include the default state encoding of the FSM but also allow the logic synthesis tool to change the parameter value when exploring the solution space of the different parameters. In the C-Element example, if the one-hot state encoding is chosen as the default then the following code is generated.

```
parameter [3:0] S0_ENCODING = 4'b0001;
parameter [3:0] S2_ENCODING = 4'b0010;
parameter [3:0] S4_ENCODING = 4'b0100;
parameter [3:0] S6_ENCODING = 4'b1000;
```

The state register holds the current state of the FSM and at each clock edge it is assigned to the next state unless the reset is activated. Naming the current and next states of the leftmost FSM of the C-Element as `SM1_q` and `SM1_d` results to the following Verilog description.

```
reg[3:0] SM1_q, SM1_d;
always @(posedge(clk))
 if(reset)
   SM1_q = S6_ENCODING;
 else
   SM1_q = SM1_d;
```

The state transition and local output functions are described in an `always` block of combinational logic. Local are the outputs for which the value is only specified by a single FSM of the MSFSM. In the C-Element example, output `x` value is dictated by both FSMs and hence it is not local. The Verilog HDL description of the transition function for one of C-Element's FSMs is shown below.

```
always @(*)
 if(reset)
   SM1_d = S6_ENCODING;
 else
 begin
   if(SM1_q == S0_ENCODING)
    if(tb1)
```

```
          SM1_d = S2_ENCODING;
        else
          SM1_d = S0_ENCODING;
      else if(SM1_q == S2_ENCODING)
        if(~a)
          SM1_d = S4_ENCODING;
        else
          SM1_d = S2_ENCODING;
      ...
    end
```

6.3.2 Synchronization Primitives

The RTL description of synchronization primitives is a combinational function. In particular, a transition barrier is formed as a boolean AND of the transition functions comprising it. The RTL description of both synchronization primitives of the C-Element are as follows:

```
  assign tb1 = (SM1_q == S0_ENCODING) & (SM2_q == S1_ENCODING);
  assign tb2 = (SM4_q == S4_ENCODING) & (SM5_q == S5_ENCODING);
```

It should be noted that the transition barriers function can not use the outcome of the boolean function calculating the next states as this would incur combinational loop (`tb1=f(t0)` and `t0=g(tb1)`).

6.3.3 Global Outputs

Global are the outputs which require more than one FSM to define their value. They are described as clocked set-reset flip-flops. The set function is the boolean AND of all the underlying FSM transitions which set high the output. Accordingly, the reset part is comprised of all the transitions de-asserting the output. If the above transitions are part of a transition barrier, then instead of the transition function, the transition barrier function can be used instead. In the C-Element example there a single output, `x` which is global. Transition barrier tb1 asserts `x` and hence it comprises the set part of the flip-flop. Accordingly, tb2 is the reset part of the flip-flop. The resultant RTL level description of the global output is as follows:

```
always @(posedge clk)
if(reset)
  x = 0;
else
begin
  if(tb1)
    x = 1;
  else if(tb2)
    x = 0;
  else
    x = x;
end
```

6.4 Synchronous MSFSM Logic Implementation

The previous Section described the transformation of the MSFSM to an RTL model. The RTL model can be synthesized by logic synthesis tools to an optimized circuit. Thus, the algorithms developed for the MSFSM model can be evaluated by providing the resultant MSFSM to a third party logic synthesis tool and evaluating the QoR of the final implementation. However, having a logic synthesis tool in a closed loop with algorithms exercised on an MSFSM incurs significant execution time penalties. This Section introduces a rapid logic realization procedure of an MSFSM which can be

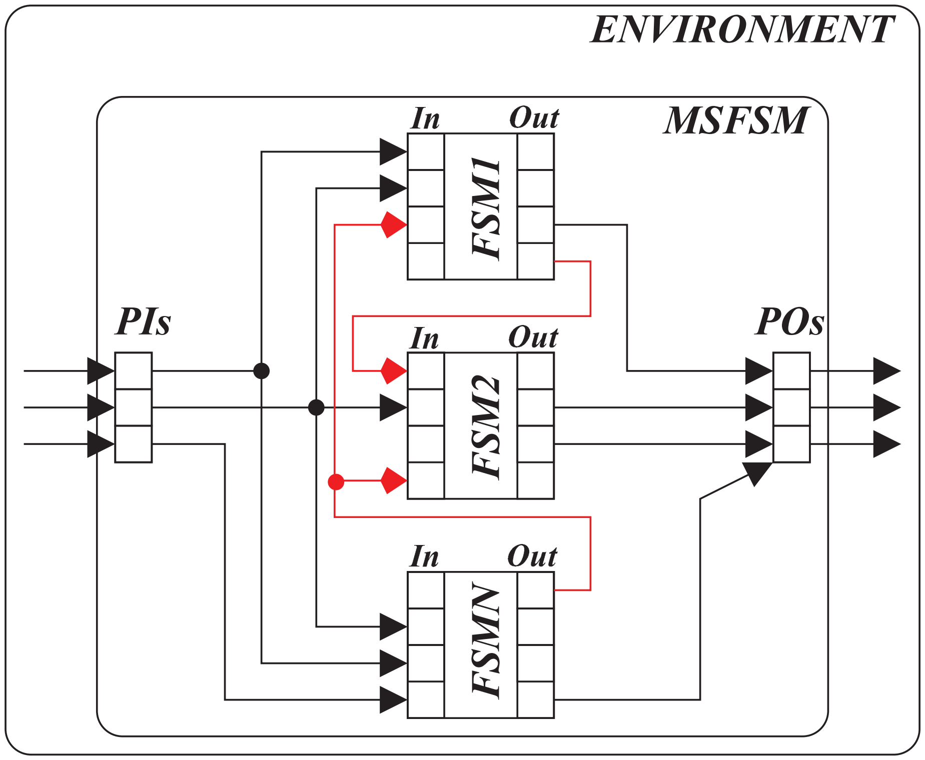


Figure 6.3: Synthconous MSFSM Architecture

used as a cost evaluation vehicle for any logic synthesis tool which supports MSFSMs. Figure 6.3 illustrates a possible architecture of a Synchronous MSFSM comprised of N FSMs.

6.4.1 Primary Inputs and Outputs

The Primary Inputs (PIs) and Outputs (POs) typically constitute the interface between the controller and its environment. Changes on the value of PIs happen synchronously with the edges of controller's clock, CLK. Accordingly, the controller places new values on the POs on the CLK's edges. Besides the above environment interface signals, inter-FSM communication is also realized by additional FSM inputs and outputs which interconnect communicating FSMs. In Figure 6.3, black arrows depict PIs and POs whereas red arrows depict inter-FSM communication.

6.4.2 FSM Logic Implementation

Figure 6.4 illustrates a possible logic implementation an MSFSM's FSM. The FSM logic entities are the interface signals, the transition logic which realizes the boolean functions for the state changes, the reset logic which forces the FSM to its initial state and initial output values and the state decoder which specifies FSM's active state.

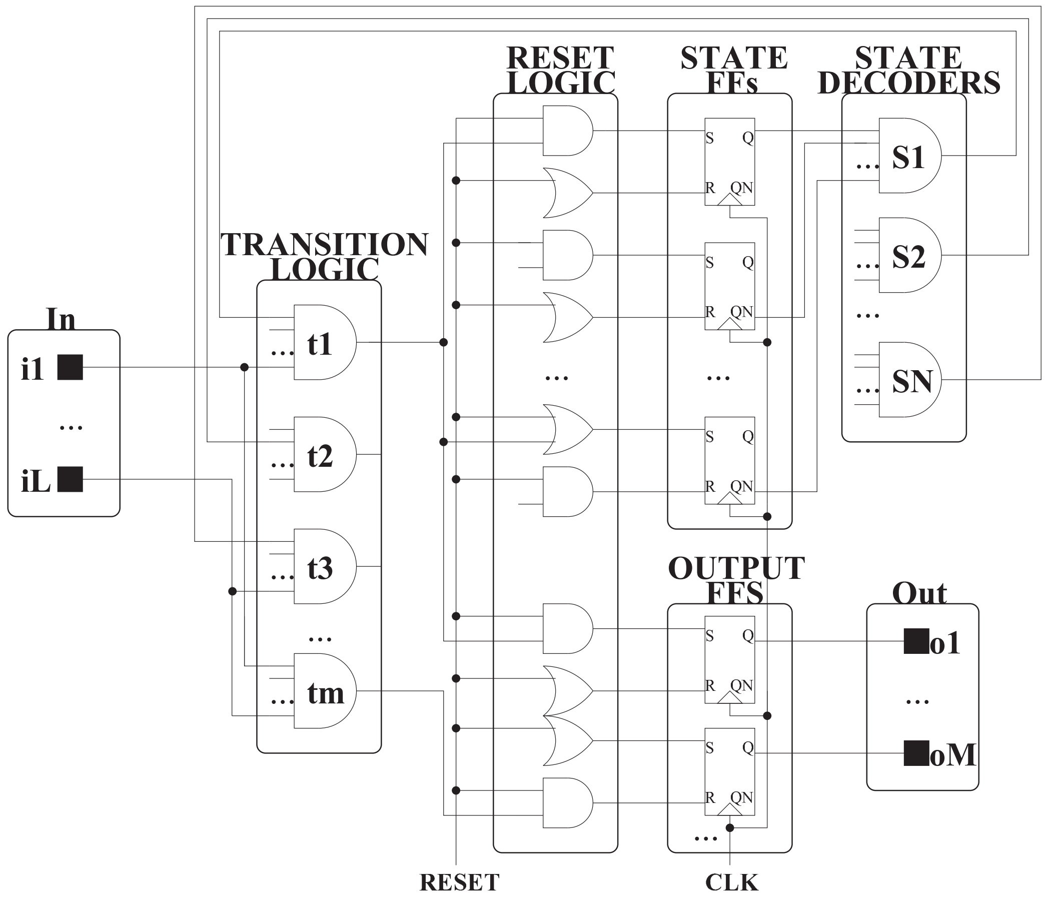


Figure 6.4: Synthconous FSM Architecture

6.4.2.1 State FFs

The state encoding applied to each FSM dictates the number of state holding elements required to store the state of the FSM. In the proposed architecture Set/Reset Clocked FFs are used. The initial state of the controller is forced by asserting the global reset signal, $RESET$. In the case where a state bit should be set in the initial state, an

OR gate at the R input of the RS-FF and an AND gate at its S input guarantee that the FF will be set (cross-coupled NAND implementation is assumed).

6.4.2.2 Output FFs

All the outputs of the proposed architecture are registered. Hence, output changes propagate to controller's environment on CLK's edges. The FSM style chosen is Mealy, hence outputs are boolean functions of both states and inputs. The output values are reset during initialization in the same manner as with the state FFs, using a global reset signal and a pair of AND and OR gates which are placed at S and R FF inputs according to output's initial value.

6.4.2.3 State Decoders

The state decoders, have the form $Q_0Q_1...Qn$, where product literals represent the state holding elements of the FSM. A single decoder which represents the active state of the FSM is set during operation. As a typical MSFSM is comprised of more than one FSMs, there are multiple active states during MSFSM's operation.

6.4.2.4 Transition Logic

The transition logic is a sum of products comprised of inputs and state bits. The input signals are categorized to the environment interface signals and the inter-FSM communication signals. During MSFSM operation, only a single transition logic should be satisfied at the edge of the corresponding clock.

6.5 Path Groups

The proposed architecture defines a set of different path groups. The optimization operations impact specific path groups, hence if the clock period is the main metric of the overall optimization, it is possible to choose the operation which is more likely to affect it.

6.5.1 PI to FFs and FFs to PO

A PI to FF path starts from the environment and ends at a state FF or at an output FF. Its logic depth is affected by the transition and reset logic. Accordingly the FF to PO paths start from an output FF and finish at the environment having no logic in between.

The PI to FF logic depth can be minimized by either performing boolean logic optimization techniques or by modifying the inter-FSM communication. For instance, removing an FSM from a transition barrier may decrease the logic depth of its transition logic, as the inputs required to realize the synchronization are removed.

6.5.2 FFs to FFs

There are four kinds of FF to FF paths according to the proposed MSFSM architecture. State FF to state FF paths within a single FSM are comprised of portions of the state decoder, transition logic and reset logic. Such paths can be optimized by changing FSM's encoding, by optimizing the transition logic and by modifying the inter-FSM communication which contributes to transition logic's Boolean functions. State FF to output FF paths within a single FSM cross the same logic portions as with state FF to state FF paths. Output FF to state FF paths across two FSMs realize the inter-FSM communication. These paths can be removed by either removing the corresponding synchronization points or by collapsing the communicating FSMs to a single FSM. Otherwise the logic depth of the paths can be optimized by minimizing the transition logic. Output FF to output FF paths across two FSMs realize the output functions which reside on synchronized transitions. Optimization strategies for these paths matches the ones of the output FF to output FF paths.

Chapter 7

Asynchronous Synthesis

The previous Chapter introduced a flow for synthesizing synchronous PTNets through transformation to MSFSMs. This chapter targets the problem of synthesizing asynchronous circuits from corresponding PTNet specifications.

The ITRS 2009 [7] Roadmap predicted that by 2012, the first synthesis tools able to implement both synchronous and asynchronous circuits would have had reached the EDA market. Asynchronous circuit design is an emerging technology, and has been a highly active research field for the last two decades. Its advantages over the conventional synchronous design methodology have attracted the attention of almost all leading semiconductor companies [59] [81] [73] [34] [80]. Although results were indeed promising, it was not until after the year 2000, that the first asynchronous technology products [13] [2] [1] [3] were produced. As certain of them are technically superior compared to their synchronous counterparts [90], there will be an increasing pressure on the Semiconductor and EDA industries to integrate asynchronous techniques into standard tool flows, so that more products will enjoy the benefits of asynchronous technology. The continuously increasing number of EDA startup companies [83] introducing cutting edge asynchronous technology in standard industrial flows, is an indication that it is only a matter of time before asynchronous technology becomes mainstream.

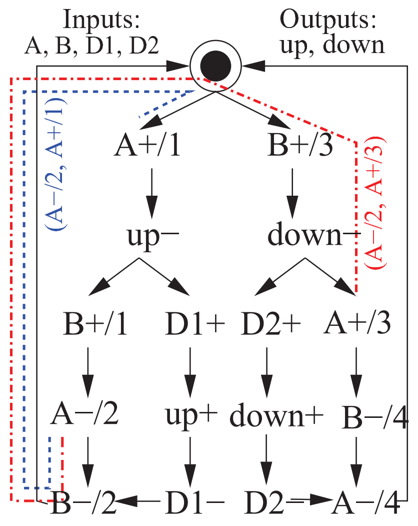
The contribution of this chapter is a flow which actually synthesizes PTNets to asynchronous circuits without suffering from the state explosion problem. The existing heuristic algorithms performing at the FSM level are exploited to target implementation metrics, such as area or performance. Furthermore, the algorithms developed for MSFSMs are also used to optimize the specification at a higher level of abstraction where the inter-FSM communication is explicitly described.

The flow of this Chapter is an evolution of the one presented in Chapter 3, whereby each FSM of the MSFSM set should be a legal BM-FSM. The decomposition flow consists of the following seven steps, two of which, *i.e.* 2 and 3 are identical to the flow in Section 3.

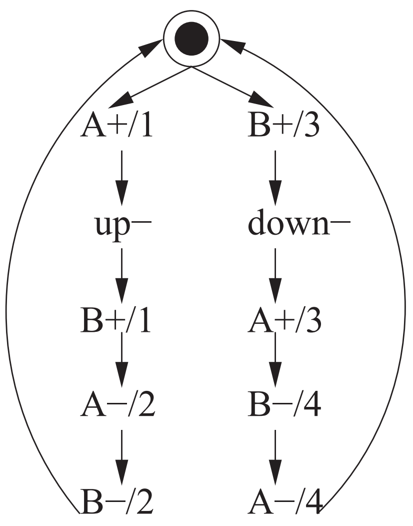
1. *FCPTNet S-covering* (Section 7.1)
2. *S-Component to FSM transformation* (Chapter 4)
3. *Synchronization Primitive Extraction and Integration* (Chapter 4)
4. *FSM to BM-FSM transformation* (Section 7.2)
5. *Insertion of Synchronizing BM-FSMs* (Section 7.3)
6. *Empty Input Bursts Removal* (Section 7.4)
7. *Maximal Set Property Satisfaction* (Section 7.5)

To illustrate the operation of our flow steps, we have used as an example the benchmark `count` [94], an STG with an irreducible CSC conflict for STG synthesis.

7.1 FCPTNet S-covering



(a) PTNet and A-/2, next(A-/2) paths



(b) S-Component covering A transitions

Figure 7.1: *count*2 S-Covering

Algorithm 4 - signal_cover

```
Require: An STG, $stg = (P, T, W, MN, In, Out, l)$, and a signal $s$ in $stg$'s $I \cup O$
Ensure: An S-Component $N' = (P', T', F')$
 1: $N' \leftarrow \{ \emptyset, \emptyset, \emptyset \}$
 2: for all adjacent transition pairs $(s*, \text{next}(s*))$ of $s$ do
 3:     Remove places and transitions concurrent to $s$
 4:     Find a path $path_{ij}$ from $s*$ to next($s*$)
 5:     Add to $N'$ all places and transitions of $path_{ij}$
 6:     for all transitions $t_p$ of $path_{ij}$ do
 7:         Disconnect $t_p$ from all its predecessors other than the one in $path_{ij}$
 8: for all places $p_i$ in $P'$ do
 9:     if there are transitions $t_i$ in $p_i$ preset not in $T'$ then
10:         $S \leftarrow P'$
11:         $S' \leftarrow P$ - $P'$
12:         Find a handle $h \leftarrow$ get_handle($S$, $S'$, $stg$, $p_i$, $t_i$) [58]
13:         Add to $N'$ all places and transitions of $h_i$
14: return $N'$;
```

The S-Covering step extracts a set of S-Components covering all the places and transitions of the specification. In [74] an algorithm was used which allocated a minimal siphon for each place, p. This algorithm was comprised of two steps. During the first one, a DFS was performed, and a single path was extracted starting and ending at p. The above path formed a strongly connected subnetwork, N'. The second step allocated a handle [58] for each transition not in N' which was at the preset of N''s places. Including all the above handles, N' resulted to a siphon, which was also minimal, as N' was strongly connected and the above steps ensured that $|\bullet t \cap P'| = 1$ [56]. Additionally, for live and bounded FCPTNets, a minimal siphon is also an S-Component, which means that for all of its places it holds that $|\bullet t \cap P'| = |t \bullet \cap P'| = 1$. The absence of fork and join transitions in N' allowed transforming it to an FSM for which a mature, synthesis-based implementation path exists.

Using the above S-Components to generate BM-FSMs is not feasible as there is no guarantee that the resulting FSMs will be valid BM-FSMs as well. The above is accomplished with a novel algorithm introduced in this work which guarantees that the extracted minimal siphons can be transformed to valid BM-FSMs. Each such minimal siphon covers all the transitions of one or more signals. Hence, individual minimal siphons, which also form S-Components for FCPTNets, are capable of identifying all the events of one or more input signals, and/or of generating all the events of one or more output signals, resulting to a consistent BM-FSM.

The process of extracting an S-Component reduces for live and bounded FCPT-Nets to finding a subnetwork $N' = (P', T', F')$ which is strongly connected and its places form a minimal siphon [42]. The introduced algorithm, shown in Figure 4, creates in a bottom up fashion a subnetwork with the aforementioned properties for each signal s.

Initially, the algorithm iterates through all signal's adjacent pairs, ($s*$, next($s*$)) (line 2). For the `count2` example, next$(A-/2) = \{A+/1, A+/3\}$ and hence the extracted pairs are ($A-/2$, $A+/1$) and ($A-/2$, $A+/3$). Afterwards, a single path is found for each one of the aforementioned transition pairs (line 4) corresponding to one of the possible sequences σ, comprised of places and transitions which are in $IR(s*, next(s*))$. For `count2` the extracted paths corresponding to pairs ($A-/2$, $A+/1$) and ($A-/2$, $A+/3$) are shown in blue and red colors respectively.

Scanning all the paths between $s*$ and next($s*$) results to undesirable high computation complexity and hence a preprocessing step is performed, removing temporarily from the graph the places and transitions which are concurrent to s (line 3). Then, a simple DFS finds a path between the two adjacent transitions. Path's places and transitions are then added to N' (line 5). As we want to end up with a minimal siphon, it should be guaranteed that $| \bullet t \cap P| = 1$. Hence, path's transitions are disconnected from all the places at their preset, except the one which is in the path (line 7). At this point of the flow, N''s places are not yet a siphon as there may be transitions at the preset of some of its places not in the postset at any of its places. Therefore, for each such transition a handle is found starting from a place of N' and ending at another place of it, as shown in [58] (line 10). Extracted handle's elements are then inserted in N' (line 11). The above procedure is repeated until N's places form a (minimal) siphon.

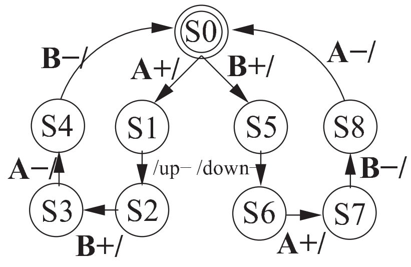
Algorithm 5 - stg_cover

```
Require: An STG, $stg = (S, T, F, M_0, In, Out, l)$
Ensure: SMCover, $smc$
1: for all STG signals, $s$ do
2:     if $s$ is not covered by an S-Component in $smc$ then
3:         $smc \leftarrow smc \cup$ signal_cover(copy($stg$), $s_i$)
4: for all STG places, $p$ do
5:     if $p$ is not covered in $smc$ then
6:         $smc \leftarrow smc \cup$ S-Component($stg$, $p$) [56]
```

The above procedure, as described in Algorithm 5, is performed for all signals. Afterwards, an S-Component is extracted for each place remaining uncovered, following the procedure presented in [56]. The aforementioned S-Components will not be associated with any signal, only synchronizing the operation of the MSFSM system. The S-Component extracted from the *signal_cover* algorithm for the `count2` benchmark and the A signal is shown in Figure 7.1b. Signal B is covered by the above S-Component as well. The complexity for extracting an SMCover is O(max($|T|^3|P|$,

$|P|^2|T|^2))$ as extracting a handle is $O(|P||T|)$. Algorithm's execution time can be reduced if the *next* function is precomputed.

7.2 FSMs to BM-FSMs Transformation



state	in(A)	in(B)	state	in(A)	in(B)
S0	0	0	S5	0	1
S1	1	0	S6	0	1
S2	1	0	S7	1	1
S3	1	0	S8	1	0
S4	0	1			

Figure 7.2: BM-FSM covering A and B input signals

The S-Component extracted in the first step is very close to an FSM as each place may have more than one transition at its postset and preset sets, but each transition may only have a single source and a single sink. The second step transforms each place of the S-Component to an FSM state and each transition to elements in FSM δ and λ sets. This step refines the FSM to conform to the burst-mode semantics.

Initially, the support of each BM-FSM is computed so as to match the support of the FSM which in turn includes signals which are covered by the S-Component, *i.e.* all transitions of the signal are included in the S-Component. The states and transitions of the BM-FSM are modeled with the vertex and arc sets of its graph representation. Hence, a vertex and an arc are added for each element of FSM's S and δ sets respectively.

In BM-FSMs signal values are defined at the vertices as a vector $< i_0, i_1, ..., i_m,$ $o_1, o_2, o_n >$, where $(i_0, i_1, ..., i_m)$ and $(o_1, o_2, .., o_n)$ are the values of the n input and m output signals respectively. Each vertex should be completely specified and hence dont care values are not supported by the model. The key to form BM-FSM functions is the structure of the S-Component as extracted in the first step, where all the places,

p, are in the IR relation of two transitions of each signal s in the BM-FSM support and hence they have values either 1 or 0 for the corresponding literal in their cover cube, C_p^s (cf. [82] for the definition of Cover Cubes). Figure 7.2 shows the BM-FSM which corresponds to the S-Component of Figure 7.1b. As the S-Component covers A and B, the BM-FSMs support includes them both. It should be noted that *up* and *down* transitions are **not** part of this BM-FSM and do **not** generate output signal transitions, but they are used to synchronize this BM-FSM with the BM-FSMs covering *up* and *down* respectively. The vector of the A and B values is shown in the corresponding array. As expected, values for the covered signals are defined for all BM-FSM states and hence there are no dont care values. This step's complexity is equal to $O(P + I + O)$ where P, I and O are STG's places, inputs and outputs respectively.

7.3 Insertion of Synchronising FSMs

The previous steps transformed the S-Components to individual FSMs. Each such FSM is associated with a subset of the signals and hence the FSMs are independent with respect to the interface of the environment. However, BM-FSM transitions stemming from the same PTNet transition should be synchronized to maintain the original control flow. The aforementioned synchronization is modeled as TBs. This step transforms the aforementioned barriers to synchronizing BM-FSMs, merely a mechanism emulating barriers operation in burst mode semantics.

In order to activate a TB $\{t_1, t_2, ..., t_n\}$, all the states $\{s_1, s_2, ..., s_n\}$ which precede the corresponding synchronized transitions should be activated. To detect the activation of a state s_i in FSM_i at another FSM, FSM_j, an additional signal with the same name as s_i is added in FSM_i's O set. With respect to s_i's value it holds that $out(s, s_i)$=1 iff s=s_i and 0 elsewhere.

The activation of a TB is identified by a synchronizing FSM which is added in the MSFSM. This FSM has two states, s and sN. s gets activated and deactivated when all the states preceding a TB's transitions are activated and deactivated respectively. Thus, s's activation denotes the synchronization of TB's underlying FSMs.

Formally, for each TB $\{e_1,\, e_2,\, ...,\, e_n\}$, a BM-FSM, $tb\text{-}fsm{=}(V, E, I, O, v_0, in, out)$, is introduced. The characteristics of the above FSM are the following:

- $V = \{s,\, sN\}$
- $E = \{(s,\, sN),\, (sN,\, s)\}$
- $I = \{s_1,\, s_2,\, ...,\, s_n\}$
- $O = \{sync\}$
- $in(s,\, s_i){=}0$ for $i \in [1, n]$, $in(sN,\, s_i){=}1$ for $i \in [1, n]$
- $out(s,\, sync){=}0$ for $i \in [1, n]$, $out(sN,\, sync){=}1$ for $i \in [1, n]$

The interlocking of the synchronizing FSMs with system's FSMs is achieved by inserting the generated *sync* signal in the I set of the formers. In addition, for each state s_i, such that transition $(s_i,\, s_j)$ is in TB_i, an additional state s_{ij} is inserted in FSM's V set. The E set is transformed as follows:

- $E = (E$ - $\{(s_i,\, s_j)\}) \cup \{(s_i,\, s_{ij}),\, (s_{ij},\, s_j)\}$

With respect to the *in* and *out* functions it holds that $in(s_{ij},\, i) = in(s_i,\, i)$ for each $i \in I$ and $out(s_{ij},\, o) = out(s_i,\, o)$ for each $o \in O - \{s_i\}$. Regarding the output signal s_i it holds that $out(s_{ij},\, s_i){=}0$. The `count2` benchmark after inserting the extra state-outputs and the synchronizing FSMs is shown in Figure 7.3.

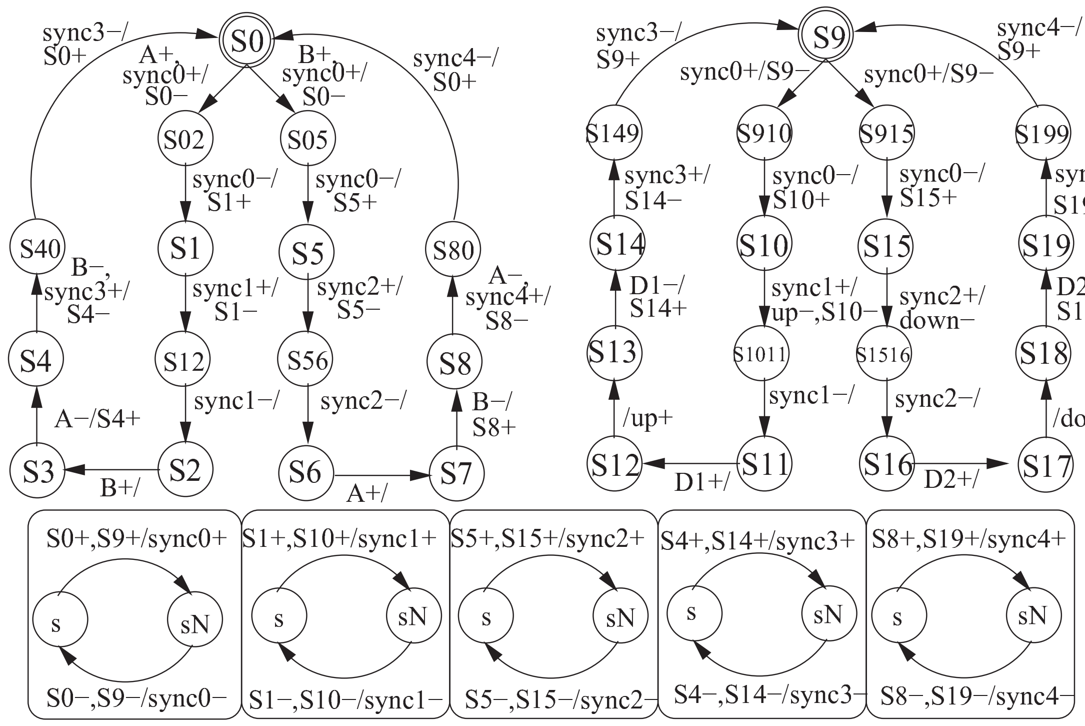


Figure 7.3: BM-FSMs

7.4 Empty Input Bursts Removal

A BM-FSM specification with empty input bursts is not synthesizable. Hence, such edges, if any, should be collapsed with their predecessors until a synthesizable FSM is derived.

Initially, it is checked whether any two consecutive states v_{is} and v_{it} share the same value for all the input signals which means that the input burst at the transition e_i between them is empty. In order to tackle the above issue, the edge e_i is collapsed with all the transitions e_j driven by v_{it} and then v_{it} is removed as well. If there are any output transitions at e_j's burst, they are moved earlier by assigning v_{it} output vector to v_{is}. It should be noted that there are situations where e_j carries an empty input burst and its output burst cannot be moved neither earlier (at the transitions preceding v_{is}) nor later (at the e_j transitions) as it may be the case that both these transitions already have in their burst the same output signal. The above would result to an inconsistent BM-FSM which is not synthesizable.

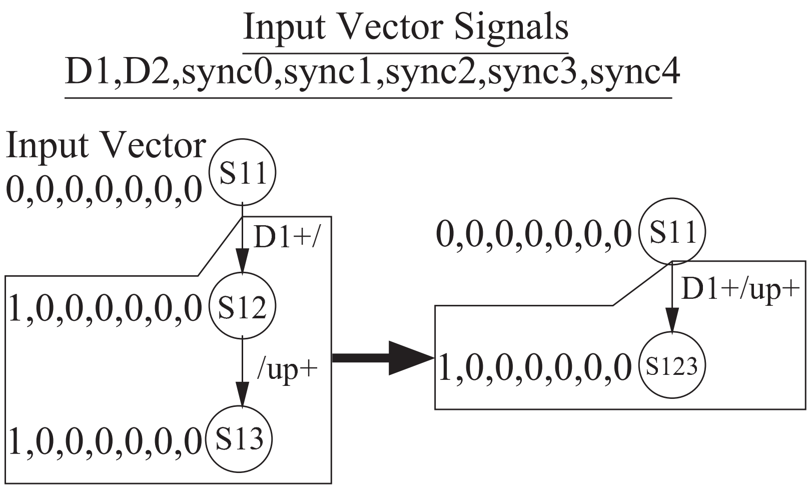


Figure 7.4: Collapsing Empty Input Bursts

The `count2` benchmark has two edges with empty input bursts which can be collapsed with their predecessors. One of them is shown in Figure 7.4. States S_{12} and S_{13} have the same input vectors which means that the input burst between them is empty.

7.5 Maximal Set Property Satisfaction

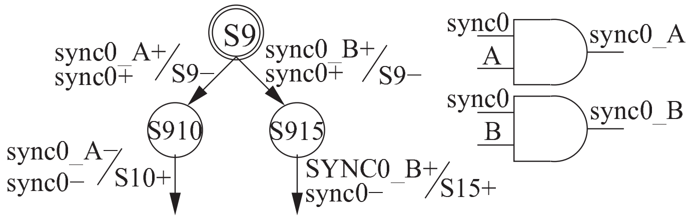


Figure 7.5: Satisfying the Maximal Set Property

Distributing the inputs and outputs of the initial STG to multiple BM-FSMs may incur non-deterministic choice states. The above is shown in Figure 7.3 where the choice state $S9$ does not satisfy the maximal set property, as $\{sync_0+\}\subseteq \{sync_0+\}$. To overcome the previous problem, the bursts following synchronized choice states should be reassembled at each one of the FSMs synchronized at this state.

Local copies of the input signals contributing to the synchronised choice states are generated by ANDing the synchronised state output with the input itself. If the specification is deterministic only a single burst of local input signals will be activated. Local signals will be deactivated at the state following the synchronised choice state as the state-output will be driven low. The above guarantees that the rising event of the local input copies is a dont care in all the states except the synchronised state itself and the falling edge is also a dont care event for all the states but the ones following the synchronised state.

An example is shown in Figure 7.5 where a $sync_0$ signal is used to poll A's and B's values generating $sync0_A$ and $sync0_B$ signals respectively. After either of the polled values rises a choice in the control flow is made and the polling signal $sync0$ falls deactivating the polling mechanism.

Chapter 8

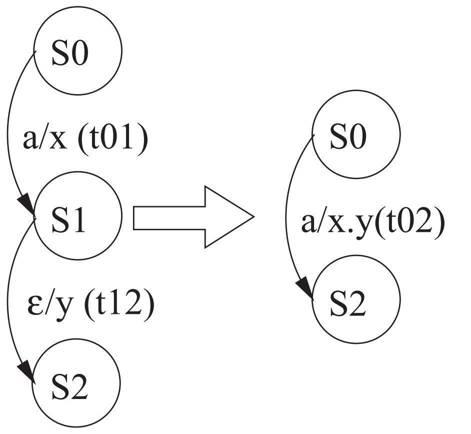
Optimizations

The previous chapters introduced the MSFSM control model and they presented flows which exploit MSFSM properties for transforming FCPTNets to interacting FSMs and vice versa. This Chapter introduces two classes of algorithms which optimize the number of states at the MSFSM level. The first class collapses concurrent states in a single FSM and multiple FSMs in a single one. The second class is based on the X-Compatibles, a novel property characterizing states and transitions across FSMs which can be exploited in the same manner as the conventional monolithic FSM compatibles to minimize the total number of states of an MSFSM system.

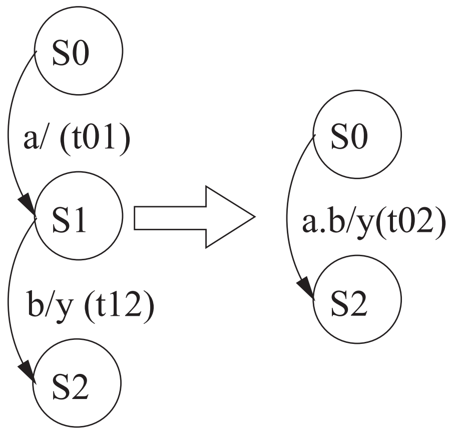
8.1 Horizontal and Vertical Collapsing

8.1.1 Vertical Collapsing

Vertical, *i.e.* Intra-FSM collapsing operations collapse consecutive transitions in a single FSM if the input-output behavior of the system is maintained. These operations leverage input functions which always evaluate to logic one and transitions with no associated output functions.



(a) Empty Input Transition Collapsing



(b) Empty Output Transition Collapsing

Figure 8.1: Vertical Collapsing Operations

In the former case, a transition with input function always evaluating to true can be collapsed with its previous transitions unless it belongs to a transition barrier. An example is shown in Figure 8.1a, where FSM waits at state S_0 for input function a to evaluate to true and then concurrently sets output x to logic one and moves to state S_1. Afterwards, FSM unconditionally and instantly moves to state S_2 while also setting output signal y to logic 1. As the system operates in input-output mode, it can only guarantee ordering between input and output events and hence state S_1 which dictates an ordering between output signal transitions is redundant and can be thus removed. The resultant FSM is shown in the right side of Figure 8.1a.

The second operation collapses two consecutive transitions if the first one has no associated output functions. An example is shown in Figure 8.1b, where FSM waits at state S_0 for signal a to go high and then moves to state S_1. While moving at state S_1, no output signal values are changed and hence the environment is not able to distinguish the current active state. As soon as input function b is satisfied FSM can move to state S_2 while also forcing output signal y to logic one. From environment's perspective, input signal events a and b are provided in order. However, as the system follows the input-output mode of operation the ordering of consecutive input events cannot be realized. Hence, output signal y will be set to logic one when both input events have been set, irrespectively from their order. The above observation allows removing state S_1 as it only preserves the ordering between consecutive input events. In the resultant system shown in the right side of Figure 8.1b, transitions t_{01} and t_{02} are collapsed to a single transition with input function the complement of the two previous input functions.

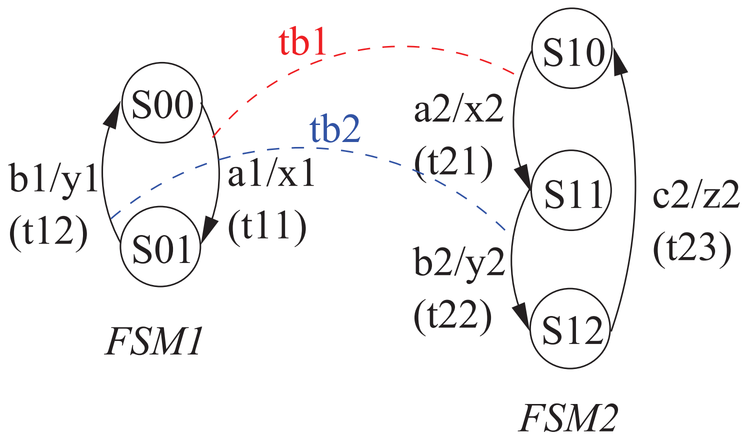
The aforementioned minimization cannot be performed in the case that the first transition contributes to a transition barrier, as in this case the ordering between input and output events could be violated.

The above minimizations can also be captured by state minimization algorithms of incompletely specified FSMs [88], as in both cases compatible state pairs (S_0, S_1) and (S_1, S_2) can be collapsed to a single state. However, as identifying the full compatible states set is intractable, the introduced operation of vertical collapsing is a linear complexity operation which acts as an enabler for the following horizontal collapsing operation.

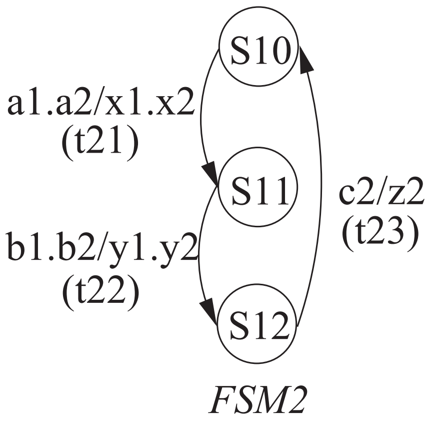
8.1.2 Horizontal Collapsing

The Horizontal, *i.e.* Inter-FSM collapsing operation is performed on an FSM pair (FSM_1, FSM_2) in the case where all the transitions of FSM_1 are synchronized with transitions of FSM_2. Input and output boolean functions of the synchronized transitions can be collapsed without changing the input/output behavior of the system, as each synchronized pair of transitions stems from a single PTNet transition.

An example of the horizontal collapsing operation is shown in Figure 8.2, where all of FSM_1's transitions are synchronized with FSM_2's transitions. Both of FSM_1's transitions cannot fire unless FSM_2's corresponding transitions are enabled. Hence, FSM_1 is redundant, as none of its transitions is independent.



(a) Initial MSFSM



(b) Collapsed MSFSM

Figure 8.2: Horizontal Collapsing Operation

Initial system's concurrency is limited between the input or the output boolean functions of the synchronized transitions. Hence, input functions a_1 and b_1 can concurrently fire with input functions a_2 and b_2 (The same holds for the concurrent output pairs (x_1, x_2) and (y_1, y_2). The resultant FSM maintains the above concurrency as a single FSM is able to model and implement concurrency between the signals in the input and the output functions.

8.2 X-Compatibles

In conventional synthesis of incompletely specified FSMs, two states can be collapsed into a single state if they are compatible, *i.e.* they lead to identical outputs values and their next states are compatible as well. In the MSFSM context, the states span across multiple FSMs. The current inter-FSM state minimization techniques are based on the interface signals between FSMs. In particular, if words between FSMs are never produced then the corresponding words signal values are used as DCs (sequential DCs [36]) in the conventional monolithic FSM minimization of the communicating FSMs.

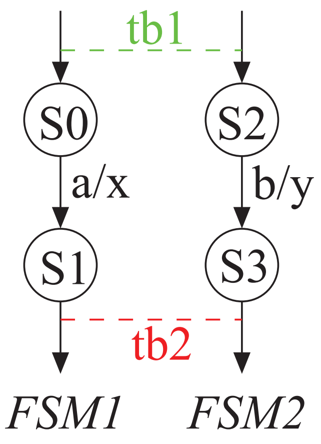
In this section we present Cross-Compatibles (X-Compatibles), which is a novel approach for minimizing interacting FSMs in a MSFSM system. X-Compatibles exploit the concurrency between FSM states and transitions. Although X-Compatibles were empirically identified by engineers as inter-FSM minimization opportunities [19], they were neither formalized nor automated in a synthesis tool. The reason for the above could be the lack of a mechanism to efficiently identify concurrency between FSM components. Additionally, in asynchronous systems synthesis, concurrency is traded with states by logic synthesis tools. However, the above trade-off is realized on the SG which is exponential in size when compared to the initial specification [27].

In this work, we formalize X-Compatibles and we introduce novel ways for extracting them and using them to optimize the MSFSM system.

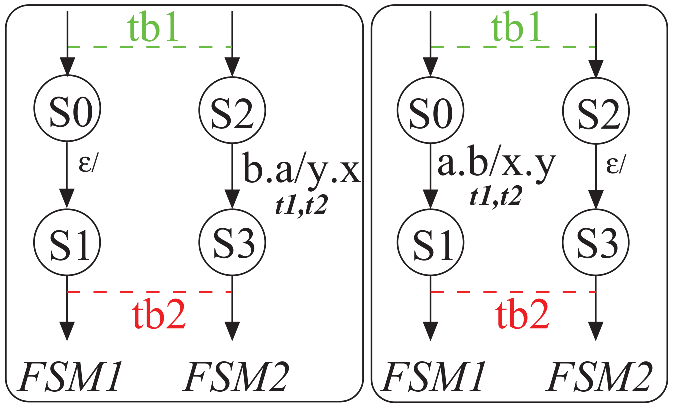
8.2.1 Definition

Definition 8.2.1. *A set of transitions are X-Compatibles if they are concurrent and mutually inclusive.*

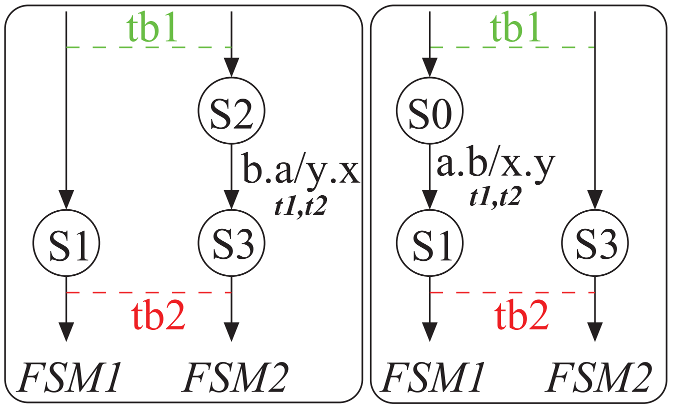
The concurrency states that the transitions can fire in any order. The mutual inclusiveness requires that a transition fires if the other transitions are activated as well. Hence, during MSFSM execution, either all the transitions fire or none of them. It should be stressed that the mutual inclusiveness allows transitions to fire in any order as long as all of them eventually fire.



(a) Initial MSFSM



(b) Minimized MSFSM Scenaria



(c) Minimized MSFSM Scenaria

Figure 8.3: X-Compatibles

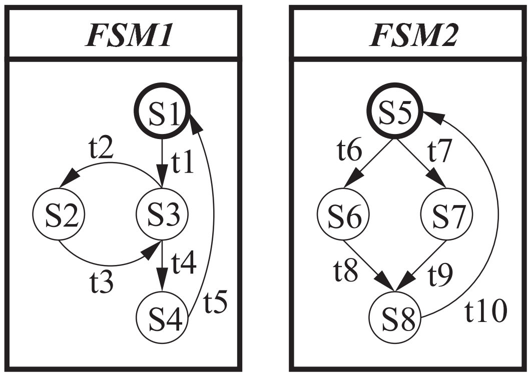
Figure 8.3b shows two FSM portions of an MSFSM system. Transition barriers tb_1 and tb_2 synchronize FSMs before states S_0 and S_2, and after states S_1 and S_3 respectively. Transitions $t1$ and $t2$ are X-Compatibles as each time transition barrier $tb1$ fires both of them will eventually fire (mutual inclusiveness) in any order (concurrency).

X-Compatible transitions can be exploited to minimize the MSFSM system. Figure 8.3a shows the previous system after collapsing the two X-Compatible transitions either to $FSM2$ or to $FSM1$. In either case, an ϵ transition substitutes the transition which was collapsed to the other FSM. Thus, vertical collapsing can be applied, resulting to the two possible systems shown in Figure 8.3c.

Before X-Compatibles minimization, the system had four states and modeled concurrency between signals of $FSM1$ and $FSM2$. After the minimization, the concurrency is reduced to inputs a and b and outputs x and y. However, the system size and presumably implementation area is reduced.

8.2.2 Extraction

Extracting the set of X-Compatibles in an MSFSM system requires analyzing the concurrency between states and the mutual inclusiveness. In a conventional interacting FSMs system, the problem of deciding the concurrency at the state level is intractable as the dependencies between the FSMs are encoded in the transition and output functions. On the other side, the MSFSM system can be transformed to a PTNet as described in Section 5 and at this level the concurrency between places can be efficiently (in the case of FCPTNet) decided in polynomial time [62]. With respect to mutual inclusiveness, the dominator theory [87] originally developed for compiler optimization is used to decide whether a set of states at different FSMs are always activated together during the system execution. The computational complexity of the dominator relation extraction is almost linear [69].



Transition Barriers	
t5	t10

Figure 8.4: X-Compatibles Extraction testcase

The steps of the X-Compatible extraction will be demonstrated using a representative test case shown in Figure 8.4. FSMs $FSM1$ and $FSM2$ are synchronized at

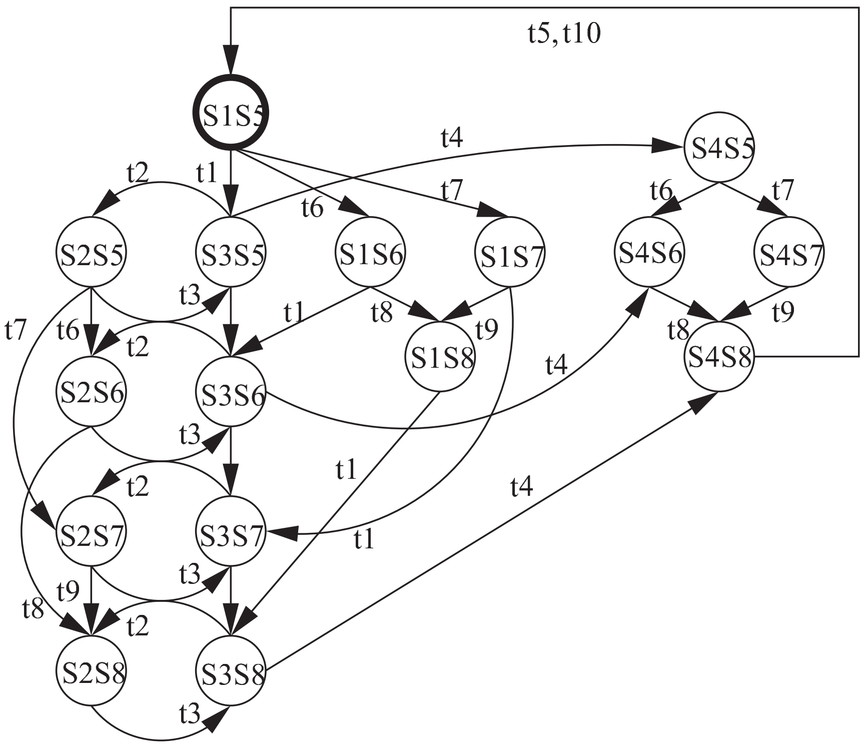


Figure 8.5: X-Compatibles Extraction testcase equivalent monolithic FSM

the transition barrier formed by transitions $\{t5, t10\}$. Optimizing the above MSFSM system requires either minimizing the number of states at each FSM separately or generating the MSFSM product and then minimize the resultant monolithic FSM. The former has the disadvantage that the FSM interactions are not exploited during minimization, whereas the latter suffers from the state explosion problem. The equivalent monolithic FSM of the MSFSM system is shown in Figure 8.5 where a state named S_iS_j stems from the composition of states S_i and S_j of the original FSMs.

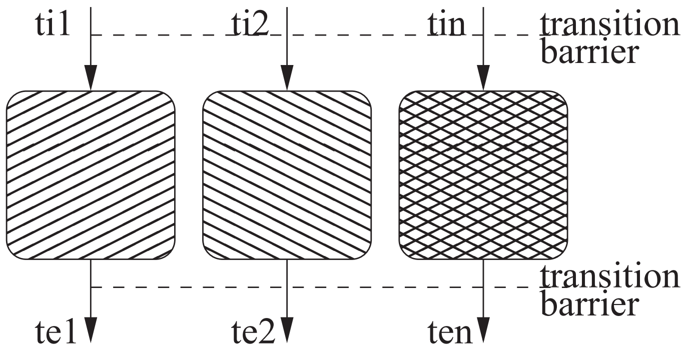


Figure 8.6: Heuristic X-Compatibles Extraction

In the first step of the X-Compatibles extraction, the MSFSM is transformed to an equivalent PTNet and the concurrency relations are computed for places and transitions. In the case where the resultant PTNet does not belong to the FCPTNet class and hence computing the concurrency relations becomes intractable, a heuristic approach is followed which identifies a portion of the X-Compatibles. In the above approach, transition barriers which are shared by a set of FSMs are used as boundaries into which X-Compatibles are extracted. This is shown in Figure 8.6 where three FSMs are synchronized at two transition barriers, $\{ti1, ti2, tin\}$ and $\{te1, te2, ten\}$. All three FSMs enter the former transition barrier in a synchronized manner and they execute the region of states and transitions following it, concurrently. Additionally, none of the FSMs is allowed to re-enter the region between the two transition barriers without the other FSMs due to the transition barrier $\{te1, te2, ten\}$. Hence the regions between the transition barriers a not only concurrent to each other but also mutually inclusive. The above regions will be from now on called as X-Compatible segments. In the testcase of Figure 8.4 the single transition barrier defines two X-Compatible segments, $\{S1, S2, S3, S4, t1, t2, t3, t4\}$ and $\{S5, S6, S7, S8, t6, t7, t8, t9, t10\}$ for $FSM1$ and $FSM2$ respectively.

Each X-Compatible segment has an arbitrary control structure. Hence it should be guaranteed that collapsing states and transition from different segments preserves the initial behavior of the system. In order to extract the X-Compatible states and transitions across different X-Compatible segments the dominator theory is exploited. Initially, the X-Compatible segments are transformed to control flow graphs [6] (CFGs), by transforming both FSM states and transitions to CFG nodes and following the FSM connectivity for the CFG as well. The transitions $\{ti1, ti2, ..., tin\}$ and $\{te1, te2, ..., ten\}$ become the underlying CFG entry and exit points respectively. The CFGs of testcase's FSMs are shown in Figure 8.7.

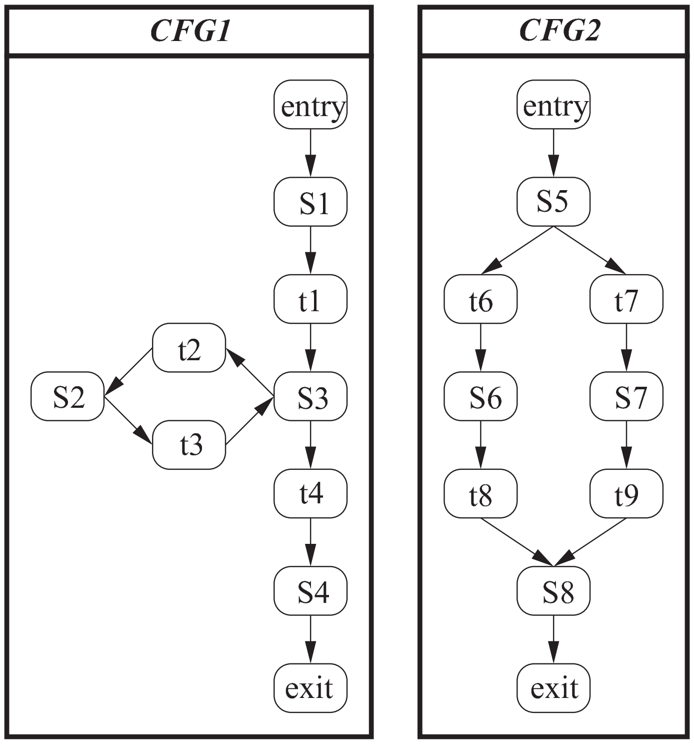


Figure 8.7: MSFSM transformation to CFGs

Figure 8.8 illustrates the dominator trees for the CFGs. Following each dominator tree from the entry point to the exit point, it is deduced which states and transitions will be eventually activated, independently from the input values each time the control flow is executed. Hence these paths contain sets of states and transitions which are mutually inclusive and which hence define X-Compatibles. The dominator trees of the testcase are shown in Figure 8.8. The paths which are unconditionally executed each time and which define X-Compatibles are enclosed in blue polygons.

8.2.3 X-Compatible Minimization

The X-Compatibles can be used to minimize the number of states of an MSFSM system. The above can be performed at the X-Compatible Segment level or at the X-Compatible states and transitions level. In the former case, the operations on the X-Compatible Segments target minimizing the total state space of the MSFSM

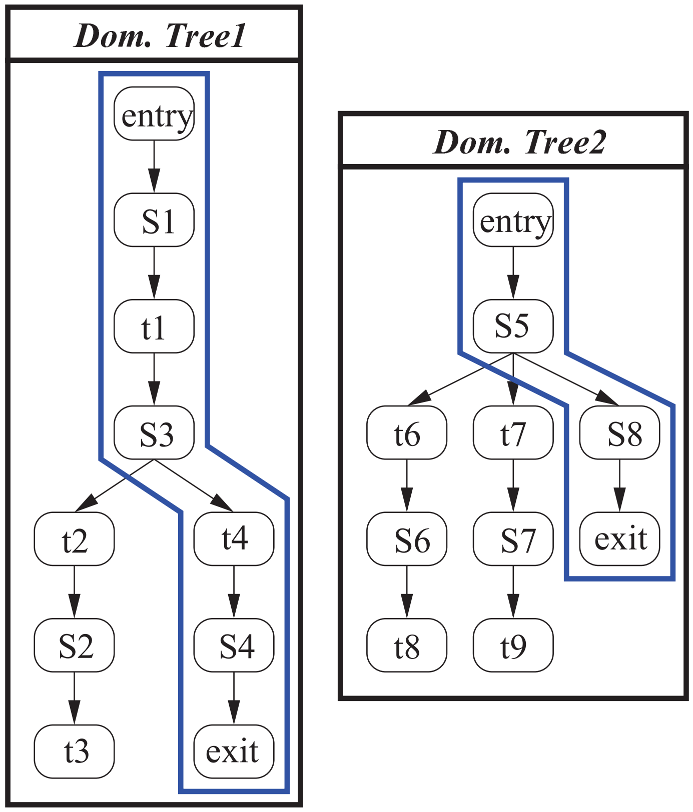


Figure 8.8: Testcase Dominator Trees and X-Compatibles

whereas in the latter the minimization is performed at a finer grain by collapsing X-Compatible states and transitions across FSMs.

The X-Compatible Regions Cascading approach moves X-Compatible Segments across FSMs. Although this method does not directly reduce the number of states of the MSFSM system, it operates as an enabler for the horizontal and vertical collapsing operations, by targeting a single FSM and removing all the non-synchronized transitions of this FSM. Additionally, it can be used to decrease the set of input and output signals of specific FSMs by removing segments which include transitions of these signals.

Figure 8.9 shows two FSMs, synchronized at transition barriers tb_1 and tb_2. There is an X-Compatible Segment in each FSM, defined by the aforementioned transition barriers. The transitions in each X-Compatible Segment are X-Compatibles with the transitions in the other X-Compatible Segment. The X-Compatible Segments

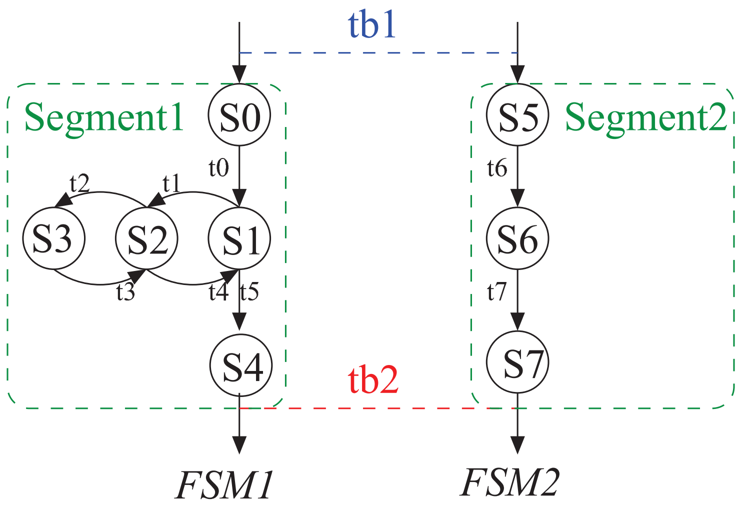


Figure 8.9: X-Compatibles Testcase

cascading approach moves either of the Segments to the other FSM, or swaps the Regions between FSMs if this reduces the per-FSM number of inputs and outputs.

If $Segment_2$ is cascaded at the end of $Segment_1$ then the resulting MSFSM is shown in Figure 8.10. It should be emphasized, that although the signal orders are preserved, transitions which were concurrent are now ordered which results to concurrency reduction. This reduction may or may not result to overall performance degradation, however it results to significant reduction of the states in the state space.

The X-Compatible Segments Product applies the conventional FSM composition algorithm on the target Segments. As with the cascading approach, the transition orders are preserved. However, the state number is significantly increased, as in worst case the product FSM's number of states equals to the product of number of states of the two FSMs.

Figure 8.11 shows the test-case of Figure 8.9 after generating the product of the X-Compatible Segments and using it to substitute $Segment_1$ while also removing $Segment_2$ from FSM_2. Product's total number of states is equal to the product of the two Segments number of states if the states preceding the exit transition barriers are maintained, *i.e.* eight. This great increase should prohibit the use of this approach

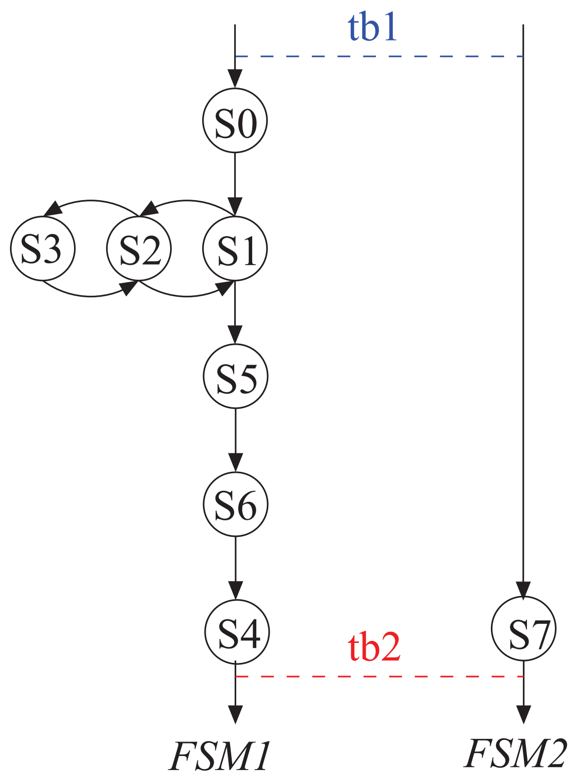


Figure 8.10: Cascading X-Compatible Segments

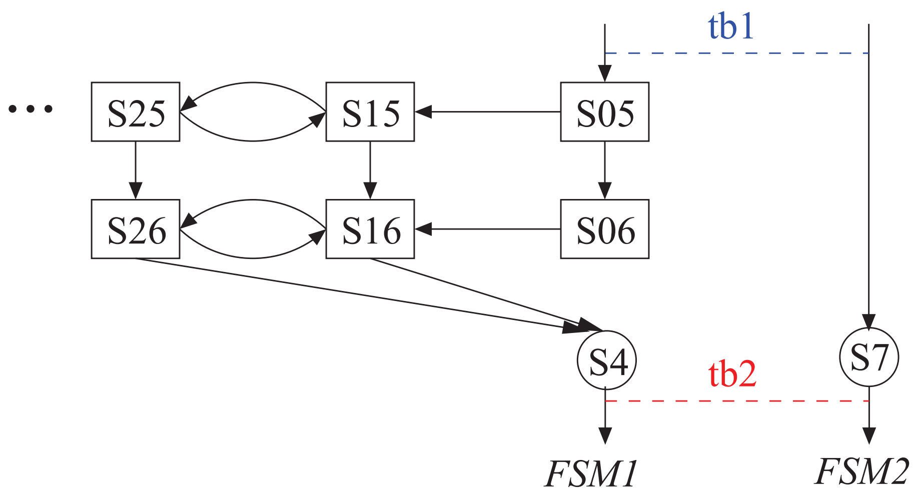


Figure 8.11: Generating the product of X-Compatible Segments

in FSM pairs which are not tightly synchronized unless preserving the concurrency between transitions is also the target of the optimization.

The X-Compatible Segments Folding Approach synchronizes the X-Compatible Segments in a finer grain by exploiting the X-Compatibility between transitions.

In this way the cost of applying the FSM composition algorithm is reduced when compared with the Segment product approach. The inserted synchronization has the form of transition barriers and it should be guaranteed that the system remains deadlock free. Hence, it should hold that whenever a transition in a new transition barrier is activated, the corresponding transition on the other FSM contributing to the same FSM is eventually activated as well, before its FSM reaches the exit transition barrier. The above is guaranteed if X-Compatible transitions are chosen in order when traversing the dominator trees.

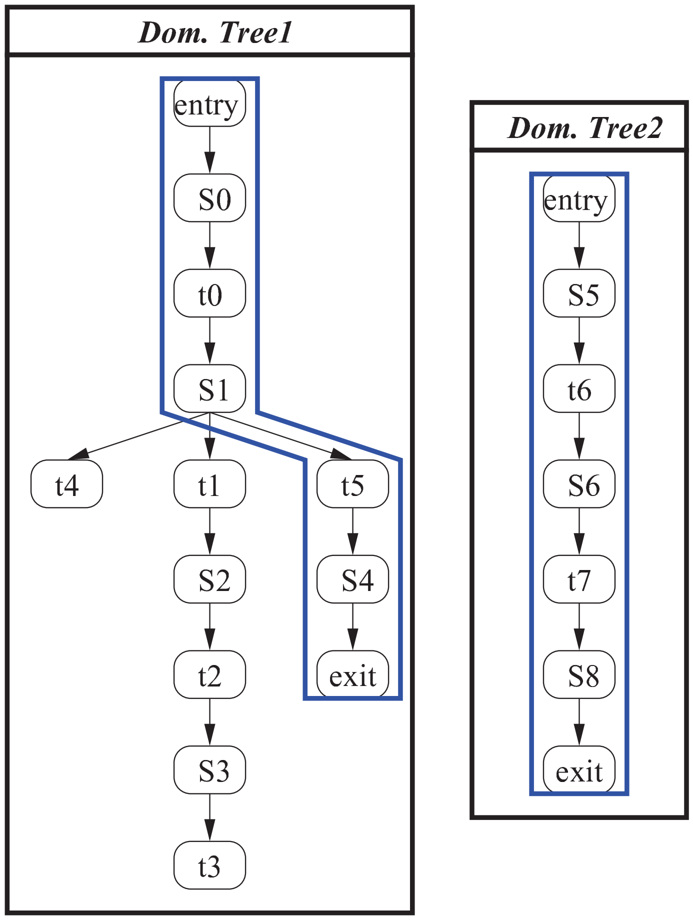


Figure 8.12: Dominator Trees of the testcase at Figure 8.9

Figure 8.12 shows the dominator trees which correspond to the MSFSM of Figure 8.9. The states which are activated for every scenario are the ones in the path from the state which succeeds the entering transition barrier to the state which precedes

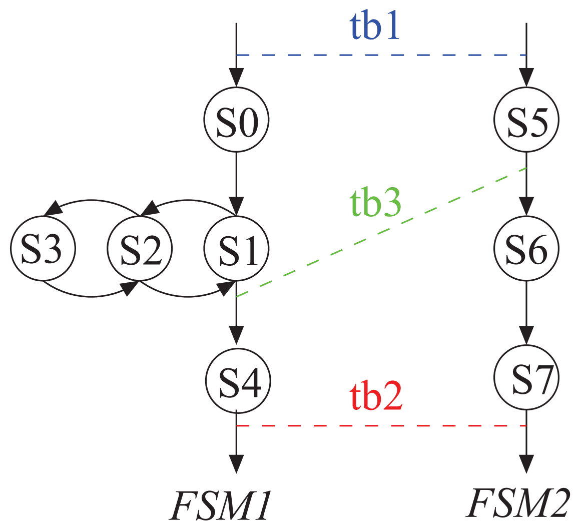


Figure 8.13: MSFSM after synchronizing in a finer grain

the exiting transition barrier. These paths are highlighted with a blue frame in both dominator trees. There are many opportunities for minimizing the total number of states. For instance, combining the X-Compatible transitions $t5$ and $t6$ into a new transition barrier results to the MSFSM shown in Figure 8.13. Extracting the new X-Compatible Segments and generating their product results to a significantly reduced number of states, *i.e.* five. However the folded MSFSM has also reduced concurrency when compared to the initial one as $S6$ and $t7$ are no more concurrent to the states and transitions of $FSM1$ preceding transition $t5$.

The total state space of the folded MSFSM is significantly reduced with respect to its initial one as the initial MSFSM had an equivalent monolithic FSM of fifteen states whereas the resultant MSFSM's has one with six states. In conclusion, the folding operation trades concurrency for reduced state space by examining the X-Compatible states and transitions in the dominator tree structure. Incrementally folding an MSFSM may result to an MSFSM system where FSMs can be collapsed with other FSMs as described in Section 8.1.2.

Chapter 9

Results

This Chapter presents the results of the algorithms introduced in this work. Initially, the PTNet to interacting FSMs algorithm, presented in Chapter 4 is evaluated with respect to memory footprint and execution time. Then, the synchronous synthesis flow presented in Chapter 6 is used as a basis to quantify the optimization algorithms introduced in Chapter 8. Finally, the asynchronous synthesis flow introduced in Chapter 7 is evaluated with not only synthetic scalable benchmarks but also with real asynchronous circuit specifications which are distributed in the asynchronous circuits community.

9.1 FCPTNet to Interacting FSMs Transformation

We now present experimental results of the FCPTnet to Interactive Synchronized FSM polynomial complexity flow presented, on a set of 25 PTnet benchmarks. The benchmarks used stem from the asynchronous circuit implementation field and represent asynchronous control circuit specifications, whereby PTnet events represent the assertion or deassertion of the relevant circuit signals. We contrast the state space and execution time of our flow to that of tool Petrify [25], which, in order to implement a PTnet specification, it expresses the latter's state space as a monolithic FSM, *i.e.* the event State Graph (SG).

Table 9.1 illustrates state space and execution time results of the flow presented in this paper, as well as the corresponding results of the Petrify tool. The first column of Table 9.1 classifies the PTnet of the relevant benchmark, between Marked Graph (MG), State Machine (SM), FC (Free-choice), or State-Machine Decomposable (SMD). The latter class refers to a General PTnet, coverable by S-Components.

Correctness of the transformation flows has been verified by converting the resultant MSFSMs, generated by the FCPTnet to MSFSM flow, back to PTnets, and validating their equivalence to the original FCPTnet specification. The latter was performed both by static PTnet analysis and dynamic simulation, through the use of an MSFSM simulation tool which has been developed [46].

Petrify results clearly indicate several cases of state explosion occurring while the tool explores the PTnet specification's state space, particularly for the more concurrent benchmarks of the set. As an example, `master_read2`, which consists of 52 places, requires a SG with $8{\times}10^7$ states for state-based, SG analysis! In comparison, the total number of states produced by our flow is 62, and correspond to a total of 10 Interacting, Synchronized FSMs. With respect to execution time, for a fair comparison, this includes solely the required time for state-space generation, without any additional time for SG processing required to generate the implementation. The execution time results illustrate significant runtime improvement over Petrify, with several orders of magnitude difference. We should emphasize that the resultant set of Interacting FSMs are not optimized in any way, for instance by running a state minimization algorithm per FSM.

Benchmark	**Original PTnet**			**Petrify [25]**		**MSFSM**			
	Type	Num. of Places	Num. of Trans.	Num. of States	Exec. Time(s)	Num. of FSMs	Total Num. of States	Total Num. of Trans.	Exec. Time(s)
`art_jordi_10_9`	MG	216	198	9.3×10^{17}	39	19	216	216	0.31
`two_pipes_weak3`	MG	23	14	160	0.03	7	28	28	<0.01
`two_pipes_weak6`	MG	47	26	$10\ \times10^{3}$	0.33	13	52	52	0.01
`two_pipes_weak9`	MG	71	38	6.5×10^{5}	1.54	19	76	76	0.02
`two_pipes_weak12`	MG	95	50	4.2×10^{7}	11.22	25	100	100	0.03
`two_pipes_arb3`	SMD	38	24	$1\ \times10^{3}$	0.17	11	61	61	0.01
`two_pipes_arb6`	SMD	62	36	6.9×10^{4}	1.19	23	115	115	0.02
`two_pipes_arb9`	SMD	86	48	4.4×10^{6}	19.2	33	158	158	0.04
`two_pipes_arb12`	SMD	95	50	2.8×10^{8}	550.4	25	100	100	0.07
`three_pipes_weak3`	MG	34	20	1.6×10^{3}	0.13	11	44	44	<0.01
`three_pipes_weak6`	MG	70	38	8.5×10^{5}	1.31	20	80	80	0.02
`three_pipes_weak9`	MG	106	56	4.3×10^{8}	38.4	29	116	116	0.04
`three_pipes_weak12`	MG	142	74	2.2×10^{11}	356.94	38	152	152	0.06
`three_pipes_arb3`	SMD	56	36	1.4×10^{4}	0.56	16	121	126	0.01
`three_pipes_arb6`	SMD	70	38	7.3×10^{6}	9.3	34	192	197	0.02
`three_pipes_arb9`	SMD	106	56	3.7×10^{9}	200.5	51	280	287	0.07
`three_pipes_arb12`	SMD	142	74	1.9×10^{12}	1045.7	69	335	340	0.11
`dup-4-pull.sl.3`	FC	121	112	155	0.62	17	1360	1496	0.88
`count2`	FC	19	16	27	0.01	5	41	46	<0.01
`dup-4-ph`	FC	133	123	169	0.95	20	1814	2014	0.93
`hybridf`	MG	26	16	80	0.03	8	48	48	<0.01
`master_read2`	MG	74	52	8×10^{7}	1.12	20	102	102	<0.01
`master_read`	MG	38	26	1882	0.17	10	62	76	<0.01
`mmu`	MG	20	16	174	0.24	5	31	31	<0.01
`tangram`	MG	98	92	426	0.48	6	348	348	0.06

Table 9.1: MSFSM Generation for FCPTnets for Concurrent Benchmarks

9.2 Logic Synthesis of Synchronous PTNets

This section presents the results of the methodologies developed in Chapter 6 and 8. In particular it uses the flow of Chapter 6 to generate a structural view of the synthesized circuit which allows evaluating it with respect to technology independent metrics of area and timing. Then using the above result as a basis, the optimization methodologies introduced in Chapter 8 are evaluated.

Algorithm 6 - Synchronous Synthesis Basic Flow

```
Require: PTNet
Ensure: Minimized Set of Interacting FSMs
 1: PTNet to MSFSM Transformation;
 2: for all $fsm \in$ MSFSM do
 3:    Vertical FSM Collapsing
 4: for all $fsm$ pairs $\in$ MSFSM do
 5:    Horizontal MSFSM Collapsing
 6: for all $fsm$ pairs $\in$ MSFSM do
 7:    if The Percentage of Inter-Synchronized Transitions $\geq \alpha$ then
 8:       for all Common Transition Barriers do
 9:          X-Compatibles Minimization
10:          Vertical MSFSM Collapsing
11:          Horizontal MSFSM Collapsing
```

Algorithm 6 shows a basic flow for synchronizing a PTNet to a minimized set of Interacting FSMs. Initially, the PTNet is transformed to an MSFSM using the flow introduced in Chapter 4 (line 1). Then, consecutive states and redundant FSMs are collapsed using the algorithms of Sections 8.1.1 and 8.1.2 respectively (lines 3 and 5). Finally, the X-Compatibles minimization algorithm is applied on each pair of FSMs which have more than a % of their transitions synchronized. The above constraint prohibits the state explosion problem from appearing during the synthesis.

A set of concurrent specification benchmarks, originally developed for asynchronous circuit implementation were synthesized using the clocked flow presented in this Section. Figures 9.2 and 9.1 illustrate the literal count and maximum logic levels results after optimizing the MSCFSMs resulting from the FCPTNet specifications

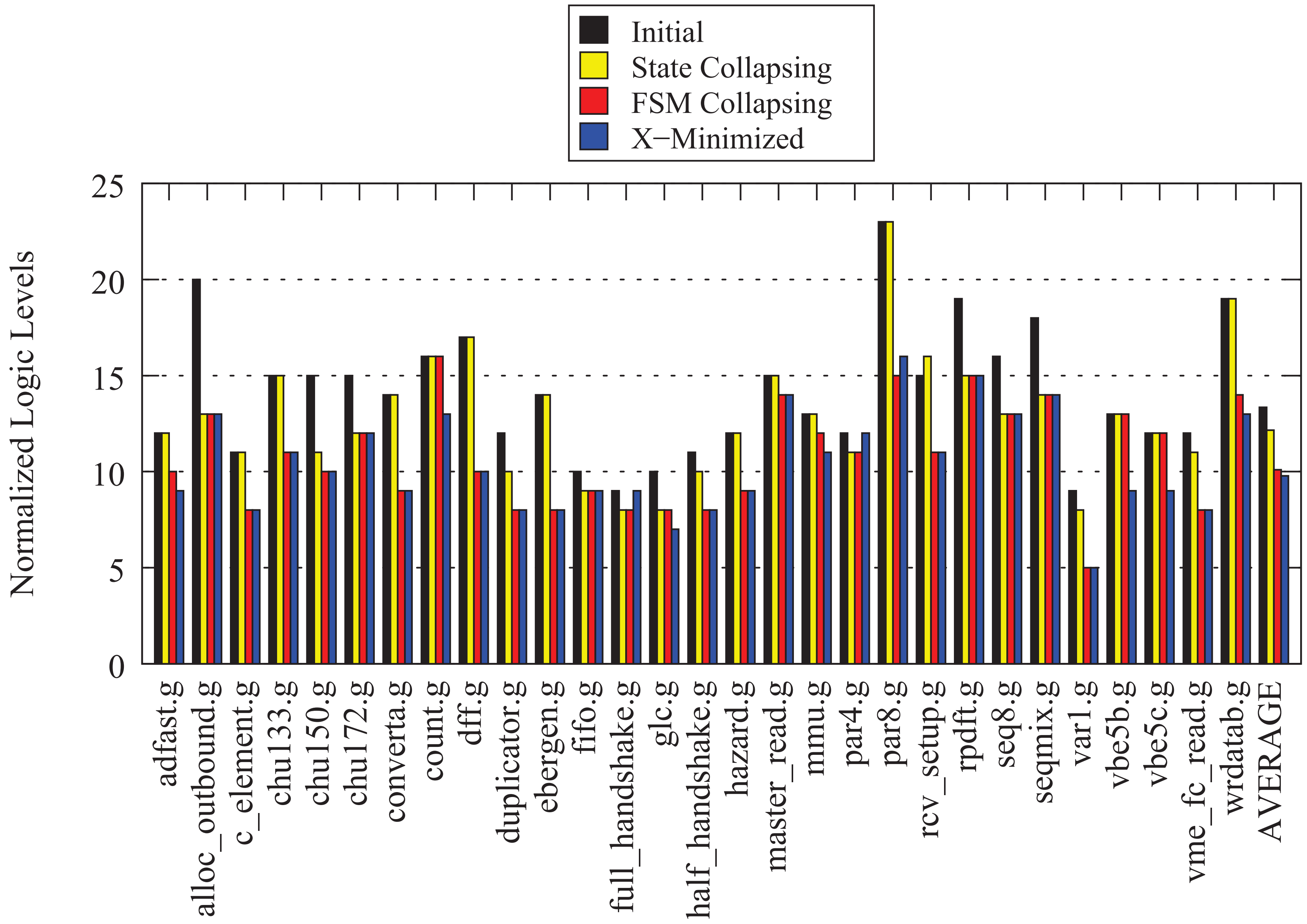


Figure 9.1: Synchronous MSFSM Logic Levels Results. *AVERAGE* column illustrates the average logic levels across all benchmarks for each optimization.

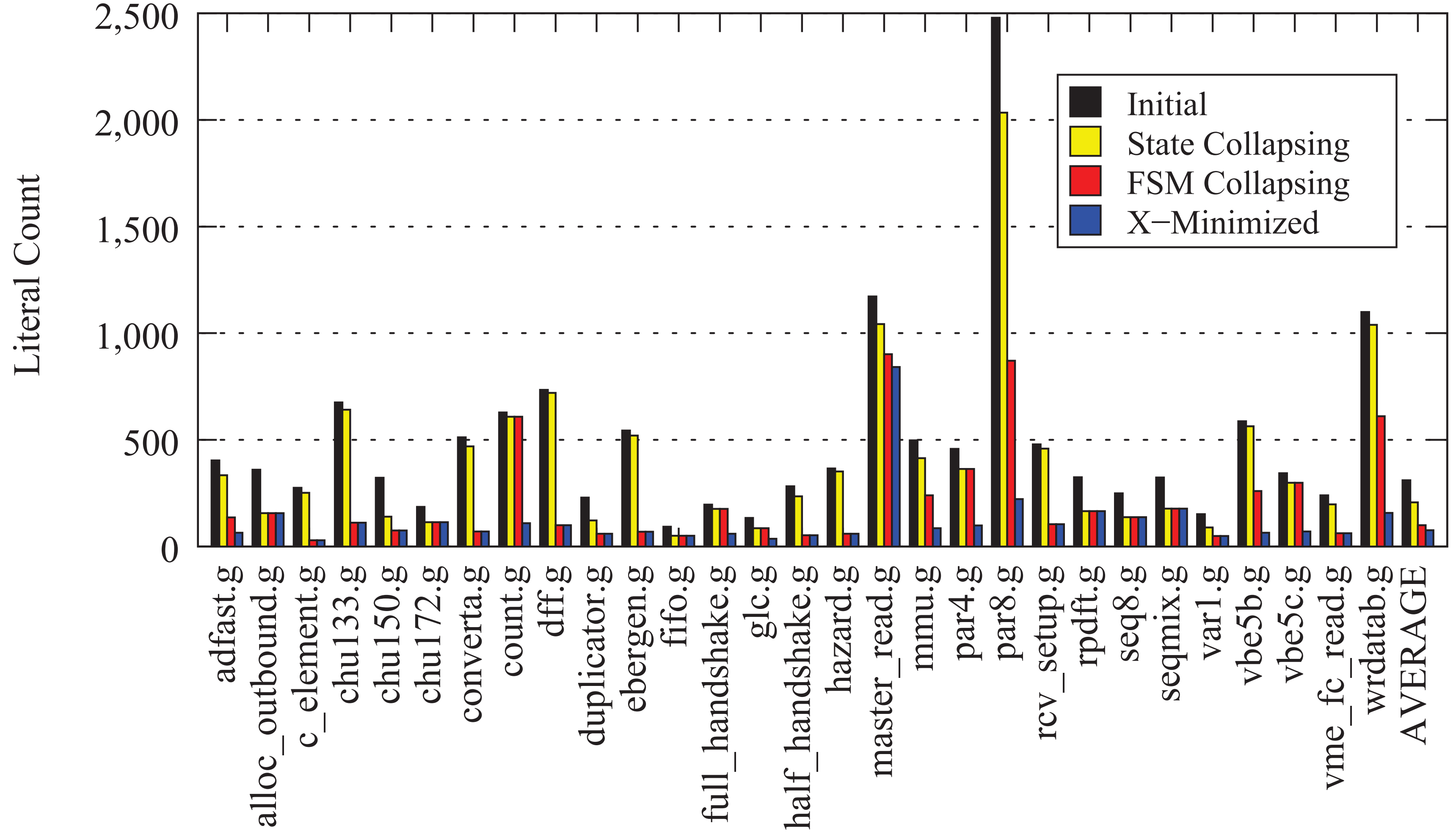


Figure 9.2: Synchronous MSFSM Literal Count Results. *AVERAGE* column illustrates the average literal count across all benchmarks for each optimization.

respectively It was assumed that the factor a was equal to 0 so that the circuits with the lowest possible area were generated.

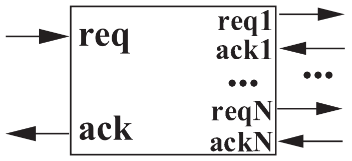
In each circuit there are four different bars. The black one is the initial circuit, the yellow one is the circuit after collapsing consecutive states in each separate FSM, the red one shows the result of collapsing FSMs which only show concurrency between inputs or between outputs and finally the blue bar illustrates the qualitative results after extracting the X-Compatibles and using them to further minimize the area and timing of the circuit at the expense of expressed concurrency. There are specification characteristics which specify the effectiveness of each optimization. FSM collapsing reduces significantly the literal count of the specifications which exhibit concurrency only between inputs or outputs. For instance, the `dff` benchmark which reduces to a single FSM and thus only exhibits input or output concurrency is effectively optimized by FSM collapsing. With respect to state collapsing, specifications which have consecutive not-synchronized states can be significantly optimized. As an example `seq8` and `seqmix` exhibit no actual concurrency as they are comprised by a linear (single predecessor and successor) sequence of places and transitions and thus it is only state collapsing which can optimize the above specifications.

The results illustrate that the optimization algorithms, combined in a single script, are able to significantly optimize all the exercised circuits in both area and performance. In particular, the average logic levels are reduced from 14 to 10 which is expected to be translated in clock period reduction after technology mapping the given circuit. With respect to area, a reduction of more than 80% is obtained on overage after applying all the optimizations in each circuit.

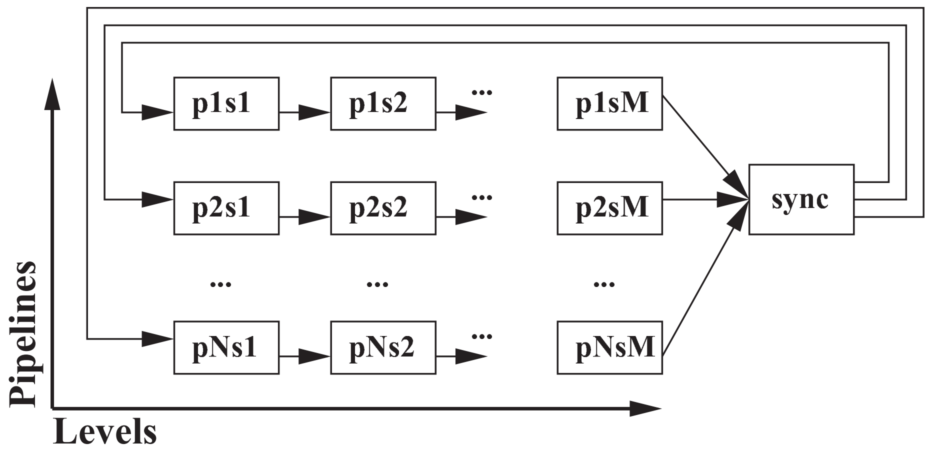
9.3 Logic Synthesis of Asynchronous PTNets

The algorithms described in Section 7 were realized in an EDA tool for logic synthesis of highly concurrent specifications, named *Expose*. In order to demonstrate the applicability and the effectiveness of the proposed synthesis flow, two scalable synthetic benchmarks and a set of control specifications, introduced in the bibliography, are used.

9.3.1 Parallel Handshakes Synthetic Benchmark



(a) Parallel Handshakes Block

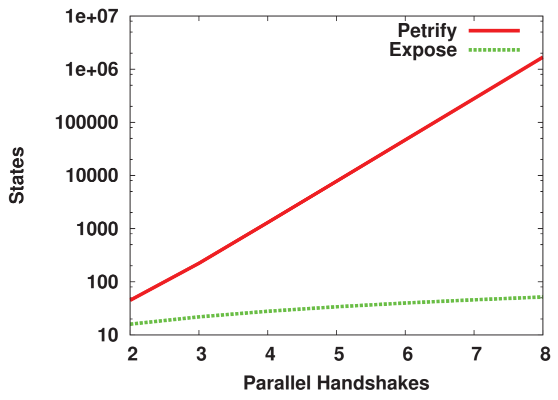


(b) Loosely Synchronised Pipelines Block

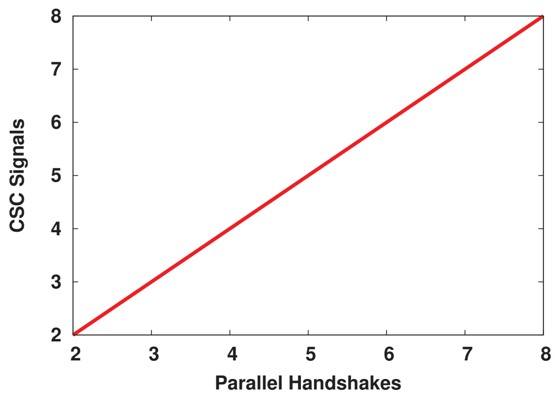
Figure 9.3: Synthetic Benchmarks

Figure 9.3a shows the block diagram of a controller which accepts a request event *req* and generates N concurrent requests independently acknowledged by N corresponding ack_i signals.

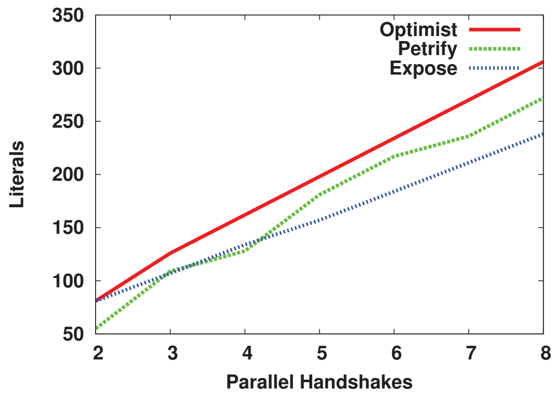
The Petrify synthesis flow requires the generation of the SG, which grows exponentially w.r.t. PTNet, as shown in Figure 9.4a. On the contrary, the state space



(a) State Space (**Log.cale**)



(b) Inserted CSC Signals

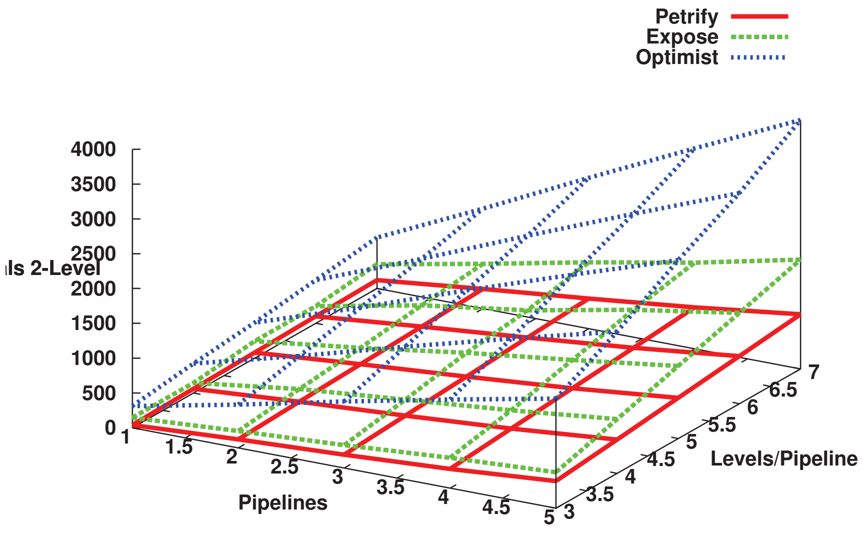


(c) Petrify, Optimist and Expose L.C.

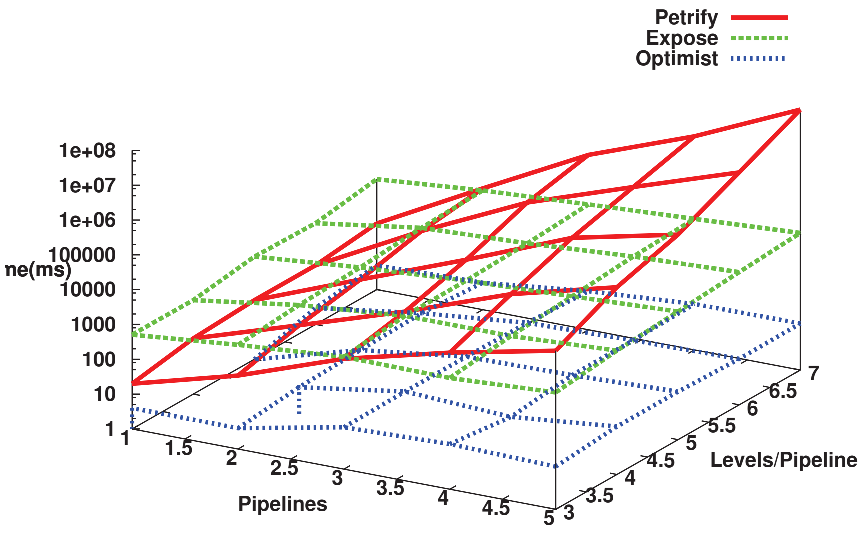
Figure 9.4: Parallel Handshakes Synthetic Benchmark

generated for the MSFSM flow grows linearly in size (y-axis is in logarithmic scale). It should be stressed that Petrify fails to synthesize controllers handling more than 8 parallel handshakes.

Figure 9.4b shows the number of CSC signals inserted in the SG to disambiguate states with matching encodings. Signals equal to the number of handshakes are required to satisfy the CSC property, negatively affecting the quality of the resultant implementation, as shown in Figure 9.4c. Expose, on the contrary, is not affected by the increase in required CSC signals and thus delivers low cost implementations.



(a) Literal Count Comparison



(b) Execution Time Comparison (**Log. Scale**)

Figure 9.5: Loosely Synchronized Pipelines Synthetic Benchmark

9.3.2 Loosely Synchronized Pipelines Synthetic Benchmark

Figure 9.3b shows a synthetic benchmark comprised of N parallel pipelines, each one having M stages.

The pipelines are loosely synchronised with a *sync* signal, meaning that the request signals of the M stage of all pipelines should be asserted to set *sync*, but only one of them resets it.

Figures 9.5a and 9.5b show the area and execution time results respectively for pipelines with variable number of parallel pipelines and stages per pipeline. As expected, Optimist generates implementations in negligible execution time. However, the circuit size grows linearly with the size of the specification as synthesis is not exploited. Petrify on the other side, explores a large solution space, resulting to highly optimized circuits. However, Petrify's execution time explodes, making it almost impossible to synthesize larger specifications. Expose aims at exploring the state space residing between the direct mapping and the synthesis approaches. As synthesis targeting area is performed for each one of the decomposed BM-FSMs, the total area is significantly lower than the one generated by Optimist. However, the area is higher than Petrify, as the latter examines the total state space. It is expected that composing Expose's FSMs in a single FSM (as Petrify's SG) will produce results close to the ones of Petrify, at the expense of exponential execution time. Finally, Expose enjoys low execution times, as its underlying algorithms are polynomial and the resultant FSMs are small enough to be synthesised by Minimalist.

9.3.3 Asynchronous Circuits Benchmarks

Figure 9.2 shows that Expose can also derive low area implementations for asynchronous controllers with relatively small state space. `count2` and `monkey` cannot be synthesized by Petrify due to an irreducible conflict and a tool crash respectively. For three of the designs, Expose shows the best results, as a single FSM suffices to model the specification. For the remaining benchmarks Expose shows the same trend as with the parallel pipelines synthetic benchmark, producing results between the ones of Petrify and the ones of Optimist.

Benchmark	**Petrify** [25]	**Optimist** [94]	**Expose** (thiswork)
`alloc-outbound`	66	258	30
`c3`	12	198	13
`count2`	N/A	302	178
`dff`	44	304	20
`duplicator`	93	228	111
`full`	44	264	102
`half`	43	198	148
`monkey`	N/A	1148	181
`rpdft`	34	214	25
`semi-decoupled`	86	242	208
`vbe6a`	132	732	237

Table 9.2: Asynchronous Circuits Benchmarks

Chapter 10

Conclusions

The aim of this work was to raise the abstract level of designing and synthesizing hardware control systems with concurrency. A novel control formalism was introduced which not only allows manually specifying concurrent specifications, but also acts as an intermediate model for synthesizing existing concurrent systems described as PTNets and for verifying properties relevant to concurrency for systems described as interacting FSMs. More specifically, the three most significant contributions of this thesis are the following.

The first contribution is a polynomial complexity flow for transforming an event-driven FCPTnet into a state-based Interacting FSMs model. This flow tackles the deficiencies of the PTnet and monolithic FSMs models, *i.e.* state explosion and efficient representation of concurrency, by acting as bridge between the these two models. The key to the flow is the definition of a new formalism for Interacting FSMs, which exposes the inter-FSM synchronizing interactions, using a set of synchronization primitives, *i.e.* Wait States and Transition Barriers. The new flow can be used for algorithms which need to explore the state space of a PTnet specification, *e.g.* PTnet implementation. Experimental results, on a set of 25 PTnet benchmarks from the field of asynchronous circuit design, demonstrate a significant reduction in both state space and execution time for our approach, when compared to the corresponding results of the Petrify tool.

The second contribution is the introduction of two viable paths for synthesis of concurrent specification to synchronous and asynchronous logic. The synchronous path assumes that transitions are synchronized with a global clocking signal whereas the asynchronous path generates burst-mode FSM specifications satisfying the fundamental mode operation mode. These two paths can handle an arbitrary well formed FCPTNet and they are realized in a new logic synthesis tool, named Expose. Comparative experimental results of our tool, Expose, coupled with Minimalist for BM-FSM synthesis, and existing tools Petrify and Optimist, which perform state-based STG synthesis and STG direct-translation illustrate significant reduction in execution time for synthesis coupled with LC results close to the those of Petrify, and significantly better than Optimist's.

The third contribution is a set of optimizations which operate on the MSFSM and which are confluent with the final implementation with respect to area and performance metrics. Hence, more efficient implementations can be obtained without generating the full state space of the system.

10.1 Future Work

The research conducted in this work has revealed new directions for the field of logic synthesis.

The first direction is the combination of synchronous and asynchronous concurrent control logic synthesis. In Chapters 6 and 7 algorithms were introduced to tackle the problem of synthesis of synchronous and asynchronous concurrent specifications respectively. However, there exists a solution space in which the solutions are partly synchronous and partly asynchronous. Examining the above solutions could result to implementations more efficient than the purely synchronous or asynchronous ones.

The second direction is the exploitation of the concurrency relations during synchronous and asynchronous logic synthesis. In this work it was assumed that the user targets the lowest possible area without preserving the initial concurrency as long as the input/output behavior is maintained. In a new flow the user could dictate which concurrency relations should be maintained during logic synthesis and thus the logic synthesis procedure should come up with a solution which trades the remaining concurrencies for area and/or performance.

The third direction, is the support of speed independent asynchronous circuits. In this work, MSFSMs are transformed to burst-mode FSMs which satisfy the fundamental mode assumption. However, the MSFSM can also be synthesized to speed independent implementations. The above enhancement would require introducing new encoding algorithms as the ones applying to the burst mode FSMs do not guarantee the correct operation of the implementation under the input/output mode.

Bibliography

[1] Focalpoint switches in the datacenter. `http://www.fulcrummicro.com/product_library/applications/datacenter.pdf`.

[2] Fulcrum white paper on scalable networks. `http://www.fulcrummicro.com/product_library/applications/clos.pdf`.

[3] Opus, asynchronous dsp core. `http://www.octasic.com/en/tech/opus_dsp.php`.

[4] Synopsys DC. `http://www.synopsys.com/tools/implementation/rtlsynthesis/dcgraphical/Pages/default.aspx`.

[5] OpenMP Application Program Interface, 2011.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[7] Semiconductor Industry Association. The international technology roadmap for semiconductors, 2009 edition, 2009.

[8] David Bañeres, Jordi Cortadella, and Mike Kishinevsky. Dominator-based partitioning for delay optimization. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, GLSVLSI '06, pages 67–72, New York, NY, USA, 2006. ACM.

[9] Edwards Bardsley. The balsa asynchronous circuit synthesis system, 2000.

[10] Kamel Barkaoui and Jean-François Pradat-Peyre. On liveness and controlled siphons in petri nets. In *Application and Theory of Petri Nets*, pages 57–72, 1996.

[11] Luca Benini, Giovanni De Micheli, Antonio Lioy, Enrico Macii, Giuseppe Odasso, and Massimo Poncino. Computational kernels and their application to sequential power optimization. In *Proc. ACM/IEEE Design Automation Conference*, pages 764–769, 1998.

[12] Krzysztof Biliński, E. L. Dagless, Jonathan M. Saul, and Marian Adamski. Parallel controller synthesis from a petri net specification. In *Proc. European Conference on Design Automation (EDAC)*, 1994.

[13] A. Bink and R. York. Arm996hs: The first licensable, clockless 32-bit processor core. *Micro, IEEE*, 27(2):58 –68, mar. 2007.

[14] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. Mis: A multiple-level logic optimization system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(6):1062 – 1081, november 1987.

[15] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1984.

[16] Josep Carmona, Jordi Cortadella, and Enric Pastor. A Structural Encoding Technique for the Synthesis of Asynchronous Circuits. *Fundam. Inf.*, 50:135–154, February 2002.

[17] Josep Carmona et al. Synthesis of Asynchronous Hardware from Petri Nets. In *Advances in PetriNets, LNCS 3098*, pages 345–401. Springer, 2004.

[18] Tiberiu Chelcea, Steven M. Nowick, Andrew Bardsley, and Doug A. Edwards. A burst-mode oriented back-end for the balsa synthesis system. In *DATE*, pages 330–337, 2002.

[19] Chih-Tung Chen and Kayhan Küçükçakar. High-level scheduling model and control synthesis for a broad range of design applications. In *ICCAD*, pages 236–243, 1997.

[20] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 326–337, London, UK, UK, 1993. Springer-Verlag.

[21] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. *Theor. Comput. Sci.*, 147:117–136, August 1995.

[22] Sue-Hong Chow, Yi-Cheng Ho, TingTing Hwang, and C. L. Liu. Low power realization of finite state machines - a decomposition approach. *ACM Trans. Design Autom. Electr. Syst.*, 1(3):315–340, 1996.

[23] T.-A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[25] J. Cortadella et al. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. Inf. and Syst.*, E80-D(3):315–325, March 1997.

[26] J. Cortadella et al. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[27] Jordi Cortadella, Michael Kishinevsky, Steven M. Burns, Alex Kondratyev, Luciano Lavagno, Ken S. Stevens, Alexander Taubin, and Alexandre Yakovlev. Lazy transition systems and asynchronous circuit synthesis withrelative timing assumptions. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(2):109–130, 2002.

[28] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and technology mapping of speed-independent circuits using boolean relations. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(9):1221–1236, 1999.

[29] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos P. Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[30] Olivier Coudert. On solving binate covering problems. In *Proceedings of the IEEE Design Automation Conference*, pages 197–202, 1996.

[31] Philippe Coussy and Adam Morawiec. *High-Level Synthesis*. Springer-Verlag, 2008.

[32] Clifford E. Cummings. Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs, 2000. Synospsys User Group, Boston.

[33] Clifford E. Cummings. The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and BuildGates, 2002. International Cadence Usergroup Conference.

[34] Al Davis, Ken Stevens, and Bill Coates. The post office experience: Designing a large asynchronous chip. In *Integration, the VLSI Journal*, pages 409–418. IEEE Computer Society Press, 1993.

[35] Jorg Desel and Javier Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995. ISBN-10 0-521-01945-1.

[36] S. Devadas. Optimizing interacting finite state machines using sequential don't cares. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(12):1473 –1484, dec 1991.

[37] S. Devadas, Hi-Keung Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. Mustang: state assignment of finite state machines targeting multilevel logic implementations. *IEEE Transactions on Computer-Aided Design*, 7(12):1290 –1300, dec. 1988.

[38] S. Devadas and A.R. Newton. Decomposition and Factorization of Sequential Finite State Machines. pages 148 –151, nov. 1988.

[39] Srinivas Devadas and A. Richard Newton. Decomposition and factorization of sequential finite state machines. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8(11):1206–1217, 1989.

[40] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(7):798–807, 1989.

[41] D Edwards. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[42] J. Esparza, E. Best, and M. Silva. Minimal deadlocks in free choice nets. 89-09, Universidad de Zaragoza, departamento de ingenieria electrica e informatica, Research, May 1989. NewsletterInfo: 33.

[43] Francisco Fernández-Nogueira and Josep Carmona. Logic synthesis of handshake components using structural clustering techniques. In *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 188–198, 2008.

[44] R.M. Fuhrer and S. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools.* Springer-Verlag GmbH, 2001.

[45] Robert M. Fuhrer, Bill Lin, and Steven M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 604–611, Washington, DC, USA, 1995. IEEE Computer Society.

[46] Labros Gkavogiannis. MSFSMsim A Tool for MSFSM Functional Analysis, Jun 2010. Diploma Thesis, University of Crete, Greece.

[47] Steve Golson. State machine design techniques for Verilog and VHDL, 1994. Synospsys User Group, San Jose.

[48] J. Grabowski. On the analysis of switching circuits by means of petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 14(15):611–617, Dec, 1978.

[49] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, 1996. ISBN 0-7923-9746-0.

[50] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[51] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, 1966.

[52] Juris Hartmanis. Symbolic analysis of a decomposition of information processing machines. *Information and Control*, 3(2):154–178, 1960.

[53] John Hopcroft. An nlogn Algorithm for Minimizing States in a Finite Automaton. Technical Report STAN-CS-71-190, Stanford University, Computer Science Department, January 1971.

[54] Li Jiao, To-Yat Cheung, and Weiming Lu. On liveness and boundedness of asymmetric choice nets. *Theor. Comput. Sci.*, 311:165–197, January 2004.

[55] Peter Kemper. Linear time algorithm to find a minimal deadlock in a strongly connected free-choice net. In *Application and Theory of Petri Nets*, pages 319–338, 1993.

[56] Peter Kemper. O(PT) - Algorithm to Compute a Cover of S-components in EFC-nets. Technical report, Informatik der Universitt Dortmund, 1994.

[57] Peter Kemper and Falko Bause. An efficient polynomial-time algorithm to decide liveness and boundedness of free-choice nets. In *Application and Theory of Petri Nets*, pages 263–278. Springer, 1992.

[58] Peter Kemper and Falko Bause. An efficient polynomial-time algorithm to decide liveness and boundedness of free-choice nets. In *Application and Theory of Petri Nets*, pages 263–278, 1992.

[59] Shai Rotem Ken, Ken Stevens, Ran Ginosar, Peter Beerel, Chris Myers, Kenneth Yun, Rakefet Kol, Charles Dike, Marly Roncken, and Boris Agapiev. Rappid: An asynchronous instruction length decoder. In *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 60–70, 1999.

[60] Joep Kessels and Ad Peeters. The tangram framework (embedded tutorial): asynchronous circuits for low power. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 255–260, New York, NY, USA, 2001. ACM.

[61] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, June 1978.

[62] Andrei Kovalyov and Javier Esparza. A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs. In *In Proc. of the International Workshop on Discrete Event Systems (WODES*, pages 1–6, 1995.

[63] P. Kudva, G. Gopalakrishnan, H. Jacobson, and S.M. Nowick. Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes. In *Proc. ACM/IEEE Design Automation Conference*, pages 77–82, June 1996.

[64] Prabhakar Kudva, Ganesh Gopalakrishnan, and Hans Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proceedings of the 33rd annual Design Automation Conference*, DAC '96, pages 67–70, New York, NY, USA, 1996. ACM.

[65] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proc. ACM/IEEE Design Automation Conference*, June 1992.

[66] Luciano Lavagno et al. Solving the state assignment problem for signal transition graphs. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–572, 1992.

[67] Luciano Lavagno, Cho W. Moon, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. An efficient heuristic procedure for solving the state assignment problem for event-based specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 14(1):45–60, 1995.

[68] Dong-Ik Lee, Shinzo Kodama, and Sadatoshi Kumagai. Decomposition Algorithms for Live and Safe Free Choice Nets. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 78, 1995.

[69] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[70] ZhiWu Li and MengChu Zhou. On siphon computation for deadlock control in a class of petri nets. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 38(3):667–679, May 2008.

[71] Jørn Lind-Nielsen et al. Verification of Large State/Event Systems using Compositionality and Dependency Analysis. In *TACAS'98 Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, 1998.

[72] M.P. Marcus. Derivation of maximal compatibles using boolean algebra. November 1964.

[73] Alan Marshall, Bill Coates, and Polly Siegel. The design of an asynchronous communications chip. *IEEE Design & Test of Computers*, 11:8–21, 1994.

[74] P. M. Mattheakis, C. P. Sotiriou, and P. A. Beerel. A Polynomial Time Flow for Implementing Free-Choice Petri-Nets. In *Proc. International Conf. Computer Design (ICCD)*, 2012.

[75] J.C. Monteiro and A.L. Oliveira. Implicit fsm decomposition applied to low-power design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(5):560 – 565, oct. 2002.

[76] Tadao Murata, Boris Shenker, and Sol M. Shatz. Detection of ada static deadlocks using petri net invariants. *IEEE Trans. Software Eng.*, 15(3):314–326, 1989.

[77] Chris J. Myers. *Asynchronous circuit design*. Wiley, 2001.

[78] Faten Nabli, François Fages, Thierry Martinez, and Sylvain Soliman. A boolean model for enumerating minimal siphons and traps in petri nets. In *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming*, CP'12, pages 798–814, Berlin, Heidelberg, 2012. Springer-Verlag.

[79] S.M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford, 1995.

[80] Steven M. Nowick, Montek Singh, Sergey Rylov, Alexander Rylyakov, and Jose A. Tierno. An adaptively-pipelined mixed synchronous-asynchronous digital fir filter chip operating at 1.3 gigahertz. *Asynchronous Circuits and Systems, International Symposium on*, 0:84, 2002.

[81] Steven M. Nowick, Kenneth Y. Yun, and David L. Dill. Practical asynchronous controller design. In *ICCD*, pages 341–345, 1992.

[82] E. Pastor et al. Structural Methods for the Synthesis of Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108 –1129, November 1998.

[83] A Percy. Time is right for clockless design. *EE Times*, 2010.

[84] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, 1962.

[85] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[86] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1):82–126, 1996.

[87] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. *Managing Requirements Knowledge, International Workshop on*, 0:133, 1959.

[88] June-Kyung Rho et al. Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines. *IEEE Transactions on Computer-Aided Design*, 13(2):167–177, 1994.

[89] June-Kyung Rho and F. Somenzi. Don't care sequences and the optimization of interacting finite state machines. *IEEE Transactions on Computer-Aided Design*, 13(7):865–874, June 1994.

[90] M. Schlansker, J. Tourrilhes, Y. Turner, and J.R. Santos. Killer fabrics for scalable datacenters. In *Pric. International Conference on Communications (ICC)*, pages 1 –6, May 2010.

[91] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-vincentelli. Sis: A system for sequential circuit synthesis. Technical report, 1992.

[92] V. Sklyarov and I. Skliarova. Specification and synthesis of parallel hierarchical finite state machines for control applications. In *Asian Control Conference, 2009. ASCC 2009. 7th*, pages 1085–1090, 2009.

[93] Douglas J. Smith. *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog.* Doone Publications, 1998.

[94] Danil Sokolov, Alexandre V. Bystrov, and Alexandre Yakovlev. Direct Mapping of Low-Latency Asynchronous Controllers from STGs. *IEEE Transactions on Computer-Aided Design*, 26(6):993–1009, 2007.

[95] J. Sparsø and S.B. Furber. *Principles of asynchronous circuit design: a systems perspective.* Springer Netherlands, 2001.

[96] Maxim Teslenko and Elena Dubrova. An efficient algorithm for finding double-vertex dominators in circuit graphs. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 406–411, Washington, DC, USA, 2005. IEEE Computer Society.

[97] Michael Theobald and Steven M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1130–1147, 1998.

[98] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, 1969.

[99] Anrong Wang, ZhiWu Li, Jianyuan Jia, and MengChu Zhou. An effective algorithm to find elementary siphons in a class of petri nets. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 39(4):912 –923, july 2009.

[100] Huey-Yih Wang and Robert K. Brayton. Multi-level logic optimization of fsm networks. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, ICCAD '95, pages 728–735, Washington, DC, USA, 1995. IEEE Computer Society.

[101] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. Gadara: dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 281–294, Berkeley, CA, USA, 2008. USENIX Association.

[102] Dominic Wist et al. Signal transition graph decomposition: internal communication for speed independent circuit implementation. *IET Computers & Digital Techniques*, 5(6):440–451, 2011.

[103] Wayne Wolf. The fsm network model for behavioral synthesis of control-dominated machines. In *Proc. ACM/IEEE Design Automation Conference*, pages 692–697, New York, NY, USA, 1990. ACM.

[104] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits. II. (automatic synthesis). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(2):118–132, 1999.

[105] K.Y. Yun and D.L. Dill. Automatic synthesis of extended burst-mode circuits. I. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(2):101 –117, feb. 1999.