School of Electrical, Electronic & Computer Engineering

Conditional Partial Order Graphs

Andrey Mokhov

Technical Report Series

NCL-EECE-MSD-TR-2009-150

September 2009

Contact:

andrey.mokhov@ncl.ac.uk

Supported by EPSRC grants EP/C512812/1 and EP/F016786/1

NCL-EECE-MSD-TR-2009-150

School of Electrical, Electronic & Computer Engineering,

Merz Court,

University of Newcastle upon Tyne,

Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Newcastle University

School of Electrical, Electronic and Computer Engineering

Conditional Partial Order Graphs

by Andrey Mokhov

PhD Thesis

September 2009

Contents

List of Figures

List of Tables

Abstract

This work presents a new formal model for specification and synthesis of microcontrol circuits. The model, called Conditional Partial Order Graph, captures concurrency and choice in a system's behaviour in a compact and efficient way. It is especially beneficial for a class of systems which have many behavioural scenarios defined on the same set of primitive events or actions, e.g. CPU microcontrollers. The key feature of the model is its ability to specify systems in a compact functional form as opposed to the existing models which either have a direct event traces representation or use an explicit notion of states and transitions between the states.

In this approach a system is specified with a set of scenarios, each of them being a partial order of events. The scenarios are further composed into a single graph containing all the partial orders in a functional form. This superposition is obtained by assignment of Boolean conditions to the events and dependencies between them, hence the name of the model. As a result the specification has different levels of abstraction for control and data paths: control flow is represented with partial orders of events, while data path is modelled at the level of Boolean functions. Such separation helps to avoid exponential explosion in the size of system specification which happens in other models.

At the stage of microcontroller synthesis the obtained graph is structurally mapped into logic equations without the explicit exploration of the system state space, thereby leading to algorithms with high time and memory efficiency.

The model has potential applications in the area of microcontrol synthesis and brings new methods for modelling concurrency into the domain of modern and future processor architectures. Several application examples are presented, namely basic handshake components and processor microcontrollers, as well as phase encoding circuits used in digital communication channels.

As a result of this work several synthesis, verification, optimisation and mapping tools have been developed to facilitate specification and synthesis of microcontrollers using the proposed methodology. The tools have been successfully incorporated into the Workcraft framework for visualisation and simulation support.

Acknowledgements

Many people have contributed to this work and to my education in the area of computing science and engineering.

My supervisor, Alex Yakovlev, introduced me to the world of asynchronous systems and gave me invaluable guidance during my PhD research. I am also grateful to Gennady Desyatkov, who supervised my undergraduate study and all other teachers who gave me the background knowledge in computing science.

I would like to thank Crescenzo D'Alessandro whose research motivated me to study the partial order theory. This eventually led to creation of Conditional Partial Order Graphs which are the main topic of this thesis. Special thanks to Danil Sokolov, Victor Khomenko, Fei Xia, Maciej Koutny, Alex Bystrov, and my other colleagues for educational and inspiring discussions, necessary criticism and help.

Lastly but, perhaps, most importantly I am immensely grateful to my family for their support and love. Many thanks to all my friends who have always been sources of education, inspiration and motivation for me.

This research was supported by the ORS Awards Scheme grant, the EPSRC grants EP/F016786/1 and EP/C512812/1.

Chapter 1

Introduction

Electrical engineering has come a long way in its development: from the early era when it was easier to design a machine rather than to build it, to the modern era when the technology is far more advanced and capable than the available design methodologies. Charles Babbage described his Analytical Engine in detail but could not build it during his lifetime, and many great projects of his followers (Percy Ludgate, Leonardo Torres y Quevedo, Vannevar Bush, and others [86]) remained on paper due to difficulties in their physical realisation. Finally, works by John von Neumann [16] and Howard Aiken [5] led to the emergence of first computers — more than a hundred years after the Babbage's initial design. Although the basic architectural principles of today's computers remain the same, their complexity has grown enormously: transistor counts are exponentially increasing due to smaller feature sizes and higher consumer demand for functionality [3]. System design and validation have become extremely difficult; they exceeded the bounds of human comprehension a long time ago, thereby calling for extensive Electronic Design Automation (EDA) support [90].

The increasing gap between manufacturing capability and ability of available EDA tools leads to low productivity, lengthy time-to-market, costly system verification, diagnostics and test [3]. This creates plenty of research opportunities and challenges in the area of design methodology and system modelling. The key point of leverage to approach these challenges concerns management of the increasing system complexity. The

efficiency of EDA tools in this respect depends more on the efficiency of the underlying mathematical models rather than on the computational power of the machines running them: the lack of adequate models equates the lack of adequate tools. There are a number of requirements a model should satisfy in order to adequately describe the behaviour of a large System-on-Chip (SoC) or Network-on-Chip (NoC): it should capture concurrency and multiple choice, it should be expressive enough to cover a wide range of solutions for different optimisation criteria, and yet be manageable, i.e. it should not overrefine the specification with unnecessary details. The latter requirement becomes especially important in designing asynchronous (or self-timed) systems [96] which do not rely on global clocking for synchronisation, thus leading to massive parallelism and, in turn, to difficulties in system modelling and validation (see Section 2.1 for additional details). Long debates between the synchronous and asynchronous camps have recently came to a peaceful resolution: modern SoCs exploit advantages of both design paradigms (a good example is the Globally Asynchronous Locally Synchronous (GALS) architecture [17]), hence the next requirement to the model: it should be able to handle asynchronous systems (note that synchronous systems are easier for modelling and can be considered as a subclass of asynchronous ones when all concurrent actions are executed in single steps).

To date there are several methodologies for asynchronous systems design, e.g. [103] and [94]. Some approaches such as Tangram (or Haste) [102][104] and Balsa [1][10] use CSP-like hardware description languages (HDLs) and syntax-directed translation for synthesis. They are not well suited for control logic specification because they describe the entire system as a collection of processes and channels; control is implicit in them. Other models such as Burst-mode Finite State Machines (FSMs) [78], as well as Petri Nets (PNs) and Signal Transition Graphs (STGs) [25] are able to capture concurrency and choice at a very fine level and are more suitable for control logic design: they produce more compact and faster circuits than the methods based on syntax-directed translation from HDLs [94] (Subsections 2.2.1-2.2.3 contain the background on FSMs and STGs). However, these models are built on the explicit enumeration of all the event traces and causal relations of a system and their applicability is limited to microcontrollers with a

small state space as demonstrated by a set of examples in Chapter 3.

In this thesis we present a new model and design methodology for system specification and synthesis which is targeted at systems with many behavioural scenarios defined on the same set of primitive events or actions, e.g. CPU microcontrollers. The key features of the model are: ability to describe systems in a compact visual form without the explicit listing of all its behavioural scenarios, and structural synthesis methods which significantly improve performance of the whole design flow. These features make the model very efficient for representation and management of causal information in hardware and EDA software. The next section outlines the proposed methodology and overall contribution of the thesis.

1.1 Methodology and contribution

The proposed methodology is based on the Conditional Partial Order Graph (CPOG) model introduced recently [71] which is a formalism that is capable of capturing behavioural patterns of a system in a functional form. A CPOG is a superposition of a set of *partial orders* [56] (see definition in Section 4.1) which can be extracted from it by providing the corresponding codewords as shown in Figure 1.1 (centre). It can be regarded as a custom associative memory for storing cause and effect relations within a predefined set of events.

There are different kinds of systems which can be described with this model. For example, a CPU microcontroller executes partial orders (or *instructions*) of primitive computational steps (or *microinstructions*) defined on a set of data path operational units, see Figure 1.1 (top). The order is determined by an *instruction code* — a combination of logical conditions presented to the controller by the environment [62]. To this end, the microcontroller can be seen as an entity which communicates with two parts of the environment: one part is the source of condition signals (an instruction decoder) and the other part is a set of controlled objects with a request-acknowledgement interface (data path operational units which execute the microinstructions). Thus the condition signals dynamically reconfigure the microcontroller according to the instruction being executed.

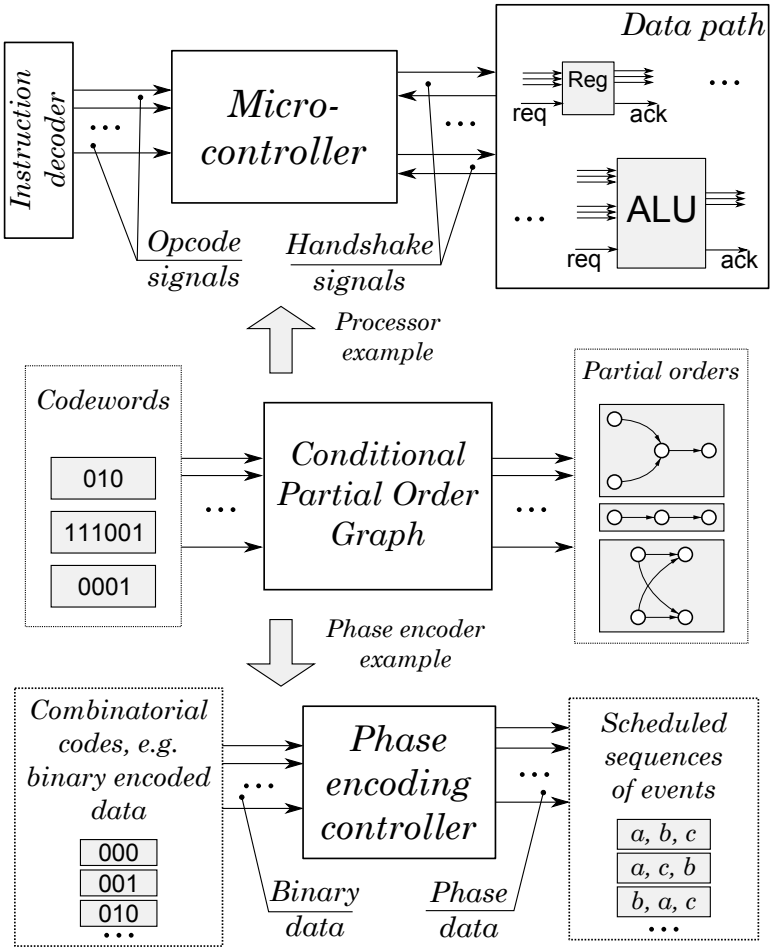


Figure 1.1: Examples of dynamically reconfigurable microcontrollers

Microcoded control synthesis presented in [62] is applicable to this class of systems. However, it is based on synchronous FSMs and stores the event orders separately in look-up tables, thus having a limited degree of parallelism and certain area penalties.

Another class of controllers suitable for CPOG specification is a family of phase encoders [65]. A phase encoder is a circuit that converts data between two conceptually different information domains, see Figure 1.1 (bottom). The first one corresponds to the combinatorial data encoding, e.g. binary or m-*of*-n *encoded* data symbols [105]. The second domain is comprised of sequences of events ordered in time. CPOGs are capable of specifying such controllers without the explicit representation of all the contained behavioural scenarios thereby avoiding the combinatorial explosion of the specification. Section 8.2 contains a detailed study of phase encoding circuits.

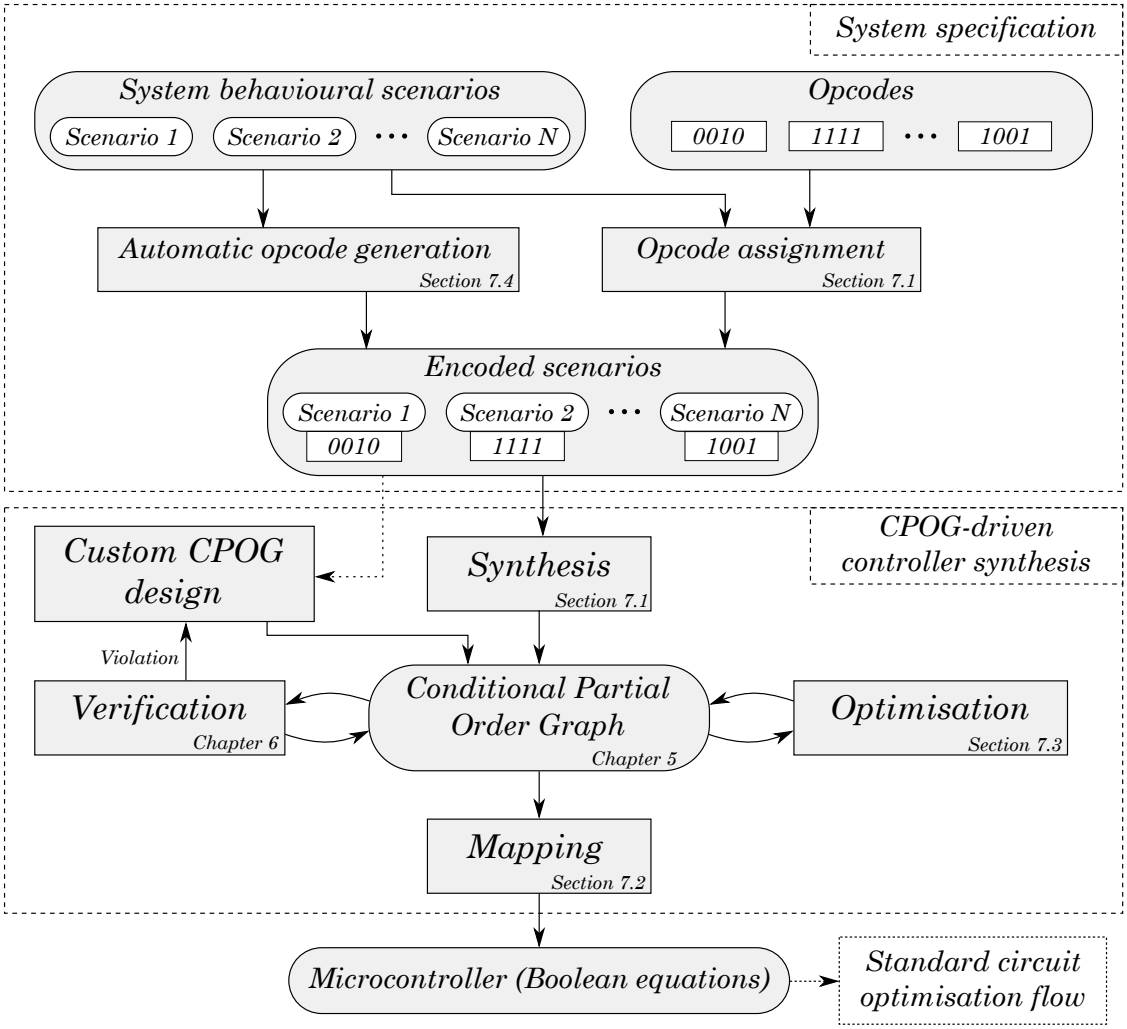


Figure 1.2: CPOG-based flow for control specification and synthesis

Figure 1.2 shows the proposed CPOG-driven flow for automated synthesis of microcontrollers. At first, the designer has to specify all the execution scenarios of the controller. A scenario is a schedule of basic events or actions that have cause and effect relationships between them. Consider a simple scenario of adding two numbers which consists of actions {a, b, c, d}:

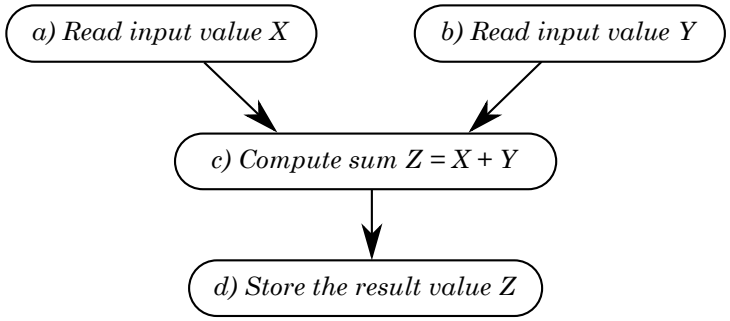
a) Read input value X;

b) Read input value Y;

c) Compute sum $Z = X + Y$;

d) Store the result value Z.

One can see that action c depends on actions a and b (there is no way to compute sum Z without having values X and Y), and action d, in turn, cannot happen until action c has been completed. This is captured by the following diagram:



The diagram depicts a *partial order* [56], a basic precedence relation on the set of actions. Every scenario of the system is specified independently from the others.[1] This makes our approach substantially different from the STG- or FSM-driven approaches that require the designer to specify the controller as a whole, often resulting in complicated and incomprehensible specifications. This is demonstrated by a simple controller with three basic scenarios used as an example in Section 3.1.

As soon as all the scenarios of the controller have been defined the flow proceeds to the stage of CPOG synthesis described in Section 7.1. At this stage all the scenarios are combined within a single mathematical structure — a Conditional Partial Order Graph (introduced formally in Chapter 5). Later on, every scenario can be extracted from this structure by providing its *opcode*. Therefore it is necessary to set up a correspondence between the scenarios and their opcodes. This correspondence is called an *encoding scheme*. Sections 7.1 and 7.4 discuss several encoding schemes which are often encountered in practice, and provide an automated procedure for the optimal encoding of the scenarios. A CPOG can also be obtained manually or by using custom synthesis methods which do not guarantee the correctness of the synthesised CPOG. In this case the CPOG has to be verified; automated procedures for verification of various CPOG properties are presented in Chapter 6.

[1] There are cases when many scenarios match a certain pattern and can be specified together in a functional form without their explicit enumeration as demonstrated in Subsection 8.2.3.

The obtained CPOG can be mapped into an interconnection of logic gates to produce the physical implementation of the microcontroller as explained in Section 7.2. The area and speed of the microcontroller depends on the size and structural properties of its CPOG representation. Therefore a CPOG can undergo various optimisation procedures which exploit similarities between the original scenarios and functional characteristics of their encodings (see Section 7.3). The obtained gate-level implementation of the controller can be further processed using the standard circuit design tools, e.g. it might require *technology mapping* for a particular gate library, or a custom technology-dependent performance optimisation which are out of the scope of this work.

The CPOG-based synthesis flow requires the designer's involvement only at the stage of system specification and scenario encoding. Therefore the rest of the stages can be automated and the designer might even be unfamiliar with CPOGs and the underlying theory. It is also important to note that all of the CPOG-related stages (synthesis, optimisation, and mapping) rely only on structural methods and do not require exploration of the entire controller state space or explicit enumeration of all its behavioural scenarios, which results in a high efficiency of the whole design flow.

1.2 Organisation of the thesis

This thesis is organised as follows:

Chapter 1 (Introduction) outlines the motivation behind this thesis and briefly discusses its main contributions (the CPOG model and the CPOG-based design flow).

Chapter 2 (Background) describes the main classes of asynchronous circuits, and introduces the FSM and STG models.

Chapter 3 (Motivation for the new model) presents several motivational examples which reveal limitations of the FSM and STG models for specification and synthesis of certain class of microcontrollers.

Chapter 4 (Background for CPOGs) gives definitions of the basic mathematical structures which form the backbone of the CPOG model: partial orders and directed acyclic graphs.

Chapter 5 (Conditional Partial Order Graphs) gives the formal definition of the static and dynamic CPOG models, discusses their properties, and introduces algebraic operations over CPOGs.

Chapter 6 (Verification) presents a set of SAT-based verification techniques that can be used for analysis of systems specified with CPOGs.

Chapter 7 (Synthesis and optimisation) addresses issues of CPOG synthesis from partial orders, optimisation of the synthesis result, and automated synthesis of instruction opcodes in the context of microarchitecture design.

Chapter 8 (Application examples) studies several design cases which demonstrate high efficiency of the proposed CPOG-driven methodology for specification and synthesis of microcontrol circuits.

Chapter 9 (Conclusions) summarises contributions of the thesis and outlines areas of future work.

Appendix (Tool) describes the set of developed software tools supporting different stages in the CPOG-based design flow.

Chapter 2

Background

This chapter introduces asynchronous systems, their basic properties and classes, and describes two models that are used for specification, verification and synthesis of asynchronous circuits: Finite State Machines (Subsection 2.2.1) and Signal Transition Graphs (Subsection 2.2.3). This chapter may be safely skipped if the reader is familiar with these concepts.

2.1 Asynchronous systems

Asynchronous, or self-timed, systems are systems without clocks, i.e. they do not synchronise all the internal events with the global clock signal. Instead, they operate under distributed control, with concurrent hardware components communicating and synchronising using local handshakes [99]. Asynchronous circuits have started to gain popularity both in industry and academia due to the following advantages [3].

Modularity

Asynchronous systems are constructed out of modular hardware components, each having a formally defined protocol for communication with others. The components may be implemented using different design approaches and silicon technologies, and are allowed to operate at different speeds. Due to their well-defined interfaces they can be safely combined into a correctly working system, thus meeting the need for 'correct by construction' designs of large-scale heterogeneous systems. By contrast, every part of a

synchronous system is physically tethered to the same clock signal and therefore must be fast enough to finish its computation task within a clock cycle; delay variations in any part of a synchronous system have global implications.

Low power consumption

In traditional synchronous designs almost half of power is consumed by the clock distribution tree leading to chip overheating and short battery life. Moreover, the power is often wasted on unproductive clock cycles: even if a synchronous component has no data to process in a particular clock cycle, it still dissipates dynamic power because it is clocked (meaningless data is being propagated through the combinational logic layers and latched in registers). Asynchronous systems have no clock tree, and they do not perform any unneeded computation; they are event-driven, i.e. they compute only when there is a request for that. In the absence of the requests, an asynchronous component consumes virtually no power because there is no switching activity [100].

Average-case performance

The clock frequency of a synchronous system is determined by its slowest component, with an added margin taking care of worst-case process, temperature, and data parameters. Therefore, under the normal operating conditions the system wastes a lot of time waiting for the current clock cycle to finish (so called *worst-case performance*). This problem can be alleviated to a certain extent by fine-grain partitioning of pipeline stages and splitting the main clock cycle into several sub-cycles, however, this significantly increases the design complexity. Asynchronous systems, on the other hand, are more flexible by their nature: a new computational cycle may start as soon as the previous one has been completed; there are no strict deadlines and performance of the system is determined by the current internal and external parameters, not the worst ones. Therefore, an asynchronous system may exhibit the *average-case performance* [11]. It should be noted that in order to exploit the average-case performance, asynchronous data-path components require completion detection which may be expensive in terms of area and power. Many asynchronous data-path methodologies [37][52][112] and completion detection techniques have been proposed [18][69][79][92].

Low electromagnetic emission

Due to the absence of clock asynchronous systems have lower levels of electromagnetic emission in comparison with synchronous ones. Switching activity is not concentrated near the edges of the clock signal resulting in evenly-spread, much less aggressive noise profile. This allows a higher degree of integration, reduces the need for electromagnetic shielding, and makes asynchronous systems advantageous for security applications (it is easier to extract information from a synchronous device by using the distinct clock patterns in its noise profile as a data flow reference).

Tolerance to process variation

Asynchronous systems are more tolerant to the increasing variation in process parameters caused by continuous technology scaling. Robust *delay insensitive* communication protocols [31][98][105] make no assumptions on the delay of wires or gates, which represent the major concern of synchronous systems. There are many asynchronous design styles varying in terms of their robustness and performance/area characteristics; they are discussed in the next subsection.

2.1.1 Classes of asynchronous circuits

There are several important classes of asynchronous circuits. The *Delay Insensitive* (DI) class is the most robust one with respect to process and environmental variations, as it makes no assumptions on the delay of wires or gates: DI circuits are designed to operate correctly with the *unbounded* gate and wire *delay models* [98]. The concept of delay insensitive circuits originates from [21]; formalisations were given in [98]. Unfortunately, the class of DI circuits is quite limited because they are very difficult to build out of simple gates [59] and usually have large area and performance overheads.

It is necessary to relax the DI requirements in order to build practical circuits out of simple gates. *Speed-independent* (SI) circuits assume the unbounded delay model for gates, while wire delays are considered to be negligible (or added to the gate delays). This means that whenever a gate changes its output, this change is immediately propagated along the wire [12][75].

A different approach to relaxation of the DI requirements is adopted in the class of *Quasi Delay Insensitive* (QDI) circuits. Instead of assuming zero wire delays, they rely on the concept of an *isochronic fork*: it is assumed that the difference in delays between the branches of a wire fork is negligible [58].

SI and QDI circuits are more practical and are widely used in asynchronous systems. Methods for specification and approaches to synthesis of implementation of these classes of circuits are discussed in the next section.

2.2 Behavioural models

This section introduces behavioural models used for specification of asynchronous circuits: Finite State Machines (FSMs), Petri Nets (PNs) and Signal Transition Graphs (STGs) – a special type of PNs where events are associated with signal transitions in an asynchronous circuit.

2.2.1 Finite State Machines

FSMs (also known as *finite state automata*) have been studied a long time ago [7] and dominate the area of specification and synthesis of sequential synchronous systems [62]. They can also be applied to asynchronous systems, in particular Burst-mode FSMs [78], however, specification of concurrent processes is very awkward and problematic with them, because FSMs use *interleaving semantics* to capture concurrency.

A FSM is composed of a finite number of *states*, *transitions* between these states, and *actions*. A transition indicates a state change and is described by a condition that has to be satisfied to enable the transition. An action is a description of an activity that is to be performed at a given moment (entering a state, executing a transition, etc). There is a direct correspondence between the states of a FSM, and the states of the modelled system. This often leads to problems in terms of concurrency specification. For example, to specify n concurrent events, the designer has to specify 2^n states each of them corresponding to a state, when a subset of these n events have already happened. There are 2^n different subsets of a set of n events, hence the number of required states. This

is a serious limiting factor for FSMs to be used for specification of large asynchronous systems. Chapter 3 contains several examples, demonstrating these limitations.

Petri Nets formally introduced in the next subsection are more suitable for specification of concurrency. They admit the *true concurrency semantics*, i.e. PNs can model concurrent events directly, without considering all possible sequences of their execution.

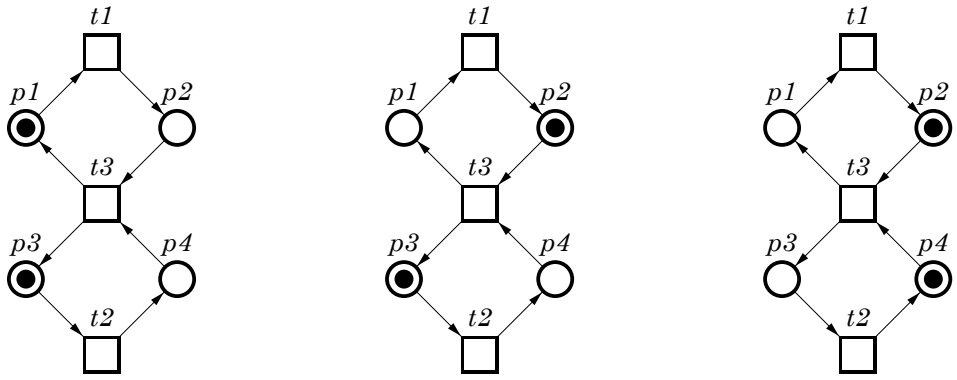
Self-Modifying Finite Automata

Many extensions of FSMs have been introduced in order to increase their expressive power and ease specification for certain classes of systems. One of such extensions, the Self-Modifying Finite Automaton (SMFA) model [44][91], has an interesting relation to this work. An SMFA is an ordinary FSM with the additional property of being able to modify itself during a transition from one state to another, e.g. it can add/delete states or transitions in itself. These self-modification abilities dramatically extend the model's expressive power but they do not help to deal with concurrency — inherently, an SMFA describes a sequential process and the only way to model concurrent behaviour is to use interleaving semantics. That is why they are not suitable for our purposes.

In the same way as the SMFA model is built around FSMs, the Conditional Partial Order Graph model is built around partial orders: one can say that a CPOG is a partial order that can be modified according to certain conditions — new events or dependencies can be added/deleted from it. Unfortunately, it is not possible to reuse any results from the SMFA theory in our work because of the fundamental differences between FSMs and partial orders. However, we think that the relation between SMFAs and CPOGs is interesting and worth mentioning.

2.2.2 Petri Nets

The *Petri Net* model, introduced half a century ago by Carl Adam Petri [80], is a well-studied formalism for modelling concurrent system behaviour. PNs are particularly useful for modelling asynchronous systems [25], because unlike FSMs they do not have the explicit notion of a global system state, therefore being better in combating the *state explosion problem* [101].



(a) Initial state (b) t1 has fired (c) t2 has fired

Figure 2.1: Evolution cycle of a Petri Net

Definition 2.1. A Petri Net is a quadruple $PN(P, T, F, m_0)$ where P and T are disjoint finite sets of *places* and *transitions* respectively, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, and $m_0 : P \to \mathbb{Z}_+$ is the *initial marking*. A PN is a directed bipartite graph [22] with nodes P and T forming its two disjoint sets of vertices and relation F containing arcs between them. $m_0(p)$ denotes the number of *tokens* that are initially contained in place $p \in P$.

Given a node $x \in P \cup T$, the set $\bullet x = \{y \mid (y, x) \in F\}$ is the *preset* of x, and the set $x\bullet = \{y \mid (x, y) \in F\}$ is the *postset* of x.

Figure 2.1(a) depicts a simple PN consisting of four places $P = \{p1, p2, p3, p4\}$ (denoted as circles ○) and three transitions $T = \{t1, t2, t3\}$ (denoted as boxes □) connected with a number of arcs (arrows ⟶). The initial marking is $m_0 = (1, 0, 1, 0)$: places {p1, p3} contain tokens (a place with a token is denoted as a marked circle ◉), while places {p2, p4} do not. The preset of place p2 is {t1}, the postset of transition t3 is {p1, p3}, etc.

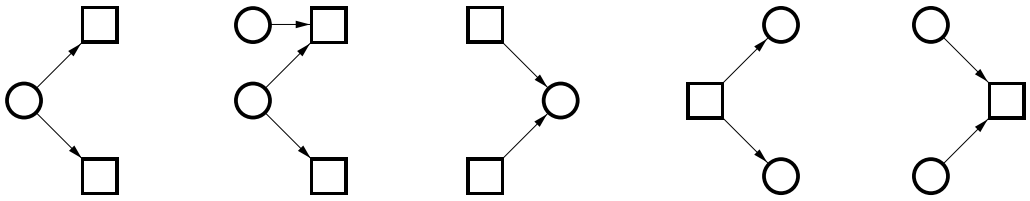
A transition $t \in T$ is *enabled* at a marking m iff $\forall p \in \bullet t,\ m(p) > 0$, i.e. every preset place of t contains at least one token. In the PN from Figure 2.1(a) transitions {t1, t2} are enabled, while transition t3 is not (it is *disabled*).

A transition enabled at marking m can *fire* producing a new marking m' such that $m' = m - \bullet t + t\bullet$, i.e. one token is consumed from each preset place and one token is

produced to each postset place:

$$m'(p) = \begin{cases} m(p) - 1 & \text{if } p \in {\bullet t} \setminus t{\bullet} \\ m(p) & \text{if } p \in {\bullet t} \cap t{\bullet} \\ m(p) + 1 & \text{if } p \in t{\bullet} \setminus {\bullet t} \end{cases}$$

Figure 2.1(b) shows the result of transition t1 firing: a token from place p1 has been removed and placed into p2. Only transition t2 is enabled according to the new marking. If t2 fires the marking changes in such a way that transition t3 becomes enabled, see Figure 2.1(c). Its firing returns the PN into the initial state shown in Figure 2.1(a). Note that in the initial state both transitions {t1, t2} are enabled, therefore they can fire concurrently.



(a) Free-choice (b) Controlled-choice (c) Merge (d) Fork (e) Join

Figure 2.2: Modelling choice and concurrency Petri Nets

A place $p \in P$ with more than one postset transition ($|p\bullet| > 1$) is called a *choice* place. A choice place p is called *free-choice* if every transition in its postset has only one preset place (and it is p), i.e. $\forall t \in p\bullet$, $|\bullet t| = 1$; otherwise the place is called *controlled-choice*. In other words, a choice is free if it cannot be influenced by the rest of the system: once a free-choice place gets a token all of its postset transitions become enabled and one of them is 'freely' chosen to fire [32]. A place with more than one transition in its preset ($|\bullet p| > 1$) is called a *merge* place. A transition $t \in T$ such that $|t\bullet| > 1$ is called a *fork*, and $|\bullet t| > 1$ – a *join*. Figure 2.2 shows PN fragments containing choice, merge, fork and join structures.

A marking m' is *reachable* from marking m if there exists a *firing sequence* (or *trace* for short) of transitions $(t_1, t_2, \ldots, t_n)$ leading from m to m'. A *reachability set* of a PN is

a set of all markings reachable from its initial marking m_0. The *reachability problem* is to decide if a given marking m belongs to the reachability set of a given PN. This problem is very important in practice; the most efficient methods for reachability analysis are based on the *PN unfolding theory* [45]. Several common reachability properties [76] of PNs are listed below.

A *deadlock* is a marking which does not enable any transitions. A PN is *deadlock-free* if its reachability set contains no deadlocks.

A PN is k-*bounded* if the number of tokens in every place does not exceed a finite number k at any reachable marking. A 1-bounded PN is called *safe*.

A PN is *live* if for every reachable marking m and transition t it is possible to reach a marking m' enabling t. In other words, every transition can always fire again. The PN shown in Figure 2.1 is deadlock free, safe and live.

These and other properties of PNs correspond to the properties of the modelled system. For example, normally there is a one to one correspondence between deadlocks in a system and in its PN representation. This is why PNs are often used for model checking and verification: if one would like to know if a certain property of a concurrent system holds, it is easy to formulate the corresponding PN property and verify it using the variety of available PN tools [82]. In the general case, most of the PN properties are very expensive (in terms of computational complexity) to verify. However, there are several subclasses of PNs that have enough expressive power yet allow simpler verification algorithms. A survey of PN decision problems, their decidability and complexity results can be found in [35][36]. The following are three basic subclasses of PNs.

A PN is a *Marked Graph* (MG) if every place has exactly one preset and one postset transition: $\forall p \in P,\ |{\bullet}p| = |p{\bullet}| = 1$. This subclass does not allow any choice, but allows concurrency, thus it can be used to describe deterministic concurrent systems.

A dual subclass consists of *State Machines* (SMs). A PN is a SM if every transition has exactly one preset and one postset place: $\forall t \in T,\ |{\bullet}t| = |t{\bullet}| = 1$. No concurrency can be modelled with a safe SM. This subclass represents non-deterministic sequential systems.

A PN is called a *Free Choice Net* (FCN) if its every choice place is free-choice [32]. This subclass of PNs is capable of modelling both concurrency and choice but does not allow their interference, or *confusion*. FCNs have a good balance between the expressive power and verification complexity.

The three most common extensions of PNs are *Coloured Petri Nets* which model data using tokens of different colour [43], *Timed Petri Nets* which introduce the notion of time into PNs [106], and *Labelled Petri Nets* which are discussed in the next subsection.

2.2.3 Signal Transition Graphs

It is often necessary to label transitions of a PN in order to attach a meaningful semantics to them, e.g. each transition can be labelled with the name of an event in an asynchronous system, thereby setting a direct correspondence between behaviour of the system and that of its PN model [109].

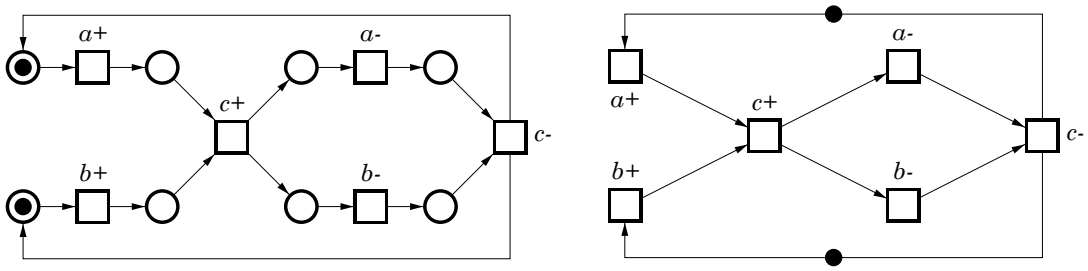
Definition 2.2. A *Labelled Petri Net* (LPN) is a triple $LPN(PN, \Sigma, \lambda)$ where $PN(P, T, F, m_0)$ is a PN, Σ is a finite alphabet, and λ is a *labelling function* which associates a *label* $\lambda(t)$ with every transition $t \in T$ of the LPN. Note that different transitions can have the same label.

Labelled (or interpreted) PNs, where transitions represent changes of circuit signals, were independently proposed in [87] and [19]. The former called the model *Signal Graphs*, while the latter – *Signal Transition Graphs*; this name became widely adopted. Both works proposed to label every transition in a PN with an element of $X \times \{+, -\}$, where X is the set of circuit signals and $\{+, -\}$ stand for their rising and falling transitions, i.e. event $x+$ means that signal $x \in X$ changes from 0 to 1, while event $x-$ corresponds to its change from 1 to 0. Following is the formal definition of STGs.

Definition 2.3. A *Signal Transition Graph* (STG) is quadruple $STG(PN, I, O, \lambda)$, where $PN(P, T, F, m_0)$ is a PN, I and O are disjoint sets of *input* and *output signals* respectively (their union is $X = I \cup O$ – the set of all signals of the circuit; *internal* signals are traditionally included in the O set), and the *labelling function* $\lambda : T \to X \times \{+, -\}$ labels

every transition of the PN with a signal event. An STG inherits behavioural semantics from its underlying PN, including the firing rules, the notions of concurrency, reachability, etc. (which can be found in the previous subsection).

Figure 2.3(a) shows the STG specification of a *C-element*, a basic asynchronous circuit introduced by Muller [75]. It has two inputs $I = \{a, b\}$ and one output $O = \{c\}$, and behaves in the following way. Initially all signals are 0. As soon as both inputs change to 1 (concurrent events $\{a+, b+\}$), the circuit has to acknowledge that by changing its output to 1 (event $c+$) – this is the end of the *set phase*. The *reset phase* is symmetric: inputs concurrently reset to 0 (events $\{a-, b-\}$) enabling the output reset (event $c-$); its firing brings the circuit back into the initial state. According to the specification this process continues forever.



(a) Full notation (b) Simplified notation

Figure 2.3: Signal Transition Graphs

As STGs tend to have many 'trivial' places, i.e. those having exactly one preset and one postset transition, they are often omitted for clarity as shown in Figure 2.3(b). Note that the tokens are placed directly on the corresponding arcs in this case.

STGs are very natural for describing behaviour of small asynchronous microcontrollers and they are successfully used for this purpose in the thesis (see, for example, Subsection 8.3.4). However, the STGs describing large systems with multiple choice, or mixed data/control path controllers can be very large and incomprehensible as will be demonstrated in the next chapter.

There are two main approaches to synthesis of the physical implementation of an asynchronous controller from its STG specification: *logic synthesis* [25][47] and *direct mapping* [39][50][92]. The logic synthesis methods try to capture dependencies between

signals in a circuit by analysing its STG specification which, unfortunately, requires exploration of its complete state space. To alleviate the state explosion problem it is possible to employ methods for efficient representation of the state space, e.g. unfoldings [47], or to use the STG decomposition techniques [89], or both [48]. The direct mapping techniques, on the other hand, are based on the direct translation of the whole STG structure into circuits using presynthesised handshake components, such as *David cells* [30] (this approach is similar to the syntax-directed translation from HDLs [102]). These methods have low algorithmic complexity and are applicable to an STG of any size, but produce circuits that are larger and slower than those obtained using the logic synthesis approach.

The next chapter presents several practical examples which demonstrate that the STG model (as well as the FSM model introduced in the previous section) cannot be directly applied to specification and synthesis of a certain class of asynchronous microcontrollers. This provides the motivation for a new model which is formally described in Chapter 5. In Chapter 8 the proposed model is shown to be significantly more efficient on the same specification and synthesis examples.

Chapter 3

Motivation for the new model

There are many models targeted at microcontrol specification and synthesis (the most commonly used have been addressed in the previous chapters), and a strong reason is demanded if yet another model is to be introduced into this well-studied and established domain. This chapter demonstrates limitations of the existing control specification models, in particular STGs and FSMs (see Chapter 2). The limitations arise in situations when a specified system contains a mixture of data and control path interfaces. This leads to combinatorial explosion in the size of specification because data path modelling requires exploration of all possible combinations of signal arrivals within a single data codeword which in fact can be avoided by using different abstraction levels for data and control related events. A basic example of such situation is presented in Section 3.1 outlining the first part of the motivation.

Another source of problems for the conventional approaches comes from systems having many similar behavioural patterns, or event orders, defined on the same domain of events. It is generally impossible to specify such systems in a functional form within the existing models: each event order from the patterns have to be explicitly enumerated and specified thereby blowing the specification size beyond what is practically feasible. This issue was first addressed in [71], which also presented the Conditional Partial Order Graph model capable of specifying such systems in a compact functional form. Section 3.2 discusses this class of systems and completes the motivation for the new model.

Note that this chapter does not present any solutions to the problems discussed here. It's purpose is only to highlight these problems and convince the reader of the necessity of a new approach to microcontrol specification. The corresponding solutions can be found in Chapter 8, however, they are based on Conditional Partial Order Graphs and it might be difficult to understand them being unfamiliar with the new model. The next two chapters give the necessary background and the reader is advised to go through them before viewing the solutions.

3.1 ParSeq controller

This section outlines limitations of the STG and FSM models for the specification of systems containing a mixture of data and control path interfaces.

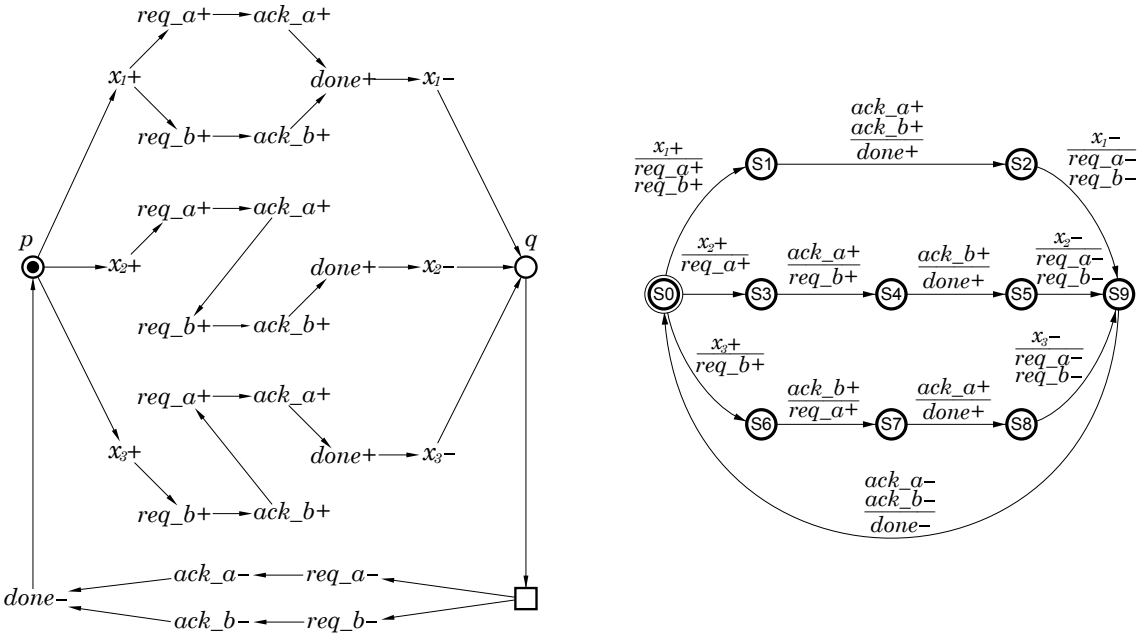
Consider a generalised *ParSeq controller*[1], which manages two handshakes $A = (\mathtt{req\_a},\ \mathtt{ack\_a})$ and $B = (\mathtt{req\_b},\ \mathtt{ack\_b})$ on its right side according to the *operation code (opcode)* provided by an asynchronous data path interface on its left side as shown in Figure 3.1.

Figure 3.1: ParSeq controller interface

Depending on the opcode signals the handshakes are to be initiated either in parallel (concurrent events A and B) or in sequence (in two possible event orders $A \rightarrow B$ or $B \rightarrow A$), hence the name of the controller. As soon as both handshakes are completed the controller issues signal $\mathtt{done}$. The reset phase is similar but the handshakes are always reset concurrently regardless of the opcode.

The three operational scenarios can be encoded (i.e. given distinct opcodes) in different ways. The following subsections demonstrate how a chosen data path encoding affects the controller specification.

[1] ParSeq controller and its variations have several practical applications, e.g. in Balsa synthesis flow [10] or in phase encoding controllers [65].



(a) STG specification

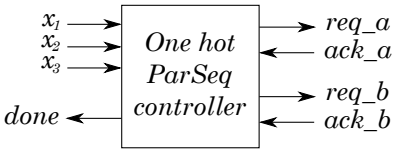
(b) FSM specification

Figure 3.2: Specifications of one hot ParSeq controller

3.1.1 One hot encoding

One hot encoding [105] is the most natural data path encoding in this case. It uses three signals $\{x_1, x_2, x_3\}$ to select one of the three scenarios:

Scenario	x_1	x_2	x_3
A‖B	1	0	0
A → B	0	1	0
B → A	0	0	1



Note, that combination (0, 0, 0) represents a *spacer* value which separates two consecutive data symbols.

STG specification of a ParSeq controller with a one hot interface is shown in Figure 3.2(a). The STG has a global choice (place p) and the three scenarios are specified as three independent branches starting with input signals x_1+, x_2+, and x_3+. The upper branch corresponds to parallel handshakes A‖B; the two lower branches correspond to sequential handshakes A → B and B → A. After the global merge (place q) the hand-

shakes are reset concurrently and the system returns to the initial state. This STG seems to be convenient, understandable and can be designed by hand but it duplicates events in different branches which can cause exponential explosion in the size of the specification for larger controllers. This particular issue, however, is not addressed in this section and will be investigated in Section 3.2.

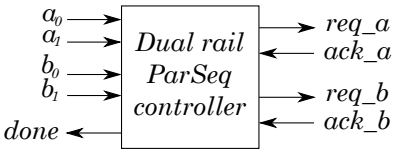
Figure 3.2(b) shows an FSM specification of one hot ParSeq controller. Its structure is similar to the STG: it has a global choice in the initial state *S0*, three separate branches describing different scenarios, and the concurrent reset of the handshakes via state *S9*. Apart from the same event duplication issue, this specification is compact and can be easily obtained manually.

The presented specifications do not exhibit any mentioned problems of data path interface communication because of the fact that one hot encoding transfers all the information within a single signal transition (which is the essence of one hot encoding, being its advantage and disadvantage at the same time). Unfortunately, one hot encoding is generally not an option for data path interfaces as it requires too many wires for data transmission. The next subsection shows the effect of using binary (dual rail) encoding for the opcodes.

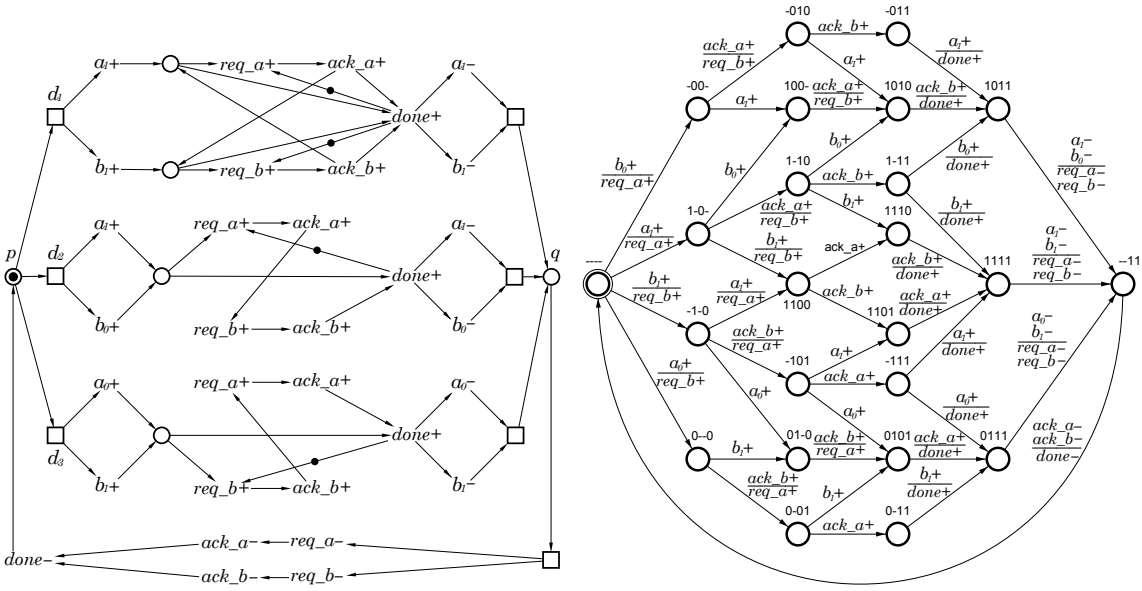
3.1.2 Dual rail encoding

Dual rail encoding [105] uses two wires (a_0, a_1) for one data bit a encoding: combination (1, 0) stands for Boolean value 0; (0, 1) represents 1; and (0, 0) is a spacer value. The three scenarios can be encoded with two dual rail signals $\{a, b\}$ as shown below:

Scenario	(a_0, a_1)	(b_0, b_1)
A‖B	(0, 1)	(0, 1)
A → B	(0, 1)	(1, 0)
B → A	(1, 0)	(0, 1)



Here, bit a can be interpreted as a permission to handshake A to happen without waiting for handshake B (bit b has a symmetric interpretation).



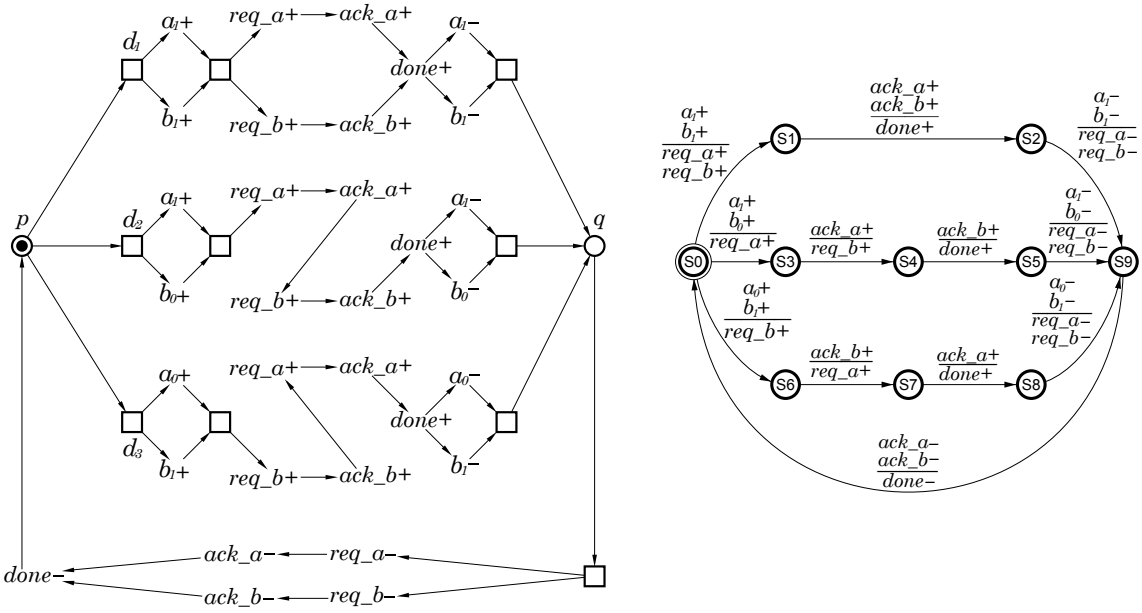
(a) STG specification (b) FSM specification

Figure 3.3: Specifications of dual rail ParSeq controller

As can be seen from Figure 3.3(a) the STG specification changes dramatically due to this modification of the data path interface. The reason is that the new data encoding uses two concurrent transitions to transfer an opcode (instead of only one in one hot encoding) and all the arrival scenarios of these two transitions have to be explicitly reflected in the specification. Dummy events d_1, d_2, and d_3 help to simplify the specification: they do not correspond to any hardware signals but rather represent the choice of the environment, e.g. arrival of signals a_1+ and b_1+ signifies that the environment has chosen the first scenario associated with the d_1 branch of the STG, etc. (note that signal a_1+ alone is not enough to deduce the choice).

Unexpectedly, the new STG has to model *OR-causality* [108]: handshake A can be initiated as soon as at least one of signals a_1+ or b_0+ is received. As a result the opcode decoding process propagates further into the controller specification and it is already impossible to clearly separate data and control related event flows in the STG.

The situation with the FSM specification is even worse because FSMs are not well suited for modelling concurrency in the arrival of dual rail bits, which, coupled with OR-causal behaviour, leads to a very complicated FSM shown in Figure 3.3(b) (for convenience



(a) STG specification (b) FSM specification

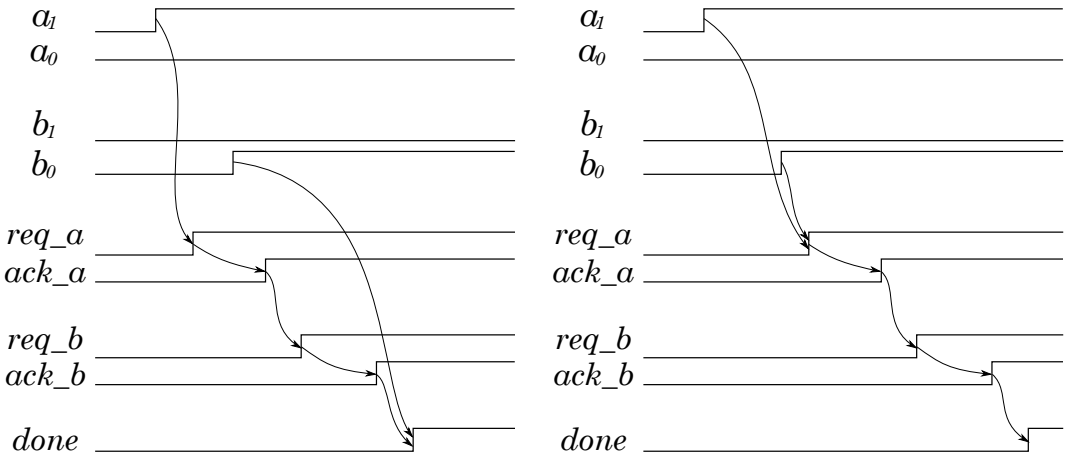
Figure 3.4: Dual-rail ParSeq specifications after concurrency reduction

every state is marked with a signal vector (a, b, req_a, req_b) where '0' stands for dual-rail signal (1, 0), '1' stands for (0, 1), and '−' stands for the spacer (0, 0), e.g. '1−0−' means that a = (0, 1), b = (0, 0), req_a = (1, 0), and req_b = (0, 0)).

3.1.3 Dual rail encoding (concurrency reduction)

In order to simplify the specification of the dual rail ParSeq controller one can try to get rid of OR-causality by *concurrency reduction* [24]: the controller can be restricted to wait for both dual rail signals to arrive before generating handshakes. This greatly simplifies both STG and FSM specifications (see Figure 3.4) bringing their sizes back to those of one hot controller (cf. Figure 3.2). See Figure 3.5 for timing diagrams comparing two versions of the dual-rail ParSeq controller; note that the former one generates the first handshake in the OR-causal manner (a_1+ is enough to generate req_a+), while the latter one does it in the AND-causal manner (req_a+ is generated only after both a_1+ and b_0+ have arrived).

The presented examples demonstrate a high degree of sensitivity of STG and FSM specifications to minor changes in the data path interface protocol, yet from a high-level



(a) With OR-causaility (b) With AND-causaility (concurrency reduction)

Figure 3.5: Timing diagrams of dual-rail ParSeq controllers

designer perspective an actual data encoding may be unimportant at all, or sometimes may even be unknown (or undecided) until the later design stages. However, even such a simple controller with three basic behavioural scenarios becomes a real challenge for manual design, and a subtle modification of data encoding requires its complete redesign.

This motivated the author to propose a new specification model that has different levels of abstraction for data and control events. The model, called Conditional Partial Order Graph, separates control event flow within scenarios from the encoding of these scenarios and associated data path interface events. This allows specification to stay structurally unchanged under different data encodings. Moreover, the model provides an opportunity to synthesise encodings of the scenarios targeting various design optimality criteria, e.g. controller latency, as explained in Section 7.4.

3.2 n-permutators

This section discusses a 2-permutator controller (a simplified version of the ParSeq controller without the parallel handshakes scenario) and its generalisation to n-*permutator*, which reveals another weakness in the existing control specification models [71]. n-permutators arise naturally in the context of *phase encoding* communication controllers addressed in Section 8.2.

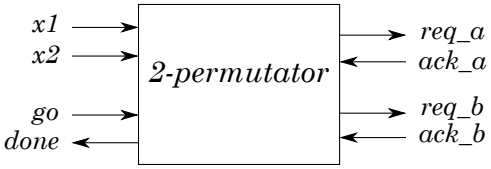


Figure 3.6: 2-permutator controller interface

The interface of the controller is shown in Figure 3.6. Depending on the opcode signals {x1, x2} the controller has to initiate two handshakes either in order A → B or in order B → A. The start of the handshake sequence is prompted by signal go and as soon as the handshakes are completed the controller issues signal done. This leads us to the STG shown in Figure 3.7. It has a global choice and the two scenarios are specified as two separate branches starting with input signals x_1+ and x_2+. The first scenario (the upper branch) is handshake sequence A → B; the second one (the lower branch) corresponds to B → A. After the global merge the handshakes are reset concurrently and the system returns to the initial state.

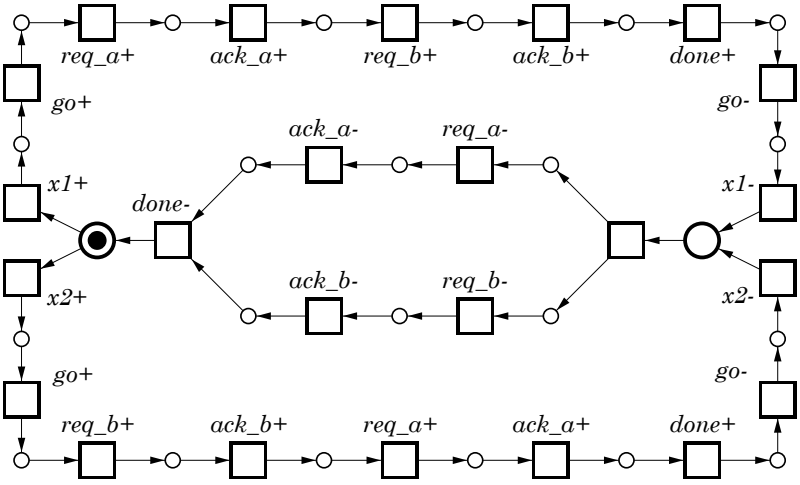


Figure 3.7: STG specification of 2-permutator

This straightforward STG specification has a flaw: it duplicates events in different branches. This flaw may seem to be unimportant at the first glance, but in general there can be exponential number of different scenarios composed of linear number of events: for instance, we can generalise 2-permutator to control more than two handshakes, and for n handshakes there will be n! different scenarios. It is not efficient to have an STG

specification containing $n!$ different branches and it turns from inefficient to infeasible for large values of n.[2]

To specify the controller in a more compact way we can construct a behaviourally equivalent STG without multiple event occurrences (e.g. by using PETRIFY [23] tool). The result is shown in Figure 3.8. Such compositional STGs tend to be much more complicated and contain a lot of additional choice places tracking the current system state. Even for such a simple controller as a 2-permutator the obtained STG is non-trivial and difficult for manual design. In practice the only way to produce an optimal STG specification is to start with an inefficient global choice STG and feed it to an optimisation tool but this is infeasible for larger controllers, because synthesis of compositional STGs involves construction of a state graph and examining all reachable states, which is a very time and memory consuming process.

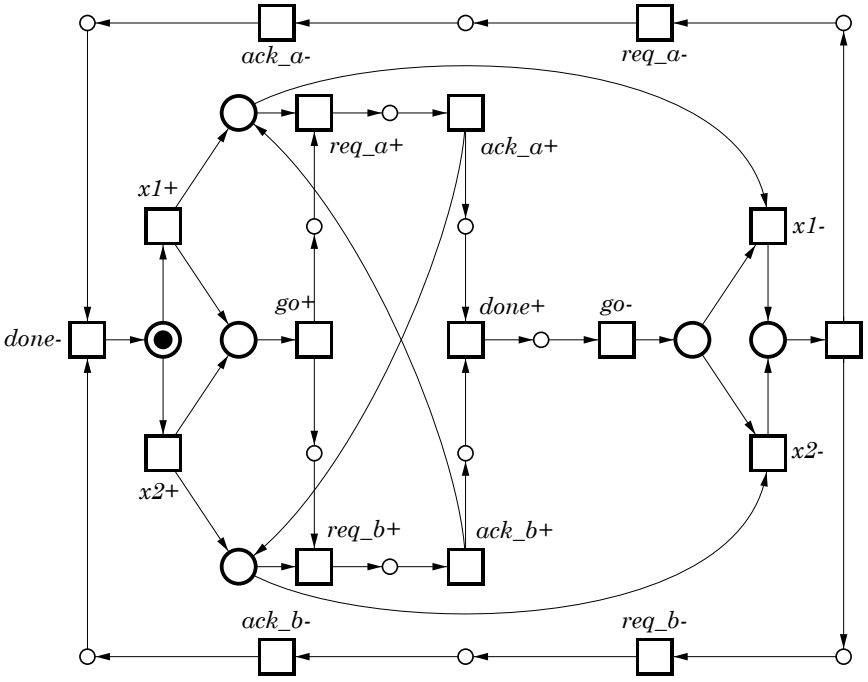


Figure 3.8: Optimised STG specification of 2-permutator

The FSM-driven approach has the same event duplication issues. The state space of n-permutator is huge (its size is proportional to $n \cdot n!$ even if we assume that all the

[2] In practice, 4-permutator was the largest n-permutator controller which was possible to synthesise using the STG-driven approach.

opcode signals arrive simultaneously). An FSM specification has to explicitly list all the states and quickly becomes impractical with the growth of n. It should be mentioned, however, that both STG and FSM approaches produce very efficient controllers when they manage not to run out of time and memory resources. Notably, the size of the synthesised controllers is polynomial with respect to n, which implies that there exists a polynomial specification as well.

This is another motivation factor behind the new Conditional Partial Order Graph model. Chapter 8 demonstrates that it is possible to specify and synthesise ParSeq and n-permutator controllers efficiently using techniques of polynomial complexity. The next two chapters introduce the new model formally.

Chapter 4

Background for CPOGs

This chapter introduces partial orders (Section 4.1) and directed acyclic graphs (Section 4.2) which constitute the basis for the new model. These two formalisms are comprehensively studied [13][22][56] and are closely related to each other as shown in Section 4.3.

4.1 Partial orders

Partial orders are widely used to formalise the intuitive concept of ordering of events (or actions) with cause and effect relationships between them.

Definition 4.1. A *strict partial order* $P(S, \prec)$ is a binary precedence relation $\prec$ over a set of events S which satisfies two properties [13][56]:

1. *Irreflexivity*: $\forall a \in S, \neg(a \prec a)$;
2. *Transitivity*: $\forall a, b, c \in S, (a \prec b) \bigwedge (b \prec c) \Rightarrow (a \prec c)$.

This work focuses only on strict partial orders and from now on the qualifier 'strict' will be omitted for brevity. Peculiarities of strict and non-strict partial orders are discussed in [70].

The following example demonstrates the concept of event schedule specification by means of a partial order.

Example 4.1. Consider the operation of addition $Z = X + Y$. It is possible to break it into the following four primitive actions:

a) Read input value X;

b) Read input value Y;

c) Compute sum $Z = X + Y$;

d) Store the result value Z.

Let us order these actions taking into account the cause and effect relationships between them. One can see that action c depends on actions a and b (there is no way to compute sum Z without having values X and Y; actions a and b are so-called passive, or material causes for action c), and action d in turn cannot happen until action c is completed. Note that although d does not depend on a and b directly, it has them as indirect causes, what should also be reflected in the partial order. Such indirect dependencies are called *transitive* (see Definition 4.2).

The above causal relationships are captured with the following partial order $P(S, \prec)$:

$$P = \begin{cases} S = \{a,\ b,\ c,\ d\} \\ \prec = \{a \prec c,\ b \prec c,\ a \prec d,\ b \prec d,\ c \prec d\} \end{cases}$$

It is possible to depict it graphically as shown in Figure 4.1. The actions are represented with boxes and the relationships between them – with arcs (the transitive ones, namely $a \prec d$ and $b \prec d$, are dashed).

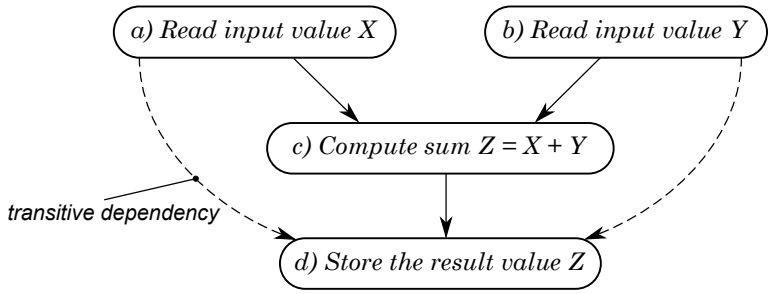


Figure 4.1: Partial order for the operation of addition $Z = X + Y$

Depicting all the transitive dependencies on a partial order diagram is usually unnecessary. For example, it is possible to omit transitive arcs $\{a \prec d,\ b \prec d\}$ in Figure 4.1 without losing the essential information about the depicted partial order: one can always keep the transitivity property of partial orders in mind and realise that any two relationships $a \prec b$ and $b \prec c$ in a diagram imply the transitive relationship $a \prec c$. *Hasse diagrams* [13] are widely used as a compact way of graphical representation of partial orders, as explained in Subsection 4.1.2.

The notions of *transitive dependency* and *concurrent/sequential events* with respect to a partial order are introduced below.

Definition 4.2. A dependency $a \prec b$ between events $a,\ b \in S$ in a partial order $P(S,\ \prec)$ is called *transitive* (denoted as $a \prec\!\prec b$) iff there exists an event $x \in S$ such that both conditions $a \prec x$ and $x \prec b$ hold:

$$(a \prec\!\prec b) \stackrel{df}{=} \exists x \in S,\ (a \prec x) \wedge (x \prec b)$$

Definition 4.3. Two events $a,\ b \in S$ $(a \neq b)$ are called *concurrent* or *parallel* (denoted as $a \parallel b$) with respect to a partial order $P(S,\ \prec)$ iff neither $a \prec b$ nor $b \prec a$ holds:

$$(a \parallel b) \stackrel{df}{=} \neg(a \prec b) \wedge \neg(b \prec a)$$

Concurrent events can happen at any time independently from each other, possibly simultaneously. The partial order from Example 4.1 has only two concurrent events: $a \parallel b$.

Definition 4.4. Two events $a,\ b \in S$ are called *sequential* (denoted as $a \not\parallel b$) with respect to a partial order $P(S,\ \prec)$ iff either $a \prec b$ or $b \prec a$ holds:

$$(a \not\parallel b) \stackrel{df}{=} (a \prec b) \vee (b \prec a) = \neg(a \parallel b)$$

To conclude, every two events in a partial order are either sequential or concurrent. The sequential events $a \not\parallel b$ can have either direct dependency $a \prec b$ or an indirect (transitive) one $a \prec\!\prec b$.

4.1.1 Total orders

Definition 4.5. A *total order* is a partial order $P(S, \prec)$ which has an additional property called *totality*: every two events $a, b \in S$ $(a \neq b)$ are sequential $a \not\parallel b$.

In other words, a total order is a partial order with no two concurrent events: all the events are totally ordered in one of $|S|!$ possible ways. Total orders are natural to represent *phase encoded* data symbols (see Section 8.2).

Definition 4.6. A *chain* or a totally ordered subset $C \subseteq S$ of a partial order $P(S, \prec)$ is a set of events $C = \{a_1, a_2, \ldots, a_{|C|}\}$ such that it contains no two concurrent events: $a_k \prec a_{k+1}$, $1 \leqslant k < |C|$. It is denoted as $a_1 \prec a_2 \prec \cdots \prec a_{|C|}$.

Partial order $P(S, \prec)$ from Example 4.1 is not total: it contains two concurrent events $a \parallel b$. Subsets $\{a, c, d\} \subset S$ and $\{b, c, d\} \subset S$, however, are totally ordered, so $a \prec c \prec d$ and $b \prec c \prec d$ are chains.

4.1.2 Hasse diagrams

A partial order $P(S, \prec)$ normally contains a lot of transitive dependencies, for instance, a chain of n events $a_1 \prec a_2 \prec \cdots \prec a_n$, $a_k \in S$, $k = 1 \ldots n$ implies $\binom{n-1}{2} = \frac{(n-1)(n-2)}{2} = O(n^2)$ transitive relationships $a_j \prec\prec a_k$, $1 \leqslant j < k-1 < n$ ($a_1 \prec\prec a_3$, $a_1 \prec\prec a_4$, $a_2 \prec\prec a_4$ etc). So, there can be a lot more transitive dependencies than non-transitive, essential ones.

Hasse diagram [13] is a graphical representation of a partial order based on the *transitive reduction* [22]: it depicts only non-transitive dependencies between the events thereby keeping the diagram as simple as possible. One can always reconstruct all the reduced transitive dependencies performing the *transitive closure* of a Hasse diagram. The transitive reduction and closure are formally described in terms of graphs in Section 4.2. There is also a convention of arranging the events in a Hasse diagram in such a way that all the arrows point only downward or upward thus forming the event levels which sometimes help understanding the depicted partial order.

Example 4.2. Hasse diagram of the partial order from Example 4.1 is shown in Figure 4.2. Note the difference from the diagram in Figure 4.1: the transitive dependencies are

reduced resulting in a clear and intuitively understandable form.

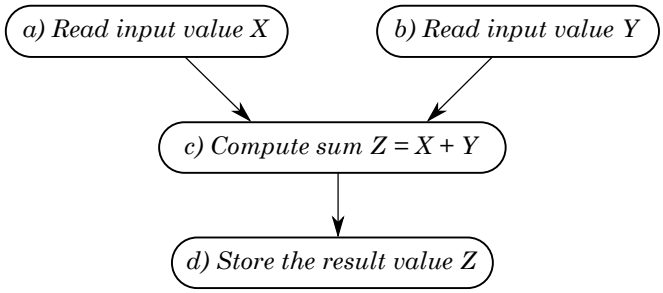


Figure 4.2: Hasse diagram of the partial order from Figure 4.1

4.2 Directed acyclic graphs

Definition 4.7. A *directed graph* is a tuple $G(V, E)$ where V is a set of *vertices* (or *nodes*) and $E \subseteq V \times V$ is the set of ordered pairs of vertices, called *arcs* [22][56].

A directed graph is typically depicted as a set of labelled circles ○ (standing for vertices) and a set of arrows ⟶ between the circles (standing for arcs). Figure 4.3(a) shows an example of a directed graph $G(V, E)$ containing $|V| = 7$ vertices and $|E| = 6$ arcs.

A sequence of $n \geqslant 2$ vertices $(x_1, x_2, \ldots, x_n)$, $x_k \in V$, $k = 1...n$ such that $(x_{k-1}, x_k) \in E$, $k = 2...n$ is called a *path* from x_1 (start vertex) to x_n (end vertex) and is denoted as $\langle x_1, x_n \rangle$. The fact that graph G contains path $\langle x, y \rangle$ is denoted as $\langle x, y \rangle \in G$. For instance, graph G in Figure 4.3(a) contains paths $(c, d, f, g) = \langle c, g \rangle \in G$ and $(a, b) = \langle a, b \rangle \in G$ but does not contain path $(d, f, e) = \langle d, e \rangle \notin G$ because $(f, e) \notin E$.

A *cycle* is a path $\langle x, y \rangle$ whose start and end vertices coincide: $x = y$.

Definition 4.8. A *directed acyclic graph* (*DAG*) is a directed graph $G(V, E)$ that does not contain any cycles: $\forall x \in V$, $\langle x, x \rangle \notin G$. All the graphs in Figure 4.3 are DAGs.

Definition 4.9. The *transitive closure* of a graph $G(V, E)$ is graph $G^*(V, E^*)$ such that:

$$\forall x, y \in V, \langle x, y \rangle \in G \Leftrightarrow (x, y) \in G^*$$

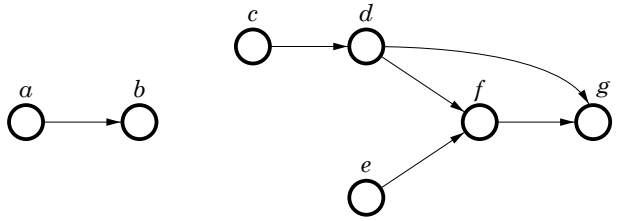
In other words graph G^* contains arc $(x, y) \in E^*$ for every two vertices $x, y \in V$ that

are connected with a path $\langle x, y\rangle \in G$ in the original graph (and vice versa). The transitive closure of the graph from Figure 4.3(a) is shown in Figure 4.3(b).

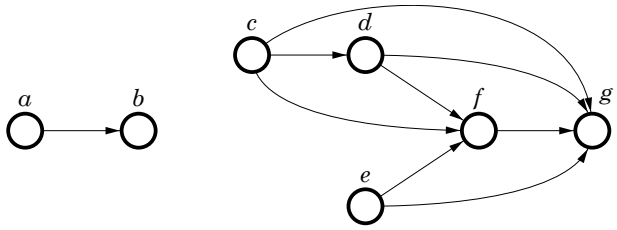
An arc $(x, y) \in E$ of a graph $G(V, E)$ is called *transitive* iff

$$\exists z \in V \backslash \{x, y\},\ \langle x, z\rangle \in G \wedge \langle z, y\rangle \in G$$

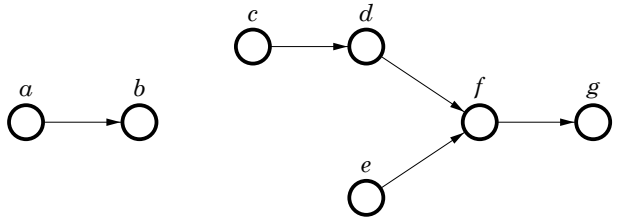
i.e. there is an indirect path from x to y in the graph. Arcs $\{(c, f), (c, g), (d, g), (e, g)\}$ in Figure 4.3(b) are transitive.



(a) Directed acyclic graph $G(V, E)$



(b) The transitive closure $G^*(V, E^*)$



(c) The transitive reduction $G'(V, E')$

Figure 4.3: Directed acyclic graph, its transitive closure and transitive reduction

Definition 4.10. The *transitive reduction* of a graph $G(V, E)$ is the smallest (with respect to the number of arcs) graph $G'(V, E')$ such that:

$$\forall x, y \in V,\ \langle x, y\rangle \in G \Leftrightarrow \langle x, y\rangle \in G'$$

Hence, the transitive reduction preserves all the paths in a graph but minimises the

number of arcs: all the transitive arcs are reduced. Figure 4.3(c) shows the transitive reduction of graph from Figure 4.3(a): the transitive arc $(d, g) \in E$ has been removed.

4.3 DAGs and partial orders correspondence

There is a strong correspondence between partial orders and DAGs: every partial order is a DAG, and the transitive closure of a DAG is both a partial order and a DAG itself. The graph in Figure 4.3(b) directly matches a partial order relation E^* over the set of vertices V while the graph in Figure 4.3(a) does not because it violates the transitivity condition. For instance, it contains arcs $(e, f) \in E$ and $(f, g) \in E$ while the corresponding transitive arc is not present: $(e, g) \notin E$.

This correspondence between partial orders and DAGs provides an intuitive way of partial order specification. A DAG $G(V, E)$ defines a corresponding partial order $P(V, E^*)$. Note that there can be more than one DAG with the same corresponding partial order. For example, all the DAGs in Figure 4.3 have the same transitive closure and therefore they define the same partial order. The graph in Figure 4.3(c) is the simplest, however, and is preferable in most cases. This is equivalent to the approach used in Hasse diagrams (see Subsection 4.1.2).

Example 4.3. Figure 4.4 shows the four possible DAG specifications of partial order from Example 4.1. The leftmost graph is the simplest but all of them are valid. Their transitive closure is the same and is equal to the rightmost graph.

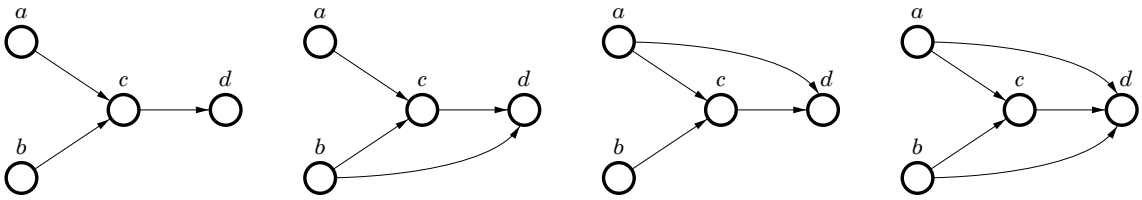


Figure 4.4: Possible specifications of a strict partial order using directed acyclic graphs

Chapter 5

Conditional Partial Order Graphs

This chapter formally defines the CPOG model and algebra over well-formed CPOGs. The model was originally introduced in [71] as a formalism that has a distinctive feature of capturing similar behavioural patterns in a compact functional form as opposed to the existing models[1] which either have a direct event traces representation or an explicit notion of states and transitions between the states. A CPOG is a superposition of a set of partial orders (see Chapter 4) which can be extracted from it by providing the corresponding codewords as shown in Figure 5.1. It can be regarded as a custom associative memory for storing cause and effect relations within a predefined set of events.

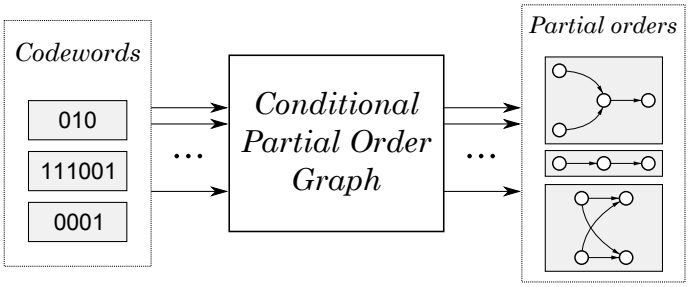


Figure 5.1: A Conditional Partial Order Graph as a superposition of partial orders

The basic definition of the CPOG model [71] restricted a codeword to remain constant throughout retrieval and execution of the corresponding partial order. This basic model is therefore called *static* (it is defined in Section 5.1). Although it has many practical applications the class of the modelled systems is significantly limited. In order to deal with

[1] Chapters 2 and 3 describe the STG and FSM models and provide their comparison with CPOGs.

these limitations the model was further developed in [72] to handle the dynamic codeword evaluation. The extended model is called *dynamic* and is presented in Section 5.2.

The dynamic model revealed a strong need for the verification support. SAT-based verification methods were provided in [72] and are discussed in Chapter 6. They are computationally expensive that motivated the author to introduce the algebraic approach [70] which eliminated the need for verification in most cases, because the synthesis and optimisation can be based on operations which are proved to preserve the correctness of CPOGs. These algebraic operations are common between static and dynamic CPOGs and are presented in Section 5.1.

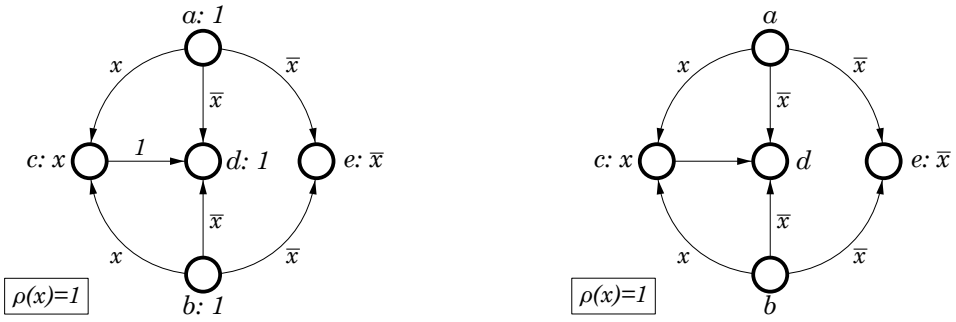
5.1 The static model

Definition 5.1. *Conditional Partial Order Graph* (further called *CPOG* or *graph* for short) is a quintuple $H(V, E, X, \rho, \phi)$ where:

- V is a finite set of *vertices* which correspond to the events in the modelled system. V defines the system's *event domain*.
- $E \subseteq V \times V$ is a set of *arcs* representing dependencies between the events.
- *Operational vector* X is a finite set of Boolean variables. An *opcode* is an assignment $(x_1, x_2, \ldots, x_{|X|}) \in \{0, 1\}^{|X|}$ of these variables. An opcode selects a particular partial order from those contained in the graph.
- $\rho \in \mathcal{F}(X)$ is a *restriction function,* where $\mathcal{F}(X)$ is the set of all Boolean functions over variables in X. ρ defines the *operational domain* of the graph: X can be assigned only those opcodes $(x_1, x_2, \ldots, x_{|X|})$ which satisfy the restriction function, i.e. $\rho(x_1, x_2, \ldots, x_{|X|}) = 1$. A graph is called *singular* iff its operational domain is empty, i.e. function ρ is a contradiction: $\rho = 0$.
- Function $\phi : (V \cup E) \to \mathcal{F}(X)$ assigns a Boolean *condition* $\phi(z) \in \mathcal{F}(X)$ to every vertex and arc $z \in V \cup E$ in the graph. Let us also define $\phi(z) \stackrel{\text{df}}{=} 0$ for $z \notin V \cup E$ for convenience.

Conditional Partial Order Graphs are represented graphically by drawing a labelled circle ○ for every vertex $v \in V$, and drawing a labelled arrow ⟶ for every arc $e \in E$. The label of a vertex $v \in V$ consists of the vertex name, semicolon and the vertex condition $\phi(v)$, while every arc $e \in E$ is labelled with the corresponding arc condition $\phi(e)$. The restriction function ρ is depicted in a box next to the graph; operational variables X can therefore be observed as parameters of ρ.

Example 5.1. Figure 5.2(a) shows an example of a graph containing $|V| = 5$ vertices and $|E| = 7$ arcs. The restriction function is $\rho(x) = 1$, and the operational vector consists of a single variable $X = \{x\}$. Vertices $\{a, b, d\}$ have constant $\phi = 1$ conditions and are called *unconditional*, while vertices $\{c, e\}$ are *conditional* and have conditions $\phi(c) = x$ and $\phi(e) = \overline{x}$ respectively. Arcs also fall into two classes: *unconditional* (arc (c, d)) and *conditional* (all the rest). As CPOGs tend to have many unconditional vertices and arcs it is reasonable to use a simplified notation in which conditions equal to 1 are not depicted in the graph. This is demonstrated in Figure 5.2(b).



(a) Full notation (b) Simplified notation

Figure 5.2: Graphical representation of Conditional Partial Order Graphs

The purpose of vertex and arc conditions is to 'switch off' some vertices and/or arcs in the graph according to the given opcode. This makes CPOGs capable of specifying multiple DAGs, and consequently multiple partial orders (due to the correspondence between DAGs and partial orders which was demonstrated in Section 4.3). Figure 5.3 shows an example of a graph and its two *projections* (see Definition 5.2). The leftmost projection is obtained by keeping in the graph only those vertices and arcs whose conditions evaluate to Boolean 1 after substitution of the operational variable x with Boolean 1. Hence,

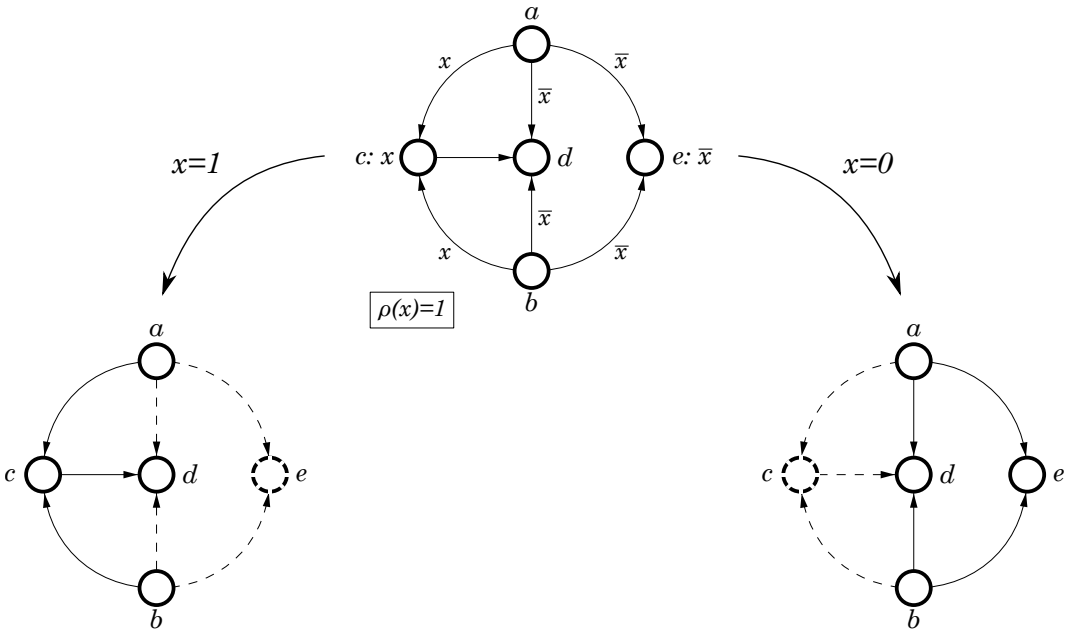


Figure 5.3: Multiple DAGs contained in a single CPOG

vertex e disappears (denoted as a dashed circle ⌀), because its condition evaluates to 0: $\phi(e) = \overline{x} = \overline{1} = 0$. Arcs $\{(a, d), (a, e), (b, d), (b, e)\}$ disappear for the same reason (denoted as dashed arrows ⇢). The rightmost projection is obtained in the same way with the only difference that variable x is set to 0. Note also that although the condition of arc (c, d) evaluates to 1 (in fact it is constant 1) the arc is still excluded from the resultant graph because one of the vertices it connects (vertex c) is excluded and obviously an arc cannot appear in a graph without one of its vertices. The restriction function of the graph does not affect anything in this particular case because it evaluates to 1 for both possible opcodes ($x = 1$ and $x = 0$): $\rho(0) = \rho(1) = 1$. Its role will be explained in details later (in particular, see graph H_b in Example 5.3).

Each of the obtained projections can be treated as a specification of a partial order of events in a particular behavioural scenario of the modelled system. Potentially, a CPOG $H(V, E, X, \rho, \phi)$ can specify an exponential number of different partial orders of events in V according to one of $2^{|X|}$ different possible opcodes.

The concept of system specification with a set of partial orders contained within a single graph is clarified with the following example.

Example 5.2. Consider a processing unit that has the accumulator register A and the general purpose register B, and performs two different operations: addition and exchange of two variables stored in memory. The event domain of the system consists of the following five events:

1. Load register A from memory;
2. Load register B from memory;
3. Add a value (a constant or a value from a register) to accumulator A;
4. Save register A into memory;
5. Save register B into memory.

Table 5.1 describes the two operations. The addition operation consists of loading of the two operands from memory (concurrent events a and b), their addition (event c), and saving the result (event d). This is reflected in the table with the corresponding partial order and DAG of this scenario (cf. also partial order in Example 4.1). The operation of exchange consists of loading of the operands (concurrent events a and b), and saving them into swapped memory locations (concurrent events d and e). Note that in order to start saving one of the registers it is necessary to wait until both of them have been already loaded to avoid overwriting one of the values.

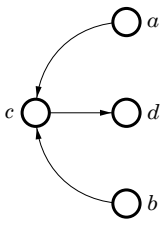
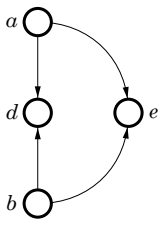
Operation		Addition	Exchange
Events description		a) Load A b) Load B c) Add B to A d) Save A	a) Load A b) Load B d) Save A e) Save B
Partial order	S	$\{a, b, c, d\}$	$\{a, b, d, e\}$
	$\prec$	$\{a \prec c,\ b \prec c,\ a \prec d,\ b \prec d,\ c \prec d\}$	$\{a \prec d,\ b \prec d,\ a \prec e,\ b \prec e\}$
DAG specification			

Table 5.1: Two behavioural scenarios specified as two CPOG projections

One can see that the two DAGs in Table 5.1 appear to be the two projections shown in Figure 5.3. Thus both of the operations of the discussed processing unit can be specified with the single graph. Two important characteristics of such specification are that the common events $\{a, b, d\}$ are overlaid and the choice between the two operations is distributed in the vertex/arc conditions of the graph, i.e. there is no 'nodal point' of choice which tend to appear in the alternative specification models (an STG would have a choice place, an FSM — a choice state, and an HDL specification would describe the two operations in two branches of a conditional statement `if` or `case`).

The rest of the section contains the formal definitions of projections and algebra over CPOGs.

5.1.1 Projections

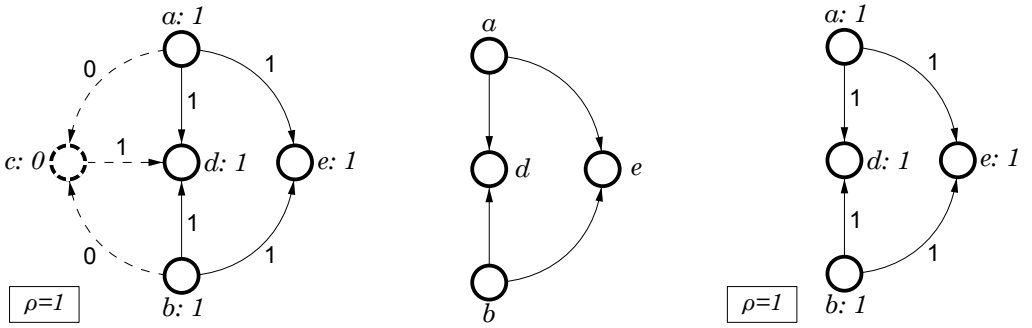
Definition 5.2. A *projection* of a graph $H(V, E, X, \rho, \phi)$ under constraint $x = \alpha$ (where $x \in X$, $\alpha \in \{0, 1\}$) is denoted as $H|_{x=\alpha}$ and is equal to graph $H'(V, E, X \setminus \{x\}, \rho|_{x=\alpha}, \phi|_{x=\alpha})$ where notations $\rho|_{x=\alpha}$ and $\phi|_{x=\alpha}$ mean that variable x is substituted with a constant Boolean value α in ρ and all functions $\phi(z)$, $z \in V \cup E$, which implies that $\rho|_{x=\alpha}$ and $\phi|_{x=\alpha}(z)$ belong to $\mathcal{F}(X \setminus \{x\})$.

Projection is a *commutative operation*, i.e. $(H|_{x=\alpha})|_{y=\beta} = (H|_{y=\beta})|_{x=\alpha}$, hence the following short notation can be used without any ambiguity: $H|_{x=\alpha,\, y=\beta}$.

Definition 5.3. A *complete projection* of a graph H is such a projection that all the operational variables X are constrained to constants. It is denoted as $H|_\psi$ where $\psi : X \to \{0, 1\}$ is an opcode that assigns a Boolean value to every variable in X. A complete projection is a graph whose restriction function and vertex/arc conditions are only Boolean constants $\rho|_\psi$ and $\phi|_\psi$ (either 0 or 1), and the operational vector is empty: $X = \emptyset$.

A (complete) projection is called *singular* iff the resultant graph is singular.

Figure 5.4(a) shows a complete projection of the graph from Example 5.1 under opcode $x = 0$ in full notation. The same projection in the simplified notation (without the trivial $\{0, 1\}$ conditions) is shown in Figure 5.3 (to the right).



(a) Complete projection H (b) Directed graph $G = \mathbf{dg}\ H$ (c) Complete projection $H' = \mathbf{dg}^{-1}\ G$

Figure 5.4: Complete projection, operation **dg**, and its inverse

Definition 5.4. Given a non-singular complete projection $H(V,\ E,\ \emptyset,\ 1,\ \phi)$, operation $G = \mathbf{dg}(H)$ generates directed graph $G(V_G,\ E_G)$ such that[2]

$$\begin{cases} V_G = \{v \in V\ ,\ \phi(v) = 1\} \\ E_G = \{e = (a,\ b) \in E,\ \phi(a) \cdot \phi(b) \cdot \phi(e) = 1\} \end{cases}$$

In other words, G includes only those vertices and arcs whose conditions in H are constant 1. Note that exclusion of a vertex also leads to exclusion of all its adjacent arcs. Brackets around the operation argument may sometimes be omitted for clarity: $G = \mathbf{dg}\ H$.

The inverse operation is $H' = \mathbf{dg}^{-1}(G)$. Here $H'(V,\ E,\ X,\ \rho,\ \phi)$ is defined in terms of $G(V_G,\ E_G)$ as follows: $V = V_G$, $E = E_G$, $X = \emptyset$, $\rho = 1$ and $\phi(z) = 1$, $z \in V \cup E$. Note that $\mathbf{dg}^{-1}$ is a *right inverse* operation i.e. $\mathbf{dg}(\mathbf{dg}^{-1}\ G) = G$ but $\mathbf{dg}^{-1}(\mathbf{dg}\ H)$ is not necessarily equal to H. This is demonstrated in Figure 5.4 which shows an example of complete projection H, its conversion into directed graph $G = \mathbf{dg}\ H$, and complete projection $H' = \mathbf{dg}^{-1}\ G$. One can see, that $H \neq H'$ but both **dg** H and **dg** H' are the same and equal to G.

Definition 5.5. A complete projection $H|_\psi$ is called *valid* iff it is not singular and its corresponding directed graph $\mathbf{dg}\ H|_\psi$ is acyclic (DAG).

An opcode ψ is called *valid* with respect to a graph H iff $H|_\psi$ is valid.

[2]In this work symbols '+' and '·' are used to denote Boolean OR and AND operations, respectively.

Definition 5.6. Graph $H(V,\ E,\ X,\ \rho,\ \phi)$ is *well-formed* iff its every non-singular complete projection $H|_\psi$ is valid. In other words, every opcode ψ which is allowed by the restriction function ($\rho|_\psi = 1$) is valid, i.e. it produces an acyclic directed graph **dg** $H|_\psi$.

The set of all CPOGs is denoted as $\mathcal{C}$, and the set of all well-formed CPOGs — as $\mathcal{W}$.

A SAT-based approach for verification of the well-formedness of a graph is presented in Chapter 6. It is computationally expensive so its use should be kept to a minimum by employing the 'safe' operations from the CPOG algebra which are closed over the set of well-formed graphs $\mathcal{W}$ (see Subsections 5.1.3 through 5.1.6). In fact, it will be shown in Chapter 7 that it is possible to synthesise and optimise graphs without any verification of intermediate results using only the 'safe by construction' techniques. However, verification may still be required for the custom graph design/optimisation.

Definition 5.7. Given a DAG $G(V_G,\ E_G)$ operation $P = \mathbf{po}(G)$ generates partial order $P(S,\ \prec)$ such that $S = V_G$ and $\prec = E_G^*$ where $G^*(V_G,\ E_G^*)$ is the transitive closure of G.

The inverse operation is $\mathbf{po}^{-1}$: $G = \mathbf{po}^{-1}(P)$. The obtained $G(V_G,\ E_G)$ contains all the transitive arcs from $P(S,\ \prec)$: $V_G = S$, $E_G = \prec$. As was shown in Section 4.3, a partial order has more than one DAG specification, therefore $\mathbf{po}^{-1}$ is also a right inverse operation: $\mathbf{po}(\mathbf{po}^{-1}\ P) = P$ but $\mathbf{po}^{-1}(\mathbf{po}\ G) = G^*$ and $G \neq G^*$ in general.

Using operations **dg** and **po** it is possible to write equations operating over CPOGs, DAGs and partial orders. For example, the partial order defined by the rightmost projection of graph H in Figure 5.3 can be denoted as $\mathbf{po}(\mathbf{dg}\ H|_{x=0})$:

$$\mathbf{po}(\mathbf{dg}\ H|_{x=0}) = \begin{cases} S = \{a,\ b,\ d,\ e\} \\ \prec = \{a \prec d,\ b \prec d,\ a \prec e,\ b \prec e\} \end{cases}$$

It should be noted that although a non-singular complete projection $H(V,\ E,\ \emptyset,\ 1,\ \phi)$ has no operational variables ($X = \emptyset$), it still defines a partial order $\mathbf{po}(\mathbf{dg}\ H)$. In this case we consider the corresponding opcode ψ to be *empty*, i.e. an empty opcode $\psi = \varepsilon$ does not constrain any variable and is considered to be a valid opcode for generality.

Definition 5.8. The set of all partial orders defined by a well-formed graph $H(V, E, X, \rho, \phi)$ is denoted as $\mathcal{P}(H)$ and is formally defined as

$$\mathcal{P}(H) \stackrel{\text{df}}{=} \{P = \mathbf{po}(\mathbf{dg}\ H|_{\psi}),\ \rho|_{\psi} = 1\}$$

For example, the set of all partial orders defined by graph in Figure 5.3 is

$$\mathcal{P}(H) = \{P_1,\ P_2\} = \{\mathbf{po}(\mathbf{dg}\ H|_{x=0}),\ \mathbf{po}(\mathbf{dg}\ H|_{x=1})\}$$

where P_1 and P_2 are shown in Table 5.1.

Remark 5.1. There is no restriction on the number of opcodes for a particular partial order within a graph, i.e. more than one opcode may yield the same partial order P:

$$P = \mathbf{po}(\mathbf{dg}\ H|_{\psi_1}) = \mathbf{po}(\mathbf{dg}\ H|_{\psi_2}),\ \psi_1 \neq \psi_2$$

Remark 5.2. The following identity is true for any partial order P:

$$\mathcal{P}(\mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P)) = \{P\}$$

(by Definitions 5.4, 5.7 and 5.8). Clearly, graph $H = \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P)$ contains only partial order P, which is induced by the empty opcode ε.

5.1.2 Encoding conflicts

Definition 5.9. Two well-formed graphs H_1 and H_2 are said to be in an *encoding conflict* with respect to their restriction functions ρ_1 and ρ_2 iff $\rho_1\rho_2 \neq 0$. An encoding conflict implies the existence of an opcode ψ such that both of the restriction functions are satisfied: $\rho_1|_{\psi} = \rho_2|_{\psi} = 1$. This leads to ambiguity in some cases (for instance, in case of graph addition introduced in Subsection 5.1.4), when two graphs describe different behaviour for the same opcode ψ. Depending on whether these two graphs actually specify the same or different scenarios under ψ the conflict can be either true or false.

An encoding conflict is *true* if the partial orders generated with ψ are different:

$$\exists\psi,\ (\rho_1\rho_2)|_\psi = 1,\ \mathbf{po}(\mathbf{dg}\ H_1|_\psi) \neq \mathbf{po}(\mathbf{dg}\ H_2|_\psi)$$

Conversely, an encoding conflict is *false* if the partial orders generated with ψ are in fact the same:

$$\forall\psi,\ (\rho_1\rho_2)|_\psi = 1,\ \mathbf{po}(\mathbf{dg}\ H_1|_\psi) = \mathbf{po}(\mathbf{dg}\ H_2|_\psi)$$

The verification issues of true and false conflict detection are addressed in Subsection 6.1.3. Examples 5.4 and 5.5 in Subsection 5.1.6 demonstrate both types of encoding conflicts and ways of their resolution.

5.1.3 Equivalence

Definition 5.8 provides the background for a natural *equivalence relation* [56] $\sim$ over the set of well-formed graphs $\mathcal{W}$.

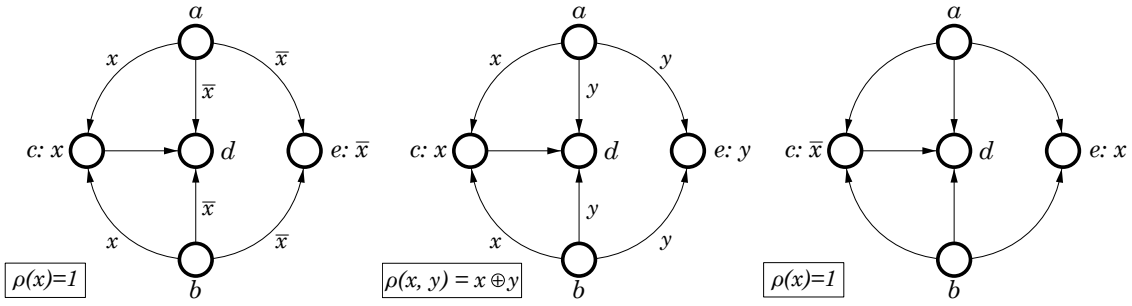
Definition 5.10. Graphs $H_1 \in \mathcal{W}$ and $H_2 \in \mathcal{W}$ are *equivalent* (denoted as $H_1 \sim H_2$) iff they define the same set of partial orders:

$$(H_1 \sim H_2) \stackrel{\text{df}}{=} \mathcal{P}(H_1) = \mathcal{P}(H_2)$$

Pair $(\mathcal{W},\ \sim)$ satisfies all the required properties of an equivalence relation [56]:

- *Reflexivity*: $\forall H \in \mathcal{W},\ (H \sim H)$
- *Symmetry*: $\forall H_1,\ H_2 \in \mathcal{W},\ (H_1 \sim H_2) \Rightarrow (H_2 \sim H_1)$
- *Transitivity*: $\forall H_1,\ H_2,\ H_3 \in \mathcal{W},\ (H_1 \sim H_2) \wedge (H_2 \sim H_3) \Rightarrow (H_1 \sim H_3)$

Example 5.3. Figure 5.5 shows three equivalent graphs $H_a \sim H_b \sim H_c$. Graph H_a in Figure 5.5(a) is taken from Example 5.1. Figure 5.5(b) shows graph H_b with the modified operational vector. It contains two variables $X = \{x,\ y\}$ which are restricted in the *one hot* manner: only encodings $(0,\ 1)$ and $(1,\ 0)$ are allowed with the restriction function



(a) Example graph (b) Two control variables (c) No redundant conditional arcs

Figure 5.5: Equivalent graphs

$\rho(x, y) = x \oplus y$. Graph H_c in Figure 5.5(c) does not contain any arc conditions (which are in fact redundant[3]) and also has inverted encodings compared to H_a.

In spite of the seeming difference between the three graphs, they are equivalent as they define the same set of two partial orders $\mathcal{P}(H_a) = \mathcal{P}(H_b) = \mathcal{P}(H_c) = \{P_1, P_2\}$:

$$\begin{cases} P_1 = \mathbf{po}(\mathbf{dg}\ H_a|_{x=0}) = \mathbf{po}(\mathbf{dg}\ H_b|_{x=0,\ y=1}) = \mathbf{po}(\mathbf{dg}\ H_c|_{x=1}) \\ P_2 = \mathbf{po}(\mathbf{dg}\ H_a|_{x=1}) = \mathbf{po}(\mathbf{dg}\ H_b|_{x=1,\ y=0}) = \mathbf{po}(\mathbf{dg}\ H_c|_{x=0}) \end{cases}$$

Checking whether two given graphs are equivalent or not is an important practical problem. A SAT-based solution is presented in Subsection 6.1.2.

It is useful to introduce a measure of graph complexity in order to compare them within the same equivalence class. For instance, graph H_c in Figure 5.5 has the simpler description in comparison with graphs H_a and H_b and is preferred in most cases.

Definition 5.11. The *complexity* (or *size*) $C(H)$ of graph $H(V, E, X, \rho, \phi)$ is measured in the number of literals contained in the restriction function ρ and conditions $\phi(z)$, $z \in V \cup E$:

$$C(H) \stackrel{\text{df}}{=} C(\rho) + \sum_{v \in V} C(\phi(v)) + \sum_{e \in E} C(\phi(e))$$

where $C(f)$, $f \in \mathcal{F}(X)$ denotes the literal count of a Boolean function f (see [107]).

[3]All of the arc conditions in this example are implied by the vertex conditions. See, for example, Figure 5.5(a): arc (a, c) has condition $\phi = x$, therefore it is switched off when $x = 0$. However, vertex c is also switched off when $x = 0$ and, as arc (a, c) cannot exist in the graph without vertex c, the arc condition can be relaxed to $\phi_{opt} = 1$. See Subsection 7.3.2 for a formal description of this optimisation.

Looking at the graphs in Figure 5.5 one can see that $C(H_a) = 0+2+6 = 8$, $C(H_b) = 2+2+6 = 10$, and $C(H_c) = 0+2+0 = 2$. So, graph H_c can be called *optimal* in this context. Methods for graphs size optimisation are addressed in Section 7.3. The removal of all redundant conditions from a graph leads to the *canonical CPOG description* of a set of encoded partial orders.

5.1.4 Addition

Definition 5.12. The result of *addition* of two graphs $H_1(V_1, E_1, X_1, \rho_1, \phi_1)$ and $H_2(V_2, E_2, X_2, \rho_2, \phi_2)$ is graph $H(V_1 \cup V_2, E_1 \cup E_2, X_1 \cup X_2, \rho_1 + \rho_2, \phi)$ where the vertex/arc conditions ϕ are defined as

$$\forall z \in V_1 \cup V_2 \cup E_1 \cup E_2,\ \phi(z) \stackrel{\text{df}}{=} \rho_1 \overline{\rho_2} \phi_1(z) + \overline{\rho_1} \rho_2 \phi_2(z)$$

Addition is denoted using the standard notation $H = H_1 + H_2$.

Remark 5.3. We consider events in V_1 and V_2 to belong to some universe $\mathcal{V}$ containing all the events of the modelled system, so their intersection can be non-empty: $V_1 \cap V_2 \neq \emptyset$; they can also coincide $V_1 = V_2$ or be disjoint $V_1 \cap V_2 = \emptyset$. The same holds for operational variables: $X_1 \subseteq \mathcal{X}$ and $X_2 \subseteq \mathcal{X}$ for some universe $\mathcal{X}$ of variables, and also for arcs ($E_1 \subseteq \mathcal{E}$ and $E_2 \subseteq \mathcal{E}$ respectively).

Theorem 5.1. *The pair* $(\mathcal{W}, +)$ *is a commutative semigroup* [56]*, i.e. the set of well-formed graphs* $\mathcal{W}$ *is closed under addition* $+$*, which is an associative and commutative operation.*

Proof. 1) Closure: $(H_1 \in \mathcal{W}) \wedge (H_2 \in \mathcal{W}) \Rightarrow (H_1 + H_2 \in \mathcal{W})$.

Let $H = H_1 + H_2$. According to Definition 5.6, graph H is well-formed iff its every non-singular complete projection $H|_\psi$ is valid.

Consider a non-singular complete projection $H|_\psi$. Non-singularity implies $\rho|_\psi = \rho_1|_\psi + \rho_2|_\psi = 1$ which is possible in one of the following three cases:

- $\rho_1|_\psi = \rho_2|_\psi = 1$ (H_1 and H_2 are in conflict with respect to opcode ψ). In this case, all the vertex and arc conditions $\phi(z)$ evaluate to zero: $\forall z,\ \phi(z)|_\psi = (\rho_1 \overline{\rho_2} \phi_1(z) +$

$\overline{\rho_1}\rho_2\phi_2(z))|_\psi = 1\cdot\overline{1}\cdot\phi_1(z)|_\psi + \overline{1}\cdot 1\cdot\phi_2(z)|_\psi = 0$. This projection generates an empty directed graph **dg** $H|_\psi$ which is obviously acyclic. Therefore, $H|_\psi$ is valid.

- $\rho_1|_\psi = 1$ and $\rho_2|_\psi = 0$. Here it is possible to show that **dg** $H|_\psi$ is equal to **dg** $H_1|_\psi$ and therefore $H|_\psi$ is valid due to the well-formedness of graph H_1 and validity of all its complete projections. For every $z \in V_1 \cup V_2 \cup E_1 \cup E_2$ condition $\phi(z)|_\psi$ in the complete projection $H|_\psi$ is equal to

$$\phi(z)|_\psi = (\rho_1\overline{\rho_2}\phi_1(z) + \overline{\rho_1}\rho_2\phi_2(z))|_\psi = 1\cdot\overline{0}\cdot\phi_1(z)|_\psi + \overline{1}\cdot 0\cdot\phi_2(z)|_\psi = \phi_1(z)|_\psi$$

  dg $H|_\psi$ has the same set of vertices and arcs as **dg** $H_1|_\psi$, hence $H|_\psi$ is valid.

- $\rho_1|_\psi = 0$ and $\rho_2|_\psi = 1$. This case is symmetric to the previous one: $H|_\psi$ is valid because **dg** $H|_\psi =$ **dg** $H_2|_\psi$.

The above cases show that any non-singular complete projection $H|_\psi$ is valid, and thus $H = H_1 + H_2$ is well-formed.

2) Associativity: $\forall H_1,\ H_2,\ H_3 \in \mathcal{W},\ (H_1 + H_2) + H_3 = H_1 + (H_2 + H_3)$.

Follows from the associativity of set union ($(V_1 \cup V_2) \cup V_3 = V_1 \cup (V_2 \cup V_3)$ etc.) and Boolean disjunction ($(\rho_1 + \rho_2) + \rho_3 = \rho_1 + (\rho_2 + \rho_3)$). To prove associativity with respect to conditions ϕ, let us define ρ' and ϕ' to be the restriction functions and conditions of graph $H' = H_1 + H_2$: $\rho' = \rho_1 + \rho_2$ and $\phi' = \rho_1\overline{\rho_2}\phi_1 + \overline{\rho_1}\rho_2\phi_2$. In the same way, let ρ and ϕ denote the restriction function and conditions of the final graph $H = H' + H_3$. So, $\rho = \rho' + \rho_3 = \rho_1 + \rho_2 + \rho_3$ while ϕ is equal to

$$\phi = \rho'\overline{\rho_3}\phi' + \overline{\rho'}\rho_3\phi_3 = (\rho_1 + \rho_2)\overline{\rho_3}(\rho_1\overline{\rho_2}\phi_1 + \overline{\rho_1}\rho_2\phi_2) + \overline{(\rho_1 + \rho_2)}\rho_3\phi_3 =$$

$$= (\rho_1 + \rho_2)(\rho_1\overline{\rho_2}\,\overline{\rho_3}\phi_1 + \overline{\rho_1}\rho_2\overline{\rho_3}\phi_2) + (\overline{\rho_1}\,\overline{\rho_2})\rho_3\phi_3 = \rho_1\overline{\rho_2}\,\overline{\rho_3}\phi_1 + \overline{\rho_1}\rho_2\overline{\rho_3}\phi_2 + \overline{\rho_1}\,\overline{\rho_2}\rho_3\phi_3$$

The result remains the same if the order of addition of the three graphs is altered: $H' = H_2 + H_3$, $H = H_1 + H'$. So, independently of the order, function $\phi(z)$ for a particular z will eventually be equal to $\rho_1\overline{\rho_2}\,\overline{\rho_3}\phi_1(z) + \overline{\rho_1}\rho_2\overline{\rho_3}\phi_2(z) + \overline{\rho_1}\,\overline{\rho_2}\rho_3\phi_3(z)$. Observe the correct scaling of the orthogonal coefficients from $\{\rho_1\overline{\rho_2},\ \overline{\rho_1}\rho_2\}$ to $\{\rho_1\overline{\rho_2}\,\overline{\rho_3},\ \overline{\rho_1}\rho_2\overline{\rho_3},\ \overline{\rho_1}\,\overline{\rho_2}\rho_3\}$.

3) Commutativity: $H_1 + H_2 = H_2 + H_1$.

Follows from the commutativity of set union ($V_1 \cup V_2 = V_2 \cup V_1$ etc.) and Boolean disjunction ($\rho_1 + \rho_2 = \rho_2 + \rho_1$ etc.) operations. □

Remark 5.4. When adding more than two graphs the redundant brackets can be omitted without ambiguity: $H_1 + H_2 + H_3$.

Corollary 1. *The general equation for conditions* ϕ *in graph* $H(V, E, X, \rho, \phi)$ *in case of addition of* $n \geqslant 2$ *graphs* $H_k(V_k, E_k, X_k, \rho_k, \phi_k)$, $1 \leqslant k \leqslant n$ *is*

$$\phi = \sum_{1 \leqslant k \leqslant n} (\phi_k \rho_k \prod_{\substack{1 \leqslant j \leqslant n \\ j \neq k}} \overline{\rho_j})$$

e.g. if $n = 3$ *the equation is* $\phi = \rho_1 \overline{\rho_2}\,\overline{\rho_3} \phi_1 + \overline{\rho_1} \rho_2 \overline{\rho_3} \phi_2 + \overline{\rho_1}\,\overline{\rho_2} \rho_3 \phi_3$.

In the same way as graphs H_1 and H_2 are considered to be specifications of certain behavioural scenarios over event domains V_1 and V_2, graph $H_1 + H_2$ is considered to be specification of the scenarios from both the graphs over the joint event domain $V = V_1 \cup V_2$. This is formally stated in the following theorem.

Theorem 5.2. *If* H_1 *and* H_2 *are well-formed graphs that are not in conflict then*

$$\mathcal{P}(H_1 + H_2) = \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$$

i.e. graph $H_1 + H_2$ *contains partial orders from both* H_1 *and* H_2.[4]

Proof. Let $H = H_1 + H_2$. At first let us show that $\mathcal{P}(H_1) \cup \mathcal{P}(H_2) \subseteq \mathcal{P}(H)$. Consider a partial order $P \in \mathcal{P}(H_1)$ (the proof for the case when $P \in \mathcal{P}(H_2)$ is similar due to symmetry between H_1 and H_2). By Definition 5.8, there must exist at least one possible valid opcode ψ such that $\rho_1|_\psi = 1$ and $P = \mathbf{po}(\mathbf{dg}\ H_1|_\psi)$. It is possible to show that $\mathbf{dg}\ H_1|_\psi = \mathbf{dg}\ H|_\psi$ (and thus H also defines P under the same assignment function):

1. The restriction function $\rho_2|_\psi$ of H_2 is not satisfied because H_1 and H_2 are not in conflict: $(\rho_1 \rho_2)|_\psi = \rho_1|_\psi \cdot \rho_2|_\psi = 1 \cdot \rho_2|_\psi = \rho_2|_\psi = 0$.

[4] An even stronger property holds: addition preserves the initial opcodes of the partial orders.

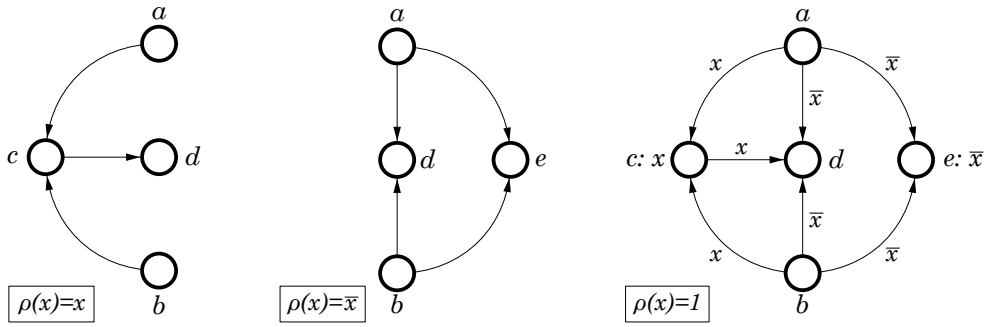
2. The restriction function ρ of H is satisfied with ψ: $\rho|_\psi = (\rho_1 + \rho_2)|_\psi = 1 + 0 = 1$.

3. Vertex/arc conditions $\phi(z)$ for $\forall z \in V_1 \cup E_1$ in $H|_\psi$ evaluate to the same values as in $H_1|_\psi$: $\phi(z)|_\psi = (\rho_1\overline{\rho_2}\phi_1(z) + \overline{\rho_1}\rho_2\phi_2(z))|_\psi = 1 \cdot \overline{0} \cdot \phi_1(z)|_\psi + \overline{1} \cdot 0 \cdot \phi_2(z)|_\psi = \phi_1(z)|_\psi$.

4. Vertex/arc conditions $\phi(z)$ for $\forall z \notin V_1 \cup E_1$ in $H|_\psi$ evaluate to 0: $\phi(z)|_\psi = \phi_1(z)|_\psi = 0$ (by Definition 5.1 of ϕ).

Therefore, the sets of vertices and arcs of **dg** $H|_\psi$ are the same as those of **dg** $H_1|_\psi$. Consequently, $P = \mathbf{po}(\mathbf{dg}\ H_1|_\psi) = \mathbf{po}(\mathbf{dg}\ H|_\psi)$ and therefore $P \in \mathcal{P}(H)$.

Now let us prove the reverse statement: $\mathcal{P}(H) \subseteq \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$. Consider a partial order $P \in \mathcal{P}(H)$. There must exist at least one possible valid opcode ψ such that $P = \mathbf{po}(\mathbf{dg}\ H|_\psi)$. The restriction function $\rho = \rho_1 + \rho_2$ must be satisfied which means that either ρ_1 or ρ_2 is satisfied but not both of them. Let it be ρ_1: $\rho_1|_\psi = 1$ and $\rho_2|_\psi = 0$ (the other case is again symmetric). This leads to the same conclusion as in the first part of the proof (see points (3) and (4)): **dg** $H_1|_\psi =$ **dg** $H|_\psi$. Therefore $P \in \mathcal{P}(H_1) \subseteq \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$. This completes the proof. □



(a) H_1 (b) H_2 (c) $H_1 + H_2$

Figure 5.6: Graph addition

Consider an example of addition shown in Figure 5.6. Both H_1 and H_2 specify a single scenario (see Table 5.1 for the details of the scenarios). The graphs are not in conflict ($\rho_1\rho_2 = x\overline{x} = 0$), the result of their addition $H_1 + H_2$ is shown in Figure 5.6(c). It contains both of the scenarios (as was demonstrated in Figure 5.3).

5.1.5 Scalar multiplication

Definition 5.13. Graph $H(V, E, X, \rho, \phi)$ can be *multiplied* by a Boolean function $f \in \mathcal{F}(Y)$ (which in our context can be called *scalar*). The resultant graph is $H'(V, E, X \cup Y, f\rho, \phi)$. The standard notation is used for scalar multiplication: $H' = f \cdot H$.[5]

Theorem 5.3. *For every Boolean function* f *and a well-formed graph* H*, graph* $H' = fH$ *is also well-formed and* $\mathcal{P}(H') \subseteq \mathcal{P}(H)$.

Proof. Every opcode ψ which is valid with respect to H is either singular with respect to H' (when $f|_\psi = 0$) or also valid (when $f|_\psi = 1$). In the latter case the partial order $P = \mathbf{po}(\mathbf{dg}\ H|_\psi)$ defined by ψ remains the same in H': $P = \mathbf{po}(\mathbf{dg}\ H'|_\psi)$ (because conditions $\phi(z)$ in H' are the same as in H). Thus, function f only 'filters out' some of the partial orders defined in H by setting an additional constraint to the restriction function ρ, and no new partial orders are introduced. □

Remark 5.5. Multiplication by $f = 1$ does not change a graph: $1 \cdot H = H$ and $\mathcal{P}(1 \cdot H) = \mathcal{P}(H)$.

Remark 5.6. Multiplication by $f = 0$ produces a singular graph: $\mathcal{P}(0 \cdot H) = \emptyset$.

Definition 5.14. A *linear combination* of $n \geqslant 1$ graphs H_1, H_2, ..., H_n and scalars f_1, f_2, ..., f_n is

$$\sum_{1 \leqslant k \leqslant n} f_k H_k = f_1 H_1 + f_2 H_2 + \cdots + f_n H_n$$

Any linear combination of well-formed graphs is also well-formed due to the closure of addition and scalar multiplication operations over well-formed graphs (Theorems 5.1 and 5.3).

5.1.6 Encoding conflict resolution

The operation of addition introduced in Subsection 5.1.4 produces a conservative result in case of an encoding conflict in the added graphs. In particular, if there is a false conflict between graphs H_1 and H_2 for a particular encoding ψ their sum $H_1 + H_2$ does

[5] As usual, multiplication by juxtaposition (omitting the '·') is allowed: $H' = fH$.

not contain the conflicting partial order $P = \mathbf{po}(\mathbf{dg}\ H_1|_\psi) = \mathbf{po}(\mathbf{dg}\ H_2|_\psi)$ at all, therefore $\mathcal{P}(H_1 + H_2) \neq \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$ (cf. Theorem 5.2).

In order to be able to add graphs with true and false conflicts preserving the conflicting partial orders in the sum, the following concept of *asymmetric addition* is introduced.

Definition 5.15. The result of *asymmetric addition* of two graphs $H_1(V_1, E_1, X_1, \rho_1, \phi_1)$ and $H_2(V_2, E_2, X_2, \rho_2, \phi_2)$ is linear combination $H_1 \vec{+} H_2 \stackrel{\text{df}}{=} H_1 + \overline{\rho_1} H_2$. Asymmetric addition is a *left-associative* operation, i.e. it is conventionally evaluated from left to right: $H_1 \vec{+} H_2 \vec{+} H_3 \stackrel{\text{df}}{=} (H_1 \vec{+} H_2) \vec{+} H_3$.

Remark 5.7. Asymmetric addition is closed over well-formed graphs ($(H_1 \in \mathcal{W}) \wedge (H_2 \in \mathcal{W}) \Rightarrow (H_1 \vec{+} H_2 \in \mathcal{W})$) but because of the asymmetry it is neither commutative ($H_1 \vec{+} H_2 \neq H_2 \vec{+} H_1$) nor associative ($(H_1 \vec{+} H_2) \vec{+} H_3 \neq H_1 \vec{+} (H_2 \vec{+} H_3)$) unlike normal addition.

Remark 5.8. It is possible to generalise the notion of linear combination for asymmetric addition of more than two graphs. Let ρ' be the the restriction function of graph $(H_1 \vec{+} H_2)$: $\rho' = \rho_1 + \overline{\rho_1}\rho_2 = \rho_1 + \rho_2$. This leads to $(H_1 \vec{+} H_2) \vec{+} H_3 = (H_1 + \overline{\rho_1} H_2) \vec{+} H_3 = H_1 + \overline{\rho_1} H_2 + \overline{\rho'} H_3 = H_1 + \overline{\rho_1} H_2 + \overline{\rho_1}\,\overline{\rho_2} H_3$. In general, linear combination for asymmetric addition of $n \geqslant 2$ graphs $H_k(V_k, E_k, X_k, \rho_k, \phi_k)$, $1 \leqslant k \leqslant n$ is

$$H_1 \vec{+} H_2 \vec{+} \ldots \vec{+} H_n = \sum_{1 \leqslant k \leqslant n} \Big(\prod_{1 \leqslant j < k} \overline{\rho_j} \Big) H_k$$

Theorem 5.4. *If* H_1 *and* H_2 *are well-formed graphs that are not in a true conflict then* $\mathcal{P}(H_1 \vec{+} H_2) = \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$.

Proof. Let $H = H_1 \vec{+} H_2 = H_1 + \overline{\rho_1} H_2$. At first, notice that graphs H_1 and $\overline{\rho_1} H_2$ are not in conflict: $\rho_1(\overline{\rho_1}\rho_2) = 0$. According to Theorems 5.2 and 5.3, $\mathcal{P}(H) = \mathcal{P}(H_1 + \overline{\rho_1} H_2) = \mathcal{P}(H_1) \cup \mathcal{P}(\overline{\rho_1} H_2) \subseteq \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$.

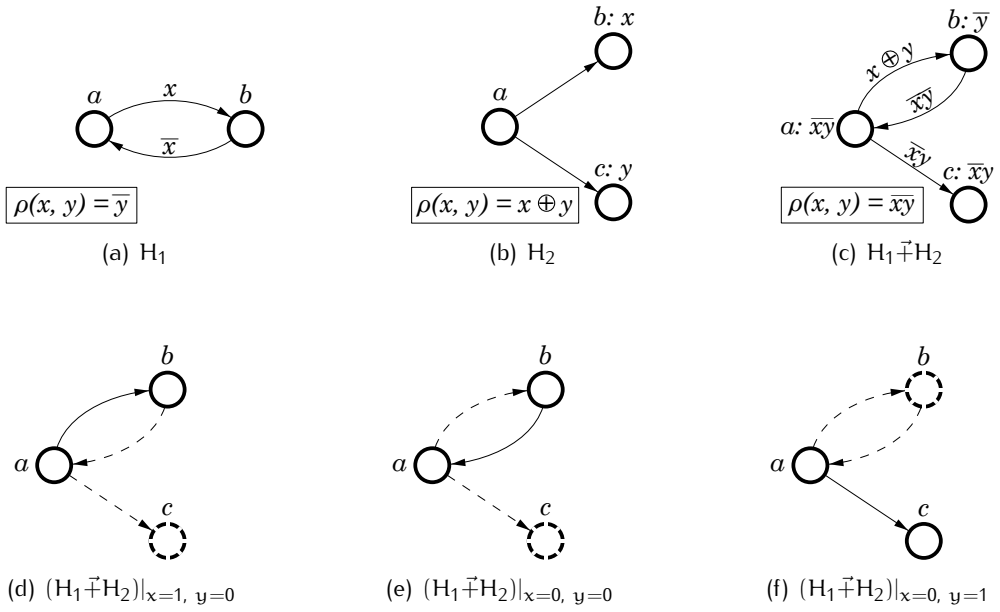
Now, let us prove the reverse statement $\mathcal{P}(H_1) \cup \mathcal{P}(H_2) \subseteq \mathcal{P}(H)$. Any partial order $P \in \mathcal{P}(H_1)$ must belong to $\mathcal{P}(H) = \mathcal{P}(H_1 + \overline{\rho_1} H_2)$ (by Theorem 5.2). Consider a partial order $P \in \mathcal{P}(H_2)$ which has encoding ψ: $P = \mathbf{po}(\mathbf{dg}\ H_2|_\psi)$. There can be two cases with respect to $\rho_1|_\psi$ value:

- $\rho_1|_\psi = 0$: $\mathcal{P}(\overline{\rho_1}H_2) = \mathcal{P}(1 \cdot H_2) = \mathcal{P}(H_2)$ (due to Remark 5.5). So, $P \in \mathcal{P}(H_2)$ also belongs to $\mathcal{P}(\overline{\rho_1}H_2)$ and thus $P \in \mathcal{P}(H)$.

- $\rho_1|_\psi = 1$, which means that ψ is a conflicting opcode. If the conflict is false, then $P = \mathbf{po}(\mathbf{dg}\ H_1|_\psi)$ and as was already shown, any partial order from graph H_1 is included into $\mathcal{P}(H)$.

So, both $\mathcal{P}(H_1) \subseteq \mathcal{P}(H)$ and $\mathcal{P}(H_2) \subseteq \mathcal{P}(H)$ hold. Together with $\mathcal{P}(H) \subseteq \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$ this proves that $\mathcal{P}(H_1 \vec{+} H_2) = \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$. □

Remark 5.9. If well-formed graphs H_1 and H_2 are in a true conflict with respect to ψ, i.e. $\mathbf{po}(\mathbf{dg}\ H_1|_\psi) \neq \mathbf{po}(\mathbf{dg}\ H_2|_\psi)$ then their asymmetric sum $H_1 \vec{+} H_2$ includes $\mathbf{po}(\mathbf{dg}\ H_1|_\psi)$ but not $\mathbf{po}(\mathbf{dg}\ H_2|_\psi)$.



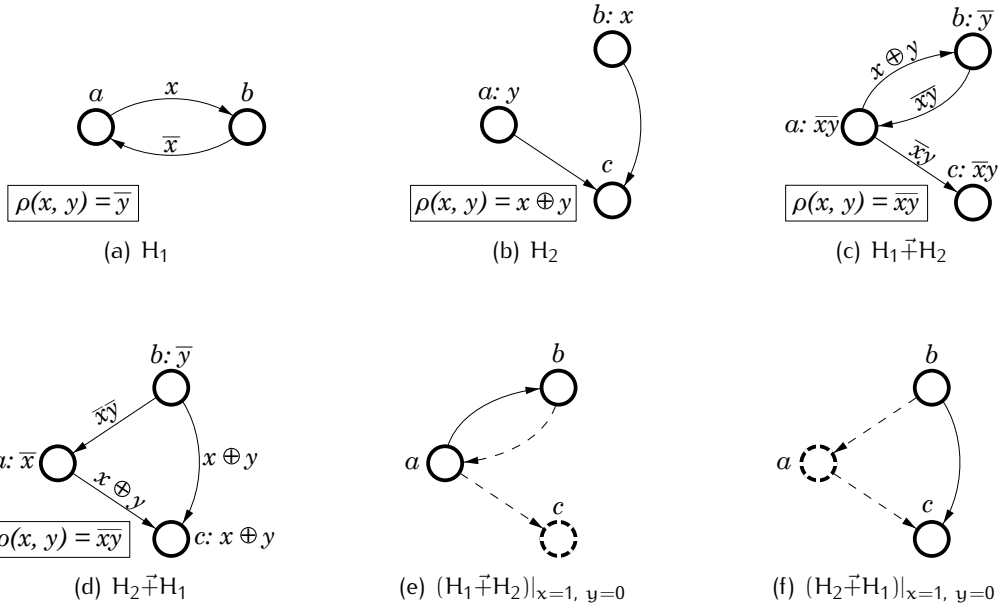
(a) H_1 (b) H_2 (c) $H_1 \vec{+} H_2$

(d) $(H_1 \vec{+} H_2)|_{x=1,\ y=0}$ (e) $(H_1 \vec{+} H_2)|_{x=0,\ y=0}$ (f) $(H_1 \vec{+} H_2)|_{x=0,\ y=1}$

Figure 5.7: Asymmetric addition: a false encoding conflict

Example 5.4. Consider an example of asymmetric addition of two graphs with a false encoding conflict shown in Figure 5.7. Graph H_1 (Figure 5.7(a)) defines two simple partial orders $P_1 = \{a \prec b\} = \mathbf{po}(\mathbf{dg}\ H_1|_{x=1,\ y=0})$ and $P_2 = \{b \prec a\} = \mathbf{po}(\mathbf{dg}\ H_1|_{x=0,\ y=0})$, while graph H_2 (Figure 5.7(b)) defines $P_1 = \{a \prec b\} = \mathbf{po}(\mathbf{dg}\ H_2|_{x=1,\ y=0})$ and $P_3 = \{a \prec c\} = \mathbf{po}(\mathbf{dg}\ H_2|_{x=0,\ y=1})$. One can see that $\psi = (1,\ 0)$ is a conflicting opcode, but the

conflict is false, because the corresponding partial orders are equal: $\mathbf{po}(\mathbf{dg}\ H_1|_{x=1,\ y=0}) = \mathbf{po}(\mathbf{dg}\ H_2|_{x=1,\ y=0}) = P_1$. Asymmetric sum $H_1 \vec{+} H_2$ shown in Figure 5.7(c) contains all the three partial orders: $\mathcal{P}(H_1 \vec{+} H_2) = \{P_1,\ P_2,\ P_3\} = \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$. The corresponding projections are demonstrated in Figures 5.7(d), (e), (f).



(a) H_1 (b) H_2 (c) $H_1 \vec{+} H_2$

(d) $H_2 \vec{+} H_1$ (e) $(H_1 \vec{+} H_2)|_{x=1,\ y=0}$ (f) $(H_2 \vec{+} H_1)|_{x=1,\ y=0}$

Figure 5.8: Asymmetric addition: a true encoding conflict

Example 5.5. Asymmetric addition of graphs with a true encoding conflict is demonstrated in Figure 5.8. Graph H_1 (Figure 5.8(a)) defines partial orders $P_1 = \{a \prec b\} = \mathbf{po}(\mathbf{dg}\ H_1|_{x=1,\ y=0})$ and $P_2 = \{b \prec a\} = \mathbf{po}(\mathbf{dg}\ H_1|_{x=0,\ y=0})$, while graph H_2 (Figure 5.8(b)) defines $P_3 = \{b \prec c\} = \mathbf{po}(\mathbf{dg}\ H_2|_{x=1,\ y=0})$ and $P_4 = \{a \prec c\} = \mathbf{po}(\mathbf{dg}\ H_2|_{x=0,\ y=1})$. Conflict under $\psi = (1,\ 0)$ is true, because the corresponding partial orders are different: $P_1 = \mathbf{po}(\mathbf{dg}\ H_1|_{x=1,\ y=0}) \neq \mathbf{po}(\mathbf{dg}\ H_2|_{x=1,\ y=0}) = P_3$. Two asymmetric sums $H_1 \vec{+} H_2$ and $H_2 \vec{+} H_1$ are shown in Figures 5.8(c) and (d). The difference between them is due to the different conflict resolution choice: the former graph keeps partial order $P_1 = \{a \prec b\}$ while the latter keeps $P_3 = \{b \prec c\}$. This fact is demonstrated in Figures 5.8(e) and (f) which show the complete projections of these graphs under the conflicting opcode $(x,\ y) = (1,\ 0)$. So, the result of asymmetric sum depends significantly on the order of arguments: $\mathcal{P}(H_1 \vec{+} H_2) = \{P_1,\ P_2,\ P_4\}$, while $\mathcal{P}(H_2 \vec{+} H_1) = \{P_2,\ P_3,\ P_4\}$ (the leftmost argument has the highest 'priority').

The presented set of algebraic operations (addition, scalar multiplication, and asymmetric addition) provides the necessary toolkit for 'safe by construction' structural CPOG synthesis and optimisation (see Chapter 7). The operations are also applicable to the dynamic CPOG model introduced in Section 5.2.

5.2 The dynamic model

Conditional Partial Order Graphs introduced in Section 5.1 are *static* in the sense that every behavioural scenario specified by graph H under particular opcode ψ is a partial order $\mathbf{po}(\mathbf{dg}\ H|_\psi)$ which does not have any internal choice in its behaviour: all the choice is embedded in the encoding ψ. Therefore, it is impossible to pass any information from earlier events to later ones, and there is no way to specify any internally branching process within a static CPOG model.

Example 5.6. Consider operation $r = \mathit{diff}(p,\ q)$ of computing the difference between two values p and q stored in memory: $\mathit{diff}(p,\ q) \stackrel{\text{df}}{=} |p - q|$. The flow of execution of the operation breaks up into the following primitive actions:

- Load register A from memory: *mov* A, p;
- Load register B from memory: *mov* B, q;
- Compute their difference and store the result in A:
  - Compare registers A and B: *cmp* A, B;
  - If $(A < B)$ then swap the registers: *swap* A, B;
  - Subtract B from A, store the result in A: *sub* A, B;
- Save register A into memory: *mov* r, A.

Let the result of comparison (action *cmp* A, B) be denoted as y: $y \stackrel{\text{df}}{=} (A < B)$. Note that y is used in the next conditional action of swapping *swap* A, B. Clearly, y is undefined in the beginning of the computation, because values A and B can only be compared after they have been loaded from memory. The basic CPOG model does not

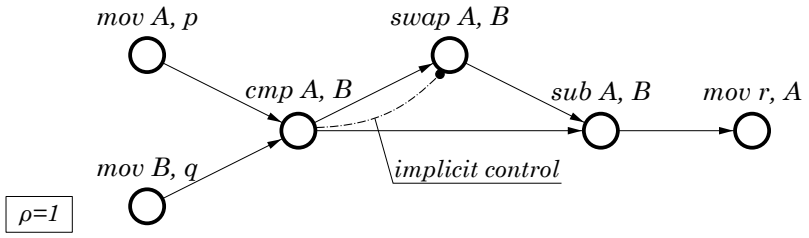


Figure 5.9: A static CPOG with implicit control specifying *diff*(p, q) operation

allow such dynamic evaluation of variable y during the execution phase. Figure 5.9 shows an unsophisticated workaround for the specification which uses an implicit dependency between actions *cmp* A, B and *swap* A, B (denoted with a dash-dotted arc) that can switch vertex *swap* A, B on or off according to the result of comparison. Certainly, such an implicit specification cannot be considered acceptable.

This section introduces the extended model which supports the dynamic opcode evaluation (further referred to as the *dynamic CPOG model*). It was first presented in [72], which addressed the limitations of the static model and provided two different approaches to overcome them. The first one used hierarchical static graphs composition (see Subsection 5.2.1). The second approach proposed dynamic graphs as a natural generalisation of the basic model.

The formal definition of the dynamic model is given in Subsection 5.2.2. Behavioural semantics and classification of different types of starting and terminating states of dynamic graphs are discussed in Subsections 5.2.3 and 5.2.4.

5.2.1 Hierarchical static graphs composition

The general schematic view of CPOG-based microcontrollers is shown in Figure 5.10. A microcontroller receives an opcode via the opcode interface and schedules the corres-

Figure 5.10: Schematic view of a CPOG-based microcontroller

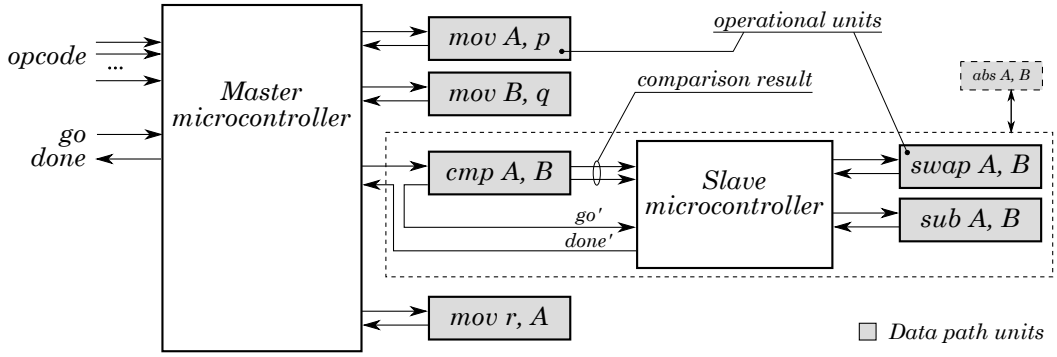
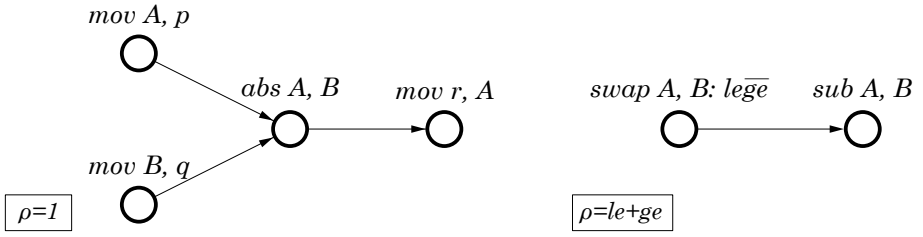


Figure 5.11: Hierarchical composition of CPOG-based microcontrollers

ponding partial order of events (in this example we consider the events to be data path operational units with a request/acknowledgement interface).

It is possible to use hierarchical composition of *master* and *slave* CPOG-based microcontrollers to implement control for $r = \mathit{diff}(p, q)$ operation as shown in Figure 5.11. Three operational blocks $\{\mathit{cmp}, \mathit{swap}, \mathit{sub}\}$ are grouped together forming an equivalent of *abs* A, B operational unit. Signal go' for the slave microcontroller is the acknowledgement signal from the comparator *cmp* A, B which ensures that the comparison result is already computed and the slave controller can use it as an opcode, hence staying within the basic CPOG model. The comparison result is represented by two signals le and ge: $le = 1$ ($ge = 1$) is set iff the value in register A is less (greater) or equal to the value in register B. The slave microcontroller performs the actions according to these signals (it swaps registers A and B if $le = 1$ and $ge = 0$, and executes *sub* A, B) and sets signal $done'$ that is used as an acknowledgement by the master microcontroller.

CPOG specifications for the master and slave microcontrollers are shown Figure 5.12. The master graph is trivial and does not contain any conditions. The graph for the slave controller executes either sequence of operations *swap* A, B $\rightarrow$ *sub* A, B or just a single operation *sub* A, B depending on the result of comparison. Vertex *swap* A, B has condition $le \cdot \overline{ge}$ which is true iff value in register A is strictly less then value in register B: $le \cdot \overline{ge} = (A \leqslant B) \cdot \overline{(A \geqslant B)} = (A < B)$. Note that the slave graph has restriction function $\rho = le + ge$ which does not allow opcode $\psi = (0, 0)$ because it is inconsistent (it implies that $A < B$ and $A > B$ simultaneously).



(a) The master graph (b) The slave graph

Figure 5.12: Static CPOGs for the master and slave microcontrollers

Although the hierarchical approach helps to extend the range of systems modelled with static CPOGs, it requires non-trivial decomposition of partial orders (or, sometimes, over-structuring them at the expense of performance or area). Such decomposition is not possible for more complicated examples. Subsection 5.2.2 presents another approach to modelling internally branching systems with CPOGs.

5.2.2 Dynamic CPOGs

This subsection introduces the dynamic CPOG model formally. The model allows opcodes to be dynamically evaluated during the retrieval and execution of the corresponding partial order. Figure 5.13 shows schematic view of the microcontroller for $r = \mathit{diff}(p, q)$ operation (see Example 5.6), which uses dynamic opcode evaluation: signals $\{le, ge\}$ are considered to be a dynamic part of opcode. Their value is undefined until comparator *cmp* A, B sets them according to the result of comparison.

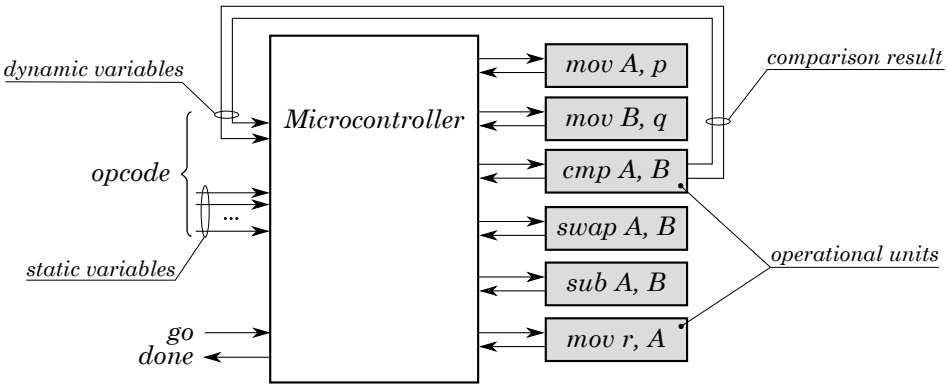


Figure 5.13: Microcontroller with dynamic opcode evaluation

The dynamic CPOG for this microcontroller is shown in Figure 5.14. It is more natural and understandable than the two separate CPOGs for the hierarchical design. Action *cmp* A, B controls variables $\{le, ge\}$ (this fact is denoted below the vertex) so that after its execution the condition $le + ge = 1$ holds. This restricts the comparison result to meaningful combinations only (the meaningless result $le = ge = 0$ is forbidden). After its execution action *swap* A, B is included into the current partial order only if needed.

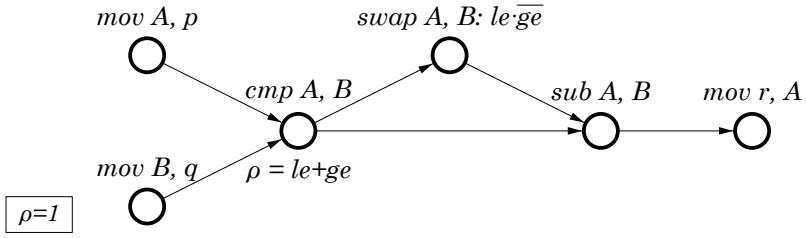


Figure 5.14: Dynamic Conditional Partial Order Graph

Definition 5.16. Conditional Partial Order Graph H(V, E, X, ρ, ϕ) is *dynamic* if every vertex $v \in V$ has an ordered pair $\langle Y_v, \rho_v \rangle$ associated with it. Y_v and ρ_v have the following semantics:

- Set Y_v contains *dynamic operational variables* that are allowed to change their values during the execution of the action corresponding to vertex v. Before execution, their values are undefined (can be either 0 or 1). Y_v is called the *controlled set* of v. Controlled sets of any two vertices u, $v \in V$, $u \neq v$ do not overlap: $Y_u \cap Y_v = \emptyset$.
- Operational variables X are called *static* and cannot change their values.
- Union of all the dynamic operational variables in the graph is denoted as Y:

$$Y \stackrel{\text{df}}{=} \bigcup_{v \in V} Y_v$$

- Function $\rho_v \in \mathcal{F}(Y_v)$ is called the *vertex restriction function*. It restricts values of the controlled variables Y_v in the same way as graph restriction function ρ restricts static variables X. Note that restriction on variables Y_v comes into force only after execution of the action associated with v.

- Domain of vertex and arc conditions $\phi(z),\ z \in V \cup E$ is extended from X to $X \cup Y$: $\phi(z) \in \mathcal{F}(X \cup Y)$. In other words, the conditions can include both static and dynamic operational variables.

The dynamic graph specifying the scenario from Example 5.6 is shown in Figure 5.14. The graph does not have any static operational variables ($X = \emptyset$), so it specifies a single scenario (computation of $r = \mathit{diff}(p,\ q)$). There are two dynamic variables $Y = \{le,\ ge\}$ controlled by vertex *cmp* A, B (the corresponding restriction function is $\rho = le + ge$ as shown below the vertex). The values of signals le and ge are not defined before the execution of the comparator, which compares registers A and B and sets the signals according to the result. Obviously, le and ge cannot be both set to 0 but the three other combinations are meaningful, thus the restriction function should be $le + ge$ (it is a contradiction only when $le = ge = 0$). The conditional action *swap* A, B is executed only if $le\overline{ge} = 1$ (this holds when $A < B$). Thus, the scenario can be dynamically changed according to the outcome of the comparison.

Dynamic CPOGs have a graphical representation similar to that of static CPOGs. In addition, vertex restriction function ρ_v is depicted below the corresponding vertex v (unless the controlled set of v is empty, i.e. $\langle Y_v,\ \rho_v \rangle = \langle \emptyset,\ 1 \rangle$, in which case ρ_v is omitted for clarity). Variables Y_v can be observed as parameters of ρ_v.

Subsection 5.2.3 introduces the *firing rules* and the notions of *opcode*, *configuration* and *state* to formally describe the behaviour of a system specified with a dynamic CPOG.

5.2.3 Behavioural semantics

Definition 5.17. *Opcode* $\psi :\ X \cup Y \to \{0,\ 1\}$ assigns Boolean values to all the static and dynamic operational variables in the graph.

The *preset* $\bullet_\psi v$ of a vertex $v \in V$ with respect to opcode ψ is

$$\bullet_\psi v \stackrel{\text{df}}{=} \{u \in V,\ \phi(u)|_\psi \cdot \phi((u,\ v))|_\psi = 1\}$$

It contains all the vertices $u \in V$ which precede vertex v in the partial order defined by complete projection $H|_\psi$.

The *postset* $v\bullet_\psi$ is defined similarly:

$$v\bullet_\psi \stackrel{\text{df}}{=} \{u \in V,\ \phi(u)|_\psi \cdot \phi((v,\ u))|_\psi = 1\}$$

Definition 5.18. *Configuration* $C \subseteq V$ is the set of vertices whose corresponding actions have been already performed.

A configuration C is called *valid* with respect to opcode ψ iff

$$\forall v \in C,\ (\bullet_\psi v \subseteq C) \wedge (\phi(v)|_\psi = 1)$$

In other words, a valid configuration may contain only those vertices which have their presets in C, and are present in complete projection $H|_\psi$.

Definition 5.19. An ordered pair $\langle C,\ \psi\rangle$ is called a *state*; it fully describes the current state of the modelled system.

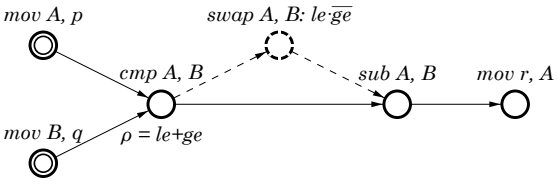
A state $\langle C,\ \psi\rangle$ is called *valid* iff configuration C is valid, restriction function ρ is satisfied ($\rho|_\psi = 1$), and restriction functions of all the vertices in the configuration are also satisfied ($\forall v \in C,\ \rho_v|_\psi = 1$).

In a given state $S = \langle C,\ \psi\rangle$ a vertex $v \in V \setminus C$ is *enabled to fire* iff

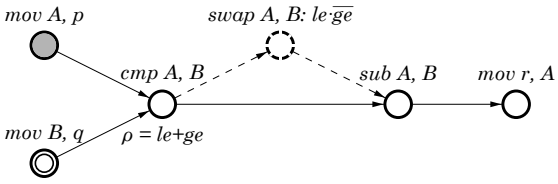
- it is present in the partial order determined by complete projection $H|_\psi$: $\phi(v)|_\psi = 1$;
- its preset is a subset of the configuration: $\bullet_\psi v \subseteq C$.

A *firing* of an enabled vertex $v \in V \setminus C$ produces a new configuration $C' = C \cup \{v\}$. Opcode ψ can also be affected by the firing if vertex v is associated with an action that changes some dynamic variables. In particular, the firing of vertex $v \in V$ affects variables in its controlled set Y_v such that the restriction function becomes satisfied: $\rho_v = 1$ (see Example 5.7). Firing is considered to be an atomic event: state S is changed momentarily into the new state $S' = \langle C',\ \psi'\rangle$.

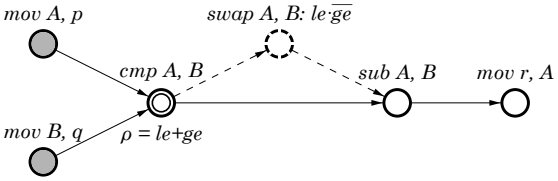
Note that every dynamic operational variable has at most one control vertex and therefore it can change at most once. This means that system state $\langle C,\ \psi\rangle$ can only change



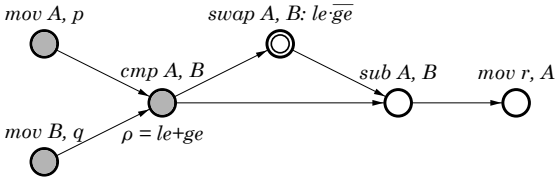
(a) The initial state: *mov* A, p and *mov* B, q are enabled to fire; $le = ge = 0$



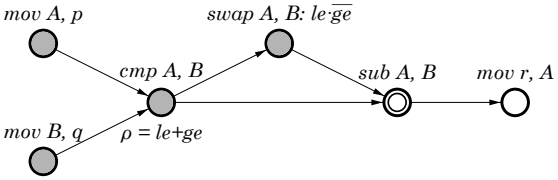
(b) *mov* A, p has fired; *mov* B, q is still enabled



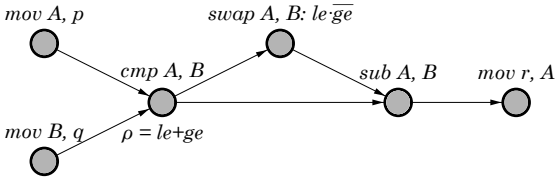
(c) *mov* B, q has fired enabling *cmp* A, B



(d) *cmp* A, B has fired changing (le, ge) to (1, 0); enabled vertex *swap* A, B appears in the graph



(e) *swap* A, B has fired enabling *sub* A, B



(f) *sub* A, B and *mov* r, A have fired in sequence

Figure 5.15: Example of dynamic CPOG evolution

monotonically during the system evolution: once a vertex is added to a configuration it cannot be removed (the *monotonicity of configuration*) and once a dynamic variable is evaluated it remains constant (the *monotonicity of control vector*).

Example 5.7. Figure 5.15 shows a step by step evolution of the system specified with the dynamic graph in Figure 5.14. At start, configuration C is empty (no actions have been performed) and two vertices are enabled to fire: *mov* A, p and *mov* B, q (enabled vertices are depicted as ◎). Dynamic variables $Y = \{le, ge\}$ can have arbitrary values, in this example we assume them being equal to 0: $le = ge = 0$. Suppose that event *mov* A, p happens first (see Figure 5.15(b)). Configuration C becomes equal to {*mov* A, p} (vertices included into C are shown as filled circles ●). After the subsequent firing of vertex *mov* B, q comparator *cmp* A, B becomes enabled because its preset is contained in C. The comparator fires and changes its controlled variables according to the result of comparison, for instance, to $(le, ge) = (1, 0)$ signifying $(A < B)$ outcome. This leads to the inclusion of vertex *swap* A, B and its arcs into the graph and it becomes the next

enabled vertex, preventing event *sub* A, B from happening too early (see Figure 5.15(d)). Any other outcome of the comparison would leave *swap* A, B excluded from the graph: it would be skipped and event *sub* A, B would become enabled immediately. The system evolution finishes after successive firings of vertices *swap* A, B, *sub* A, B and *mov* r, A and the system comes to the *final state* shown in Figure 5.15(f).

Different types of states are classified in Subsection 5.2.4.

5.2.4 Initial states, final states and deadlocks

Definition 5.20. The *initial state* of a system is $\langle \emptyset,\ \psi_0 \rangle$, i.e. no actions have been performed ($C = \emptyset$) and all (static and dynamic) variables are set to some initial values ψ_0. Initial state $\langle \emptyset,\ \psi_0 \rangle$ of a dynamic graph $H(V,\ E,\ X,\ \rho,\ \phi)$ is *valid* iff $\rho|_{\psi_0} = 1$.

Starting from the initial state the system evolves by firing enabled vertices and finally reaches a state when no vertex is enabled. Such a state can either be a *final state* or a *deadlock* (see Definitions 5.21 and 5.22). Notice that the system must terminate eventually because each firing adds a vertex $v \in V \setminus C$ into configuration C.

Definition 5.21. State $\langle C,\ \psi \rangle$ is a *final state* iff there is no vertex $v \in V \setminus C$ that is present in complete projection $H|_\psi$:

$$\forall v \in V \setminus C,\ \phi(v)|_\psi = 0$$

Definition 5.22. State $\langle C,\ \psi \rangle$ is a *deadlock* iff there is no enabled vertex but at least one vertex $v \in V \setminus C$ is present in $H|_\psi$:

$$(\forall v \in V \setminus C,\ (\bullet_\psi v \not\subseteq C) \vee (\phi(v)|_\psi = 0)) \wedge (\exists v \in V \setminus C,\ \phi(v)|_\psi = 1)$$

A deadlock is a situation wherein two or more competing events are waiting for the other to happen, and thus neither ever does. Such a situation is caused by the fact that complete projection $H|_\psi$ contains a cycle and is therefore invalid (see Definition 5.5).

Definition 5.23. Graph $H(V,\ E,\ X,\ \rho,\ \phi)$ is called *deadlock free* if there is no valid deadlock state which is reachable from any valid initial state $\langle \emptyset,\ \psi_0 \rangle,\ \rho|_{\psi_0} = 1$.

Figure 5.16 shows a graph containing both final states and deadlocks. x_1 and x_2 are static variables; dynamic variables $Y = \{y_1, y_2\}$ are controlled by vertex b. Figure 5.16(a) shows the final state reachable through the following firing sequence. The initial state is $\langle \emptyset, (x_1 = x_2 = y_1 = y_2 = 0) \rangle$; b is enabled to fire. During its firing $\{y_1, y_2\}$ change from zeroes to $(y_1 = 0, y_2 = 1)$. Then d and c fire, and the system comes to the final state: vertex a is not present in $H|_\psi$ since $\phi(a)|_\psi = (x_1 + x_2)|_\psi = 0$.

(a) Final state. $(x_1, x_2, y_1, y_2) = (0, 0, 0, 1)$ (b) Deadlock. $(x_1, x_2, y_1, y_2) = (1, 0, 1, 1)$

Figure 5.16: Final state and deadlock

A reachable deadlock state is shown in Figure 5.16(b): starting from $\langle \emptyset, (x_1 = 1, x_2 = y_1 = y_2 = 0) \rangle$ the system evolves by firing vertices a and b. Now if variables $Y_b = \{y_1, y_2\}$ evaluate to $y_1 = y_2 = 1$ then the system comes to the deadlock state: both vertices c and d exist in projection $H|_\psi$ but none of them is enabled to fire.

If a system does not contain any dynamic variables ($Y = \emptyset$) then the final state (or deadlock) can be uniquely determined from the initial state. Otherwise the system can terminate in different states according to the different dynamic variable changes during the system evolution.

Detection of reachable deadlocks, testing reachability of a particular state, and other dynamic CPOG verification problems are addressed in Section 6.2.

The overall system evolution cycle is shown in Figure 5.17. It has two phases: *active* and *reset*. The active phase starts in some initial state according to the initial assignment of operational variables; the system then goes through a sequence of states terminating in a final state (or in a deadlock state in case of a specification or implementation error). The reset phase consists of two concurrent processes: resetting the configuration to initial state $C = \emptyset$ (which might involve resetting the controllers used in the active phase), and

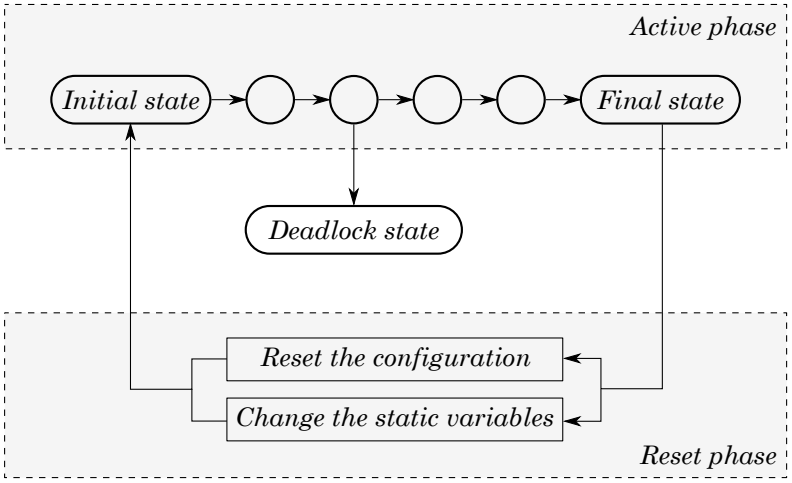


Figure 5.17: System evolution cycle

changing the static variables to select the scenario for the next evolution cycle (this might be the request for the next codeword from the environment).

Chapter 6

Verification

The static and dynamic CPOGs introduced in Chapter 5 have many important properties that require automated verification. Figure 6.1 shows a hierarchy of the verification problems addressed in this chapter (the numbers in front of the problem names indicate the corresponding subsections of the chapter). It is important to verify properties in their hierarchic order, otherwise the results may be incorrect, e.g. detection of invalid states in a dynamic graph makes sense only if the graph is deadlock free and well-formed. Therefore, before verification of a particular property one has to ensure that all the hierarchically precedent properties have been already verified.

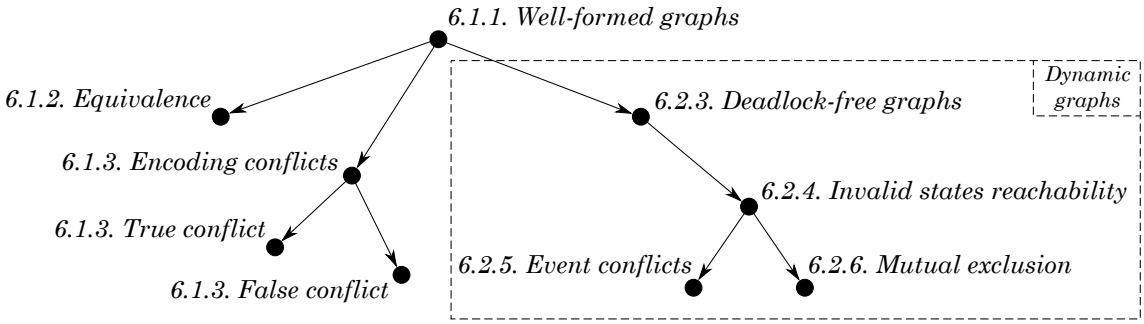


Figure 6.1: Hierarchy of CPOG verification problems

Section 6.1 presents solutions to the structural CPOG verification problems which are relevant both to static and dynamic graphs. The focus of Section 6.2, however, is limited to the dynamic CPOG properties; they are based on the behavioural semantics of dynamic graphs and are not applicable to the static model.

It is tempting to apply comprehensively studied Petri Nets for verification of CPOGs, i.e. to convert a given graph to the equivalent contextual Petri Net and to reuse the wealth of existing model checking tools available for the latter (see, for example, [83] and [93] where this approach was applied to the verification of speed-independent circuits and static data flow structures, respectively). However, the Petri Net model is more general and verification of most of its properties is PSPACE-complete [35]. The most efficient verification algorithms unfold a given Petri Net and use the SAT-based NP-complete techniques on the obtained prefix [45]. But thanks to the acyclicity of CPOGs projections (each vertex can fire at most once in every execution run) it is possible to apply the SAT techniques directly to a CPOG without its computationally expensive unfolding, thereby staying within the NP complexity class.

The *Boolean satisfiability problem* (SAT) is to decide whether a given Boolean formula $F(x_1, x_2, \ldots, x_n)$ is satisfiable or not, i.e. if it is possible to find an assignment of Boolean values $(\alpha_1, \alpha_2, \ldots, \alpha_n) \in \{0, 1\}^n$ to the variables $(x_1, x_2, \ldots, x_n)$ which makes the formula true: $F(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$. The SAT-based verification methods construct a Boolean formula F that is satisfiable if and only if a particular property under verification holds, and then give this formula to one of many available generic SAT solvers, which have become very efficient in the recent years, e.g. [34, 74].

The SAT-based approach to the verification of CPOGs was first presented in [72], which gave SAT characterisations for the most important properties of the dynamic model (see Subsections 6.2.3-6.2.6). This chapter applies the same approach to the verification of the structural properties and relations (Subsections 6.1.1-6.1.3), therefore all the techniques presented here belong to the NP complexity class.

6.1 Structural properties and relations

This section addresses verification of structural CPOG properties and relations (graph well-formedness, equivalence, true and false encoding conflicts), which were defined in Section 5.1 for the static CPOG model.

6.1.1 Well-formedness

According to Definition 5.6 a graph $H(V,\ E,\ X,\ \rho,\ \phi)$ is well-formed iff every opcode ψ which is allowed by the restriction function (i.e. $\rho|_\psi = 1$) produces an acyclic directed graph **dg** $H|_\psi$. The verification formula for this property $\mathcal{NW}$ (abbreviation for 'not well-formed') consists of two clauses. The first clause enumerates all the allowed opcodes ψ and coincides with the restriction function ρ. The second clause $\mathcal{CYCLE}$ checks if a particular opcode ψ generates a cyclic projection $H|_\psi$:

$$\mathcal{NW} = \rho \cdot \mathcal{CYCLE}$$

The intuition behind this SAT formulation is clarified by the following example.

(a) H_1 (b) H_2

Figure 6.2: Well-formed and not well-formed graphs

Example 6.1. Figure 6.2 shows two graphs H_1 and H_2 which differ from each other only in their restriction functions $\rho_1 = \bar{x}+\bar{y}+\bar{z}$ and $\rho_2 = x+y+z$. The $\mathcal{CYCLE}$ clause for the both graphs is the same: $\mathcal{CYCLE} = xyz$. It captures the fact that the graph contains cycle $a \to b \to c \to a$ iff all the operational variables $X = \{x,\ y,\ z\}$ are equal to 1. Hence, the verification formula for the first graph is

$$\mathcal{NW}_1(x,\ y,\ z) = \rho_1 \cdot \mathcal{CYCLE} = (\bar{x}+\bar{y}+\bar{z})xyz$$

It is easy to see that this formula is unsatisfiable, i.e. it is impossible to find an opcode $\psi = (\alpha,\ \beta,\ \gamma)$ such that $\mathcal{NW}_1(\alpha,\ \beta,\ \gamma) = 1$. This means that every allowed opcode produces an acyclic projection, thus H_1 is well-formed.

The situation is opposite for the second graph. The verification formula

$$\mathcal{NW}_2(x,\ y,\ z) = \rho_2 \cdot \mathcal{CYCLE} = (x+y+z)xyz$$

is satisfiable: $\mathcal{NW}_1(1,\ 1,\ 1) = 1$. This means that graph H_2 is not well-formed, as it has a cyclic complete projection induced by opcode $\psi = (1,\ 1,\ 1)$.

There are many ways to generate the $\mathcal{CYCLE}$ clause for a given graph H. The method presented here uses the fact that if a directed graph $G(V,\ E)$ has a cycle then it is possible to find a non empty set of vertices $\emptyset \subset S \subseteq V$ such that every vertex $v \in S$ has a precedent u in this set: $\exists u \in S,\ (u,\ v) \in E$.

Let $V = \{v_1,\ v_2,\ \ldots,\ v_{|V|}\}$, and a Boolean variable $s_k,\ k = 1 \ldots |V|$ equals 1 iff $v_k \in S$. Then the following Boolean formula is satisfiable iff the complete projection $H|_\psi$ has a cycle[1]:

$$\mathcal{CYCLE}(s_1,\ s_2,\ \ldots,\ s_{|V|}) = \sum_{1 \leqslant j \leqslant |V|} s_j \cdot \prod_{1 \leqslant j \leqslant |V|} \left(s_j \Rightarrow \phi(v_j) \cdot \sum_{1 \leqslant k \leqslant |V|} s_k \cdot \phi((v_k,\ v_j))\right)$$

The first clause ensures that set S is not empty, while the second clause checks that every vertex in this set has a precedent. Note that opcode ψ is constrained outside of this clause, hence for a particular ψ functions $\phi(z),\ z \in V \cup E$ are constants $\phi(z)|_\psi$ and induce a directed graph **dg** $H|_\psi$.

This generic approach applied to Example 6.1 results in formula

$$\mathcal{CYCLE}(s_1,\ s_2,\ s_3) = (s_1 + s_2 + s_3)(s_1 \Rightarrow s_3xz)(s_2 \Rightarrow s_1xy)(s_3 \Rightarrow s_2yz)$$

which is satisfiable if opcode $\psi = (1,\ 1,\ 1)$ (cf. formula $\mathcal{CYCLE} = xyz$ which was obtained manually); in this case the satisfiable assignment is $s_1 = s_2 = s_3 = 1$ which means that the cycle is formed by vertices $S = \{a,\ b,\ c\}$.

[1] In this work $a \Rightarrow b$ stands for *Boolean implication* indicating 'b if a' relation, and $a \Leftrightarrow b$ stands for *Boolean equivalence* indicating 'b if and only if a' relation [56].

6.1.2 Equivalence

This subsection provides a method to verify that two given graphs $H_1(V_1, E_1, X_1, \rho_1, \phi_1)$ and $H_2(V_2, E_2, X_2, \rho_2, \phi_2)$ define the same set of partial orders (equivalence) and that the opcodes for these partial orders are the same.

The first part of the verification process is to make sure that the sets of opcodes in the given graphs coincide, i.e. that $X_1 = X_2$ and $\rho_1 = \rho_2$. The second part is to verify that these opcodes generate the same partial orders. The corresponding SAT verification formula $\mathcal{NEQ}$ (abbreviation for 'not equivalent') is

$$\mathcal{NEQ} = \rho_1 \cdot \mathcal{TC}(H_1) \cdot \mathcal{TC}(H_2) \cdot \mathcal{DIFF}$$

Function $\mathcal{TC}(H)$ constructs the transitive closure t_{ij} of the DAG defined with the complete projection $H|_\psi$ (opcodes ψ are enumerated with term ρ_1), and clause $\mathcal{DIFF}$ searches for a mismatch in the two transitive closures:

$$\mathcal{DIFF} = \sum_{1 \leqslant i,\ j \leqslant |V_1 \cup V_2|} t^1_{ij} \oplus t^2_{ij}$$

where t^1 and t^2 are the transitive closures of the first and the second graphs respectively. If $t^1_{ij} \neq t^2_{ij}$ for at least one pair (i, j) then $\mathcal{DIFF}$ evaluates to 1 making $\mathcal{NEQ}$ satisfiable.

To conclude, the verification function $\mathcal{NEQ}$ is satisfiable if there is an opcode ψ such that partial orders defined by $H_1|_\psi$ and $H_2|_\psi$ are different.

6.1.3 Encoding conflicts

Checking for encoding conflicts in two graphs and detection of the conflict type (true or false) is very similar to the verification of equivalence which was explained in the previous subsection.

According to Definition 5.9 two graphs $H_1(V_1, E_1, X_1, \rho_1, \phi_1)$ and $H_2(V_2, E_2, X_2, \rho_2, \phi_2)$ are in an encoding conflict iff $\rho_1 \cdot \rho_2 \neq 0$. To determine if the conflict is true it is necessary to compare the partial orders generated by the

conflicting opcodes. Formula $\mathcal{TEC}$ (stands for 'true encoding conflict') is

$$\mathcal{TEC} = \rho_1 \cdot \rho_2 \cdot \mathcal{TC}(H_1) \cdot \mathcal{TC}(H_2) \cdot \mathcal{DIFF}$$

where $\mathcal{TC}(H)$ and $\mathcal{DIFF}$ are the same as in the previous subsection: $\mathcal{TC}(H_1)$ and $\mathcal{TC}(H_2)$ generate the transitive closures of **dg** $H_1|_\psi$ and **dg** $H_2|_\psi$, while $\mathcal{DIFF}$ checks whether the transitive closures are different or not. Clause $\rho_1 \cdot \rho_2$ ensures that opcode ψ is conflicting, i.e. it belongs to both the graphs ($\rho_1 = 1$ and $\rho_2 = 1$).

Therefore, if $\mathcal{TEC}$ is satisfiable then the conflict is true, otherwise the conflict is false, because despite the fact that $\rho_1 \cdot \rho_2 \neq 0$ there is no difference in the partial orders generated with the conflicting opcodes.

6.2 Dynamic properties

This section presents SAT characterisations for the dynamic CPOG properties (deadlocks, invalid state reachability, event conflicts, and mutual exclusion), and a polynomial algorithm that reconstructs a trace of events leading to a particular system state. Subsection 6.2.2 describes how to map the states of dynamic CPOGs into the domain of Boolean functions in order to enable the use of the SAT approach.

The overall verification flow of systems specified with the dynamic CPOG model is shown in Figure 6.3. A system is first checked for the absence of deadlocks (Subsection 6.2.3) and then for the impossibility to reach an invalid state from a valid one (Subsection 6.2.4). After these basic verification procedures the system can be checked for higher level properties, e.g. event conflict freedom (Subsection 6.2.5), or if two given events are mutually exclusive (Subsection 6.2.6). All the verification procedures provide the error state $\langle C, \psi \rangle$ in case of an incorrect system behaviour; this state can be passed to the trace reconstruction algorithm (Subsection 6.2.1) to obtain the trace of events leading to the failure. This information can be used to correct the given system specification and rerun the verification.

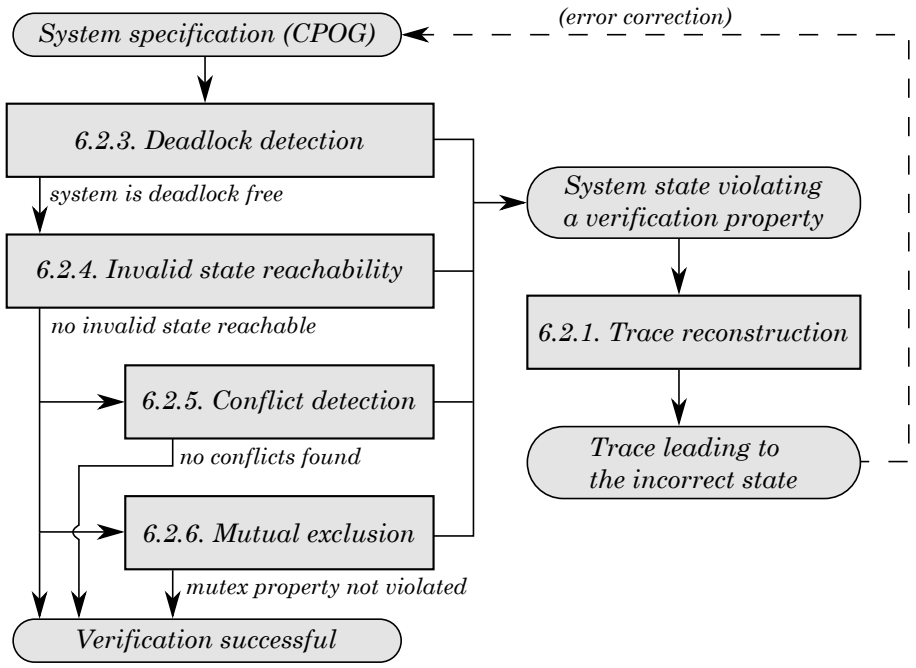
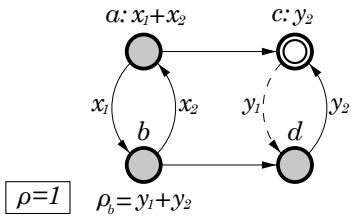


Figure 6.3: Dynamic CPOGs verification flow

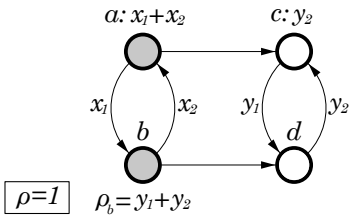
6.2.1 Trace reconstruction algorithm

It is often important to know the trace of events leading to a particular state in the system, e.g. to track the cause of a failure. This subsection presents a polynomial algorithm that given a valid state $S = \langle C, \psi \rangle$ generates trace $S_0 \to \cdots \to S$ of valid states leading from the initial state $S_0 = \langle \emptyset, \psi \rangle$ to S, or reports that such a trace does not exist.

Not every valid state $\langle C, \psi \rangle$ is reachable from the initial state $\langle \emptyset, \psi \rangle$. The reason is that configuration C can contain such a set of vertices that there is no firing sequence leading to it. Two examples of such states are shown in Figure 6.4. Both the states have subset $\{a, b\}$ in the configuration, but the static variables $x_1 = x_2 = 1$ introduce a mutual



(a) Valid state: $(x_1, x_2, y_1, y_2) = (1, 1, 0, 1)$



(b) Deadlock: $(x_1, x_2, y_1, y_2) = (1, 1, 1, 1)$

Figure 6.4: Unreachable states

dependency between vertices a and b: it is impossible to fire them in any order, thus any state with configuration $\{a, b\} \subseteq C$ is unreachable. Moreover, one can observe that the initial state $\langle \emptyset, x_1 = x_2 = 1 \rangle$ is a deadlock and it is the only reachable state provided that $x_1 = x_2 = 1$.

The following important property of deadlock free CPOGs is exploited in the verification algorithms presented in this paper.

Theorem 6.1. *If a CPOG* H *is deadlock free then every valid state* $\langle C, \psi \rangle$ *is reachable from the initial state* $\langle \emptyset, \psi \rangle$ *through a sequence of valid states.*

Proof. By induction. If $C = \emptyset$ then $\langle C, \psi \rangle$ is the initial state already, otherwise let $v \in C$ be such a vertex that configuration C does not contain any vertices of the postset of v, i.e. $C \cap v\bullet_\psi = \emptyset$. If it is impossible to select such v then C must contain a directed cycle $\langle v_0, v_0 \rangle = (v_0, v_1, \ldots, v_n = v_0)$, $v_k \in C$, $k = 0 \ldots n$ which implies that the system has a deadlock reachable from state $\langle C - \bigcup_{0 \leqslant k < n} v_k, \psi \rangle$. This contradicts the fact that H is deadlock free. Thus $v \in C$, $C \cap v\bullet_\psi = \emptyset$ can be selected. Now we can *unfire* vertex v and obtain a valid state $\langle C', \psi' \rangle = \langle C - v, \psi \rangle$ from which $\langle C, \psi \rangle$ is reachable by firing of vertex v. Note that the unfiring of vertex v does not change opcode ψ (i.e. $\psi' = \psi$). This is allowed because before the execution of v the values of its controlled variables Y_v are undefined (by Definition 5.16) and nothing restricts them from being equal to the result of the execution.

Configuration $C' = C - v$ is smaller than C and we can conclude inductively that eventually it becomes empty and the initial state $\langle \emptyset, \psi \rangle$ is reached. During the process it is possible to construct *trace* $(\langle \emptyset, \psi \rangle, \ldots, \langle C'', \psi \rangle, \langle C', \psi \rangle, \langle C, \psi \rangle)$ of valid states leading from the initial state $\langle \emptyset, \psi \rangle$ to state $\langle C, \psi \rangle$. □

The proof of Theorem 6.1 is constructive and is used as a basis for the recursive algorithm of trace reconstruction shown in Algorithm 1. The algorithm directly matches the proof but it can be further optimised using the *reverse topological sorting algorithm* [22] resulting in linear complexity $O(|V| + |E|)$.

Algorithm 1 Trace reconstruction

```
function TRACE(H, ⟨C, ψ⟩)
{
  if (C = ∅) then return (); // empty trace

  for all (v ∈ C) do
    if (∀u ∈ C, φ((v, u))|ψ = 0) then
      return TRACE(H, ⟨C − v, ψ⟩) + ⟨C, ψ⟩;

  return deadlock_found_error; // CPOG H is not deadlock free
}
```

6.2.2 SAT formulation

This subsection presents a method of mapping a system state $\langle C, \psi \rangle$ into the Boolean domain. This is needed in order to apply SAT to the dynamic CPOG verification problems.

A system state $\langle C, \psi \rangle$ is encoded with two vectors of Boolean variables:

- Configuration C is described with $|V|$ variables conf_v, $v \in V$ such that $\mathsf{conf}_v = 1$ iff vertex v is included into the configuration: $v \in C$.
- Opcode ψ is described with $|X|$ variables val_x, $x \in X$, such that $\mathsf{val}_x = 1$ iff $\psi(x) = 1$. Notation $\phi_{\mathsf{val}}(z)$, $z \in V \cup E$ is used to denote the value of a vertex/arc condition in complete projection $H|_\psi$.

A SAT verification formula is a conjunction of constraints. The constraints ensure that variables conf_v and val_x describe a system state $\langle C, \psi \rangle$ which is relevant to a particular verification problem. For instance, if we are looking only for the states with valid configurations then we have to use the following *valid configuration constraint* $\mathcal{CONF}$:

$$\mathcal{CONF} = \left(\prod_{v \in V} \mathsf{conf}_v \Rightarrow \phi_{\mathsf{val}}(v) \right) \left(\prod_{v \in V} \left(\mathsf{conf}_v \Rightarrow \prod_{u \in \bullet_\psi v} \mathsf{conf}_u \right) \right)$$

In accordance with Definition 5.18 it consists of two clauses which ensure that:

- a vertex $v \in V$ can be in the configuration ($\mathsf{conf}_v = 1$) only if it exists in $H|_\psi$;
- if a vertex $v \in V$ is in the configuration then all the vertices of its preset $\bullet_\psi v$ must be in the configuration as well.

Another constraint that is often used is the *opcode constraint* $\mathcal{CODE}$:

$$\mathcal{CODE} = \rho(\mathsf{val}_{x_1}, \ldots, \mathsf{val}_{x_{|X|}}) \cdot \prod_{v \in V} \mathsf{conf}_v \Rightarrow \rho_v(\mathsf{val}_{y_1}, \ldots, \mathsf{val}_{y_{|Y_v|}})$$

It captures the fact that the CPOG restriction function ρ must be satisfied with values $(\mathsf{val}_{x_1}, \ldots, \mathsf{val}_{x_{|X|}})$, while the restriction function ρ_v of a vertex $v \in C$ must be satisfied with values $(\mathsf{val}_{y_1}, \ldots, \mathsf{val}_{y_{|Y_v|}})$ of its controlled set Y_v.

Boolean function $\mathit{enabled}_v$, $v \in V$ is also introduced in order to simplify some of the further equations:

$$\mathit{enabled}_v = \overline{\mathsf{conf}_v} \cdot \phi_{\mathsf{val}}(v) \cdot \prod_{u \in \bullet_\psi v} \mathsf{conf}_u$$

Thus $\mathit{enabled}_v = 1$ iff vertex $v \in V$ is enabled to fire in the current state $\langle \mathsf{conf}, \mathsf{val} \rangle$ according to Definition 5.19.

6.2.3 Deadlock detection

Detection of deadlocks is different from the other verification problems because the deadlock freedom property ensures that all the valid system states are reachable from the initial state (see Theorem 6.1). Therefore if a deadlock verification procedure finds a deadlock state it is quite possible that this state is not reachable. However, the beauty of this property is that if a particular deadlock state is unreachable then there must exist another deadlock state which is reachable and which occurs before the detected one and prevents it from being reachable (this follows from the proof of Theorem 6.1). Therefore the presented approach does not guarantee that the found deadlock is reachable but it does guarantee that if it finds a deadlock then there exists a reachable deadlock in the system. Note that the found state can be checked for reachability in polynomial time using the trace reconstruction algorithm presented in Subsection 6.2.1.

Figure 6.4(b) shows an example of unreachable deadlock state $\langle C = \{a, b\}, x_1 = x_2 = y_1 = y_2 = 1 \rangle$. This deadlock is unreachable because of the existence of reachable deadlock state $\langle C = \emptyset, x_1 = x_2 = y_1 = y_2 = 1 \rangle$. To reach the former deadlock state the system should somehow resolve the latter one, what can only be done by violation of one

of the conflicting dependencies between vertices a and b.

The verification formula for this problem (named $\mathcal{RD}$, i.e. 'reachable deadlock') is conjunction of constraints $\mathcal{CONF}$, $\mathcal{CODE}$ (which together define a valid state $\langle \mathsf{conf}, \mathsf{val} \rangle$) and the following constraint $\mathcal{DEAD}$ which is true iff there is no enabled vertex but at least one unfired vertex is present in $H|_\psi$ (a deadlock state by Definition 5.22):

$$\mathcal{DEAD} = \prod_{v \in V} \overline{enabled_v} \cdot \sum_{v \in V} \left(\overline{\mathsf{conf}_v} \cdot \phi_{\mathsf{val}}(v) \right)$$

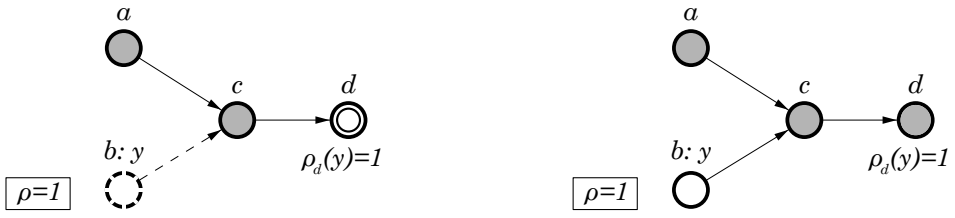
This gives us the following verification formula:

$$\mathcal{RD} = \mathcal{CONF} \cdot \mathcal{CODE} \cdot \mathcal{DEAD}$$

If a SAT solver finds such an assignment of variables conf_v and val_x that $\mathcal{RD} = 1$ then there is a deadlock state $\langle \mathsf{conf}, \mathsf{val} \rangle$ in the system, otherwise the system is deadlock free.

6.2.4 Invalid states reachability

The next basic CPOG property that we are going to verify is reachability of an invalid state from a valid one. An example of such a situation is shown in Figure 6.5. The current system configuration is $C = \{a, c\}$, the only dynamic variable y is set to zero, and it is controlled by vertex d (Subfigure (a)). This state is valid because vertex b is not present in the current complete projection $H|_\psi$: $\phi(b) = y = 0$. However, vertex d is enabled to fire and it can change the value of y leading to the invalid state: $C = \{a, c, d\}$, $y = 1$ (Subfigure (b)). This state violates validity of the configuration, because now vertex $c \in C$ has vertex b in its preset which is not in the configuration.



(a) $C = \{a, c\}$, $y = 0$ (b) $C = \{a, c, d\}$, $y = 1$

Figure 6.5: Example of a reachable invalid state

Verification formula $\mathcal{ISR}$ (i.e. 'invalid state reachability') is conjunction of constraints $\mathcal{CONF}$, $\mathcal{CODE}$ (which together define a valid state $\langle \mathsf{conf}, \mathsf{val} \rangle$) and the following constraint $\mathcal{IS}$ which is satisfiable if there is an invalid state $\langle \mathsf{conf}', \mathsf{val}' \rangle$ reachable from state $\langle \mathsf{conf}, \mathsf{val} \rangle$:

$$\mathcal{IS} = \sum_{v \in V} \mathit{enabled}_v \cdot \mathcal{FIRE}_v \cdot \mathcal{CODE}' \cdot \overline{\mathcal{CONF}'}$$

where $\mathcal{CODE}'$ is the opcode constraint for variables conf'_v and val'_x (we assume that vertex v fires correctly, i.e. it sets its controlled variables according to its restriction function and thus $\mathcal{CODE}'$ constraint is not violated); term $\overline{\mathcal{CONF}'}$ ensures that the new state $\langle \mathsf{conf}', \mathsf{val}' \rangle$ has an invalid configuration. Function $\mathcal{FIRE}_v$ constructs state $\langle \mathsf{conf}', \mathsf{val}' \rangle$ by firing vertex v in state $\langle \mathsf{conf}, \mathsf{val} \rangle$:

$$\mathcal{FIRE}_v = \mathsf{conf}'_v \cdot \prod_{u \neq v} \mathsf{conf}'_u \Leftrightarrow \mathsf{conf}_u \cdot \prod_{x \notin Y_v} \mathsf{val}'_x \Leftrightarrow \mathsf{val}_x$$

It ensures that configuration conf' differs from conf only with vertex v, and the only variables that are allowed to change are those belonging to its control set Y_v.

To conclude, there is an invalid state reachable from a valid state in the given CPOG iff the following Boolean formula is satisfiable:

$$\mathcal{ISR} = \mathcal{CONF} \cdot \mathcal{CODE} \cdot \mathcal{IS}$$

For the example in Figure 6.5 a SAT solver should find the following assignment of variables:

$$\langle (\mathsf{conf}_a = \mathsf{conf}_c = 1,\ \mathsf{conf}_b = \mathsf{conf}_d = 0),\ (\mathsf{val}_y = 0) \rangle$$
$$\langle (\mathsf{conf}'_a = \mathsf{conf}'_c = \mathsf{conf}'_d = 1,\ \mathsf{conf}'_b = 0),\ (\mathsf{val}'_y = 1) \rangle$$

If $\mathcal{ISR}$ is unsatisfiable then there is no reachable invalid state in the system.

6.2.5 Event conflicts

Two events are said to be in an *event conflict* iff there is a reachable state $\langle C, \psi \rangle$ in which both of them are enabled to fire but firing of one of them disables the other. Note

that an event conflict does not necessarily leads to a deadlock or an invalid state, and the monotonicity of configuration C and opcode ψ is not violated. Event conflicts only affect the monotonicity of function $enabled_v$, $v \notin C$ and can eventually lead to glitches or hazards in the gate-level implementation of the controller.

(a) $C = \emptyset$, $y = 0$, $enabled_c = 1$ (b) $C = \{a\}$, $y = 1$, $enabled_c = 0$

Figure 6.6: Event conflict between events a and c

An example of a simple event conflict is shown in Figure 6.6. All the three vertices $\{a, b, c\}$ are enabled to fire in the initial state $\langle \emptyset, y = 0 \rangle$ (Subfigure (a)). But if vertex a fires and sets its controlled variable y to 1 then vertex c becomes disabled due to the appearance of arc (b, c) in the graph (see Subfigure (b)).

Verification formula $\mathcal{EC}$ (abbreviation for 'event conflict')

$$\mathcal{EC} = \mathcal{CONF} \cdot \mathcal{CODE} \cdot \mathcal{CS}$$

is constructed from the valid state constraint $\mathcal{CONF} \cdot \mathcal{CODE}$ and constraint $\mathcal{CS}$ which is satisfiable iff there is a state $\langle conf', val' \rangle$ reachable from $\langle conf, val \rangle$ by firing an enabled vertex v such that another enabled vertex u becomes disabled:

$$\mathcal{CS} = \sum_{\substack{v,u \in V \\ u \neq v}} enabled_v \cdot enabled_u \cdot \mathcal{FIRE}_v \cdot \mathcal{CODE}' \cdot \overline{enabled'_u}$$

As before terms $\mathcal{CODE}'$ and $enabled'_u$ operate on variables $\langle conf', val' \rangle$ of the new state constrained with function $\mathcal{FIRE}_v$. The solution for the example in Figure 6.6 is:

$$\langle (conf_a = conf_b = conf_c = 0),\ (val_y = 0) \rangle$$
$$\langle (conf'_a = 1,\ conf'_b = conf'_c = 0),\ (val'_y = 1) \rangle$$

6.2.6 Mutual exclusion

A CPOG specification can contain more than one vertex corresponding to the same action in the modelled system. Figure 6.7 shows an example of an MSP430 (a general purpose microprocessor [2]) instruction specification. The graph represents the operational flow for an ALU operation with addressing mode **#123 to Rn/PC,** e.g. addition of a constant to a register Rn or program counter (PC). *PC++* is the program counter increment, *IR* is the next instruction word reading, *ALU* is an arithmetic operation; variable *pca* (program counter access) is set to one if the second operand is PC and to zero otherwise.

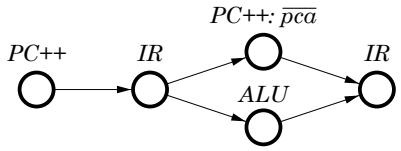


Figure 6.7: Multiple action occurrence

The scenario is to increment PC (action *PC++*), and to load the addition constant (which occupies the next word in the code memory after the instruction itself) into the instruction register (action *IR*). After that the constant is added to the second operand (action *ALU*). If the second operand is not PC then it is incremented concurrently (the second occurrence of *PC++*). The last step is to load the next instruction into the instruction register (the second occurrence of *IR*).

To avoid arbitration it is necessary to be sure that there are no concurrent requests to the same action. Looking at Figure 6.7 one can easily see that the two occurrences of actions *PC++* and IR are mutually exclusive. But there can be much more sophisticated CPOGs and an automated procedure is needed to be sure that the mutex property is not violated for a pair of given events.

The verification formula checking the mutex property for given vertices $v,\ u \in V$ is

$$\mathcal{MUTEX}(v,\ u) = \mathcal{CONF} \cdot \mathcal{CODE} \cdot \mathit{enabled}_v \cdot \mathit{enabled}_u$$

It is satisfiable iff there is a valid state $\langle \mathsf{conf},\ \mathsf{val} \rangle$ such that both vertices v and u are enabled to fire.

6.3 Summary

This chapter substantiated the use of SAT-based techniques for verification of structural and dynamic graph properties and relations. The structural verification problems have been addressed in Section 6.1 and are relevant for both the static and dynamic CPOG models. These problems include checking a given graph for well-formedness, testing equivalence, and identification of true and false encoding conflicts between given graphs (Subsections 6.1.1 through 6.1.3). It should be noted that these verification procedures are computationally intensive and their use should be kept to a minimum by exploiting the algebraic properties of CPOGs, i.e. by using the 'safe' operations from CPOG algebra which were formally proved to preserve the certain structural properties (e.g. well-formedness) in the previous chapter (Section 5.1).

Section 6.2 presented verification flow for the dynamic CPOG model. The flow consists of four verification stages (deadlocks and invalid states reachability analysis, checking for conflicts and mutual exclusion of events), and a trace reconstruction algorithm which generates the trace of events leading to a given system state and can be used for systems diagnostics and correction.

The presented verification procedures are implemented as a part of the CPOG design flow toolkit which is discussed in Appendix.

Chapter 7

Synthesis and optimisation

Chapter 5 demonstrated that a Conditional Partial Order Graph can specify many different behavioural scenarios of a system in a compact form, while Chapter 6 presented techniques for the verification of various correctness criteria of such a specification. However, it still has not been shown how to build CPOGs, i.e. how to synthesise a CPOG specification given a system described with a set of scenarios. This problem was first addressed in [71] where the initial synthesis ideas were presented. The ideas were further developed and formalised in [65] on the basis of algebraic CPOG operations that guarantee the correctness of the synthesised graph [70]. Section 7.1 discusses the initial CPOG synthesis problem and its generalisation in details.

The problem of synthesis is closely related to mapping CPOGs into Boolean equations which is aimed to produce a physical implementation of the specified microcontroller. The mapping procedure is fairly transparent, thus the final area and latency of the microcontroller strongly depend on the size and structural properties of its CPOG representation. Therefore, it is important to use appropriate techniques during the stage of synthesis to obtain the minimum cost microcontroller. A CPOG can also undergo various optimisation procedures which exploit its structural and functional properties and aim to reduce its complexity by equivalence-preserving transformations [71]. The mapping and optimisation techniques (introduced in Sections 7.2 and 7.3) are structural, i.e. they do not explore the system state space that results in algorithms of high efficiency.

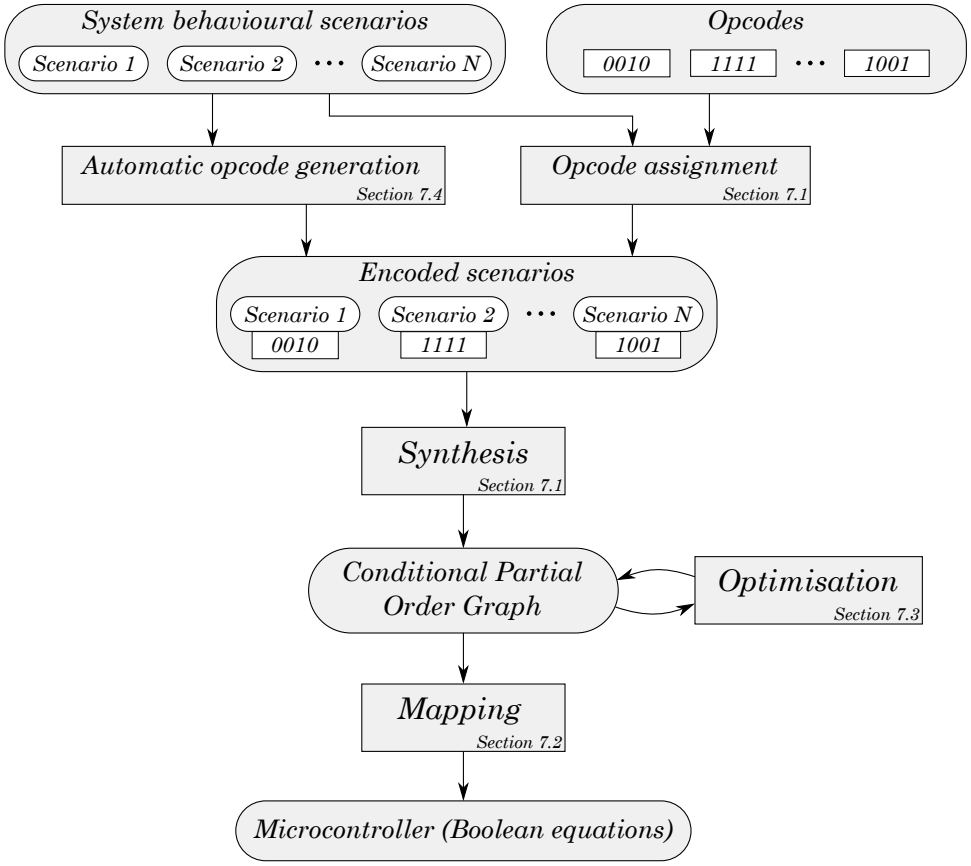


Figure 7.1: Synthesis, mapping and optimisation of CPOGs

At the stage of CPOG synthesis it is necessary to set up a correspondence between the given scenarios and their opcodes. In this way a particular scenario can be later extracted from the synthesised CPOG by providing its opcode (recall Example 5.2 from Chapter 5). This correspondence is called an *encoding scheme* and is often given as a part of system specification. Subsections 7.1.1 through 7.1.3 present several encoding schemes which are often encountered in practice. There are cases, however, when a designer does not have any requirements on the opcodes and in this case they are usually assigned arbitrarily. This does not lead to the optimal microcontroller, because a particular opcode assignment has a strong influence on the synthesised CPOG. The optimal assignment of opcodes can be computed automatically using the methods presented in [66], providing an opportunity for *implementation-aware opcode synthesis* discussed in Section 7.4.

Figure 7.1 shows interconnections between the different stages of CPOG-based microcontroller synthesis addressed in this chapter.

7.1 Synthesis

This section formally defines the problem of CPOG synthesis and solves it using a structural approach based on CPOG algebra. The initial formulation of the problem is given below, while its generalised version is introduced in Subsection 7.1.4.

Definition 7.1. (*CPOG synthesis from partial orders*). Consider a system that has n behavioural scenarios, which are specified with partial orders $\{P_1, P_2, \ldots, P_n\}$. The objective is to synthesise a CPOG $H(V, E, X, \rho, \phi)$ such that

$$\mathcal{P}(H) = \{P_1, P_2, \ldots, P_n\} \tag{7.1}$$

This short formulation captures the fact that H must contain only the given scenarios, but the opcodes of the scenarios are not specified in any way: they are implicit. If the opcodes are given explicitly as a part of system specification, it is necessary to refine (7.1) with an additional constraint.

Definition 7.2. (*CPOG synthesis from partial orders with the opcode constraint*). Let a given system be described with a set of n pairs $\{(P_1, \psi_1), (P_2, \psi_2), \ldots, (P_n, \psi_n)\}$, such that scenario P_k has opcode $\psi_k \in \{0, 1\}^{|X|}$ over set X of operational variables. Then the objective is to synthesise a CPOG $H(V, E, X, \rho, \phi)$ such that requirement (7.1) is satisfied and the following additional constraint holds:

$$\forall 1 \leqslant k \leqslant n,\ \mathbf{po}(\mathbf{dg}\ H|_{\psi_k}) = P_k \tag{7.2}$$

In other words, a complete projection $H|_{\psi_k}$ must generate partial order P_k. It is assumed that the system specification does not contain two equal opcodes, i.e. $\psi_j \neq \psi_k$ for any $1 \leqslant j < k \leqslant n$.

It is easier to solve the second problem because of the smaller search space: the opcodes are already given and cannot be changed. The first problem can be trivially reduced to the second one with an arbitrary encoding of the scenarios but this does not yield the optimal solution. A better approach is presented in Section 7.4 which provides

an algorithm for automatic optimal opcode generation. The rest of this section addresses the constrained synthesis problem from Definition 7.2.

The key idea behind the presented synthesis approach is to represent H as the following linear combination of complete projections $H_k = \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k)$:

$$H = f_1H_1 + f_2H_2 + \ldots + f_nH_n = \sum_{1\leqslant k\leqslant n} f_kH_k = \sum_{1\leqslant k\leqslant n} f_k\mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k) \tag{7.3}$$

where *encoding functions* $f_k \in \mathcal{F}(X)$ are *orthogonal*, i.e. $f_jf_k = 0$, $1 \leqslant j < k \leqslant n$ and are not contradictions: $f_k \neq 0$, $1 \leqslant k \leqslant n$. This construction is substantiated by the following proposition.

Proposition 7.1. *Linear combination (7.3) satisfies the synthesis requirement (7.1).*

Proof. We have to prove that $\mathcal{P}\big(\sum_{1\leqslant k\leqslant n} f_k\mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k)\big) = \{P_1,\ P_2,\ \ldots,\ P_n\}$. This can be done in several steps by application of definitions and theorems from CPOG algebra.

1. Let $H'_k = f_kH_k$. Observe that graphs H'_k are well-formed (by Theorem 5.3) and have no mutual encoding conflicts due to the orthogonality of encoding functions f_k. Therefore, $\mathcal{P}\big(\sum_{1\leqslant k\leqslant n} H'_k\big) = \bigcup_{1\leqslant k\leqslant n} \mathcal{P}(H'_k)$ by Theorem 5.2.

2. Now one can see that $\mathcal{P}(H'_k) = \mathcal{P}(f_kH_k) = \mathcal{P}(H_k)$ because functions f_k are not contradictions and the restriction functions of graphs H_k are equal to 1 (see Definition 5.4 and Theorem 5.3).

3. The last step is to notice that $\mathcal{P}(H_k) = \mathcal{P}(\mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k)) = \{P_k\}$ by Remark 5.2.

This completes the proof:

$$\mathcal{P}\left(\sum_{1\leqslant k\leqslant n} f_kH_k\right) = \bigcup_{1\leqslant k\leqslant n} \mathcal{P}(f_kH_k) = \bigcup_{1\leqslant k\leqslant n} \mathcal{P}(\mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k)) = \bigcup_{1\leqslant k\leqslant n} \{P_k\}$$

□

In order to satisfy the additional synthesis requirement (7.2) it is necessary to constrain encoding functions f_k. This is formalised in the next proposition.

Proposition 7.2. *Linear combination (7.3) satisfies the additional synthesis requirement (7.2) if encoding functions* f_k *and opcodes* ψ_k *meet the following condition:*

$$\forall 1 \leqslant k \leqslant n,\ f_k|_{\psi_k} = 1$$

Proof. We have to prove that $\mathbf{po}(\mathbf{dg}\ H|_{\psi_k}) = P_k$ for every $1 \leqslant k \leqslant n$. Consider complete projection $H|_{\psi_k} = \left(\sum_{1 \leqslant j \leqslant n} f_j H_j\right)|_{\psi_k}$. Due to the orthogonality of encoding functions and the fact that $f_k|_{\psi_k} = 1$ it is possible to conclude that

$$H|_{\psi_k} = \left(\sum_{1 \leqslant j \leqslant n} f_j H_j\right)\Bigg|_{\psi_k} = (0 \cdot H_1 + 0 \cdot H_2 + \cdots + 1 \cdot H_k + \cdots + 0 \cdot H_n)|_{\psi_k} = H_k|_{\psi_k}$$

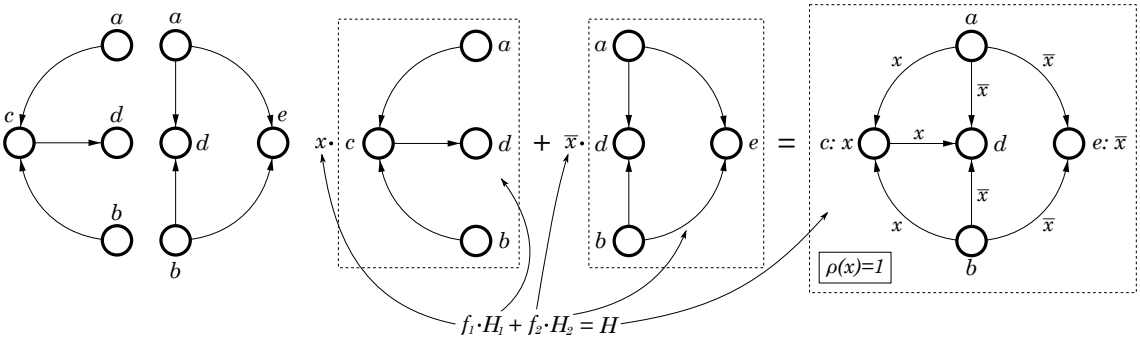
Since $H_k = \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k)$, it is known that H_k does not contain any conditional vertices or arcs (by Definition 5.4). Therefore projection $|_{\psi_k}$ does not affect it and $H_k|_{\psi_k} = H_k$. This leads us to

$$\mathbf{po}(\mathbf{dg}\ H|_{\psi_k}) = \mathbf{po}(\mathbf{dg}\ H_k) = \mathbf{po}(\mathbf{dg}\ (\mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k))) = \mathbf{po}(\mathbf{po}^{-1}\ P_k) = P_k$$

because operations **dg** and $\mathbf{dg}^{-1}$ (as well as **po** and $\mathbf{po}^{-1}$) cancel each other. This is true for every $1 \leqslant k \leqslant n$, and thus (7.2) holds. □

The following example demonstrates application of the proposed synthesis method to the simple processing unit from Example 5.2.

Example 7.1. In this example the system contains two scenarios specified with partial orders P_1 and P_2 shown in Figures 7.2(a, b). Opcodes $\psi_1 = (1)$ and $\psi_2 = (0)$ are also provided; they are defined on a single operational variable $X = \{x\}$. According to (7.3) the resultant CPOG H containing both P_1 and P_2 should be of the form $H = f_1 H_1 + f_2 H_2$, where f_1 and f_2 are orthogonal ($f_1 f_2 = 0$) encoding functions. They should also satisfy conditions $f_1|_{x=1} = 1$ and $f_2|_{x=0} = 1$ in order to meet the opcode constraints (7.2). Encoding functions $f_1 = x$ and $f_2 = \overline{x}$ trivially satisfy all the requirements: $x \cdot \overline{x} = 0$, $x|_{x=1} = 1$, and $\overline{x}|_{x=0} = 1$.



(a) P_1 (b) P_2 (c) Synthesis of graph containing both P_1 and P_2

Figure 7.2: CPOG synthesis example

Therefore, the synthesised graph H is equal to

$$H = f_1 H_1 + f_2 H_2 = x \cdot \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_1) + \overline{x} \cdot \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_2)$$

This synthesis process is shown in Figure 7.2(c) together with the resultant graph H. It can be easily checked that this graph contains both P_1 and P_2 as its projections under opcodes $\psi_1 = (1)$ and $\psi_2 = (0)$ (and it has already been demonstrated in Figure 5.3).

The correspondence between system scenarios and their opcodes is called an *encoding scheme*. Several encoding schemes are often encountered in practice, namely, *one hot*, *binary*, and *matrix* encodings. The following subsections describe these schemes and give characterisations of the synthesised CPOGs in terms of encoding functions f_k.

7.1.1 One hot encoding scheme

In this scheme n operational variables $X = \{x_1, x_2, \ldots, x_n\}$ are used and scenarios $\{P_1, P_2, \ldots, P_n\}$ are encoded with *one hot* opcodes [105], i.e. P_1 has opcode $(x_1, x_2, x_3, \ldots) = (1, 0, 0, \ldots)$, P_2 — $(x_1, x_2, x_3, \ldots) = (0, 1, 0, \ldots)$, P_3 — $(x_1, x_2, x_3, \ldots) = (0, 0, 1, \ldots)$, etc.

Functions f_k are set to

$$f_k = x_k \prod_{\substack{1 \leqslant j \leqslant n \\ j \neq k}} \overline{x_j}$$

which are orthogonal and satisfy the opcode constraints.

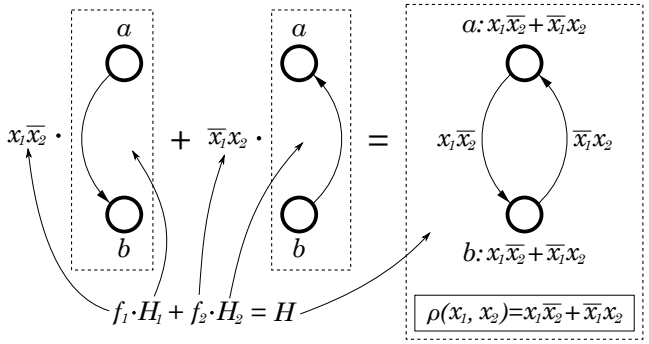
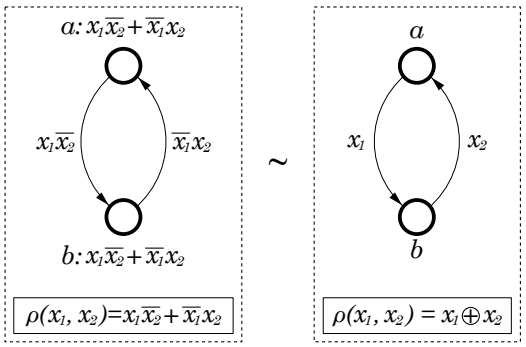


Figure 7.3: One hot CPOG synthesis

Example 7.2. Figure 7.3 shows an example of synthesis of a CPOG containing partial orders $P_1 = \{a \prec b\}$ and $P_2 = \{b \prec a\}$ with opcodes $\psi_1 = (1,\ 0)$ and $\psi_2 = (0,\ 1)$. The operational variables set is $\{x_1,\ x_2\}$ and the encoding functions are $f_1 = x_1\overline{x_2}$ and $f_2 = \overline{x_1}x_2$. The result $H = f_1H_1 + f_2H_2$ contains both partial orders as projections $H|_{x_1=1,\ x_2=0}$ and $H|_{x_1=0,\ x_2=1}$. It is possible to optimise it reducing the literal count from 16 to 4 (see Section 7.3 for details of the optimisation):



(7.4)

The one hot scheme provides a simple and intuitive way of scenario encoding but it is inefficient because of the large size of operational variables set: $|X| = n$. It is not practical for synthesis of CPOGs containing a large number of scenarios.

7.1.2 Binary encoding scheme

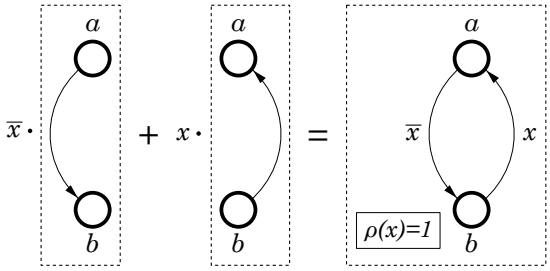
In the binary encoding scheme only $m = \lceil \log_2 n \rceil$ operational variables $X = \{x_1,\ x_2,\ \ldots,\ x_m\}$ are used to encode n given system scenarios, and this is the theoretical minimum. The opcodes of the scenarios are binary: $\psi_1 = (0,\ 0,\ 0,\ \ldots)$, $\psi_2 = (1,\ 0,\ 0,\ \ldots)$, $\psi_3 = (0,\ 1,\ 0,\ \ldots)$, $\psi_4 = (1,\ 1,\ 0,\ \ldots)$, etc.

Let b_{jk} denote j-th bit of integer number k. Then we define encoding functions f_k as:

$$f_k = \prod_{j=1}^{m} (x_j \Leftrightarrow b_{j(k-1)})$$

For example, if $n = 3$ we get $f_1 = (x_1 \Leftrightarrow 0)(x_2 \Leftrightarrow 0) = \overline{x_1}\,\overline{x_2}$, $f_2 = (x_1 \Leftrightarrow 1)(x_2 \Leftrightarrow 0) = x_1\overline{x_2}$, and $f_3 = (x_1 \Leftrightarrow 0)(x_2 \Leftrightarrow 1) = \overline{x_1}x_2$ resulting in natural binary encodings of the three partial orders: $\psi_1 = (0,\ 0)$, $\psi_2 = (1,\ 0)$ and $\psi_3 = (0,\ 1)$.

Application of the binary encoding scheme to the synthesis of a CPOG containing partial orders $P_1 = \{a \prec b\}$ and $P_2 = \{b \prec a\}$ leads to a very compact solution (only 2 literals):



Observe the difference between this result and the optimised version of the one hot solution (7.4). As one can see the selected encoding scheme does not affect the structure of the synthesised CPOG. However, it affects the complexity of its vertex/arc conditions and the size of the physical controller implementation.

7.1.3 Matrix encoding scheme

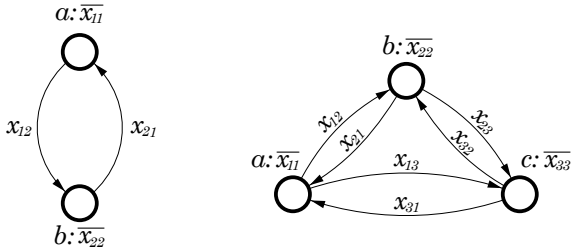
In this scheme the number of operational variables does not depend on the number of scenarios. It depends only on the number of different events in the system $|V|$. In particular, operational variables form *operational matrix* $X = \{x_{jk},\ j = 1\ldots|V|,\ k = 1\ldots|V|\}$. The matrix has enough information capacity to describe any partial order $P(V',\ \prec)$ on a subset $V' \subseteq \{e_1,\ e_2,\ \ldots,\ e_{|V|}\}$ of $|V|$ events. The opcode of the scenario specified with

partial order $P(V', \prec)$ is defined as

$$\psi(x_{jk})_{j\neq k} = \begin{cases} 1 & \text{if } (e_j \in V') \wedge (e_k \in V') \wedge (e_j \prec e_k) \\ 0 & \text{otherwise} \end{cases}$$

$$\psi(x_{kk}) = \begin{cases} 1 & \text{if } (e_k \notin V') \\ 0 & \text{otherwise} \end{cases} \tag{7.5}$$

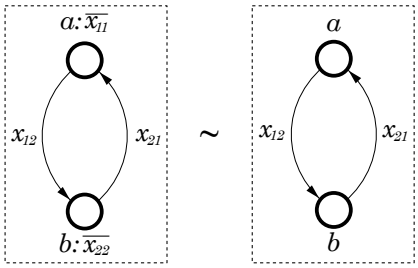
Instead of direct application of (7.3) to this encoding scheme we can use a generic solution with its subsequent optimisation taking into account the given scenarios. Generic solutions for systems with two and three events are given below[1]:



For example, application of (7.5) to the encoding of partial orders $P_1 = \{a \prec b\}$ and $P_2 = \{b \prec a\}$ gives us these opcodes:

$$\psi_{1,2} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}: \quad \psi_1 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \psi_2 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

Diagonal elements x_{kk} are constant zeros, and in this case the generic matrix graph can be reduced to the one hot solution (up to variable renaming – see (7.4)):



[1] A generic solution for an arbitrary number of events $V = \{e_1, \ldots, e_{|V|}\}$ is a fully connected graph $K_{|V|}$ with conditions $\phi(e_k) = \overline{x_{kk}}$ and $\phi((e_j, e_k)) = x_{jk}$.

The matrix encoding scheme is general in the sense that it can be used to encode any possible behavioural scenario of a system with n events in a reasonably compact and intuitive way. It is a trade-off between the one hot scheme which is straightforward but inefficient in terms of the number of operational variables and the binary scheme which has the least possible opcode length but more complicated encoding functions that are not affordable in some cases. The efficiency of the matrix encoding scheme allowed us to specify and synthesise *phase encoding repeaters* (see Subsection 8.2.2) for up to 10 wires having $10! = 3628800$ different behavioural scenarios.

7.1.4 Generalised synthesis problem

The presented synthesis problem (Definitions 7.1 and 7.2) can be generalised: it is possible to synthesise a CPOG not only from partial orders but from a mixture of partial orders and CPOGs. This allows the existing CPOG specifications to be reused without their explicit decomposition into separate partial orders. Thus, one can add new behavioural scenarios into the specification of a system and avoid its complete resynthesis.

Definition 7.3. (*Generalised CPOG synthesis problem*). Let a system be specified with a set of partial orders $\{P_1, P_2, \ldots, P_n\}$ and a set of CPOGs $\{H_1, H_2, \ldots, H_m\}$. The objective is to synthesise a CPOG $H(V, E, X, \rho, \phi)$ such that

$$\mathcal{P}(H) = \{P_1, P_2, \ldots, P_n\} \cup \mathcal{P}(H_1) \cup \mathcal{P}(H_2) \cup \cdots \cup \mathcal{P}(H_m) \tag{7.6}$$

i.e. graph H should contain partial orders $\{P_1, P_2, \ldots, P_n\}$ and also all the partial orders defined by graphs $\{H_1, H_2, \ldots, H_m\}$.

It is not difficult to modify solution (7.3) to handle this generalisation. The extended linear combination of the given partial orders and CPOGs is

$$H = \sum_{1 \leqslant k \leqslant n} f_k \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k) + \sum_{1 \leqslant k \leqslant m} H_k$$

where encoding functions f_k must be orthogonal and not contradictions (as before) and

should also satisfy the following constraint:

$$\forall 1 \leqslant j \leqslant n,\ 1 \leqslant k \leqslant m,\ f_j \rho_k = 0$$

i.e. no encoding conflicts are introduced during the synthesis.

If encoding conflicts are not a problem (for instance, if it is guaranteed that the conflicts are false), it is possible to use asymmetric addition for their resolution[2]:

$$H = \sum_{1 \leqslant k \leqslant n} f_k \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k) \vec{+} \overrightarrow{\sum_{1 \leqslant k \leqslant m}} H_k$$

In order to resolve true conflicts it is necessary to add the CPOGs and partial orders in appropriate order. For example, if a system is described with two CPOGs H_1 and H_2 plus a single partial order P_1 and we want to keep all the scenarios from H_2 but would like to substitute some scenarios from H_1 with the new ones (in case of a true conflict) then the proper order for asymmetric addition is:

$$H = H_2 \vec{+} f_1 \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_1) \vec{+} H_1$$

The correctness of the presented constructions follows from Proposition 7.1 and CPOG algebra theorems describing the properties of usual and asymmetric addition operations (see Subsections 5.1.4 and 5.1.6).

The opcode constraints can be added to the generalised problem in the same way as it was done for the basic synthesis case (see Proposition 7.2 for details): encoding functions f_k and opcodes ψ_k (if given) should satisfy condition $f_k|_{\psi_k} = 1$. This ensures that synthesised graph H specifies partial order P_k under opcode ψ_k (unless there is a true encoding conflict and opcode ψ_k has been redefined).

[2]Notation $\overrightarrow{\sum}_{1 \leqslant k \leqslant m} H_k$ is used to denote asymmetric sum $H_1 \vec{+} H_2 \vec{+} \ldots \vec{+} H_m$.

7.2 Mapping

As soon as the CPOG specification of a system is synthesised it can be mapped into Boolean equations in order to produce a physical implementation of the specified microcontroller. The mapping procedure presented below produces a set of equations whose overall size is very similar to the size of the given CPOG, hence the size and latency of the obtained microcontroller strongly correlates with the complexity of the original graph. It is therefore important to optimise the CPOG specification as far as possible by application of the CPOG optimisation techniques from Section 7.3, or by selecting the best opcode assignment as explained in Sections 7.1 and 7.4.

According to the behavioural semantics defined in Subsection 5.2.3, a vertex $v \in V$ is enabled to fire iff

1. It belongs to the current complete projection, i.e. its condition is satisfied: $\phi(v) = 1$;
2. All its preceding vertices have already fired, i.e. $\forall u \in V,\ (u \in \bullet v) \Rightarrow \mathit{fired}(u)$.

This can be captured in terms of Boolean equations as follows:

$$\mathit{enabled}(v) = \phi(v) \cdot \prod_{u \in V} \big(\phi(u) \cdot \phi((u,\ v)) \Rightarrow \mathit{fired}(u)\big)$$

Now predicates $\mathit{enabled}(v)$ and $\mathit{fired}(v)$ should be replaced with real signals. This can be done according to the microcontroller interface. For example, if the events are controlled via a request-acknowledgement handshake interface as shown in Figure 7.4, then we can substitute $\mathit{enabled}(v)$ with signal $\mathrm{req\_}v$ and $\mathit{fired}(v)$ with signal $\mathrm{ack\_}v$:

$$\mathrm{req\_}v = \phi(v) \cdot \prod_{u \in V} \big(\phi(u) \cdot \phi((u,\ v)) \Rightarrow \mathrm{ack\_}u\big)$$

In other words a request for event v execution is sent as soon as acknowledgements from the preceding events have been received. Note that the microcontroller in Figure 7.4 has also signals go and done which control its execution (the microcontroller starts generating event handshakes only upon the receipt of signal go, and as soon as all the

Figure 7.4: CPOG-based microcontroller with request-acknowledgement event interface

events have been executed it issues signal done). The following set of equations captures this fact:

$$\mathrm{req\_}v = \mathrm{go} \cdot \phi(v) \cdot \prod_{u \in V} \big(\phi(u) \cdot \phi((u,\ v)) \Rightarrow \mathrm{ack\_}u\big)$$

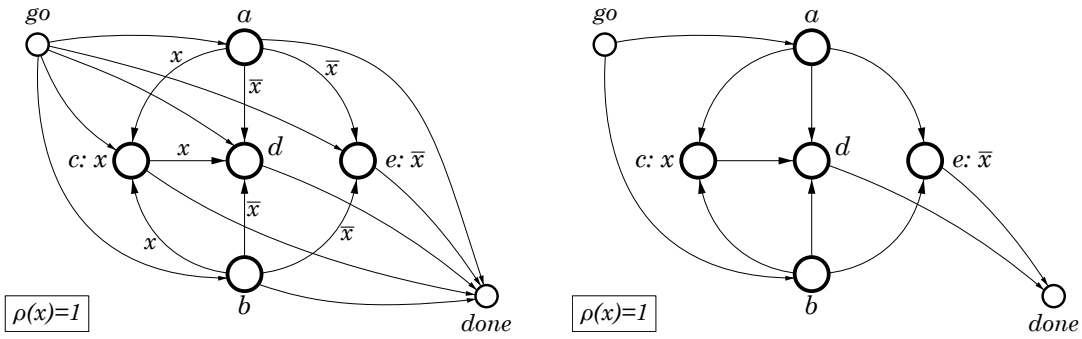
$$\mathrm{done} = \prod_{v \in V} \big(\phi(v) \Rightarrow \mathrm{ack\_}v\big)$$

Notice that it is possible to treat signals go and done as events of the modelled system for the sake of generality: event go should causally precede all the original system events, while event done should causally follow from them. This generality helps to optimise the final equations in a more consistent way. Formally, given a graph $H(V,\ E,\ X,\ \rho,\ \phi)$ we extend it to graph $H'(V',\ E',\ X,\ \rho,\ \phi')$ such that $V' = V \cup \{\mathrm{go},\ \mathrm{done}\}$, $E' = E \cup \bigcup_{v \in V}\{(\mathrm{go},\ v)\} \cup \bigcup_{v \in V}\{(v,\ \mathrm{done})\}$, and conditions on the introduced vertices and arcs are: $\phi'(\mathrm{go}) = \phi'(\mathrm{done}) = 1$ and $\phi'((\mathrm{go},\ v)) = \phi'((v,\ \mathrm{done})) = 1$ for all $v \in V$.

This is clarified with the following example.

Example 7.3. Let's map the graph obtained in Example 7.1 (Figure 7.2) into Boolean equations. Its extended version is shown in Figure 7.5(a): vertices {go, done} have been added and connected to all the original vertices of the graph. It is mapped into the following set of equations (the final equations after basic Boolean simplification are shown in boxes):

$$\begin{cases} \mathrm{req\_a} = \boxed{\mathrm{go}} \quad \mathrm{req\_b} = \boxed{\mathrm{go}} \\ \mathrm{req\_c} = \mathrm{go} \cdot x \cdot (x \Rightarrow \mathrm{ack\_a}) \cdot (x \Rightarrow \mathrm{ack\_b}) = \boxed{\mathrm{go} \cdot x \cdot \mathrm{ack\_a} \cdot \mathrm{ack\_b}} \\ \mathrm{req\_d} = \mathrm{go} \cdot (x \Rightarrow \mathrm{ack\_c}) \cdot (\overline{x} \Rightarrow \mathrm{ack\_a}) \cdot (\overline{x} \Rightarrow \mathrm{ack\_b}) = \boxed{\mathrm{go} \cdot (x \Rightarrow \mathrm{ack\_c}) \cdot (\overline{x} \Rightarrow \mathrm{ack\_a} \cdot \mathrm{ack\_b})} \\ \mathrm{req\_e} = \mathrm{go} \cdot \overline{x} \cdot (\overline{x} \Rightarrow \mathrm{ack\_a}) \cdot (\overline{x} \Rightarrow \mathrm{ack\_b}) = \boxed{\mathrm{go} \cdot \overline{x} \cdot \mathrm{ack\_a} \cdot \mathrm{ack\_b}} \\ \mathrm{done} = \boxed{\mathrm{ack\_a} \cdot \mathrm{ack\_b} \cdot (x \Rightarrow \mathrm{ack\_c}) \cdot \mathrm{ack\_d} \cdot (\overline{x} \Rightarrow \mathrm{ack\_e})} \end{cases}$$



(a) Extension of the synthesised graph (b) Optimised graph

Figure 7.5: Mapping of equivalent CPOGs into Boolean equations

It is possible to optimise the graph by removing redundant dependencies and conditions[3], which results in the significantly smaller specification shown in Figure 7.5(b). The mapping of the optimised graph into Boolean equations produces a better solution:

$$\begin{cases} \text{req\_a} = \boxed{\text{go}} \quad \text{req\_b} = \boxed{\text{go}} \\ \text{req\_c} = \boxed{x \cdot \text{ack\_a} \cdot \text{ack\_b}} \\ \text{req\_d} = \boxed{(x \Rightarrow \text{ack\_c}) \cdot \text{ack\_a} \cdot \text{ack\_b}} \\ \text{req\_e} = \boxed{\overline{x} \cdot \text{ack\_a} \cdot \text{ack\_b}} \\ \text{done} = \boxed{\text{ack\_d} \cdot (\overline{x} \Rightarrow \text{ack\_e})} \end{cases}$$

The total number of literals was reduced from 23 to 15, thus leading to the final gate-level implementation shown in Figure 7.6. This example demonstrates that the result of mapping significantly depends on the size of the given CPOG, thus it is necessary to apply optimisation techniques introduced in the next section before mapping.

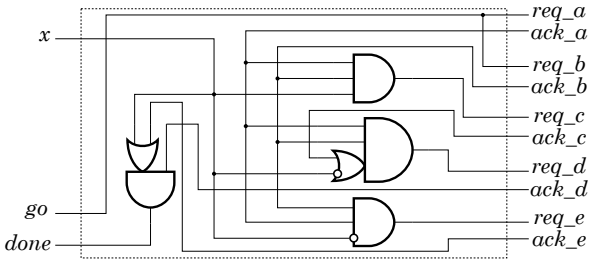
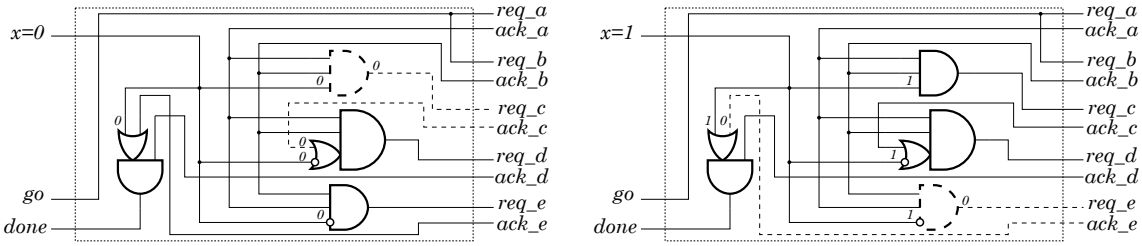


Figure 7.6: Gate-level implementation of the mapped microcontroller

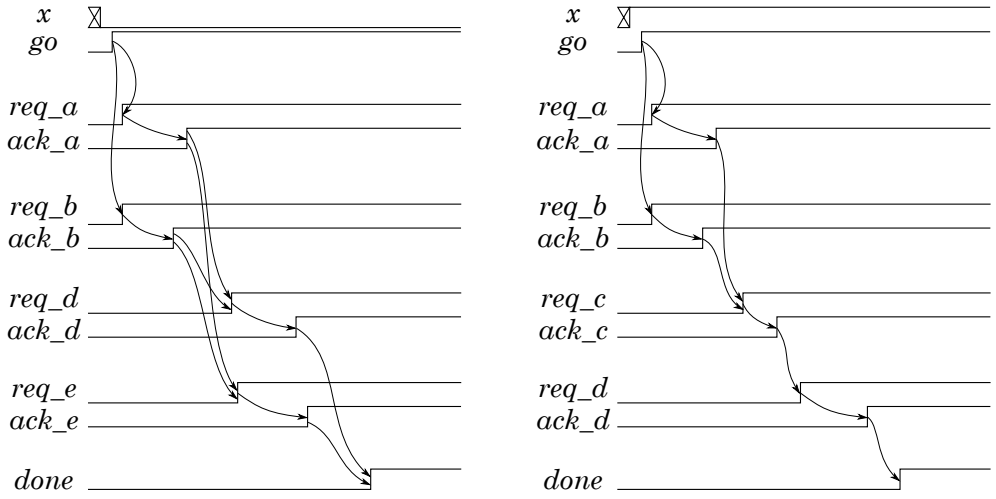
[3]A short clarification of the optimisation: dependency $a \prec d$ became unconditional because there is a transitive dependency $a \prec c \prec d$ if $x = 1$; dependency $go \prec c$ is removed because it is either transitive ($go \prec a \prec c$ if $x = 1$) or vertex c does not exist (if $x = 0$); etc. See Section 7.3 for more details.



(a) Circuit projection under x = 0 (b) Circuit projection under x = 1

Figure 7.7: 'Projections' of the synthesised controller under different opcodes

Figure 7.7 shows two 'projections' of the obtained controller under opcodes x = 0 and x = 1. Note that gates generating request signals for events c and e are logically 'switched off' according to the given opcode; the OR-AND complex gates retained only their AND functionality, e.g. the gate generating signal done is reduced to done=AND(req_d, req_e) in Figure 7.7(a). It is not difficult to see that the circuit executes two partial orders shown in Figure 7.2. See also two timing diagrams provided in Figure 7.8 that illustrate signal switching activity in the controller.



(a) Timing diagram under x = 0 (b) Timing diagram under x = 1

Figure 7.8: Timing diagrams showing behaviour of the synthesised controller

7.3 Optimisation

It has been demonstrated in the previous section that the size of the physical controller implementation is proportional to the size of its CPOG specification measured as the total number of literals in its conditions. There are different optimisation techniques [71] which reduce the size of a given CPOG by functional logic minimisation and/or by exploiting structural graph properties.

All the techniques presented here preserve the equivalence class of a given CPOG (i.e. the resultant optimised graph is equivalent to the given one) and the original encodings of partial orders (i.e. the opcodes of scenarios remain the same).

7.3.1 Logic minimisation

The most evident optimisation opportunity comes from the fact that all vertex/arc conditions ϕ in a CPOG $H(V, E, X, \rho, \phi)$ can be minimised by taking into account the *don't care set* [60][62][85] defined with restriction function ρ. In particular, a CPOG defines scenarios only for those opcodes ψ which are allowed by the restriction function, i.e. $\rho|_\psi = 1$. Otherwise system behaviour is undefined, therefore it is not important to what values conditions ϕ evaluate under disallowed opcodes ψ for which $\rho|_\psi = 0$.

Formally, let $z \in V \cup E$ be an arc or a vertex in a graph $H(V, E, X, \rho, \phi)$ having condition $f = \phi(z)$. Then it is possible to replace function f with another (possibly simpler) function g iff the following Boolean equation is a tautology[4]:

$$\rho \Rightarrow (f \Leftrightarrow g) \tag{7.7}$$

The intuition here is that function g must evaluate to the same value as f only for valid opcode variable assignments ψ (when $\rho|_\psi = 1$) and is unconstrained otherwise.

Consider one hot synthesis example shown in Figure 7.3. Restriction function $\rho = x_1\overline{x_2} + \overline{x_1}x_2$ defines two allowed opcodes (1, 0) and (0, 1). As shown in (7.4) it is possible to replace arc condition $\phi((a, b)) = x_1\overline{x_2}$ with a simpler one: $\phi_{opt}((a, b)) = x_1$. This

[4] A Boolean function is called tautology iff it is true under any possible assignment of its parameters [56].

substitution is validated by (7.7):

$$\rho \Rightarrow (f \Leftrightarrow g) \;=\; (x_1\overline{x_2} + \overline{x_1}x_2) \Rightarrow (x_1\overline{x_2} \Leftrightarrow x_1) \;=\; x_1x_2 + \overline{x_1}\,\overline{x_2} + \overline{x_1} + x_1\overline{x_2} = x_1 + \overline{x_1} = 1$$

Thus, $\phi_{opt}((a,\ b))$ is a correct optimisation of condition $\phi((a,\ b))$. See also Table 7.1 which clarifies why this works.

A general substitution $g = (\rho \Rightarrow f)$ appears to be quite good in practice (function g is forced to evaluate to 1 in all don't care entries of the truth table thus simplifying the min-terms). According to (7.7) it is a valid substitution, because

$$\rho \Rightarrow (f \Leftrightarrow g) \;=\; \rho \Rightarrow (f \Leftrightarrow (\rho \Rightarrow f)) \;=\; \rho \Rightarrow (\overline{f}\,\overline{(\overline{\rho} + f)} + f(\overline{\rho} + f)) \;=\; \rho \Rightarrow (\rho + f) \;=\; 1$$

This general substitution can be applied to the same example. Function $\phi(a) = x_1\overline{x_2} + \overline{x_1}x_2$ can be substituted with $\phi_{opt}(a) = (\rho \Rightarrow \phi(a)) = ((x_1\overline{x_2} + \overline{x_1}x_2) \Rightarrow (x_1\overline{x_2} + \overline{x_1}x_2)) = 1$ (this has been demonstrated in (7.4)).

Table 7.1 shows all four possible opcodes of two variables $(x_1,\ x_2)$ and compares the original conditions $\phi(a)$ and $\phi((a,\ b))$ with their optimised versions $\phi_{opt}(a)$ and $\phi_{opt}((a,\ b))$. Note that the conditions match under the allowed opcodes (matching values are highlighted with a bold font).

x_1	x_2	ρ	$\phi((a,\ b))$	$\phi_{opt}((a,\ b))$	$\phi(a)$	$\phi_{opt}(a)$	result
0	0	0	0	0	0	1	don't care
0	1	1	**0**	**0**	**1**	**1**	match
1	0	1	**1**	**1**	**1**	**1**	match
1	1	0	0	1	0	1	don't care

Table 7.1: CPOG logic minimisation using don't care set

7.3.2 Implicit arc exclusion

Whenever a vertex is excluded from a graph all its adjacent arcs are also excluded even if their conditions evaluate to Boolean 1. This can be exploited as follows.

Let arc $e = (a,\ b)$ have condition $f = \phi(e)$ and connect vertices with conditions $v_a = \phi(a)$ and $v_b = \phi(b)$. It is possible to substitute function f with another function g iff the

following relation is a tautology:

$$\nu_a \nu_b \rho \Rightarrow (f \Leftrightarrow g) \tag{7.8}$$

In other words, the don't care set is extended to include those opcodes in which arc e is implicitly excluded because of the exclusion of one of its vertices (cf. equation (7.7)).

For example, consider arc (a, c) in the CPOG from Figure 7.5(a). We would like to substitute condition $f = x$ with constant $g = 1$. However, this is not a valid substitution according to (7.7) because $f \neq g$ in the second scenario ($x = 0$). On the other hand vertex c is excluded from the graph in this scenario, therefore the actual value of the condition on arc (a, c) is not important. This is captured in (7.8) which considers g to be a valid substitution of f. Similar optimisation is applicable to arcs (a, e), (b, c), (b, e), and (c, d) of the graph.

7.3.3 Transitive arc reduction

Another optimisation opportunity is to reduce the transitive arc conditions. For instance, dependency (a, d) in Figure 7.5(a) is transitive with respect to path $a \rightarrow c \rightarrow d$ when $x = 1$. Clearly, the existence of an indirect dependency between events a and d is enough to establish the order relation between them, hence the condition $\phi((a, d)) = \bar{x}$ can be optimised into $\phi_{opt}((a, d)) = 1$. Arc (b, d) can be optimised in the same way leading to the graph shown to the right.

Formally, let arc $e = (a, b)$ have condition $f = \phi(e)$ and a transitive path $\langle a, b \rangle$ exist in graph $H \backslash \{e\}$ if condition t is true. Then it is possible to substitute function f with function g iff the following relation is a tautology:

$$\bar{t}\rho \Rightarrow (f \Leftrightarrow g)$$

In other words all the opcodes in which the transitive path $\langle a, b \rangle$ is activated ($t = 1$) are added to the don't care set of arc (a, b). It is possible to combine this technique with

the previous one (7.8) to obtain the general arc optimisation equation:

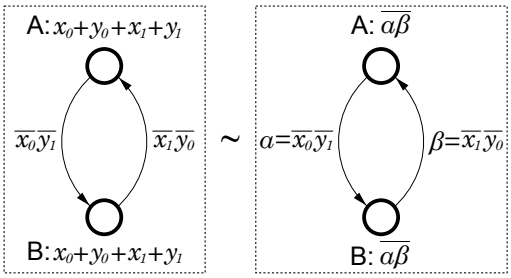
$$\nu_a \nu_b \bar{t} \rho \Rightarrow (f \Leftrightarrow g) \tag{7.9}$$

This general equation captures all the optimisation techniques presented above: logic minimisation (term ρ in the left part), implicit arc exclusion (term $\nu_a \nu_b$), and transitive arc reduction (term $\bar{t}$). It is important that the optimisation of a particular vertex or arc condition leaves the graph in the same equivalence class. This means that it is possible to optimise conditions in arbitrary order without affecting the final result what leads to efficient optimisation algorithms.

7.3.4 Common factors extraction

The three optimisation techniques presented above can be applied to every vertex or arc independently from the others. However, the actual circuit implementation of the controller may share the common factors (subexpressions) of conditions for the purpose of area minimisation. This requires the joint optimisation of graph conditions, which is a more time consuming procedure and may not be affordable for large designs.

CPOG specification[5] below can serve as a simple demonstration of common factors extraction:



Vertex conditions $\phi = x_0 + y_0 + x_1 + y_1$ can be expressed in terms of arc conditions $\alpha = \overline{x_0}\,\overline{y_1}$ and $\beta = \overline{x_1}\,\overline{y_0}$ as $\phi_{opt} = \overline{\alpha\beta}$ leading to the equivalent CPOG shown to the right. This optimisation technique can significantly reduce complexity of a large graph. However, the utilisation of common factors may slow down the resultant controller in

[5]This is a CPOG specification of dual rail ParSeq controller which is discussed in details in Chapter 8.

some cases, therefore, finding an appropriate trade-off between area and performance is necessary.

7.4 Optimal encoding of partial orders

The previous sections presented CPOG synthesis methods for systems with predefined opcodes of scenarios. There are cases, however, when there are no requirements on the opcodes and a designer is free to assign them arbitrarily. This section presents a method for optimal encoding of a given set of scenarios so that a CPOG containing all of them has the minimum complexity, thereby leading to the smallest and fastest controller [67].

The problem of optimal encoding of a set of partial orders is equivalent to that of the synthesis of optimal instruction codes in the context of micro-architecture design. Automated design of general purpose processing cores, application-specific instruction-set processors (ASIPs), and distributed Systems-on-Chip has recently gained a lot of attention from academia and industry [64]. New formalisms for data-path modelling are proposed [93], hardware/software co-design methodology [6] is actively developed and applied for ASIP performance improvement, more specific techniques (such as compiler-directed instruction set optimisation [111]) are constantly introduced into the instruction set architecture (ISA) design domain.

There are methods for automated ISA synthesis for a target platform (according to available system resources and data-path components) and for given software requirements (e.g. aimed to ease compilation or reduce program length). These methods eventually produce a structured set of instructions satisfying certain properties (orthogonality, completeness, regularity, etc.); instructions are grouped into classes and each class is allocated a certain opcode interval within the total code space [77]. At this point automation is typically stopped or becomes trivial: the instructions are given arbitrary codes within the allocated intervals. This limits performance due to instruction decoder circuitry overheads. The problem is usually approached by heuristics or application-specific optimisation techniques (see, for example, [55]). In this section we try to solve the general problem of optimal instruction encoding with the aid of the CPOG model.

Example 7.4. Consider a processing unit that has an accumulator register A and a general purpose register B, and computes four different arithmetic functions: $(-a)$, $(a+b)$, $(a-b)$, $(-a-b)$. The event domain consists of the following five events:

a) Load register A from memory;

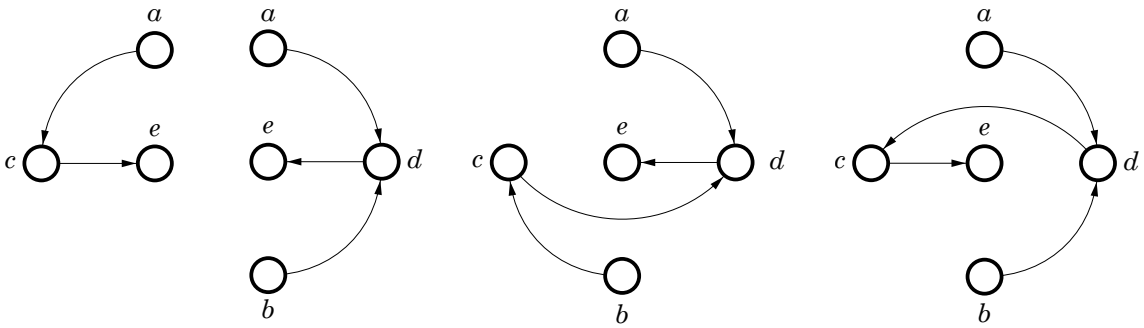
b) Load register B from memory;

c) Negate value stored in one of the registers;

d) Compute sum $A+B$ and store the result in A;

e) Save register A into memory.

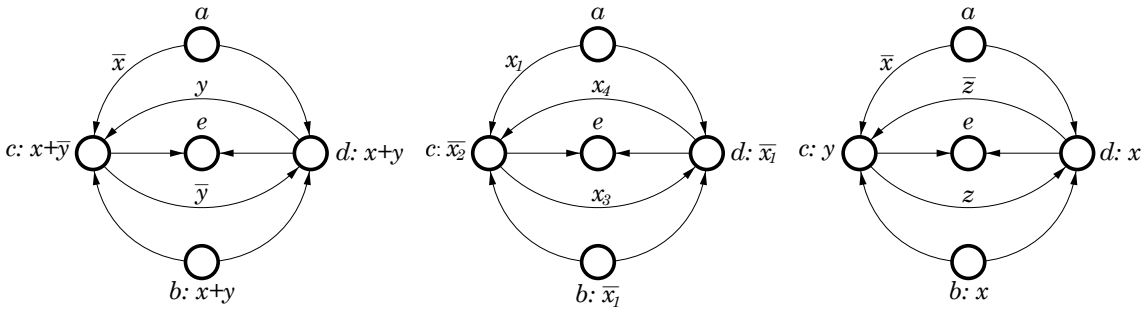
Partial orders of the four behavioural scenarios are shown in Figure 7.9. For instance, the second scenario (computation of $(a+b)$ shown in Figure 7.9(b)) consists of events a and b happening concurrently (loading of registers A and B), which are followed by event d (addition) and finally by event e (saving the result).



(a) $(-a)$ (b) $(a+b)$ (c) $(a-b)$ (d) $(-a-b)$

Figure 7.9: Four DAGs specifying the given scenarios

Before the synthesis of a CPOG $H(V, E, X, \rho, \phi)$ containing these partial orders it is necessary to encode them, i.e. to derive a set of control signals $X=\{x_1, x_2, \ldots, x_m\}$ and a set of Boolean vectors $\{\psi_1, \psi_2, \ldots, \psi_n\}$, $\psi_k \in \{1, 0\}^m$ (opcodes), each of them corresponding to a particular partial order. Note that in this section restriction function ρ is considered to be equal to the disjunction of allowed encodings for clarity. Optimisation of ρ is a separate problem because its size does not affect the size of the specified



(a) Binary encoding (b) One-hot encoding (c) Optimal encoding

Figure 7.10: CPOGs synthesised using different encoding schemes

controller, though it can be critical for verification algorithms where ρ appears as a term in SAT instances (see Chapter 6).

Let's examine several possible encoding schemes to see how a particular scheme affects the resultant CPOG.

Binary encoding scheme (Subsection 7.1.2)

This scheme uses the least possible number of operational variables $m = \lceil \log_2 n \rceil$ to encode n given partial orders. In this example two variables $X = \{x, y\}$ are used, and the opcodes are $\psi_1 = (00)$, $\psi_2 = (01)$, $\psi_3 = (10)$, and $\psi_4 = (11)$. Figure 7.10(a) shows the synthesised CPOG. It has been significantly optimised using the techniques presented in the previous section; the overall number of literals in vertex/arc conditions is 9.

One hot encoding scheme (Subsection 7.1.1)

This scheme uses four variables to encode the scenarios: $\psi_1 = (1000)$, $\psi_2 = (0100)$, $\psi_3 = (0010)$, and $\psi_4 = (0001)$. The synthesised CPOG is shown in Figure 7.10(b). Conditions on vertices $\{b, c, d\}$ became simpler, and the total literal count was reduced to 6. The price for that is the increase in the number of operational variables from 2 to 4.

Optimal encoding scheme

It turns out that there is a middle-ground solution which uses the same number of literals but only 3 variables $X = \{x, y, z\}$. The partial orders are encoded as $\psi_1 = (010)$, $\psi_2 = (100)$, $\psi_3 = (111)$ and $\psi_4 = (110)$. The synthesised CPOG is shown in Figure 7.10(c). We call this encoding scheme optimal, because it is better than the binary scheme (the resultant CPOG has fewer literals which leads to a simpler controller implementation),

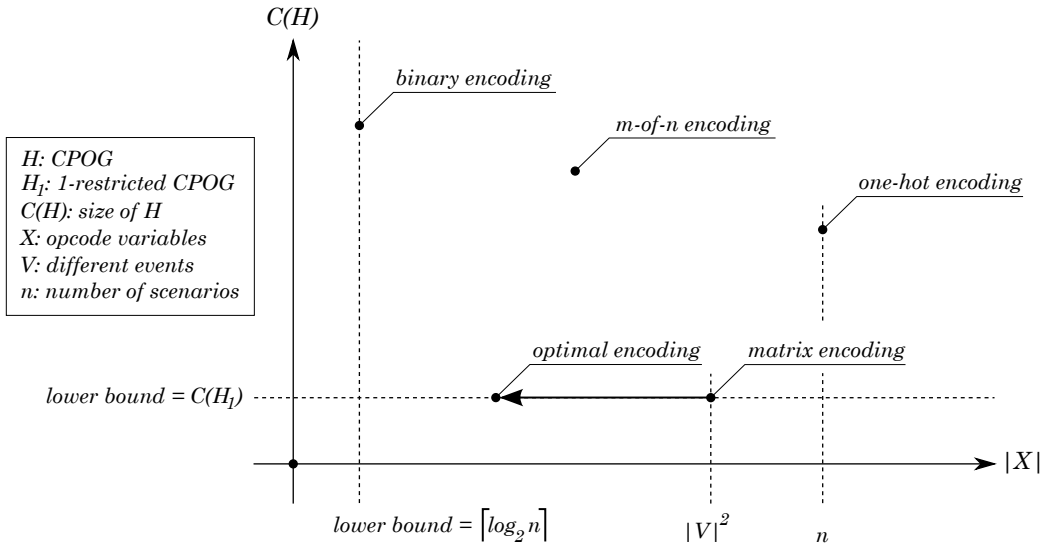


Figure 7.11: Size of specification vs number of control signals diagram

and it is better than the one-hot scheme (it uses fewer operational variables thus reducing the number of opcode wires coming to the controller from environment).

Figure 7.11 shows a Pareto diagram comparing different encoding schemes according to the number of opcode variables they use and the size of the resultant CPOG. One can see that the matrix encoding scheme (see Subsection 7.1.3) produces CPOGs with minimum possible size, but uses significantly more variables than the binary encoding scheme. This section aims to reduce the number of used variables but stay on the lower bound of CPOG complexity. In practice, the choice of a particular encoding scheme is crucial. The resultant CPOGs (and controllers) can differ in size by several orders of magnitude depending on the chosen opcodes of scenarios. The next subsection provides a formal definition of CPOG optimality criteria.

7.4.1 CPOG optimality criteria

A common measure of complexity of a Boolean function f is denoted as $C(f)$ and is defined to be the total count of literals in it [107], e.g. $C(x \cdot z + y \cdot \bar{z}) = 4$, $C(1) = 0$, etc.

Definition 7.4. A CPOG $H(V, E, X, \rho, \phi)$ is called k-*restricted* iff its vertex and arc conditions ϕ are functions having at most k literals, i.e. $\forall z \in V \cup E,\ C(\phi(z)) \leqslant k$.

Note, that this definition does not explicitly state whether H actually contains such z

that $C(\phi(z)) = k$ or not. If this fact needs to be emphasised we use a stronger definition.

Definition 7.5. A k-*restricted* CPOG is called *strongly* k-*restricted* iff there is a vertex or an arc with condition having exactly k literals, i.e. $\exists z \in V \cup E,\ C(\phi(z)) = k$.

A 0-restricted CPOG is therefore a DAG because it does not contain any conditional vertices or arcs. A 1-restricted CPOG contains only functions with single literals (possibly inverted) or constants as its vertex and arc conditions.

Section 7.2 showed that graph complexity $C(H)$ strongly correlates with the size and speed of the physical implementation of a controller specified with H. Hence, we use $C(H)$ as an adequate estimate of the efficiency of a CPOG H. The following proposition states that any optimal (with respect to measure C) CPOG is bound to be 1-restricted.

Proposition 7.3. *(Optimality). For any strongly* k*-restricted (*$k > 1$*) CPOG* H *there exists an equivalent* 1*-restricted CPOG* H_1 *such that* $C(H_1) < C(H)$.

Proof. (Constructive). Let n be the number of partial orders contained in $H(V, E, X, \rho, \phi)$, and $\{\psi_1, \psi_2, \ldots, \psi_n\}$ be their opcodes. It is possible to select a $z^+ \in V \cup E$ such that $C(\phi(z^+)) > 1$ (such z^+ must exist because H is strongly restricted).

Consider a CPOG $H'(V, E, X \cup \{x\}, \rho', \phi')$ with the extended opcode variables set (a new variable x is added), where

$$\forall z \in V \cup E,\ \phi'(z) = \begin{cases} \phi(z) & \text{if } z \neq z^+ \\ x & \text{if } z = z^+ \end{cases}$$

In other words, we replaced function $\phi(z^+)$ which had more than one literal with function $\phi'(z^+) = x$ which consists of a single literal, thereby reducing the overall CPOG size by $C(\phi(z^+)) - 1 > 0$ literals.

New encodings $\{\psi'_1, \psi'_2, \ldots, \psi'_n\}$ are obtained by adding an extra bit a_k (which is an assignment of x in k-th partial order) to the original encoding vectors: $\psi'_k = \psi_k \circ a_k$[6], where $a_k = \phi(z^+)|_{\psi_k}$.

[6]Operation '$\circ$' denotes concatenation of Boolean vectors, e.g. $1011 \circ 0 = 10110$.

This procedure simplifies the original graph by relaxing one of the conditions without affecting any contained partial orders. It can be repeated iteratively until the resultant graph becomes 1-restricted (let it be denoted as H_1). Note that every iteration reduces the size by at least 1 (because $C(\phi(z^+)) - 1 > 0$), which proves the proposition. □

Although the above proposition provides a polynomial algorithm for the reduction of any CPOG to a 1-restricted form, it is rather naive in terms of the resultant size of opcode variables set X. It adds as many new variables as there are 'heavy' ($C(\phi(z^+)) > 1$) conditions in the original CPOG. In the worst case it uses $|V|^2$ additional variables which can be impractical. The next subsection presents a method which uses the least possible number of variables.

7.4.2 Optimal encoding and synthesis

On the basis of Proposition 7.3 it is possible to formulate the problem of optimal encoding of partial orders.

Definition 7.6. (*Optimal CPOG encoding and synthesis problem*). Let $\{P_1, P_2, \ldots, P_n\}$ be a given set of n partial orders. The objective is to synthesise a 1-restricted CPOG $H(V, E, X, \rho, \phi)$ and to generate opcodes $\{\psi_1, \psi_2, \ldots, \psi_n\}$ such that synthesis requirements (7.1) and (7.2) are satisfied, i.e.

$$\Big(\mathcal{P}(H) = \{P_1, P_2, \ldots, P_n\}\Big) \wedge \Big(\forall 1 \leqslant k \leqslant n,\ \mathbf{po}(\mathbf{dg}\ H|_{\psi_k}) = P_k\Big)$$

and the size of operational variables set $|X|$ is minimised.

An *encoding constraint* $\mathbf{e}(z) \in \{0, 1, -\}^n$ for a vertex or arc $z \in (V \cup E)$ is a vector of n elements, each corresponding to one of n given partial orders. Element $\mathbf{e}(z)[k]$, $1 \leqslant k \leqslant n$ is equal to 1 iff condition $\phi(z)$ should evaluate to 1 in projection $H|_{\psi_k}$ in order to produce correct partial order P_k; $\mathbf{e}(z)[k] = 0$ iff the condition should evaluate to 0; and $\mathbf{e}(z)[k] = -$ iff $\phi(z)$ can evaluate either to 1 or to 0 (a don't care value).

Table 7.2 shows all the encoding constraints for the synthesis problem from Example 7.4. For instance, vertices a and e appear in all the four scenarios, so

Vertices/arcs	Encoding constraint $\mathbf{e}(z)$				Optimal encoding $\phi(z)$	
$z \in V \cup E$	$(-a)$	$(a+b)$	$(a-b)$	$(-a-b)$	w/o inversions	with inversions
$a,\ e$	1	1	1	1	1	1
$e \prec a$	0	0	0	0	0	0
$c \prec b$	–	–	0	0	0	0
$c \prec a,\ e \prec c$	0	–	0	0	0	0
$a \prec b,\ b \prec a,$ $d \prec a,\ d \prec b$ $e \prec b,\ e \prec d$	–	0	0	0	0	0
$a \prec d$	–	1	1	1	1	1
$a \prec e,\ b \prec e$	–	–	–	–	0	0
$b \prec c$	–	–	1	–	1	1
$b \prec d$	–	1	–	1	1	1
$c \prec e$	1	–	–	1	1	1
$d \prec e$	–	1	1	–	1	1
$b,\ d$	0	1	1	1	x	x
c	1	0	1	1	y	y
$a \prec c$	1	–	0	–	z	$\overline{x}$
$c \prec d$	–	–	1	0	w	z
$d \prec c$	–	–	0	1	z	$\overline{z}$

Table 7.2: Encoding constraints for optimal CPOG synthesis in Example 7.4

$\mathbf{e}(a) = \mathbf{e}(e) = 1111$ which means that $\phi(a) = \phi(e) = 1$. On the other hand, vertices b and d are not present in the first scenario, and therefore their encoding constraint is $\mathbf{e}(b) = \mathbf{e}(d) = 0111$. First 11 rows of the table contain *trivial encoding constraints*, i.e. constraints which do not contain $\mathbf{e}(z)[j] = 0$ and $\mathbf{e}(z)[k] = 1$ simultaneously ($j \neq k$). Vertices/arcs $z \in V \cup E$ with trivial encoding constraints can be encoded with a Boolean constant ($\phi(z) = 0$ or $\phi(z) = 1$, see column 'Optimal encoding' in the table). *Non-trivial encoding constraints* (the last 5 rows of the table) cannot be satisfied with a constant value, and therefore we need to introduce operational variables to encode them.

Don't care values appear in an encoding constraint in two cases:

- An arc $e = (v \prec u)$ is not present in a partial order together with one of its vertices. In this case function $\phi(e)$ is allowed to evaluate not only to 0 but also to 1, because if one of its vertices is excluded from the partial order ($\phi(v) = 0$ or $\phi(u) = 0$) then the arc is also excluded regardless of $\phi(e)$ value (cf. implicit arc exclusion optimisation technique from Subsection 7.3.2).

- An arc $e = (v \prec u)$ is transitive (i.e. $v \prec t \prec u$ for some $t \in V$). In this case function $\phi(e)$ is allowed to evaluate not only to 1 but also to 0, because the transitive dependency $v \prec t \prec u$ is enough to guarantee $v \prec u$ ordering (similar to transitive arc reduction used in Subsection 7.3.3).

Encoding constraint $\mathbf{e}(a \prec c) = 1{-}0{-}$ combines these two cases. The first don't care is due to exclusion of vertex c in the second scenario ($\mathbf{e}(c) = 1\underline{0}11$). And the second don't care is due to transitive dependency $a \prec d \prec c$ in the fourth scenario ($\mathbf{e}(a \prec d) = {-}11\underline{1}$ and $\mathbf{e}(d \prec c) = {-}{-}0\underline{1}$).

Two encoding constraints $\mathbf{e}_1$ and $\mathbf{e}_2$ are called *conflicting* iff

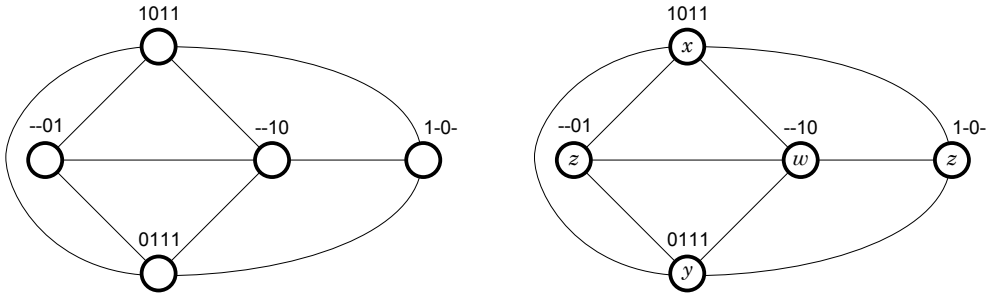
$$\exists 1 \leqslant k \leqslant n,\ (\mathbf{e}_1[k] = 1 \wedge \mathbf{e}_2[k] = 0) \vee (\mathbf{e}_1[k] = 0 \wedge \mathbf{e}_2[k] = 1)$$

In other words, it is impossible to find a Boolean vector satisfying both constraints (up to don't cares). For example, $\mathbf{e}(a \prec c) = 1{-}0{-}$ and $\mathbf{e}(c \prec d) = {-}{-}10$ are conflicting, because $\mathbf{e}(a \prec c)[3] = 0$ and $\mathbf{e}(c \prec d)[3] = 1$. On the other hand, $\mathbf{e}(a \prec c) = 1{-}0{-}$ and $\mathbf{e}(d \prec c) = {-}{-}01$ are not conflicting since vector 1001 satisfies both of them. It means that they both can be resolved with the same variable (variable z in Table 7.2, see column 'Optimal encoding w/o inversions').

The problem of optimal encoding can now be formulated in terms of conflicting encoding constraints: find an assignment of variables $\{x_1, x_2, \ldots, x_m\}$ to encoding constraints such that all pairs of conflicting constraints are assigned different variables and m is minimised. The following proposition states that this problem is within the NP-complete complexity class.

Proposition 7.4. *(NP-completeness). Optimal encoding problem is NP-complete.*

Proof. A *vertex colouring* of a DAG $G(V, E)$ is an assignment $clr : V \to \{1, 2, \ldots, m\}$ of colours to vertices such that any two adjacent vertices have different colours, i.e. $\forall (x, y) \in E,\ clr(x) \neq clr(y)$. The problem of finding a vertex colouring with the minimum number of colours m is known to be NP-complete [22]. Any instance of the optimal encoding problem is trivially an instance of the vertex colouring problem. To claim NP-



(a) Conflict graph (b) Optimal vertex coloring

Figure 7.12: Conflict graph and its optimal colouring

completeness it is also necessary to prove the reverse statement which is done by the following reduction.

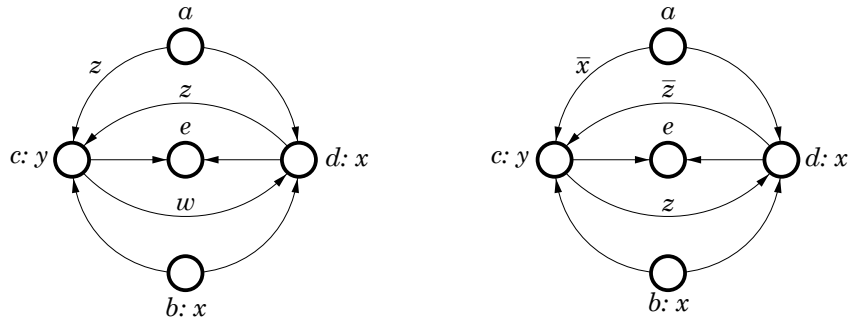
In order to obtain an instance of the optimal encoding problem, consider an incidence matrix M of a graph G(V, E). The size of matrix M is $|V| \times |E|$. Thus, it has a row for each vertex and a column for each edge. An element $M_{i,j}$ indicates how a vertex v_i is incident to an arc e_j (here we assume an arbitrary numbering of vertices and edges):

$$M_{i,j} = \begin{cases} 1 & \text{if } \exists x \in V,\ (v_i,\ x) = e_j \\ 0 & \text{if } \exists x \in V,\ (x,\ v_i) = e_j \\ - & \text{otherwise} \end{cases}$$

One can see that M represents an instance of the optimal encoding problem: any two rows i, j of M can be assigned the same variable if and only if vertices v_i and v_j are not adjacent in G. So any optimal variable assignment for M induces an optimal vertex colouring of G, and vice versa.

The described reduction uses polynomial number of steps and, therefore, proves the NP-completeness of the optimal encoding problem. □

Figure 7.12 shows the conflict graph for non-trivial encoding constraints from Table 7.2 and one of its optimal vertex colourings. It uses 4 'colours' $X = \{w, x, y, z\}$ and establishes the following encoding of partial orders: $\psi_1 = (0011)$, $\psi_2 = (0100)$, $\psi_3 = (1110)$, and $\psi_4 = (0111)$. This colouring is also given in column 'Optimal encoding w/o inversions' of Table 7.2. The synthesised CPOG is shown in Figure 7.13(a); it has the same size and



(a) Optimal encoding (no inversions) (b) Optimal encoding (with inversions)

Figure 7.13: Synthesised CPOGs

structure as the one hot solution in Figure 7.10(b) but does not contain any negative literals in its vertex/arc conditions.

Note that although encoding constraints $\mathbf{e}(c \prec d) = --01$ and $\mathbf{e}(d \prec c) = --10$ are conflicting they complement each other. This opens another optimisation opportunity: it is possible to resolve both constraints with a single variable x using conditions with complementary literals x and $\overline{x}$, thus potentially halving the number of used variables. In order to exploit this, we build an *extended conflict graph* which contains two vertices for every encoding constraint $\mathbf{e}(z)$: one for the original constraint and one for its inversion $\overline{\mathbf{e}}(z)$ which is defined as

$$\forall 1 \leqslant k \leqslant n,\ \overline{\mathbf{e}}(z)[k] = \begin{cases} 0 & \text{if } \mathbf{e}(z)[k] = 1 \\ 1 & \text{if } \mathbf{e}(z)[k] = 0 \\ - & \text{if } \mathbf{e}(z)[k] = - \end{cases}$$

Exactly one vertex of this pair $\{\mathbf{e}(z), \overline{\mathbf{e}}(z)\}$ has to be coloured. If the vertex corresponding to an inverted constraint $\overline{\mathbf{e}}(z)$ is chosen to be coloured with variable x it means that the constraint is resolved with negative condition $\phi(z) = \overline{x}$. See Figure 7.14 for the extended conflict graph of the example problem and its optimal colouring. This colouring can also be found in column 'Optimal encoding with inversions' of Table 7.2. The synthesised CPOG is shown in Figure 7.13(b) and the generated opcodes of partial orders are $\psi_1 = (010)$, $\psi_2 =$

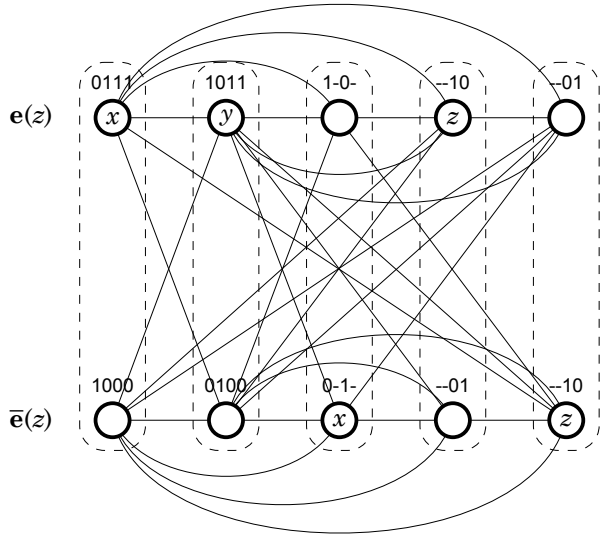


Figure 7.14: Extended conflict graph and its optimal colouring

(100), $\psi_3 = (111)$ and $\psi_4 = (110)$. This solution uses both positive and negative literals for resolution of conflicting encoding constraints, e.g. $\phi(c \prec d) = z$ and $\phi(d \prec c) = \bar{z}$, resulting in only three operational variables $X = \{x, y, z\}$. It is the optimal specification for the processing unit from Example 7.4: there is no solution which uses fewer literals.

The vertex colouring problem can be converted into an instance of the SAT problem for efficient solution (see Chapter 6 for SAT characterisations of other problems). An automated tool for optimal encoding of partial orders has been developed and is presented in Appendix.

7.5 Summary

This chapter described the stages of CPOG synthesis, mapping and optimisation that are closely related to each other: in order to produce the optimal microcontroller it is necessary to use the right combination of techniques available at every stage.

Several different CPOG synthesis problems have been formulated and solved in Sections 7.1 and 7.4:

- CPOG synthesis from partial orders (Definition 7.1);
- CPOG synthesis from partial orders with the opcode constraint (Definition 7.2);

- Generalised CPOG synthesis problem (Definition 7.3);
- Optimal encoding and synthesis problem (Definition 7.6).

The first three synthesis problems are efficiently solved using the structural techniques based on CPOG algebra, while the fourth one belongs to the NP-complete complexity class and can be characterised as a Boolean satisfiability problem.

As soon as a CPOG representation of a system is synthesised it can be passed on to the stage of mapping, to generate a physical implementation of the specified micro-controller. Section 7.2 described the process of mapping in detail and demonstrated that in order to obtain efficient microcontroller it is necessary to apply CPOG optimisation techniques which reduce complexity of a given graph by equivalence-preserving transformations. The optimisation techniques were presented in Section 7.3.

The next chapter gives practical examples of application of the presented synthesis, mapping and optimisation techniques while their tool support is discussed in Appendix.

Chapter 8

Application examples

This chapter presents three examples of application of the proposed specification and synthesis methodology. The first section returns to ParSeq controllers which were first discussed in Chapter 3 as a part of motivation behind the new model. It will be demonstrated that different levels of abstraction for data- and control-related events help to avoid combinatorial explosion in the size of system specification, something that plagues the STG and FSM models and limits their applicability to large designs.

Section 8.2 presents a detailed study of multiple rail phase encoding protocol and application of the CPOG model to synthesis of phase encoding controllers (they belong to the class of n-*permutator* circuits introduced in Section 3.2). It should be mentioned that these controllers gave the original inspiration for the new model because their specification using conventional approaches led to the explicit enumeration of all the behavioural scenarios and that was not affordable [65][71]. To deal with this problem a generic *Transition Sequence Encoder* (TSE) circuit was developed [26] and a study of its interesting properties revealed the opportunity for a novel approach to specification and synthesis of general microcontrollers using overlaid partial orders with conditional dependencies. This section presents an extensive set of design examples and benchmarks.

The last section of the chapter addresses the specification of a basic processor given a set of its instructions. The processor contains branching scenarios thereby requiring the use of the dynamic CPOG model [72] introduced in Section 5.2. Application of se-

veral synthesis methods from the previous chapter is demonstrated and the results are compared in terms of different criteria for the optimality of the obtained microcontroller. This design example highlights potential of the CPOG model in the context of synthesis of processor microarchitectures.

8.1 ParSeq controllers

In this section we finally come back to the specification and synthesis of ParSeq controllers introduced several chapters ago in Section 3.1, now with the decisive support of the CPOG model.

To refresh the memory, a ParSeq controller is a circuit managing two handshakes $A = (req\_a,\ ack\_a)$ and $B = (req\_b,\ ack\_b)$ that are executed either in parallel or in sequence according to the given opcode. The event domain in this case consists of two events $V = \{A,\ B\}$ corresponding to the handshakes. The behavioural scenarios can be represented with partial orders as shown in Table 8.1. Note that the spacer scenario (in which the controller must be idle) is explicitly defined.

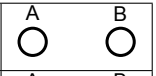
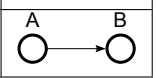
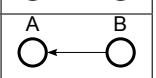
Scenario	Partial order $P(V, \prec)$			
	#	V	$\prec$	graph
Parallel A||B	P_1	$\{A,\ B\}$	$\emptyset$	
Sequential $A \rightarrow B$	P_2	$\{A,\ B\}$	$\{A \prec B\}$	
Sequential $B \rightarrow A$	P_3	$\{A,\ B\}$	$\{B \prec A\}$	
Spacer	P_4	$\emptyset$	$\emptyset$	*(empty)*

Table 8.1: Four scenarios of a ParSeq controller

8.1.1 One hot encoding

At first, consider synthesis of a CPOG specification for a one hot ParSeq controller. The set of operational signals is $X = \{x_1,\ x_2,\ x_3\}$; Table 8.2 shows the one hot opcodes ψ_k of partial orders P_k and the corresponding encoding functions f_k. The spacer scenario is assigned opcode (0, 0, 0), which is the spacer opcode in terms of one hot encoding [105].

Partial order P_k	P_1	P_2	P_3	P_4
Opcode $\psi_k = (x_1, x_2, x_3)$	(1, 0, 0)	(0, 1, 0)	(0, 0, 1)	(0, 0, 0)
Encoding function f_k	$x_1\overline{x_2}\,\overline{x_3}$	$\overline{x_1}x_2\overline{x_3}$	$\overline{x_1}\,\overline{x_2}x_3$	$\overline{x_1}\,\overline{x_2}\,\overline{x_3}$

Table 8.2: One hot encoding of ParSeq controller scenarios

Note that the four unused opcodes represent a *don't care* set [62] and can be used for logic optimisation of CPOGs and final signal equations (e.g. by using Espresso [88], a logic minimisation tool which supports don't cares). Moreover, a restriction function ρ of the synthesised graph describes this don't care set in a very compact form: all encodings ψ such that $\rho|_\psi = 0$ are don't cares. See Section 7.3 for details.

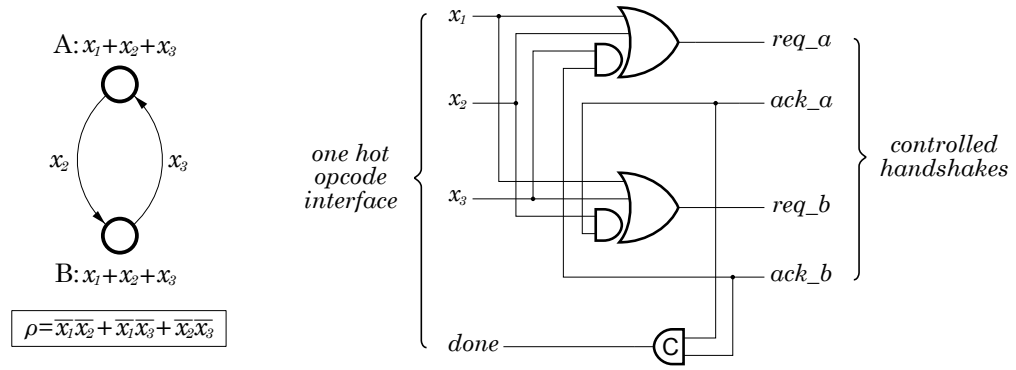
According to the CPOG synthesis method presented in Section 7.1, the resultant graph is equal to linear combination $H = f_1H_1 + f_2H_2 + f_3H_3 + f_4H_4$ (where $H_k = \mathbf{dg}^{-1}(\mathbf{po}^{-1}\ P_k)$). It is shown (after logic minimisation) in Figure 8.1(a). It can now be mapped into logic gates to produce a physical implementation of the controller as explained in Section 7.2:

$$\mathrm{req\_a} = (x_1 + x_2 + x_3)((x_1 + x_2 + x_3)x_3 \Rightarrow \mathrm{ack\_b})$$
$$\mathrm{req\_b} = (x_1 + x_2 + x_3)((x_1 + x_2 + x_3)x_2 \Rightarrow \mathrm{ack\_a})$$

which can be optimised taking into account don't cares defined by the restriction function $\rho = \overline{x_1}\,\overline{x_2} + \overline{x_1}\,\overline{x_3} + \overline{x_2}\,\overline{x_3}$:

$$\mathrm{req\_a} = x_1 + x_2 + x_3\mathrm{ack\_b}$$
$$\mathrm{req\_b} = x_1 + x_3 + x_2\mathrm{ack\_a}$$

This final equation is very easy to interpret: request to event A can be generated immediately in the two first scenarios (A||B and A → B) while in the third scenario (B → A) it should happen only upon the arrival of acknowledgement from event B; opcode (0, 0, 0) forces req_a to reset (the spacer scenario). Equation for req_b is similar, which leads to the controller shown in Figure 8.1(b). Signal done acknowledges the completion of both handshakes. It was verified that both Petrify [23] and 3D [110] synthesis tools generate the same controller given the STG and FSM specifications from Figure 3.2, so all the three specifications (in the STG, FSM and CPOG models) describe exactly the same controller.



(a) CPOG specification (b) Gate-level implementation

Figure 8.1: Specification and implementation of one hot ParSeq controller

8.1.2 Dual rail encoding

Dual rail ParSeq controller (Subsection 3.1.2) has four opcode signals: $X=\{a_0, a_1, b_0, b_1\}$. The encoding of the scenarios is shown in Table 8.3. Note that scenarios P_2 and P_3 have more than one encoding: this reflects the fact that the controller can start generating events after receiving only a partial opcode (OR-causality modelling has been shifted to the stage of scenarios encoding in CPOG based synthesis flow). The graph containing all these scenarios encoded with functions f_k is shown in Figure 8.2(a).

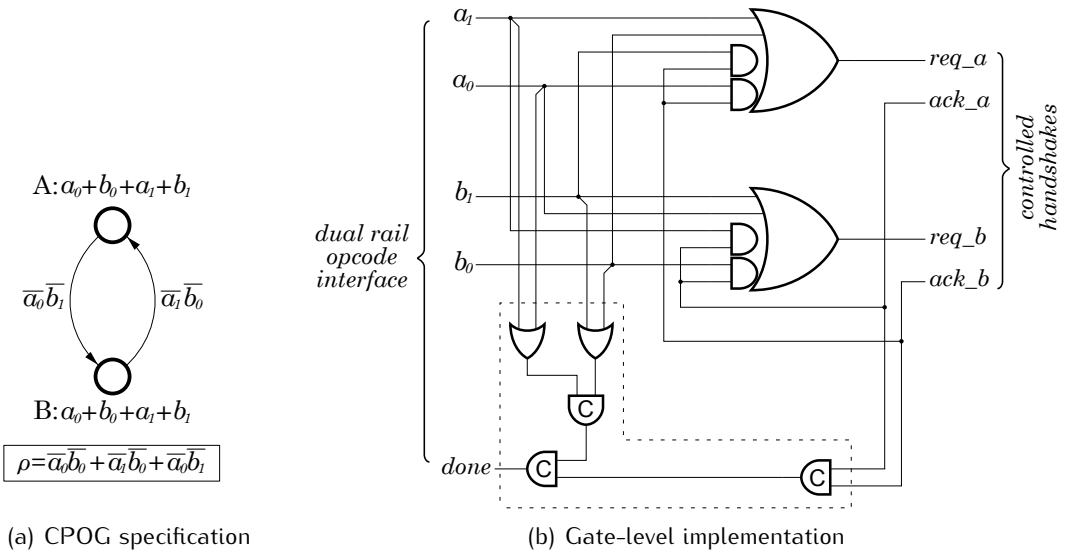
The final equations are

$$req\_a = a_1 + b_0 + (a_0 + b_1)ack\_b$$
$$req\_b = a_0 + b_1 + (a_1 + b_0)ack\_a$$

Signal *done* should acknowledge the completion of both handshakes and also the arrival of a complete opcode (this is needed because in some cases the controller can finish the handshakes having only partial opcode information). The gate-level imple-

P_k	P_1	P_2	P_3	P_4
Opcode $\psi_k=(a_0, a_1, b_0, b_1)$	(0, 1, 0, 1)	(0, 1, 1, 0) (0, 0, 1, 0) (0, 1, 0, 0)	(1, 0, 0, 1) (0, 0, 0, 1) (1, 0, 0, 0)	(0, 0, 0, 0)
Encoding function f_k	$\overline{a_0}a_1\overline{b_0}b_1$	$b_0+a_1\overline{b_1}$	$a_0+b_1\overline{a_1}$	$\overline{a_0}\,\overline{a_1}\,\overline{b_0}\,\overline{b_1}$

Table 8.3: Dual rail encoding of ParSeq controller scenarios



(a) CPOG specification

(b) Gate-level implementation

Figure 8.2: Specification and implementation of dual rail ParSeq controller

mentation of the controller is shown in Figure 8.2(b). Signal done is decomposed into several 2-input gates outlined with a dotted line. There can be several possible decompositions (note that complex gates generating signals req_a and req_b may have to be decomposed as well). Petrify and 3D produce the same controller (without signal done decomposition) given the corresponding specifications from Figure 3.3.

8.1.3 Observations

Let us draw some conclusions looking at the final ParSeq controller specification and implementation.

As in the numerous examples before, the CPOG specification stays structurally unchanged for different scenario encodings as can be seen in Figures 8.1(a) and 8.2(a). This is a very important and convenient feature: it gives the designer an opportunity to change encoding without graph resynthesis because it is possible to substitute operational variables with different ones and to rewrite the corresponding conditions. On the other hand, the STG and FSM specifications show a high degree of sensitivity to scenario encodings: a minor modification of encoding or protocol may lead to dramatic changes in the specification as have been demonstrated in Section 3.1 (in particular, see Figures 3.2, 3.3, and 3.4).

The ParSeq controller example also demonstrates that the CPOG model is beneficial for the specification and synthesis of controllers having both data and control path interfaces due to the different levels of abstraction used for data and control path modelling. Data path events and all the choices taking place in a system's behaviour are modelled with Boolean functions, while control path events and concurrency associated with them are modelled with partial orders. This combination of Boolean algebra and partial order theory allows most of the powerful optimisation techniques well-studied in these theories to be reused.

8.2 Phase encoding controllers

This section introduces a multiple rail phase encoding communication protocol and provides a scalable method for the generation of phase encoding controllers on the basis of the CPOG model.

D'Alessandro *et al.* in [27] introduced the concept of phase encoding for on-chip signalling, where the information is encoded into the sequence of events over a number of lines: this provides a way to concentrate information in symbols in a more compact form than by using binary encoding, with the added advantage of the reliability to single-event upsets [26][28]. However, in these works no satisfactory method to generate encoders and decoders for this communication scheme was provided: the presented structures were limited to a small number of wires (*rails*), and the scalability of these controllers (in terms of logic per number of wires in the channel) was not clearly described.

Phase encoding is a signalling technique that belongs to a class of *self-synchronous* (cf. *mesochronous* [29]) protocols, where the validity of data (i.e. clocking) is transmitted together with the data itself. The class of *delay insensitive* data transfer protocols [105] is a subclass of self-synchronous schemes. Subsection 8.2.1 presents asymptotic comparison between phase encoding and several well-known delay insensitive encodings in terms of information capacity, power consumption, etc.

While conventional control logic specification and synthesis methods based on STGs [25][95] or on FSMs [78] have certain advantages, they cannot be directly applied

to the problem of synthesis of phase encoders as has been shown in Section 3.2. In particular, the size of the specification of *matrix phase encoder* (see Subsection 8.2.3) is exponential with respect to the number of output rails in these models. To overcome this, the section defines and solves the problem of specification and synthesis of multiple rail phase encoding circuits using the CPOG model, providing efficient gate-level implementations for the circuits.

CPOG-based methods for synthesis of the phase encoding controllers are presented in Subsections 8.2.2 through 8.2.5. An example of the *speed-independent* [75] controller synthesis within the presented methodology is studied in Section 8.2.6. It is followed by the benchmark discussion in Section 8.2.7.

8.2.1 Phase encoding essentials

A *phase encoding* protocol was introduced by D'Alessandro *et al.* in [27]. The initial idea was to encode an information bit into a relative phase position between two switching signals. The idea was further extended into *multiple-rail phase encoding* [28], which uses several wires for communication and data is encoded in the order of occurrence of transitions on the communication lines. Figure 8.3 shows an example of a data symbol transmission over a 4-wire phase encoding communication channel. The order of rising signals on wires $\{a, b, c, d\}$ indicates that permutation $abdc$ is being sent. In total it is possible to send $n!$ different permutations over an n-wire channel. This makes the multiple rail phase encoding protocol very attractive for its information efficiency (provided that the system can maintain the minimum time interval between two transitions).

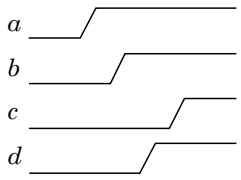


Figure 8.3: Data symbol in multiple-rail phase encoding channel

Table 8.4 contains several important characteristics of multiple-rail phase encoding protocol. The amount of information that can be sent in a symbol in n-wire channel

number of wires	number of permutations	bits per data symbol	bits per wire/time slot	transitions per bit
2	2	1	1/2	2
3	6	2	2/3	3/2
4	24	4	1	1
5	120	6	6/5	5/6
6	720	9	3/2	2/3
n (asymptotic)	$n!$	$\Theta(n \log_2 n)$	$\Theta(\log_2 n)$	$\Theta\left(\frac{1}{\log_2 n}\right)$

Table 8.4: Asymptotic characteristics of phase encoding protocol

grows faster than linearly. This is due to the fact that

$$\log_2(n!) = \sum_{k=1}^{n} \log_2 k \approx \int_1^n \log_2 x \, dx = x(\log_2 x - \ln 2) \Big|_1^n = \Theta(n \log_2 n) \tag{8.1}$$

We use the standard notation [22][51] for describing the asymptotic behaviour of functions:

- $f(n) \in O(g(n))$ means that f is bounded above by g (up to a constant factor) asymptotically, i.e.

$$\exists C > 0,\ n_0 :\ \forall n > n_0,\ |f(n)| \leqslant |Cg(n)|$$

- $f(n) \in \Theta(g(n))$ means that f is bounded both above and below by g asymptotically, i.e.

$$\exists C_1 > 0,\ C_2 > 0,\ n_0 :\ \forall n > n_0,\ |C_1 g(n)| \leqslant |f(n)| \leqslant |C_2 g(n)|$$

- $f(n) \in \Omega(g(n))$ means that f is bounded below by g asymptotically, i.e.

$$\exists C > 0,\ n_0 :\ \forall n > n_0,\ |Cg(n)| \leqslant |f(n)|$$

Asymptotic behaviour of other characteristics is based on (8.1), e.g. number of bits that can be sent in a time slot (the time separation between signals switching) is

$$\frac{\log_2(n!)}{n} = \frac{\Theta(n \log_2 n)}{n} = \Theta(\log_2 n)$$

Protocol	number of symbols	bits per symbol	time slots per symbol	bits per wire	bits per time slot	transitions per bit
phase encoding	$n!$	$\Theta(n \log_2 n)$	n	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$\Theta\left(\frac{1}{\log_2 n}\right)$
dual rail	$2^{\lfloor \frac{n}{2} \rfloor}$	$\Theta(n)$	1	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
1-of-n encoding	n	$\Theta(\log_2 n)$	1	$\Theta\left(\frac{\log_2 n}{n}\right)$	$\Theta(\log_2 n)$	$\Theta\left(\frac{1}{\log_2 n}\right)$
$\lfloor \frac{n}{2} \rfloor$-of-$n$ encoding	$\binom{n}{\lfloor \frac{n}{2} \rfloor}$	$\Theta(n)$	1	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

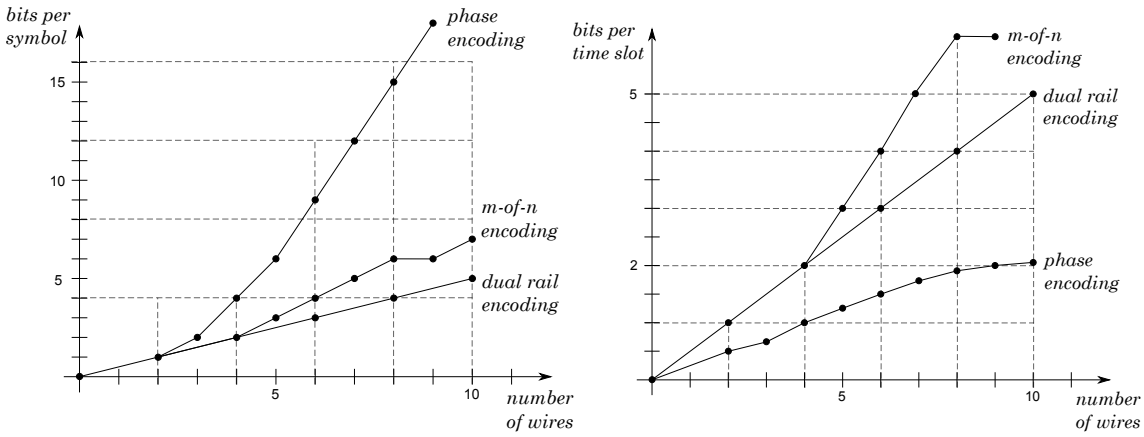
Table 8.5: Asymptotic comparison of DI communication protocols

Table 8.5 shows asymptotic comparison of the characteristics of phase encoding, *dual rail* and m-*of*-n *encoding* protocols [105] for n wires. You can see that phase encoding loses only in one parameter – bits per time slot: $\Theta(n)$ (dual-rail) and $O(n)$ (m-of-n encoding) vs $\Theta(\log_2 n)$ (phase encoding). This is because it needs n time slots to send one data symbol. Phase encoding is a clear winner in all the other parameters. The last one is particularly interesting: number of signal transitions per data bit. Phase encoding protocol needs only $\Theta\left(\frac{1}{\log_2 n}\right)$ transitions per data bit so the more wires the channel has the cheaper (in terms of power consumption) the bits are. Theoretically if we had infinite number of wires we could send a bit of information for free. A special case of m-of-n encoding with $m = 1$ (*one hot encoding* [105]) also needs only $\Theta\left(\frac{1}{\log_2 n}\right)$ transitions per data bit but its information efficiency is very low, hence it is not reasonable to use it for large values of n.

m-of-n encoding [33][105] cannot principally beat dual rail asymptotically in terms of number of bits per symbol: it reaches its maximum information efficiency when $m = \lfloor \frac{n}{2} \rfloor$. The number of bits in a symbol in this case is

$$\log_2 \binom{n}{\lfloor \frac{n}{2} \rfloor} = \log_2 \left(\frac{n!}{\lfloor \frac{n}{2} \rfloor ! \lceil \frac{n}{2} \rceil !} \right) \approx \log_2 \left(\frac{\sqrt{2\pi n} n^n e^{-n}}{\pi n n^n 2^{-n} e^{-n}} \right) = \log_2 \left(\frac{2^n}{\sqrt{\frac{\pi n}{2}}} \right) = \Theta(n) \quad (8.2)$$

Here we used *Stirling approximation* [38] of factorial $n! \approx \sqrt{2\pi n} n^n e^{-n}$. Equation (8.2) gives the upper bound of information efficiency for m-of-n encoding protocol. The lower bound is achieved for one hot encoding and is equal to $\log_2 \binom{n}{1} = \log_2 n$. For other values



(a) number of bits per symbol

(b) number of bits per time slot

Figure 8.4: Numeric comparison of DI communication protocols: information efficiency

of $0 < m < n$ information efficiency varies but remains within $\Omega(\log_2 n) \cap O(n)$ interval[1].

Therefore, asymptotically m-of-n encoding is equivalent to dual rail in terms of information efficiency, though it is better by constant factor approximately equal to 2:

$$\lim_{n\to\infty} \frac{\log_2\left(\frac{2^n}{\sqrt{\frac{\pi n}{2}}}\right)}{\log_2 2^{\lfloor \frac{n}{2} \rfloor}} = \lim_{n\to\infty} \frac{\log_2 2^n - \log_2 \sqrt{\frac{\pi n}{2}}}{\log_2 2^{\lfloor \frac{n}{2} \rfloor}} = \lim_{n\to\infty} \frac{n - \Theta(\log_2 n)}{\lfloor \frac{n}{2} \rfloor} = 2$$

In terms of power consumption m-of-n encoding can beat dual rail and approach phase encoding. The lower bound of power consumption is achieved for one hot encoding ($m = 1$) and is equal to

$$\frac{m}{\log_2 n} = \frac{1}{\log_2 n}$$

But the upper bound (when $m = \lfloor \frac{n}{2} \rfloor$) is the same as of dual rail:

$$\frac{m}{\Theta(n)} = \frac{\lfloor \frac{n}{2} \rfloor}{\Theta(n)} = \Theta(1)$$

Thus, power consumption of m-of-n encoding protocol is in interval $\Omega\left(\frac{1}{\log_2 n}\right) \cap O(1)$. Interestingly the interval spans exactly between phase-encoding and dual rail protocols.

The numeric comparison of the three protocols for up to 10-wire communication channels is shown in Figure 8.4. The results of m-of-n encoding are calculated for the most

[1] Intersection $\Omega(f(n)) \cap O(g(n))$ denotes a set of functions bounded by $f(n)$ below and by $g(n)$ above.

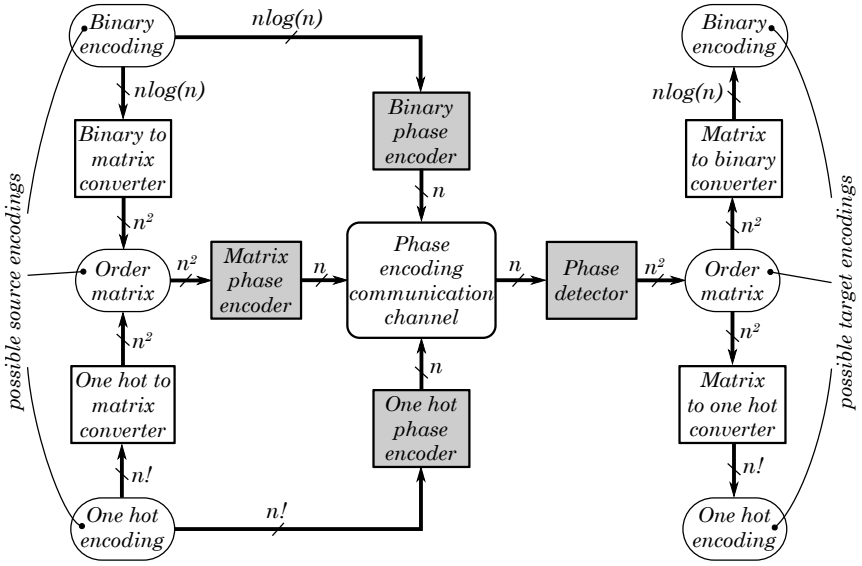


Figure 8.5: Phase encoding communication circuitry: n-wire channel

informative case when $m = \lfloor \frac{n}{2} \rfloor$. Subfigure (a) shows information efficiency with respect to a symbol size, while Subfigure (b) – with respect to a time slot. Notice that phase encoding is dominating on the first graph and shows rather bad results on the second. However this should not be misleading: although n-wire phase encoding protocol needs n time slots to send a data symbol, these time slots can be significantly shorter than that of dual rail or m-of-n protocols because each wire switches only once in these n time slots. Therefore the sending and receiving circuitry of a particular wire can work at a speed n times slower than the communication channel as a whole. It allows the time slots to be compressed much more than for dual rail and m-of-n encoding protocols and achieve higher information density over time. To summarise, phase encoding is potentially optimal in terms of area (number of bits per wire), speed (number of bits per time interval), and power (number of signal transitions per bit).

All these comparisons are theoretical and need experimental refinement. This work presents automated methods for generating multiple rail phase encoding receivers, senders, and repeaters. Figure 8.5 shows the overall phase encoding communication circuitry. Rectangular boxes represent functional units for conversion between different data encodings; the work covers the implementation of the units in highlighted boxes.

8.2.2 Phase encoding repeater

The first multiple rail phase encoding circuit that we are going to synthesise is a *phase encoding repeater* [26] – a circuit whose function is to regenerate the deteriorating phase difference between signals in the phase encoding communication channel.

Figure 8.6: Phase encoding repeater circuitry

A phase encoding repeater consists of two functional parts: a receiver (a *phase detector* which determines the order of the incoming transitions), and a sender (a *phase encoder* generating a series of transitions in the order they were received) as shown in Figure 8.6. It should be noted that we assume here that the phase encoded symbols arriving via the communication channel to the repeater are correct, i.e. all transitions are ordered with appropriate time slot condition. The issues of error behaviour and noise tolerance have been addressed in [26].

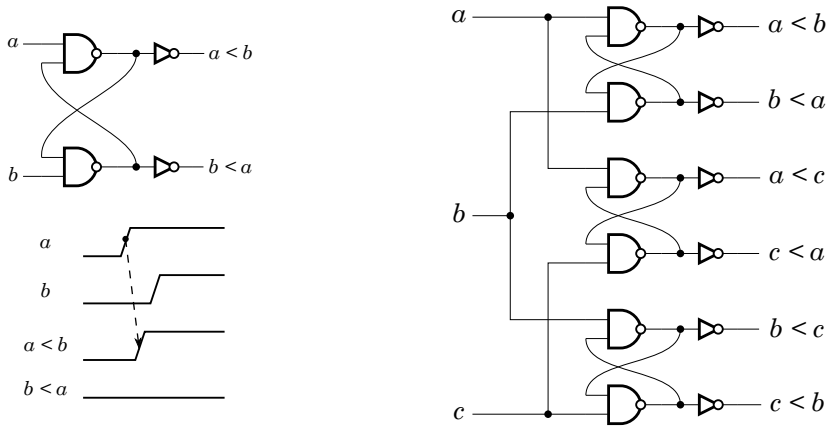
A phase detector for an n-wire communication channel consists of $\binom{n}{2}$ *mutual-exclusion* (*mutex*) *elements* [61, 73]: each for every pair of wires. A possible implementation of a mutex is shown in Figure 8.7(a): it consists of a pair of cross-coupled NAND gates (an SR-latch) and a simple metastability filter constructed from two inverters[2]. To determine the order of n transitions it is possible to compare their arrival times pairwise (see Figure 8.7(b) for an example of 3-wire phase detector).

The result of phase detection can be seen as an operational matrix with zero diagonal elements. Therefore, the subsequent phase encoder should be synthesised using the matrix encoding scheme (see Subsection 7.1.3) to avoid additional encoding conversion circuitry, as explained in the next subsection.

8.2.3 Matrix phase encoder

Given an operational matrix $X = \{x_{jk},\ j = 1\ldots n,\ k = 1\ldots n,\ j \neq k\}$, containing pairwise comparison of arrival times of n transitions, a *matrix phase encoder* should generate n

[2]This is the simplest way to construct a metastability filter, see [49] for more sophisticated solutions.



(a) Mutex element (b) 3-wire phase detector

Figure 8.7: Phase detection

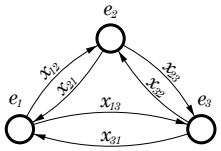
output transitions in the specified order.

Matrix X coming from the phase detector has $n!$ different possible value assignments $\psi_k : X \to \{0, 1\}$, $k = 1 \ldots n!$, each of them specifying a particular behavioural scenario of the controller. CPOG $H(V, E, X, \rho, \phi)$ containing all of them as its projections has the following generic description:

$$\begin{aligned} V &= \{e_j,\ j = 1 \ldots n\} \\ E &= \{(e_j,\ e_k),\ j = 1 \ldots n,\ k = 1 \ldots n,\ j \neq k\} \\ X &= \{x_{jk},\ j = 1 \ldots n,\ k = 1 \ldots n,\ j \neq k\} \\ \rho &= \prod_{1 \leqslant j < k \leqslant n} x_{jk} \oplus x_{kj} \prod_{\substack{1 \leqslant i,j,k \leqslant n \\ i \neq j,\ i \neq k,\ j \neq k}} x_{ij} x_{jk} \Rightarrow x_{ik} \\ \phi(e_j) &= 1,\ j = 1 \ldots n \\ \phi((e_j,\ e_k)) &= x_{jk},\ j = 1 \ldots n,\ k = 1 \ldots n,\ j \neq k \end{aligned} \tag{8.3}$$

Complexity $C(H)$ of this graph is dominated by the restriction function ρ which has a cubic size with respect to the number of wires n: $C(\rho) = \Theta(n^3)$. It restricts the operational domain of the graph to $n!$ opcodes corresponding to $n!$ different *total orders* of output transitions (see Chapter 4). However, the size of the final gate-level implementation of matrix phase encoder is quadratic because function ρ does not participate in the mapping (it is needed only for the purposes of synthesis and verification).

Example of a CPOG specification of 3-wire matrix phase encoder based on (8.3) is shown below:



Having synthesised the CPOG we can derive Boolean equations for the controller implementation. The controller should have n^2-n inputs $X=\{x_{jk},\ 1\leqslant j,\ k\leqslant n,\ j\neq k\}$ and n outputs $T=\{t_1,\ t_2,\ \ldots,\ t_n\}$. Output transition t_k is enabled to fire if all the preceding (with respect to the partial order specified by matrix X) transitions have already fired (cf. Section 7.2):

$$t_k=\phi(e_k)\cdot\prod_{\substack{1\leqslant j\leqslant n\\ j\neq k}}(\phi(e_j)\cdot\phi((e_j,e_k))\Rightarrow t_j) \tag{8.4}$$

This generic equation can be simplified taking into account the particular CPOG specification (8.3):

$$t_k=\prod_{\substack{1\leqslant j\leqslant n\\ j\neq k}}(x_{jk}\Rightarrow t_j)=\prod_{\substack{1\leqslant j\leqslant n\\ j\neq k}}(\overline{x_{jk}}+t_j)$$

Another optimisation opportunity is to exploit the fact that the operational matrix X specifies a total order. In our case it means that $\overline{x_{jk}}=x_{kj}$:

$$t_k=\prod_{\substack{1\leqslant j\leqslant n\\ j\neq k}}(x_{kj}+t_j)$$

As the phase encoder should maintain a certain time separation Δ between the generated transitions it is necessary to modify the above equation to take this fact into account:

$$t_k=\prod_{\substack{1\leqslant j\leqslant n\\ j\neq k}}(x_{kj}+t_j^{\Delta})$$

where t_j^{Δ} represents signal t_j delayed for Δ time units. For the purpose of resetting the controller into the initial state after generating the desired sequence of transitions we

should also add signal go that would serve as an initiating and resetting signal:

$$t_k = go \cdot \prod_{\substack{1 \leqslant j \leqslant n \\ j \neq k}} (x_{kj} + t_j^{\Delta})$$

The gate-level implementation of the controller specified with this final equation is shown in Figure 8.8.

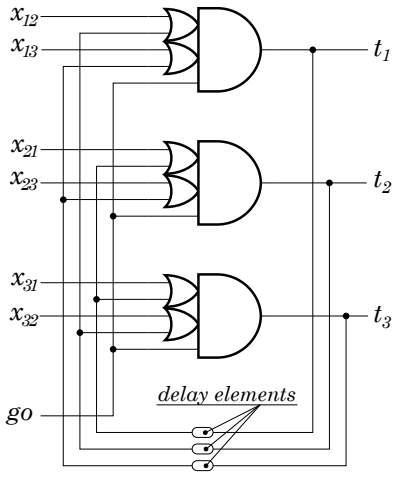


Figure 8.8: 3-wire matrix phase encoder

The implementation of the phase encoding repeater consisting of the phase detector and phase encoder is shown in Figure 8.9. Signal go can be generated in a number of ways depending on whether the repeater is *early-propagative* or not and on several other criteria which are out of the scope of this work and are discussed in detail in [26].

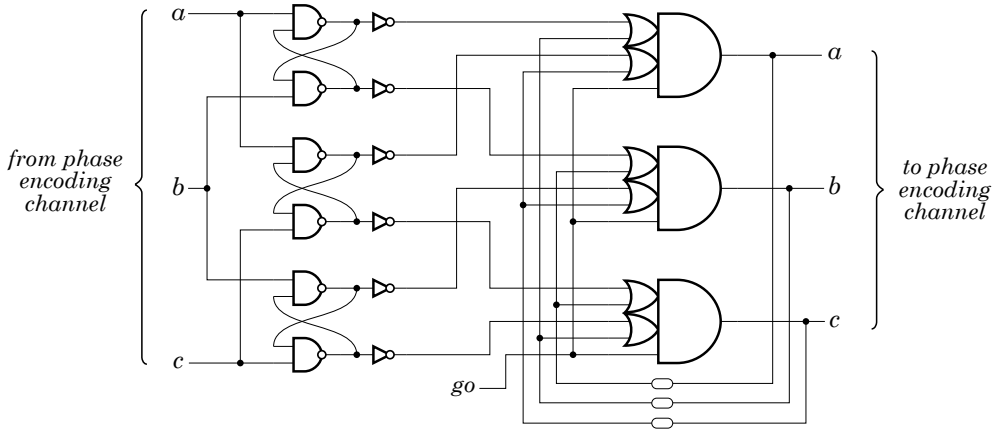


Figure 8.9: 3-wire phase encoding repeater

8.2.4 One hot phase encoder

One hot encoding can be used to specify the order of signal transitions for small values of n (for large values of n the method is inappropriate because it needs $n!$ wires). To send data presented in one hot encoding it is possible to convert it first into matrix form using a *one hot code to matrix converter* and then to send the result using a matrix phase encoder. Alternatively, to avoid unnecessary conversions it is possible to send one hot data directly using a *one hot phase encoder* as shown in Figure 8.10.

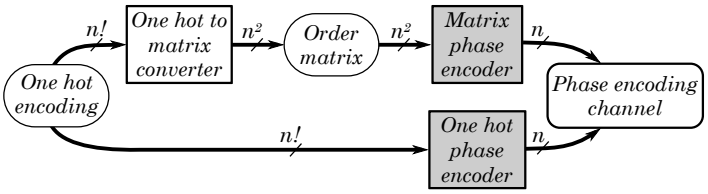


Figure 8.10: One hot phase encoder circuitry

All the $n!$ different scenarios of n output wire transitions can be specified with $n!$ partial orders $\mathcal{P} = \{P_1, P_2, \ldots, P_{n!}\}$. It is possible to synthesise a CPOG containing all of them using the one hot encoding scheme (Section 7.1.1). For example, there are 6 control signals $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and 6 partial orders corresponding to the possible permutations of output transitions $T = \{a, b, c\}$ for the case of $n = 3$ wires. See Table 8.6 which lists all the phase encoded data symbols, their one hot opcodes, and corresponding partial orders.

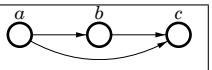
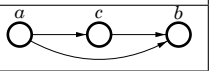
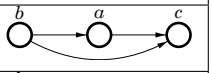
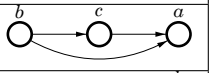
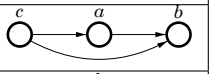
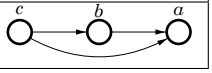
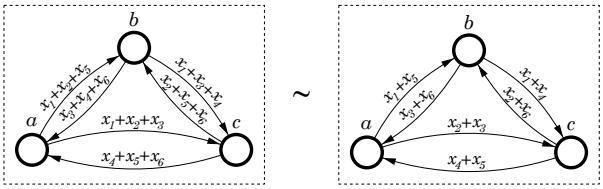
#	permutation (symbol)	one hot opcode	partial order
1	(a, b, c)	$\psi_1 = (1, 0, 0, 0, 0, 0)$	
2	(a, c, b)	$\psi_2 = (0, 1, 0, 0, 0, 0)$	
3	(b, a, c)	$\psi_3 = (0, 0, 1, 0, 0, 0)$	
4	(b, c, a)	$\psi_4 = (0, 0, 0, 1, 0, 0)$	
5	(c, a, b)	$\psi_5 = (0, 0, 0, 0, 1, 0)$	
6	(c, b, a)	$\psi_6 = (0, 0, 0, 0, 0, 1)$	

Table 8.6: Encoding of 6 scenarios of 3-wire one hot phase encoder

The synthesised CPOG is shown below (to the left); it is possible to simplify it into a slightly smaller CPOG using the transitive conditions reduction[3] (to the right):



The obtained optimal CPOG can be mapped into the gate-level implementation of a 3-wire one hot phase encoder shown in Figure 8.11.

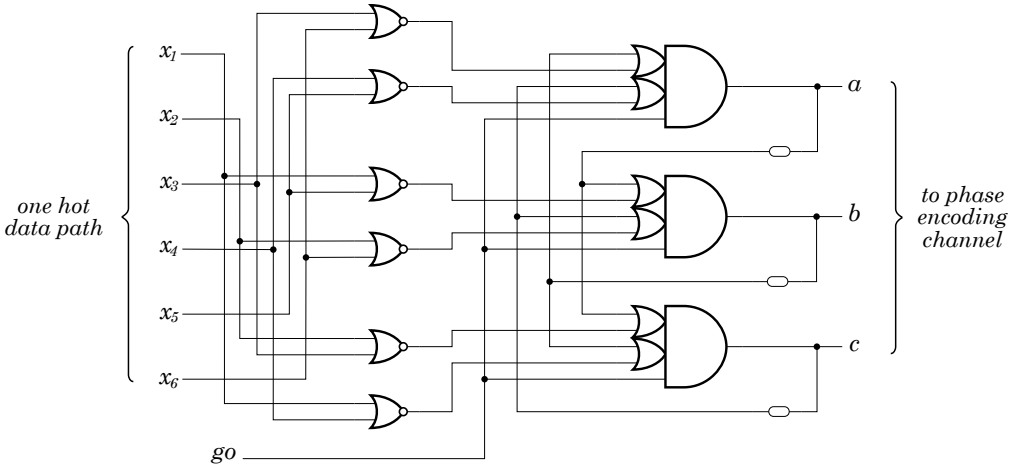


Figure 8.11: 3-wire one hot phase encoder

8.2.5 Binary phase encoder

Binary encoding is traditionally used for data transmission. To send a binary encoded symbol it is possible to convert it first into matrix form using a *binary code to matrix converter* and then to send the result using a matrix phase encoder. To avoid unnecessary conversion we can synthesise a customised *binary phase encoder* using the same principle as in the previous section for one hot encoding (cf. Figure 8.10).

The CPOG synthesis process is the same as for one hot phase encoding with the only exception that the binary encoding scheme is used (see Section 7.1.2). For the case of

[3]For example, condition $\phi((a,\ b)) = x_1 + x_2 + x_5$ can be simplified to $\phi_{opt}((a,\ b)) = x_1 + x_5$ because arc $(a,\ b)$ is transitive with respect to path $a \to c \to b$ if $x_2 = 1$. See Subsection 7.3.3 for details.

3-wire binary phase encoder, the following set of Boolean equations for output signals $T = \{a, b, c\}$ is eventually derived:

$$\begin{cases} a = ((\overline{x_1}x_2\overline{x_3} + x_1\overline{x_2}x_3) \Rightarrow b^{\Delta})((\overline{x_1}x_2x_3 + x_1\overline{x_2}\,\overline{x_3}) \Rightarrow c^{\Delta}) \\ b = ((\overline{x_1}\,\overline{x_2}\,\overline{x_3} + x_1\overline{x_2}\,\overline{x_3}) \Rightarrow a^{\Delta})((\overline{x_1}\,\overline{x_2}x_3 + x_1\overline{x_2}x_3) \Rightarrow c^{\Delta}) \\ c = ((\overline{x_1}\,\overline{x_2}x_3 + \overline{x_1}x_2\overline{x_3}) \Rightarrow a^{\Delta})((\overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_2x_3) \Rightarrow b^{\Delta}) \end{cases}$$

Taking into account the set of binary opcodes {000, 001, 010, 011, 100, 101} the above equations can be simplified (e.g. by using logic minimisation tool Espresso [88]) into

$$\begin{cases} a = \overline{x_1}\,\overline{x_2} + b^{\Delta}c^{\Delta} + \overline{x_3}(b^{\Delta} + c^{\Delta}) \\ b = x_2 + \overline{x_3}a^{\Delta} + x_3c^{\Delta} \\ c = x_1 + a^{\Delta}b^{\Delta} + x_3(a^{\Delta} + b^{\Delta}) \end{cases}$$

These resultant equations can now be mapped to gates to produce the physical implementation of the controller as shown in Figure 8.12 (go is added for start/reset purposes).

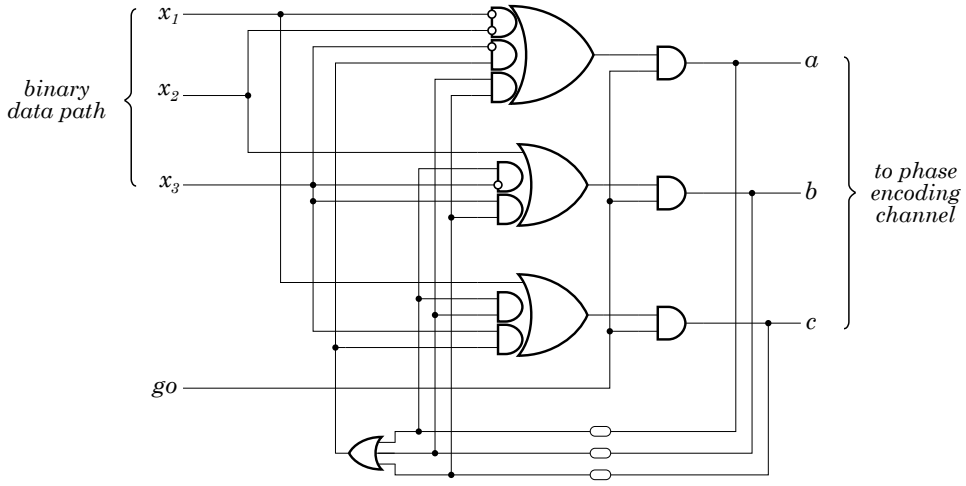


Figure 8.12: 3-wire binary phase encoder

Note that the complex gates in the obtained solution may require logic decomposition into smaller gates available in a particular technology library.

8.2.6 Speed-independent synthesis

The method of synthesis and mapping presented in Chapter 7 produces a set of Boolean equations as a result. In general it is not always possible to implement a large function using a single complex gate: libraries are often limited to 2- and 3-input elementary logic gates due to the technological constraints. This brings us to one of the key problems of circuit synthesis — the *logic decomposition* [25] of a given complex gate into an equivalent network of library gates satisfying certain correctness requirements. These requirements may vary depending on the target class of the controller being synthesised. For example, the controllers presented in Subsections 8.2.3 through 8.2.5 are not *speed-independent* [75] and operate correctly only under the timing assumptions imposed on opcode signals X and request signal go, allowing a simpler decomposition technique to be used. However, the presented CPOG based methodology can also be used for synthesis of speed-independent controllers, provided that special care is taken while mapping the resultant Boolean equations into the gate netlist (see [25] for a thorough analysis of this problem arising in a similar context of STG-driven logic synthesis).

The speed-independent synthesis can be demonstrated on a 3-wire one hot phase encoder from Subsection 8.2.4. The controller interface should be changed in order to establish a proper speed-independent communication protocol between the controller and the environment. Signal go is not needed anymore (the start and reset functions are delegated to one hot control signals $x_1 \ldots x_6$). Instead a new signal done should be introduced to prompt the environment that the controller has sent the phase encoded data and is ready for the next symbol. The delay elements should also be moved outside of the controller and become part of the environment. The implementation of the controller is shown in Figure 8.13 (the controller is separated from the environment with a dotted line). The complex gates generating the output signals are decomposed into 2- and 3-input logic gates with a subsequent negative logic optimisation[4]. The delayed output transitions are synchronised with a C-element to produce signal done. The circuit is formally verified for the compliance with the environment interface and the absence of

[4] A positive logic gate is constructed out of a negative logic gate plus an inverter in CMOS technology, therefore minimisation of the number of used positive gates is often performed.

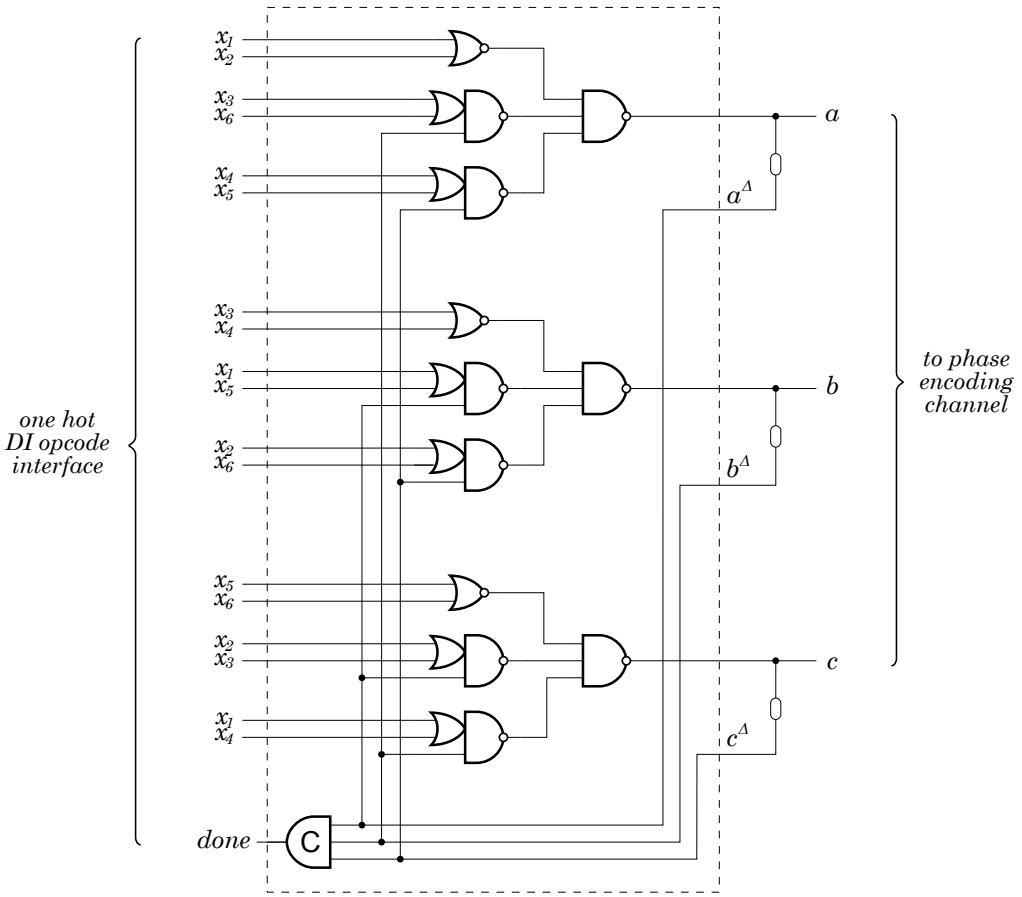


Figure 8.13: 3-wire one hot phase encoder (a speed-independent solution)

hazards using Workcraft framework [83].

In the general case a speed-independent controller has to issue signal done only after all the internal gates have switched and the communication with the environment has finished. Similarly, during the reset phase, as soon as all the gates and external handshakes have been reset to the initial states the falling transition of signal done enables the environment to initiate the next working cycle. So, the role of signal done is equivalent to that of a *completion detection* [52] signal in asynchronous data path, which informs the control path about the completion of computation within the combinational logic.

8.2.7 Benchmarks and summary

This section presented a CPOG-based approach for specification and synthesis of phase encoding controllers for different number of wires and encodings. Table 8.7 summarises the obtained results. Four leftmost columns specify a particular synthesis problem; the last two columns describe the solution size (in the number of literals, which closely correlates with the area of the decomposed gate-level implementation) and the total synthesis time (which mostly consists of logic minimisation performed by Espresso [88]).

Circuit family	# output wires	# data symbols	# opcode variables	# literals in solution	synthesis time
One hot phase encoders	3	6	6	21/27*	<10 ms
	4	24	24	160	220 ms
	5	120	120	1225	160 s
Binary phase encoders	3	6	3	21	<10 ms
	4	24	5	107	20 ms
	5	120	7	477	420 ms
Matrix phase encoders	3	6	3	15	<10 ms
	4	24	6	28	<10 ms
	5	120	10	45	<10 ms
	6	720	15	66	<10 ms
	7	5040	21	91	<10 ms

* refers to the speed-independent decomposition (Subsection 8.2.6)

Table 8.7: Synthesised phase encoding controllers

The largest synthesis instance among the presented is a 5-wire one hot phase encoder: it took almost 3 minutes to synthesise. This can be explained by the fact that the controller has $5! = 120$ one hot opcode variables blowing out the optimisation search space. It is also reflected in the huge physical implementation size which makes the controller hardly practical.

The most efficient controllers are synthesised using the matrix encoding scheme (Subsection 8.2.3): the size of matrix phase encoders grows quadratically with respect to the number of wires, even though the number of behavioural scenarios grows exponentially. Synthesis times are non-measurable because the obtained CPOGs do not require any logical optimisation. This presents a practical justification of our theoretical claims of the CPOG model efficiency. Note that the structural approach demonstrated here per-

# output wires	# data symbols	STG size			Petrify time	CPOG size (literals)	Synthesis time
		|P|	|T|	file size			
3	6	71	74	1181	$<$ 10 sec	15	$<$ 10 sec
4	24	302	322	6185	191 sec	28	$<$ 10 sec
5	120	1697	1812	36371	–	45	$<$ 10 sec

Table 8.8: Synthesis of matrix phase encoders: CPOGs vs STGs

forms only a number of operations on objects of polynomial size ($n \times n$ matrices), while application of a non-structural synthesis method would lead to exploration of the whole state space of the controller which is huge (its size is proportional to $n \cdot n!$ even if we assume all the input signals to arrive simultaneously!). See Table 8.8 for comparison of the CPOG-driven and STG-driven synthesis approaches on this class of circuits. It can be observed that the size of STG specification and the time of STG-driven synthesis (performed by Petrify synthesis tool [23]) grow exponentially. Moreover, the 5-wire instance of the problem is too large to be handled with STGs: the synthesis tool runs out of memory.

The family of binary phase encoders occupies the middle-ground: the size of controllers and synthesis times grow linearly with respect to the number of scenarios (the number of different phase encoded symbols). Existence of a generic binary solution whose size grows polynomially with respect to the number of wires is still an open question for future research.

The benchmarks demonstrate that the presented generic solutions are more scalable than those obtained without help of the CPOG model. Another advantage of the proposed methodology is an opportunity to improve the robustness of the solution by its speed-independent decomposition at the expense of the resultant controller area.

This section described synthesis of phase encoding controllers for only three source data encodings: one hot, binary, and matrix. However, it is not difficult to adapt the presented techniques to any data encoding, e.g. to other practically used m-of-n encoding communication protocols [9]: parallel 1-of-4 links [8], 2-of-7 codes [81][97], etc.

All the presented methods for synthesis of phase encoding controllers were automated in a software toolkit discussed in Appendix.

8.3 Specification and synthesis of processors

This section discusses the application of the CPOG-based methodology to specification and synthesis of processor microcontrollers. We take a simple four register processor as an example, go through all the stages of the CPOG-driven flow, and finally produce a gate-level implementation of the central microcontroller.

Specification of such a complex system as a processor usually starts at the architectural level [42][62] which helps to deal with the system complexity by structural abstraction: the system is divided into several communicating subsystems such that each of them can be designed individually, thus significantly reducing the solution search space. Design decisions which are made at this stage are crucial, and any potentially created performance bottlenecks cannot be corrected afterwards. Subsection 8.3.1 describes the architecture of the example processor.

At the next stage the processor specification is refined to the level of scenarios or instructions. Each instruction corresponds to a schedule of primitive actions such that data transfer, arithmetic operation, memory access, etc., which are performed by *operational units* – the subsystems from the previous stage. The design of instruction sets for a particular combination of operational units and software requirements is a difficult task [40]; among other optimisation objectives it contains the synthesis of instruction opcodes that can be automated within the proposed methodology as explained in Subsection 8.3.2.

The next step is to derive the behavioural description of the instruction set, which is conventionally done using Hardware Description Languages (HDLs), such as VHDL [57], Verilog [20] and others: every instruction is described as a separate functional block which are combined together using `if` or `case` branching statement. Our approach is different: the instructions are overlaid into a functional CPOG description, which is further mapped to Boolean equations. This final stage is addressed in Subsection 8.3.3. It is followed by the discussion of the handshake management issues which arise in the context of a multiresource system, e.g. arbitration of concurrent requests to the same resource, setting a limit on the number of concurrently working different resources (due to restrictions for overall system power consumption), etc.

8.3.1 Architecture

More than 60 years ago Burks, Goldstine, and von Neumann published a memo [16] which became the backbone of processor architectures for many decades. The so called *von Neumann architecture* provided a simple and effective structural abstraction for a computer design: the whole system was treated as a composition of several units: control unit, arithmetic logic unit (ALU), and memory to hold both instructions and data. The first working implementation of this architectural approach, the 'Baby' machine, was constructed in 1948 in Manchester.

At the same time an alternative approach was already used in Harvard Mark I, a relay-based computer engineered by Howard Aiken, which had separate storage for instructions and data [5]. This separation principle is now referred to as *Harvard architecture* and argued to be more efficient because it allows concurrent access to instructions and data, thus eliminating the *von Neumann bottleneck*.

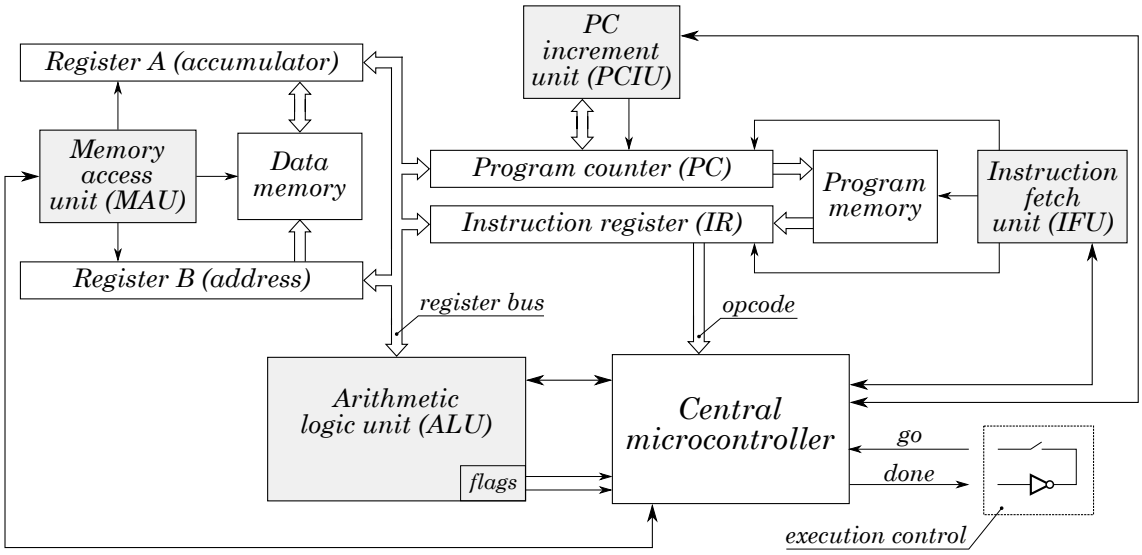


Figure 8.14: Architecture of example microprocessor

Our aim in this section is to demonstrate potential of the CPOG model for specification of highly concurrent systems, therefore the example processor is built on the basis of Harvard architecture. This can be seen in Figure 8.14, which shows separate *Program memory* and *Data memory* blocks that are accessed via *Instruction fetch* (IFU) and *Memory access* (MAU) operational units, respectively. The other two operational

units are: ALU and *Program counter increment unit* (PCIU). The units are controlled via request-acknowledgement interfaces (depicted as bidirectional arrows) by *Central microcontroller* (further called microcontroller for brevity) which is our primary specification and synthesis objective.

There are four registers: general purpose registers A and B, *Program counter* (PC) which stores the address of the current instruction in the program memory, and *Instruction register* (IR) which stores the opcode of the current instruction[5]. ALU has access to all the registers via the register bus; MAU accesses only general purpose registers; IFU reads opcode of the next instruction into IR given its address in PC; PCIU is responsible for incrementing PC (moving to the next instruction). The microcontroller has access to the opcode and ALU *flags* (information about the current state of ALU which is used in branching instructions as explained in the next subsection). Note that ALU and MAU units also need some (at least partial) information about the current opcode; this fact is not reflected in the diagram for clarity.

Execution control of the microcontroller is provided via go/done handshake: signal go prompts the microcontroller to execute one instruction, and as soon as the execution is complete it is confirmed with signal done. The simplest way of execution control would be an inverter and an ON/OFF switch as shown in the figure.

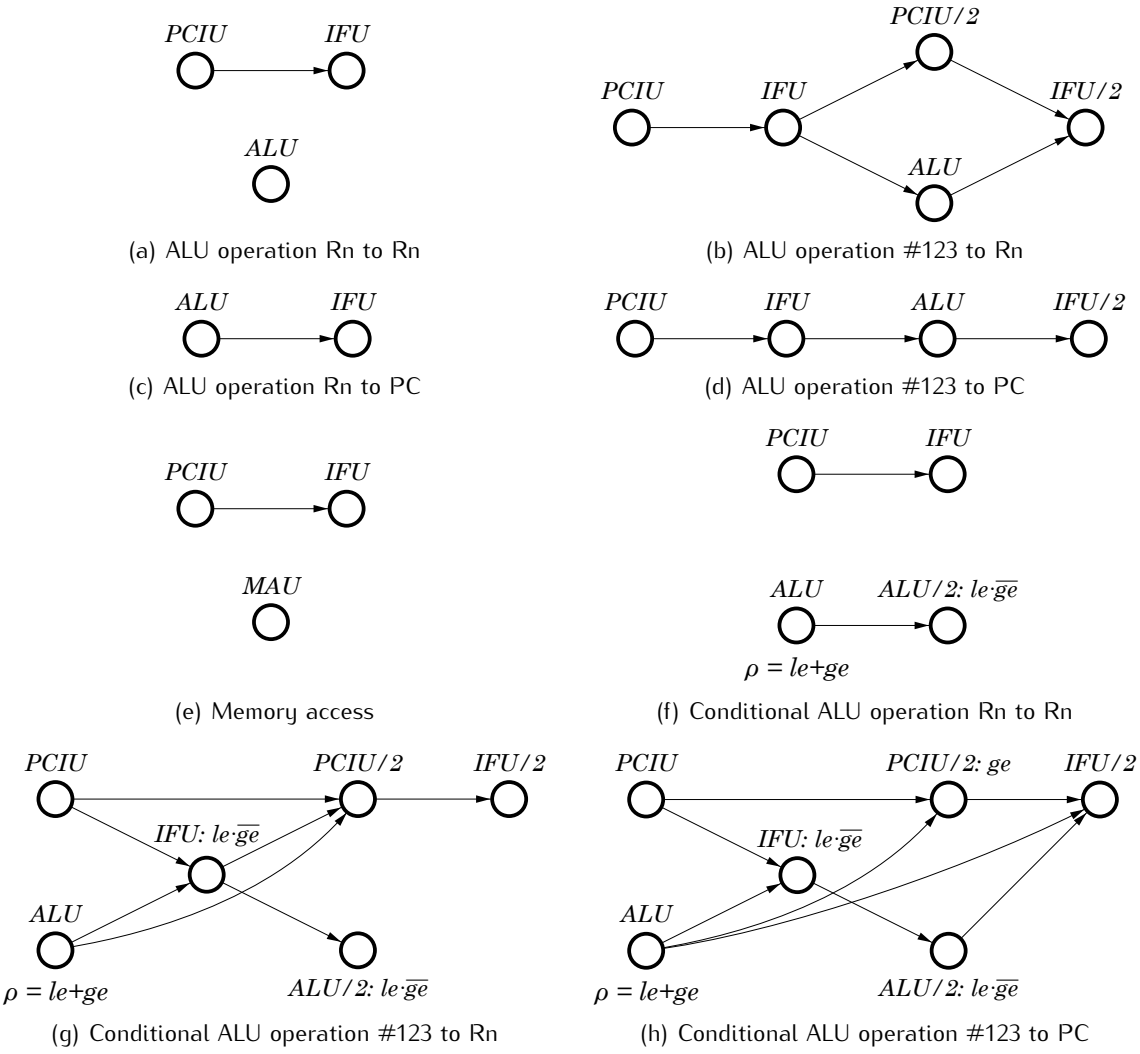
8.3.2 Design of instruction set

Now we have to define the set of instructions of the processor. Rather than to list every single instruction it is easier to describe classes of instructions with the same *addressing mode* [2] and partial order representation.

ALU operation Rn to Rn

An instruction from this class takes two operands stored in general purpose registers {A, B}, performs an operation on them, and writes the result back into one of them (so called *register direct addressing mode*). Examples: ADD A, B – addition $A = A + B$; XOR A, A – bitwise XOR of register A with itself (this effectively resets A to zero);

[5]For the purpose of this example the actual width of the registers (the number of bits they can store) is not important. Note also that different registers can have different widths.



(a) ALU operation Rn to Rn

(b) ALU operation #123 to Rn

(c) ALU operation Rn to PC

(d) ALU operation #123 to PC

(e) Memory access

(f) Conditional ALU operation Rn to Rn

(g) Conditional ALU operation #123 to Rn

(h) Conditional ALU operation #123 to PC

Figure 8.15: Graph specifications of 8 instruction classes

MOV B, A – assignment B = A. Figure 8.15(a) shows the corresponding partial order of actions that have to be performed: ALU works concurrently with PC increment (PCIU) and the next instruction fetch (IFU) actions. As soon as both concurrent branches are completed, the processor is ready to execute the next instruction.

Note that it is not important for the microcontroller which particular ALU operation is being executed (ADD, XOR, MOV, or any other) because the partial order of actions is not affected by this choice. It is the responsibility of ALU to detect which operation it has to perform according to the current opcode. Therefore, it is sufficient to specify only 8 behavioural scenarios of the microcontroller (as there are 8 classes of instructions).

ALU operation #123 to Rn

In this set of instructions one of the operands is register and another is a constant which is given immediately after the instruction opcode, hence the name: *immediate addressing mode*. Examples: SUB A, #1 – decrement A by one; MOV B, #0 – reset register B to zero.

Figure 8.15(b) shows the partial order of actions for this operation. At first, the constant has to be fetched into IR (actions PCIU and IFU). Then ALU operation is performed concurrently with another increment of PC. Finally, it is possible to fetch the next instruction opcode into IR. Note that PCIU and IFU actions have two occurrences in this partial order. This issue is addressed later in the design flow (see Subsection 8.3.4).

ALU operation Rn to PC

This class contains operations for unconditional branching, in which PC is modified thus altering the control flow of the program. Branching can be absolute or relative: MOV PC, A – absolute branch[6] to the address stored in register A; ADD PC, B – relative branch to the address B instructions ahead of the current address stored in PC.

The partial order representation for this operation is very simple: ALU and IFU actions are scheduled in sequence (see Figure 8.15(c)).

ALU operation #123 to PC

Instructions in this class are similar to those above with the exception that the branch address is specified explicitly as a constant, e.g. SUB PC, #8 – jump to the address 8 instructions before the current one.

The actions should be scheduled in the following sequence: PCIU→IFU (to fetch the constant) followed by ALU and finally another IFU, as shown in Figure 8.15(d).

Memory access

There are only two operations in this class: LOAD A and SAVE A. They load/save register A from/to memory location with address stored in register B.

As can be seen from Figure 8.15(e) access to memory can be performed concurrently with the next instruction fetch, exploiting the advantage of Harvard architecture.

[6]More common mnemonics for this instruction are JMP %A or JMP @A.

Conditional ALU operations Rn to Rn, #123 to Rn, #123 to PC

These three classes of instructions are similar to their unconditional versions above with the difference that they are performed only if the following condition is true: $A < B$, i.e. register A contains a value which is less than that in register B. In order to specify these conditional scenarios it is necessary to use the dynamic CPOG specification explained in Example 5.6. In particular, the first ALU action which is performed (comparison of registers A and B) changes the state of ALU flags $\{le, ge\}$ so as to reflect the result of the comparison. These flags are thereafter checked by the microcontroller in order to decide on the further scheduling of actions.

Let us discuss the most complicated scenario shown in Figure 8.15(h). The process starts with concurrent increment of PC and comparison of registers A and B. If condition $A < B$ holds (i.e. $le \cdot \overline{ge} = 1$) then the process is continued with the following sequence of actions: IFU→ALU/2→IFU/2 (read the constant, perform the branch, fetch the next instruction). Otherwise, the constant is skipped (PCIU/2) and the next instruction is fetched (IFU/2). Scenarios in Figures 8.15(f, g) are similar.

Now the instructions have to be encoded. The simplest way to do this is to use the binary encoding scheme, i.e. assign opcodes {000, 001, ..., 111} to the instructions in arbitrary order as shown in Table 8.9. This might not be optimal in terms of area and latency of the final microcontroller implementation. In order to obtain the smallest possible CPOG specification one has to apply the optimal encoding procedure from Section 7.4. Generated opcodes have 8 bits instead of 3 (shown in the same table). Whether 8 bit opcodes are affordable or not depends on the chosen width of instruction register IR.

#	Instructions class	Binary encoding	Optimal encoding
1	ALU operation Rn to Rn	000	00101001
2	ALU operation #123 to Rn	001	10101111
3	ALU operation Rn to PC	010	01101000
4	ALU operation #123 to PC	011	00101111
5	Memory access	100	00100001
6	Conditional ALU operation Rn to Rn	101	00111001
7	Conditional ALU operation #123 to Rn	110	11011011
8	Conditional ALU operation #123 to PC	111	01011011

Table 8.9: Possible encodings of the 8 classes of instructions

8.3.3 Microcontroller synthesis

Now that we have specified and encoded all the behavioural scenarios of the microcontroller it is possible to synthesise a CPOG containing all of them. Application of synthesis and optimisation techniques from Chapter 7 to binary encoded instructions produces graph shown in Figure 8.16; it uses three variables $X = \{x, y, z\}$ and contains 35 literals.

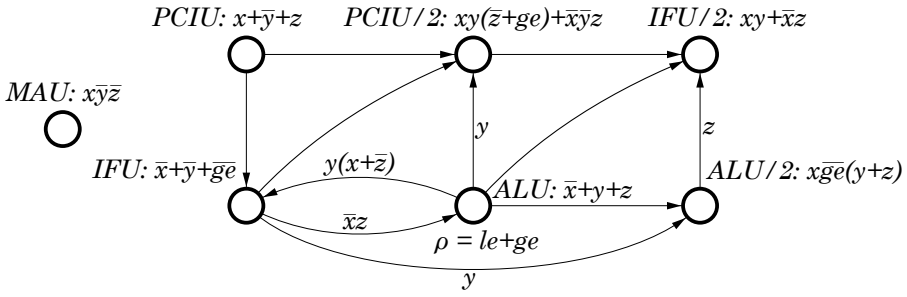


Figure 8.16: CPOG synthesised using the binary encoding scheme

The choice of a particular scenario within the graph is highly distributed: every condition is responsible for rendering only a little portion of the global picture and has a large don't care set which leads to efficient Boolean minimisation. Note that dynamic variable le turned out to be redundant and was removed from all the conditions. This is because original condition $le \cdot \overline{ge}$ (see Figures 8.15(f, g, h)) is equivalent to $\overline{ge}$ if the restriction function of ALU ($\rho_{ALU} = le + ge$) is satisfied:

$$\overline{ge} \cdot (le + ge) \Leftrightarrow le \cdot \overline{ge}$$

This means that it is enough to test only one ALU flag in order to correctly schedule all the scenarios, hence the number of wires is reduced to one, and the set of dynamic variables is $Y = \{ge\}$.

It is interesting to observe how OR-causality is modelled in the CPOG specification. For example, vertex PCIU has condition $\phi(\text{PCIU}) = x + \overline{y} + z$ and therefore the corresponding action may happen as soon as a part of an opcode is known ($x = 1$, or $y = 0$, or $z = 1$). Modelling such OR-causal behaviour using STGs or FSMs would lead to a

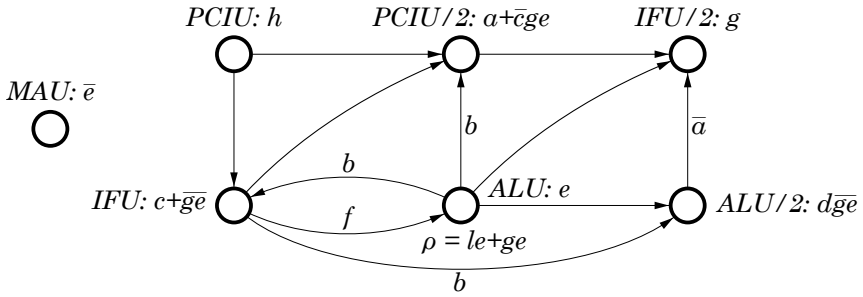


Figure 8.17: CPOG synthesised using the optimal encoding scheme

large and unobservable specifications similar to those shown in Figure 3.3 — this is a natural consequence of using event-based models for modelling the behaviour of data path logic. The CPOG model, on the other hand, employs Boolean functions for that and easily avoids such problems.

Figure 8.17 demonstrates that the optimal encoding procedure (see Section 7.4) yields a significantly smaller CPOG containing only 16 literals (which potentially leads to a microcontroller that is twice smaller and faster) but the price for this is 8 opcode variables $X = \{a, b, c, d, e, f, g, h\}$; see Table 8.9 for corresponding 8-bit opcodes. Note also that in this case it is not possible to reduce the final result to pure 1-restricted form because the graph contains dynamic variable ge which cannot be mixed with static variables for optimisation purposes as it is provided by ALU and can be changed during the execution of an ALU operation. Three non 1-restricted conditions are: $\phi(\mathrm{IFU}) = c + \overline{ge}$, $\phi(\mathrm{ALU}/2) = d \cdot \overline{ge}$, and $\phi(\mathrm{PCIU}/2) = a + \overline{c} \cdot ge$. It is impossible to use fewer literals for these conditions; this is clarified in Table 8.10: depending on the current scenario condition $\phi(\mathrm{IFU})$ has

Instructions class	$\phi(\mathrm{IFU})$	$\phi(\mathrm{ALU}/2)$	$\phi(\mathrm{PCIU}/2)$	a	c	d
ALU operation Rn to Rn	1	0	0	0	1	0
ALU operation #123 to Rn	1	0	1	1	1	0
ALU operation Rn to PC	1	0	0	0	1	0
ALU operation #123 to PC	1	0	0	0	1	0
Memory access	1	0	0	0	1	0
Conditional ALU operation Rn to Rn	1	$\overline{ge}$	0	0	1	1
Conditional ALU operation #123 to Rn	$\overline{ge}$	$\overline{ge}$	1	1	0	1
Conditional ALU operation #123 to PC	$\overline{ge}$	$\overline{ge}$	ge	0	0	1
Optimal condition	$c + \overline{ge}$	$d \cdot \overline{ge}$	$a + \overline{c} \cdot ge$			

Table 8.10: Encoding of conditions containing dynamic variable ge

to evaluate either to 1 or to $\overline{ge}$ and this choice is delegated to operational variable c such that $\phi(IFU) = c + \overline{ge}$. Condition $\phi(ALU/2)$ is similar: it must evaluate either to 0 or to $\overline{ge}$, hence $\phi(ALU/2) = d \cdot \overline{ge}$. The most complicated case is presented with condition $\phi(PCIU/2)$ which has three possible evaluations: 0, 1, and ge. Two variables are needed to handle this leading to $\phi(PCIU/2) = a + \overline{c} \cdot ge$ (note how variable c is reused here in its negated form). Optimal encoding of conditions containing dynamic variables is performed automatically together with all other conditions, thus the optimal result (in terms of the number of used literals) is guaranteed. See Appendix for description of the supporting tool.

The obtained CPOG can be mapped into logic equations using the method presented in Section 7.2:

$$\begin{cases} PCIU_{req} = h \\ ALU_{req} = e \cdot (\overline{f} + IFU_{ack}) \\ MAU_{req} = \overline{e} \\ IFU_{req} = (c + \overline{ge}) \cdot (\overline{h} + PCIU_{ack}) \cdot (\overline{b} + ALU_{ack}) \\ PCIU/2_{req} = PCIU_{ack} \cdot (a \cdot IFU_{ack} + \overline{c} \cdot ge \cdot ALU_{ack}) \\ IFU/2_{req} = g \cdot ALU_{ack} \cdot (PCIU/2_{ack} + \overline{a} \cdot (f + ALU/2_{ack})) \\ ALU/2_{req} = d \cdot \overline{ge} \cdot ALU_{ack} \cdot (\overline{b} + IFU_{ack}) \end{cases}$$

The total literal count is 28; the mapping of the binary CPOG from Figure 8.16 results in the solution with 46 literals (note how the size of the final solution correlates with the size of the corresponding CPOG specification). Figure 8.18 shows the gate-level implementation of the microcontroller. Signal go was added for start/reset purposes; signal $done$ can be implemented in different ways depending on whether it should be early-propagative or not during the reset phase (recall that behaviour of signal $done$ can be easily modelled using the extended graph specification explained in Section 7.2).

The presented microcontroller is not speed-independent but can be modified accordingly if needed (see Subsection 8.2.6). From the practical point of view speed-independence

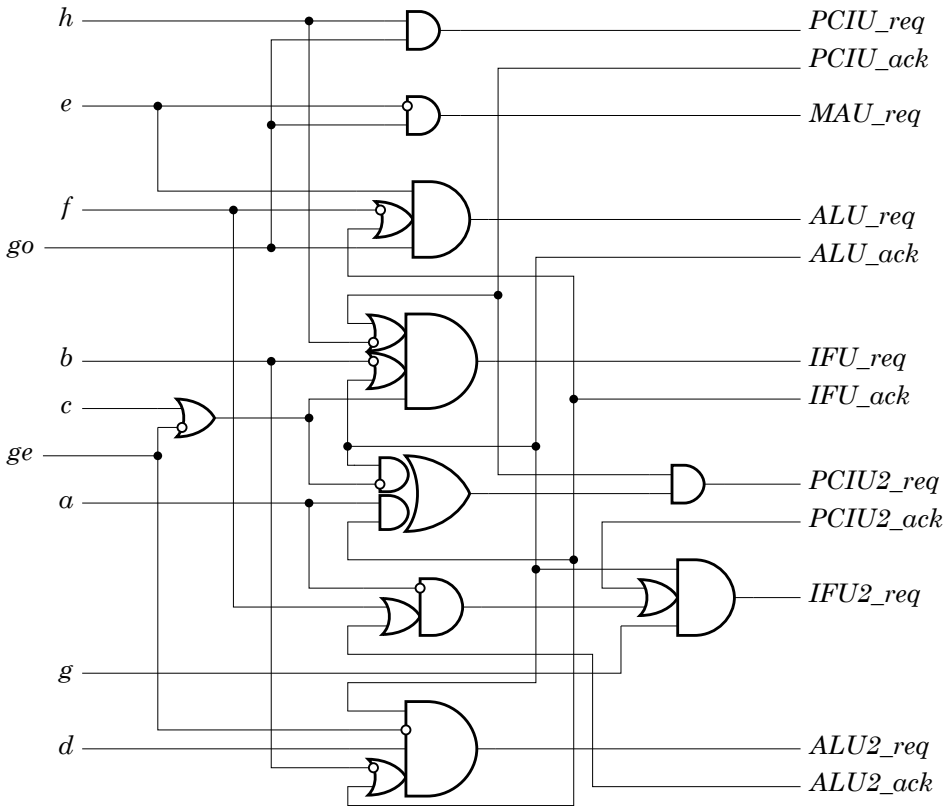


Figure 8.18: Gate-level implementation of the microcontroller

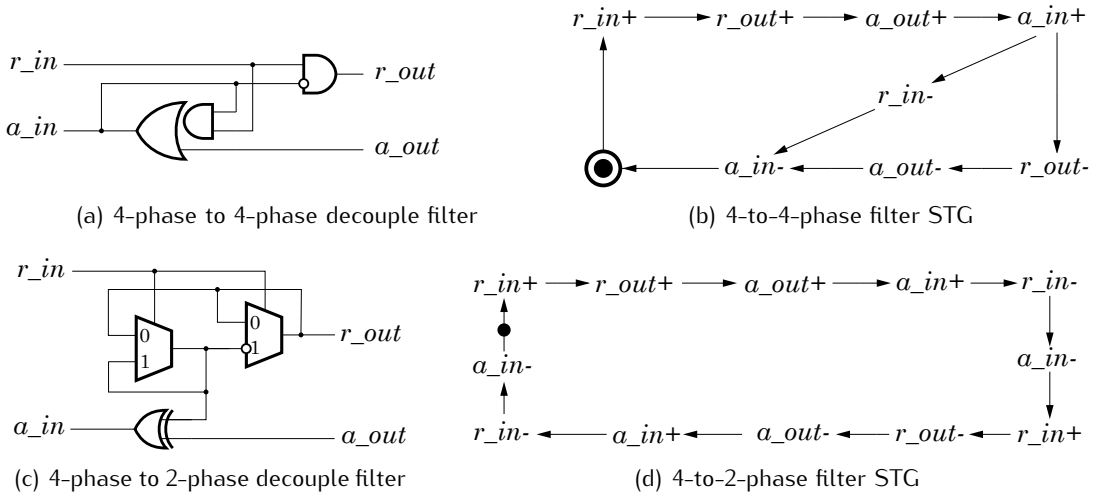
within the microcontroller may lead to significant penalties in area and latency and thus become the bottleneck of the whole processor, therefore it is reasonable to keep speed-independence only on the external level (i.e. delay-insensitive handshake communication with operational units).

The obtained microcontroller can be further optimised and/or decomposed targeting a particular gate library using standard CAD tools. It should be mentioned that the presented CPOG-based synthesis flow requires the designer's involvement only at the stage of the instruction set specification; all the later stages can be automated and the designer might even be unfamiliar with CPOGs and the underlying theory. The resultant gate-level netlist can be passed on to existing tools withing the conventionally used flow using a hardware description language, e.g. Verilog [20].

8.3.4 Handshake management

This subsection discusses several handshake related issues that arise in the context of processor microcontroller synthesis.

At first, notice that a CPOG-based microcontroller does not reset handshakes until all of them are completed (i.e. a scenario is finished). This is because a high value on an event acknowledgement wire is used as an indication of the completion of this event; the microcontroller itself does not have any memory. This leads to two potential problems: 1) an operational unit is not allowed to be reset before a scenario is finished; 2) a 4-phase protocol is imposed on communication between the microcontroller and operational units. To deal with these problems it is necessary to decouple the microcontroller from operational units. Figure 8.19(a) presents a simple solution to the first problem: this decouple filter allows the output handshake (r_out, a_out) to be reset before the input handshake (r_in, a_in) is reset (both are 4-phase handshakes). It was synthesised from the STG specification shown in Figure 8.19(b) with the Petrify tool [23].



(a) 4-phase to 4-phase decouple filter

(b) 4-to-4-phase filter STG

(c) 4-phase to 2-phase decouple filter

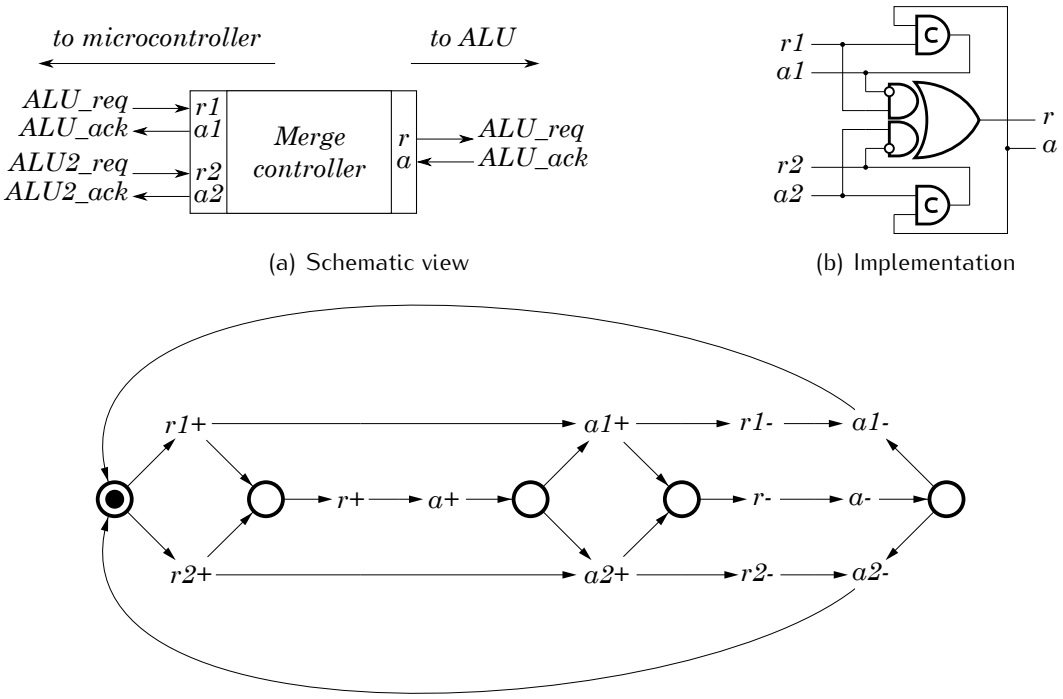
(d) 4-to-2-phase filter STG

Figure 8.19: Decoupling the microcontroller from operational units

Figure 8.19(c) presents a more sophisticated controller which converts 4-phase handshake (r_in, a_in) into 2-phase handshake (r_out, a_out). 2-phase protocol is more efficient in terms of latency and power consumption especially for long distance communication between the microcontroller and operational units; it also eliminates the first

problem, because a 2-phase handshake does not require the reset phase. The STG specification of the filter is shown in Figure 8.19(d).

The next issue to deal with is multiple occurrences of the same action in a scenario, e.g. ALU may be called twice during the execution of a single instruction. It is easy to see that these calls are mutually exclusive (this can also be checked automatically using the verification procedure provided in Subsection 6.2.6), therefore no arbitration is required for merging two requests $\mathrm{ALU}_{\mathrm{req}}$ and $\mathrm{ALU}/2_{\mathrm{req}}$ into a single one as demonstrated in Figure 8.20(a). A possible speed-independent implementation of the merge controller performing this task is shown in Figure 8.20(b). It has been synthesised with PETRIFY from the STG given in Figure 8.20(c).



(a) Schematic view

(b) Implementation

(c) STG specification

Figure 8.20: Handshakes merge controller

This is a 4-phase implementation. A 2-phase merge controller can also be synthesised using the STG approach which fits perfectly for this kind of controllers. It is also possible to generalise the merge controller for more than two requests. We omit implementation details here as this is a very well-known specification and synthesis task [25].

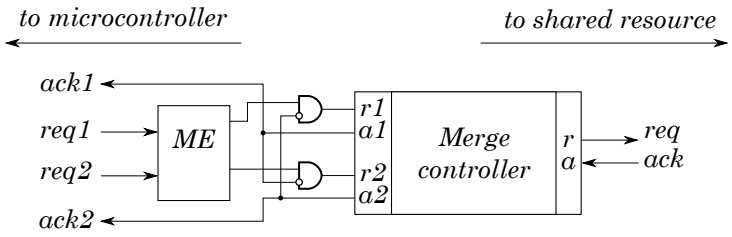


Figure 8.21: Arbitrating concurrent requests to the same operational unit

Merge controllers can only be used if the requests are mutually exclusive. If this is not the case, then there are two possible solutions: duplicate the corresponding operational unit, or set an arbiter to guard access to the unit (which in this context can be considered a shared resource). The first solution may not always be affordable either because the resource duplication is too expensive, or due to architectural constraints (for example, it is not reasonable to duplicate ALU in our case, as it requires introduction of an arbiter managing access of two ALUs to the same register bus – see Figure 8.14). Figure 8.21 shows how a merge controller can be converted into an arbiter by adding a mutex element (see Subsection 8.2.2 for details).

8.3.5 Further thoughts

This section presented application of the CPOG-driven approach to specification and synthesis of central processor microcontrollers. The example is purely academic, however, it captures many important features of real processors. It has been demonstrated that the CPOG model is capable of modelling concurrency between different subsystems and handling multiple choice during event scheduling. One of the main directions of future work is specification and synthesis of a real processor.

The other research directions include further development of optimisation techniques, in particular, the final gate-level implementation of the microcontroller (Figure 8.18) still has a room for some optimisation despite the fact that its CPOG representation has the minimum possible size. To obtain the minimum implementation size it is necessary to use the mapping information during automatic synthesis of instruction opcodes. This is possible within the CPOG methodology and has to be exploited.

8.4 Summary

This chapter provided several application examples of circuit specification and synthesis within the CPOG methodology. The examples vary from small controllers like ParSeqs (Section 8.1) to large control structures containing millions of behavioural scenarios like n-wire phase encoders (Section 8.2). ParSeq and phase encoding controllers were discussed in the motivational chapter 3 which showed that they cannot be efficiently handled with conventionally used models. This chapter demonstrated that the CPOG model is capable of providing elegant solutions for such a class of controllers.

The processor example (Section 8.3) summarised all the presented microcontroller specification and synthesis techniques and provided a design flow for processors, which starts at the architectural level and goes through stages of instruction set design, opcode generation, CPOG synthesis, circuit mapping, and handshakes management. Most of these stages are already automated in the software tool which is described in Appendix.

The next chapter summarises the contribution of the thesis and discusses further research directions.

Chapter 9

Conclusions

This thesis presented a new formal model for specification and synthesis of microcontrol circuits in the context of asynchronous systems design. Verification, synthesis, optimisation, and mapping techniques of the proposed design flow are supported with a number of software tools and were tested on a set of benchmarks. The obtained results demonstrate significant improvements over the conventionally used specification models and synthesis methods.

This chapter summarises key contributions of this thesis and outlines areas of future research.

9.1 Main contributions

It was demonstrated in Chapters 2 and 3 that the existing approaches for specification and synthesis of asynchronous microcontrollers are not applicable to a certain class of systems which contain many behavioural scenarios defined on a set of common events, because every scenario has to be explicitly defined and encoded, thereby blowing the specification size beyond what is practically feasible. The key contributions of this thesis are the new model, named Conditional Partial Order Graph, and the design flow built around it which provides an automated way to specify and synthesise microcontrol circuits without the explicit enumeration of the system scenarios or states.

The proposed CPOG-driven design flow consists of the following main stages.[1]

Specification of scenarios. In this stage the designer has to describe a system as a set of behavioural scenarios. Importantly, this description can be either implicit (see Subsection 8.2.3 for the implicit specification of $n!$ scenarios of an n-wire matrix phase encoder) or explicit (see the processor example in Section 8.3) – whichever way is more reasonable for a particular design case. This stage relies on the graphical representation of partial orders and the CPOG algebra introduced in Chapter 4 and Section 5.1, respectively. Note that in some cases it is necessary to use the dynamic CPOG model to specify an internally branching scenario as explained in Section 5.2.

Encoding of scenarios. In order to distinguish between the scenarios they are given different encodings represented with Boolean vectors. These encodings are either provided as a part of system specification (see Subsections 7.1.1 through 7.1.3) or can be assigned arbitrarily. Size and latency of the final microcontroller depends significantly on the chosen encoding of the scenarios, thus arbitrary encoding may lead to non-optimal results. Section 7.4 presented an automated procedure for the optimal encoding of a given set of scenarios.

Synthesis. Once the system scenarios are specified and encoded it is possible to synthesise a CPOG containing all of them using the algebraic techniques from Section 7.1. Resolution of true and/or false encoding conflicts may be required if the given encodings are not unique. A CPOG obtained at this stage can be regarded as an intermediate model for a compact representation of the system behaviour which will be further mapped into a gate-level circuit. Therefore the designer does not necessarily have to be familiar with the CPOG theory in order to use the CPOG-driven design flow.

Verification. The algebraic synthesis approach guarantees many properties of the synthesised CPOG automatically, therefore it is not required to verify correctness of the result. However, in cases of the custom CPOG design or manual modification/optimisation of the synthesis result the designer needs an automated support for verification. It is provided in Chapter 6 and employs SAT-based reachability analysis techniques.

[1] This list of stages is quite flexible: in some cases certain stages can overlap; others can be omitted.

Optimisation. Every system has an infinite number of equivalent CPOG representations which differ in terms of opcode variables, encodings, complexity, etc. Those with the least complexity are of a particular interest because they lead to the smallest and fastest physical implementation of the controller. Section 7.3 presented several optimisation techniques which reduce the size of a given CPOG by functional logic minimisation and/or by exploiting the structural graph properties.

Mapping. The final stage of the CPOG-driven design flow is mapping of the obtained CPOG representation into an interconnection of logic gates to produce a gate-level netlist of the microcontroller. This stage is described in Section 7.2; issues of speed-independent synthesis are addressed in Subsection 8.2.6.

The presented CPOG-based methodology was successfully applied to several design examples including: the ParSeq controllers (Section 8.1), the phase encoding communication circuits (Section 8.2), and the central processor microcontroller (Section 8.3). The results demonstrate that the CPOG model is beneficial for these classes of controllers. In particular, different levels of abstraction for data- and control-related events help to avoid combinatorial explosion in the size of system specification which is a strong limiting factor for other models (such as the STG or FSM models), especially for large design cases, such as processors, multiway routers or arbiters [68].

Extension of the CPOG model application domain and other goals of future research are discussed in the next section.

9.2 Future research directions

The presented model is new and there are a lot of future research opportunities. They can be divided into two main directions: extension of the CPOG model and investigation of its derivatives, and further in depth study of the model itself as well as development of the supporting tools.

The first direction considers applicability of the model to different classes of systems. For example, although the presented static and dynamic CPOG models have many applications discussed in Chapter 8, they are not directly applicable to systems which

exhibit more complex behaviour than a series of alternating active and reset phases (see Figure 5.17 depicting a typical evolution cycle of a controller designed within the CPOG flow). Consider a controller which computes the GCD (the greatest common divisor) of two numbers X and Y. The simplest algorithm for GCD(X, Y) computation is due to Euclid and is given below:

1. Read X and Y;
2. Compare X and Y:
   - If $X = Y$ then the algorithm terminates with result GCD $= X$;
   - If $X < Y$ then substitute Y with $Y - X$ (i.e. $Y = Y - X$);
   - If $X > Y$ then substitute X with $X - Y$ (i.e. $X = X - Y$);
3. Return to step 2.

It is impossible to specify this controller using CPOGs because there is an internal loop involving steps 2 and 3 which terminates only when condition $X = Y$ becomes true. Partial orders are acyclic and cannot deal with repetitive actions[2]. In order to handle such cyclic systems it is possible to extend the CPOG model to the *Conditional Marked Graph* (CMG) model[3]. This greatly expands the class of the modelled systems at the cost of significant complication of the model: SAT-based verification techniques are not applicable to CMGs, thus PSPACE-hard [35] verification methods are required (similar to Petri Nets verification based on unfoldings [46]). In fact, it is reasonable to study possible 'conditional' extensions of other models besides Marked Graphs, e.g. Static Data Flow Structures (SDFSs) [93] or Structured Occurrence Nets (SONs) [53].

It should be mentioned, however, that the CPOG model has many useful properties due to acyclicity of underlying partial orders, therefore investigation of its possible extensions should not stop the research and development of the model itself, hence the second research direction. For example, it is important to formalise the speed-independent

[2]But it is possible to add an extra circuitry restarting an acyclic microcontroller as many times as required to satisfy a particular condition. This workaround, however, requires non-trivial design decisions and does not fit the automated CPOG flow easily.

[3]See definition of Marked Graphs in Chapter 2.

controller synthesis (see Subsection 8.2.6), as well as to develop the corresponding synthesis and verification tools.

Another area of future work is further development of CPOG optimisation techniques, such as: taking the mapping information into account during scenario encoding, synthesis of optimal opcodes of minimal length (or required length), synthesis of variable length opcodes based on statistical properties of scenarios using Huffman coding [41], etc.

These computationally expensive optimisation techniques may require improvement of the data structure that is used to represent a CPOG in memory. One possible way to do that is to employ BDDs [54] or ZDDs [63], which are well-known compact representations of Boolean functions. They can also provide a canonical CPOG representation due to properties of Reduced Ordered BDDs [15]. It is important to note that all the vertex/arc conditions of a graph should be stored in a shared BDD data structure as explained in [14], rather than separately, thereby significantly reducing memory consumption.

Finally, design and fabrication of a real processor using the CPOG methodology should provide a rigorous practical examination of the presented automated CPOG flow.

Appendix A

Tool support for CPOGs

The appendix explains how to use the developed set of tools for systematic manipulation with CPOGs, their visualisation and simulation, and synthesis of the physical circuit implementation in structural Verilog netlist.

The overall set of tools is divided into several toolkits:

- `ce` – the **C**POG **e**ngine, which implements the basic algebraic operations over CPOGs, algorithms for synthesis, optimisation, verification, and mapping (Section A.1);
- `wm` – the **W**ORKCRAFT **m**odel, a set of plugins supporting the visualisation and simulation functionality within the WORKCRAFT framework [82][84] (Section A.2);
- `pe` – the **p**hase **e**ncoding toolkit for synthesis of multiple rail phase encoding receivers, senders, and repeaters (Section A.3).

A.1 The CPOG engine

This toolkit performs operations from the CPOG algebra (projection, addition, scalar multiplication, asymmetric addition) introduced in Chapter 5, as well as synthesis, verification, optimisation and mapping procedures described in Chapters 6-7. The tools are implemented in C++ programming language and are listed below.

ce_assign – given a graph $H(V, E, X, \rho, \phi)$, a variable $x \in X$, and a Boolean value $\alpha \in \{0, 1\}$ the tool computes the projection $H|_{x=\alpha}$ (see Definition 5.2) and writes the result to the console or the specified file.

```
USAGE:
ce_assign INPUT_FILE_NAME VARIABLE_NAME VALUE [-o OUTPUT_FILE_NAME]
```

The variable name may be enclosed within quotes if necessary. For example, the following command reads in a graph H from file `source.cpog`, computes its projection $H|_{var=1}$, and writes it to `result.cpog`:

```
ce_assign source.cpog var 1 -o result.cpog
```

ce_add – given two graphs H_1 and H_2 the tool computes their sum $H_1 + H_2$ (see Definition 5.12) and writes the result to the console or the specified file.

```
USAGE:
ce_add INPUT_FILE_NAME1 INPUT_FILE_NAME2 [-o OUTPUT_FILE_NAME]
```

For example, the following command reads in two graphs from files `source1.cpog` and `source2.cpog`, computes their sum and writes it to `result.cpog`:

```
ce_add source1.cpog source2.cpog -o result.cpog
```

ce_mult – given a graph H and a Boolean function f the tool computes the scalar multiplication $f \cdot H$ (see Definition 5.13) and writes the result to the console or the specified file.

```
USAGE:
ce_mult INPUT_FILE_NAME FUNCTION [-o OUTPUT_FILE_NAME]
```

The function may be enclosed within quotes if necessary. For example, the following command reads in a graph H from file `source.cpog`, computes the scalar multiplication $(x+y)H$ and writes it to `result.cpog`:

```
ce_mult source.cpog "(x + y)" -o result.cpog
```

ce_asym_add – given two graphs H_1 and H_2 the tool computes their asymmetric sum $H_1 \vec{+} H_2$ (see Definition 5.15) and writes the result to the console or the specified file.

```
USAGE:
ce_asym_add INPUT_FILE_NAME1 INPUT_FILE_NAME2 [-o OUTPUT_FILE_NAME]
```

For example, the following command reads in two graphs from files `source1.cpog` and `source2.cpog`, computes their asymmetric sum and writes it to `result.cpog`:

```
ce_asym_add source1.cpog source2.cpog -o result.cpog
```

ce_synthesise – given a set of partial orders the tool synthesises their CPOG composition using a chosen encoding scheme (see Section 7.1 for details) and writes the result to the console or the specified file.

```
USAGE:

ce_synthesise INPUT_FILE_NAME ENCODING_SCHEME [-o OUTPUT_FILE_NAME]

ENCODING SCHEMES:
```

`one-hot` – the one-hot encoding scheme (see Subsection 7.1.1);

`binary` – the binary encoding scheme (see Subsection 7.1.2);

`matrix` – the matrix encoding scheme (see Subsection 7.1.3);

`optimal` – the optimal encoding scheme (see Section 7.4).

For example, the following command reads in a set of partial orders from file `source.cpog`, encodes them using the binary encoding scheme, synthesises their composition and writes the obtained graph to `result.cpog`:

```
ce_synthesise source.cpog binary -o result.cpog
```

ce_verify – given a graph (or two) and a property to verify, the tool performs the verification and reports the result.

```
USAGE:

ce_verify INPUT_FILE_NAME1 PROPERTY [INPUT_FILE_NAME2] [EVENT_NAME1 EVENT_NAME2]

PROPERTIES:
```

`well-formed` – checking well-formedness of the given graph (see Definition 5.6);

`equivalent` – checking equivalence of two given graphs (see Definition 5.10);

`encoding-conflict` – checking for encoding conflicts (see Definition 5.9);

`deadlock-free` – checking if the given graph is deadlock-free (see Subsection 6.2.3);

`invalid-states` – checking reachability of an invalid state (see Subsection 6.2.4);

`event-conflict` – checking for event conflicts in the given graph (see Subsection 6.2.5);

`mutual-exclusion` – checking for mutual exclusion (see Subsection 6.2.6).

The event names may be enclosed within quotes if necessary. Consider several examples. The following command reads in a graph from file `source.cpog` and reports if it is well-formed:

```
ce_verify source.cpog well-formed
```

The next command reads in two graphs from files `source1.cpog` and `source2.cpog`, and reports if they are equivalent taking into account their encodings (see Subsection 6.1.2 for details):

```
ce_verify source1.cpog equivalent source2.cpog
```

Mutual exclusion of events ALU and ALU/2 in a graph described with file `source.cpog` can be verified using the following command:

```
ce_verify source.cpog mutual-exclusion ALU "ALU/2"
```

Note that the second event name is enclosed within quotes to avoid a possible confusion of a command-line interpreter due to the forward slash '/' symbol.

ce_optimise – given a graph the tool tries to reduce its complexity using the CPOG optimisation techniques introduced in Section 7.3, and writes the result to the console or the specified file.

```
USAGE:
ce_optimise INPUT_FILE_NAME [-o OUTPUT_FILE_NAME]
```

For example, the following command reads in a graph from file `source.cpog`, and writes its optimised version into `result.cpog`:

```
ce_optimise source.cpog -o result.cpog
```

ce_map – given a graph the tool maps it into Boolean equations as explained in Section 7.2, and writes the equations to the console or the specified file.

```
USAGE:
ce_map INPUT_FILE_NAME [-o OUTPUT_FILE_NAME]
```

For example, the following command reads in a graph from file `source.cpog` and writes the generated equations into `result.eqn`:

```
ce_optimise source1.cpog -o result.eqn
```

A.2 The Workcraft model

The Workcraft framework [82][84] is a collection of tools designed to provide a flexible common environment for development of *Interpreted Graph Models* [82], including visual editing, (co-)simulation and analysis. The framework is platform-independent, highly customisable by means of plugins, and is freely available for academic use at [4].

This section explains how to use the Workcraft framework for visual editing and simulation of CPOGs. Subsection A.2.1 describes the process of creating a new graph and editing it, while Subsection A.2.2 addresses the process of a CPOG simulation.

A.2.1 Creating and editing a graph

To create a new graph, start the WORKCRAFT framework, select **New...** in the **File** menu (or use **Ctrl-N** keyboard shortcut), highlight the *Conditional Partial Order Graph* list item in the **New model** dialogue, and press **OK** (see the corresponding screenshot in Figure A.1). The list of available components (the leftmost panel) should update itself and contain *vertex* and *variable* components. To add a new vertex or a new variable to the graph (which is initially empty), drag the corresponding component to the main working area (or use keyboard shortcuts **V** or **X** respectively). A vertex is depicted as a circle, while a variable – as a box with its current Boolean value inside.

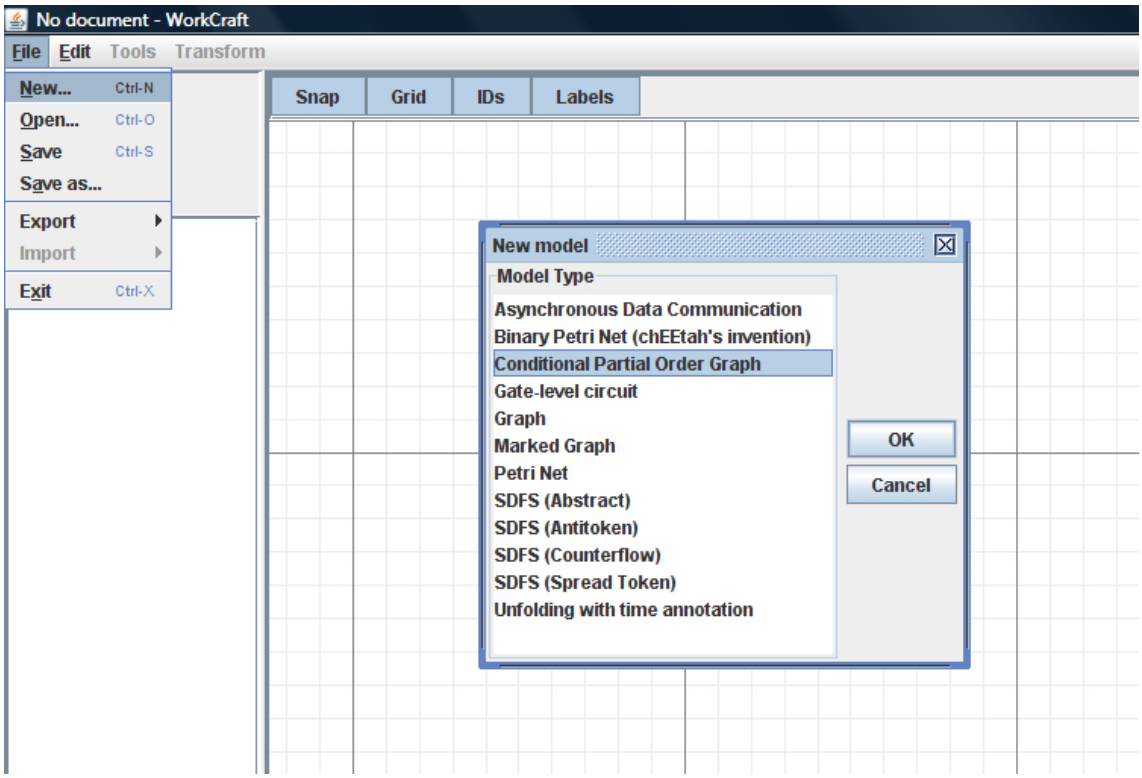


Figure A.1: Creating a Conditional Partial Order Graph in WORKCRAFT

To connect any two vertices with an arc, click button **Connect** (or press keyboard shortcut **C**) and then click on the two vertices in the appropriate order. To change the name, the label, the condition, or any other property of a vertex, arc, or variable, click on it and edit the corresponding field in the property editor window (the rightmost panel). Initially, all the created variables are static (see Definition 5.16); to make a variable

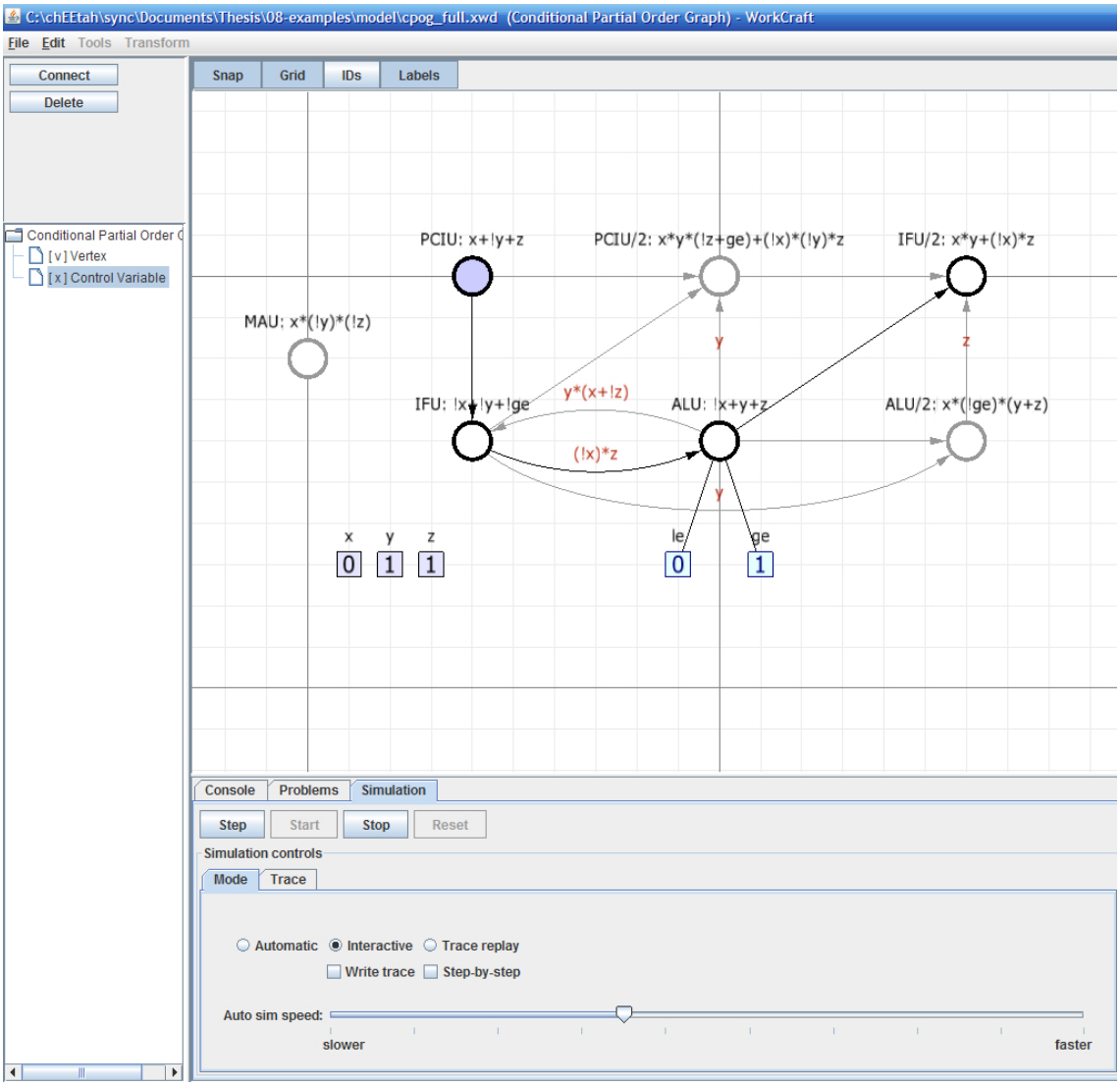


Figure A.2: CPOG simulation in WORKCRAFT

dynamic connect it to its master vertex. The framework provides all the basic visual editing features, e.g. moving the created components around the working area (left click and drag), zooming in and out (mouse wheel), panning (right click and drag), etc.

At any moment of time it is possible to save the graph into a file (**File** → **Save**/**Save as...**), or to export it (**File** → **Export**) into one of the available graphical vector formats, including SVG (Scalable Vector Graphics), EPS (Encapsulated PostScript), PDF (Portable Document Format), and others. To load a graph from the previously saved file use the **File** → **Open** command.

A.2.2 Simulation

To enter the simulation mode select **Simulation** tab in the bottom panel and click **Start**. See the screenshot in Figure A.2 which captures a moment in simulation of the CPOG describing the example processor from Chapter 8 (the CPOG can be found in Figure 8.16). During the simulation, vertices that are enabled to fire are highlighted with blue (e.g. vertex *PCIU* in the screenshot), inactive vertices and arcs are grey (e.g. vertex *MAU*, arc *ALU* → *IFU*, etc.). To fire an enabled vertex click on it, or use the automatic means of simulation provided in the simulation tab: step-by-step simulation (a random enabled vertex is chosen and fired), automatic mode with controllable simulation speed, etc. To reset the graph into the initial state use button **Reset**.

A.3 Synthesis of phase encoding controllers

This section describes the tools developed for synthesis of multiple rail phase encoding controllers, which are divided into two packages: phase encoding receivers (Subsection A.3.1) and phase encoding senders (Subsection A.3.2).

A.3.1 Phase encoding receivers

This package contains a set of tools for automated synthesis of phase encoding receivers with different parameters. The tools are combined within **pe_recv** command-line wrapper described below.

```
USAGE:
pe_recv -o OUTPUT_FILE -w NWIRES -p PHASE -lib LIBRARY [-neg] -f FORMAT [-c] [-h]
```

The parameters are:

- `-o OUTPUT_FILE` – name of the output file, if omitted the tool prints the result to the console.
- `-w NWIRES` – specifies the number of wires in the phase encoding channel (default value: 2).

- `-p PHASE` – specifies the working phase of the receiver. Supported values are: the rising phase (`-p rise`) which is the default, the falling phase (`-p fall`), or both (`-p both`).
- `-lib LIBRARY` – specifies the gate library for technology mapping. Supported values are: `-lib abstract` (default) and `-lib ams035`.
- `-neg` – performs negative logic optimisation if possible.
- `-f FORMAT` – specifies the output format. Supported values are: `-f verilog` (default) and `-f xml`.
- `-c` – compresses the resultant netlist by shortening all the internal wire names.
- `-h` – prints out all these options.

For example, the following command synthesises a 5-wire phase encoding receiver circuit operating on the rising signal edge, maps it to the AMS 0.35µ technology library, compresses the final Verilog netlist, and writes it to `receiver5.v`:

```
pe_recv -o receiver5.v -w 5 -c -lib ams035
```

To produce the same controller implemented with abstract (unmapped) gates and save it in XML format with no compression use the following command:

```
pe_recv -o receiver5.xml -w 5 -f xml
```

A.3.2 Phase encoding senders

This package contains a set of tools for automated synthesis of phase encoding senders with different parameters. The tools are combined within **pe_send** command-line wrapper described below.

```
USAGE:
pe_send -o OUTPUT_FILE -w NWIRES -p PHASE -lib LIBRARY [-neg] -f FORMAT [-c] [-h]
```

The parameters are:

- `-o OUTPUT_FILE` – name of the output file, if omitted the tool prints the result to the console.
- `-w NWIRES` – specifies the number of wires in the phase encoding channel (default value: 2).

- `-p PHASE` – specifies the working phase of the receiver. Supported values are: the rising phase (`-p rise`) which is the default, and the falling phase (`-p fall`).
- `-lib LIBRARY` – specifies the gate library for technology mapping. Supported values are: `-lib abstract` and `-lib ams035` (default).
- `-neg` – performs negative logic optimisation if possible.
- `-f FORMAT` – specifies the output format. Supported values are: `-f verilog` (default) and `-f xml`.
- `-c` – compresses the resultant netlist by shortening all the internal wire names.
- `-h` – prints out all these options.

For example, the following command synthesises a 4-wire phase encoding sender circuit operating on the falling signal edge, maps it to the AMS 0.35µ technology library, applies negative logic optimisation, and writes it to `sender4.v`:

```
pe_send -o sender4.v -w 4 -p fall -lib ams035 -neg
```

To produce the same controller implemented with abstract (unmapped) gates and save it in XML format without the negative logic optimisation use the following command:

```
pe_send -o sender4.xml -w 4 -p fall -f xml
```

Bibliography

[1] Balsa project homepage. http://intranet.cs.man.ac.uk/apt/projects/tools/balsa/.

[2] MSP430x4xx Family User's Guide. Texas Instruments. http://focus.ti.com/lit/ug/slau056i/slau056i.pdf.

[3] International Technology Roadmap for Semiconductors (ITRS'07). http://www.itrs.net/Links/2007ITRS/Home2007.htm, 2007.

[4] The Workcraft framework homepage. http://www.workcraft.org, 2009.

[5] Howard H. Aiken and Grace M. Hopper. The automatic sequence controlled calculator. *Electrical Engineering*, Vol. 65; No. 8-9: pp. 384-391 (Aug 1946); No. 10: pp. 449-454 (Oct 1946); No. 11: pp. 522-528 (Nov 1946).

[6] A. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, and M. Imai. PEAS-I: A hardware/software co-design system for ASIPs. In *Proc. of European Design Automation Conference (EURO-DAC)*, pages 2–7, 1993. ISBN: 0-8186-4350-1.

[7] Michael A Arbib. *Theories of abstract automata (Prentice-Hall series in automatic computation)*. Prentice-Hall, Inc., 1969. ISBN: 0139133682.

[8] W. J. Bainbridge and S. B. Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *ASYNC '01: Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, page 118. IEEE Computer Society, 2001. ISBN: 0-7695-1034-5.

[9] W. J. Bainbridge, W. B. Toms, David A. Edwards, and Stephen B. Furber. Delay-Insensitive, Point-to-Point Interconnect Using M-of-N Codes. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'03)*, pages 132–140, 2003.

[10] Andrew Bardsley and Doug Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, 2000.

[11] P. A. Beerel, K. Y. Yun, and W. C. Chou. Optimizing average-case delay in technology mapping of burst-mode circuits. In *Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'96)*, page 244. IEEE Computer Society, 1996.

[12] Peter A. Beerel and Teresa H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design (ICCAD'92)*, pages 581–586. IEEE Computer Society Press, 1992.

[13] G. Birkhoff. *Lattice Theory*. Third Edition, American Mathematical Society, Providence, RI, 1967. ISBN: 0821810251.

[14] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45. ACM, 1990. ISBN: 0-89791-363-9.

[15] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986. ISBN: 0018-9340.

[16] A W Burks, H H Goldstein, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Institute for Advanced Study, 1946.

[17] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous systems*. PhD thesis, Stanford University, 1984.

[18] Fu-Chiung Cheng. Practical design and performance evaluation of completion detection circuits. In *IEEE International Conference on Computer Design (ICCD)*, pages 1063–6404, 1998. ISBN: 0-8186-9099-2.

[19] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, MIT Laboratory for Computer Science, 1987.

[20] Michael D. Ciletti. *Advanced Digital Design with the VERILOG HDL*. Prentice Hall PTR, 2002. ISBN: 0130891614.

[21] Wesley A. Clark. Macromodular computer systems. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 335–336, New York, NY, USA, 1967. ACM.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. ISBN: 0262031418.

[23] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[24] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Automatic handshake expansion and reshuffling using concurrency reduction. In *Proceedings of HWPN'98*, 1998.

[25] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002. ISBN: 3540431527.

[26] Crescenzo D'Alessandro, Andrey Mokhov, Alex Bystrov, and Alex Yakovlev. Delay/Phase Regeneration Circuits. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 105–116, 2007.

[27] Crescenzo D'Alessandro, Delong Shang, Alexandre V. Bystrov, and Alexandre Yakovlev. PSK signalling on NoC buses. In *PATMOS*, pages 286–296. Springer, 2005.

[28] Crescenzo D'Alessandro, Delong Shang, Alexandre V. Bystrov, Alexandre Yakovlev, and Oleg V. Maevsky. Multiple-rail phase-encoding for NoC. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 107–116, 2006.

[29] William J. Dally and John W. Poulton. *Digital systems engineering*. Cambridge University Press, New York, NY, USA, 1998. ISBN: 0-521-59292-5.

[30] René David. Modular design of asynchronous circuits defined by graphs. *IEEE Trans. Computers*, 26(8):727–737, 1977.

[31] A. Davis and S. Nowick. An introduction to asynchronous system design. Technical report, University of Utah, September (UUCS-97-013) 1997.

[32] Jorg Desel and Javier Esparza. *Free choice Petri nets*. Cambridge University Pres, 1995. ISBN: 0521465192.

[33] Victor I. Varshavsky (Editor). *Self-Timed Control of Concurrent Processes*. Kluwer Academic Publishers, 1990. ISBN: 0792305256.

[34] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, pages 333–336, 2004. ISBN: 978-3-540-20851-8.

[35] J. Esparza. Decidability and Complexity of Petri Net Problems - an Introduction. In *Lectures on Petri Nets I: Basic Models, W. Reisig and G. Rozenberg (Eds.)*., pages 374–428, 1998. ISBN: 3-540-65306-6.

[36] Javier Esparza and Mogens Nielsen. Decidability Issues for Petri Nets. *Petri nets newsletter*, 52:245–262, 1994.

[37] K. Fant and S.A. Brandt. Null conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *Proc. Int'l Conf. Application-Specific*

Systems, Architectures, and Processors (ASAP 96), volume 19, pages 261 – 273, 1996.

[38] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. ISBN: 0201558025.

[39] Lee A. Hollaar. Direct implementation of asynchronous control units. *IEEE Trans. Computers*, 31(12):1133–1141, 1982.

[40] Bruce Kester Holmer. *Automatic design of computer instruction sets*. PhD thesis, 1993. Co-Chair-Culler, David E. and Co-Chair-Despain, Alvin M.

[41] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[42] B. Robic J. Silc, T. Ungerer. A survey of new research directions in microprocessors. *Microprocessors and Microsystems*, pages 175–190, 2000.

[43] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*. Springer-Verlag, 1997. ISBN: 3540582762.

[44] Ka-Ming Keung and Akhilesh Tyagi. State space reconfigurability: an implementation architecture for self modifying finite automata. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 83–92. ACM, 2006. ISBN: 1-59593-543-6.

[45] Victor Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, University of Newcastle upon Tyne, School of Computing Science, 2003.

[46] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting State Coding Conflicts in STG Unfoldings Using SAT. In *Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)*, page 51, 2003. ISBN: 0-7695-1887-7.

[47] Victor Khomenko, Maciej Koutny, and Alexandre Yakovlev. Logic synthesis for asynchronous circuits based on petri net unfoldings and incremental sat. In *Proceedings of International Conference on Applicatioon of Concurrency to System Design (ACSD'04)*, pages 16–25, 2004.

[48] Victor Khomenko and Mark Schäfer. Combining decomposition and unfolding for stg synthesis. In *Proc. of 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, pages 223–243, 2007.

[49] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley and Sons, 2008. ISBN: 978-0-470-51082-7.

[50] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent hardware: the theory and practice of self-timed design*. John Wiley & Sons, Inc., 1994. Translator-Yakovlev, Alex and Translator-Napelbaum, Eric and Translator-Reva, Olga. ISBN: 0-471-93536-0.

[51] Donald E. Knuth. Big Omicron and big Omega and big Theta. *SIGACT News*, 8(2):18–24, 1976. ISSN: 0163-5700.

[52] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous CAD tools. *IEEE Design & Test of Computers*, 19(4):107–117, 2002.

[53] Maciej Koutny and Brian Randell. Structured Occurrence Nets: A formalism for aiding system failure prevention and analysis techniques. Technical report, Newcastle University, August 2009.

[54] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.

[55] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable asips. In *Proc. of the International Conference on Computer-aided Design (ICCAD)*, pages 649–654, 2002.

[56] Art Lew. *Computer Science: A Mathematical Introduction.* Prentice-Hall, 1985. ISBN: 0-13-164252-9.

[57] Roger Lipsett, Cary A. Ussery, and Carl F. Schaefer. *VHDL, Hardware Description and Design*. Kluwer Academic Publishers, 1993. ISBN: 9780792390305.

[58] Alain J. Martin. Compiling communicating processes into delay-insensitive vlsi circuits. Technical report, California Institute of Technology. [CaltechCSTR:1986.5210-tr-86], 1986.

[59] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278. MIT Press, 1990.

[60] E. L. McCluskey. Minimization of boolean functions. *Bell System Technical Journal*, pages 149–175, 1959.

[61] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1979. ISBN: 0201043580.

[62] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994. ISBN: 0070163332.

[63] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC '93: Proceedings of the 30th international Design Automation Conference*, pages 272–277. ACM, 1993. ISBN: 0-89791-577-1.

[64] P. Mishra and N. Dutt. *Processor Modelling and Design Tools, Chapter 8 in 'EDA for IC Systems Design, Verification, and Testing' by L. Sheffer, L. Lavagno, and G. Martin*. Taylor and Francis Group, 2006. ISBN: 0849379237.

[65] Andrey Mokhov, Crescenzo D'Alessandro, and Alex Yakovlev. Synthesis of multiple rail phase encoding circuits. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 95–104. IEEE Computer Society, 2009. ISBN: 978-0-7695-3616-3.

[66] Andrey Mokhov, Ulan Degenbaev, and Alex Yakovlev. Optimal Encoding of Partial Orders. Technical report, Newcastle University, February 2009.

[67] Andrey Mokhov, Ulan Degenbaev, and Alex Yakovlev. Synthesis of instruction codes in the context of asynchronous microcontrol design. In *UK Asynchronous Forum*, 2009.

[68] Andrey Mokhov, Victor Khomenko, and Alex Yakovlev. Flat arbiters. In *Proc. of 9th International Conference on Applicatioon of Concurrency to System Design (ACSD'09)*, pages 99–108, 2009.

[69] Andrey Mokhov, Danil Sokolov, and Alex Yakovlev. Completion detection optimisation based on relative timing. In *UK Asynchronous Forum*, 2006.

[70] Andrey Mokhov and Alex Yakovlev. Conditional partial order graphs algebra. Technical report, Newcastle University, September 2008.

[71] Andrey Mokhov and Alex Yakovlev. Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis. In *Proceedings of Design, Automation and Test in Europe (DATE) Conference*, pages 1142–1147, 2008. ISBN: 978-3-9810801-3-1.

[72] Andrey Mokhov and Alex Yakovlev. Verification of conditional partial order graphs. In *Proc. of 8th Int. Conf. on Application of Concurrency to System Design (ACSD'08)*, pages 128–137, 2008. ISBN: 978-1-4244-1838-1.

[73] Charles E. Molnar and Ian W. Jones. Simple circuits that work for complicated reasons. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 138, Washington, DC, USA, 2000. IEEE Computer Society. ISBN: 0-7695-0586-4.

[74] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[75] D. Muller and W. Bartky. A Theory of Asynchronous Circuits. In *Proc. Int. Symp. of the Theory of Switching*, pages 204–243, 1959.

[76] Tadao Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, 1989.

[77] A. Nohl, V. Greive, G. Braun, A. Hoffman, R. Leupers, O. Schliebusch, and H. Meyr. Instruction encoding synthesis for architecture exploration using hierarchical processor models. In *Proc. of the 40th annual Design Automation Conference (DAC)*, pages 262–267. ACM, 2003.

[78] Steven Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, 1993.

[79] Steven Nowick, Kenneth Yun, Peter Beerel, and Ayoob Dooply. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, page 210, 1997. ISBN: 0-8186-7922-0.

[80] Carl Adam Petri. *Kommunikation mit automaten (Communicating with automata)*. PhD thesis, University of Bonn, 1962.

[81] Luis A. Plana, Steve B. Furber, Steve Temple, Mukaram Khan, Yebin Shi, Jian Wu, and Shufan Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design and Test*, 24(5):454–463, 2007.

[82] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Workcraft – A Framework for Interpreted Graph Models. In *Proceedings of the 30th International Conference on Applications and Theory of Petri Nets*, pages 333–342, 2009.

[83] Ivan Poliakov, Andrey Mokhov, Ashur Rafiev, Danil Sokolov, and Alex Yakovlev. Automated Verification of Asynchronous Circuits Using Circuit Petri Nets. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 161–170, 2008. ISBN: 978-0-7695-3107-6.

[84] Ivan Poliakov, Danil Sokolov, Andrey Mokhov, and Alex Yakovlev. Workcraft: a static data flow structure editing, visualisation and analysis tool. In *Proceedings of the 28th International Conference on Applications and Theory of Petri Nets*, pages 505–514, 2007.

[85] W. V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955. ISSN: 00029890.

[86] Brian Randell. From Analytical engine to electronic digital computer: the contributions of Ludgate, Torres, and Bush. *IEEE Annals of the History of Computing*, 4(4):327–341, 1982.

[87] Leonid Rosenblum and Alexandre Yakovlev. Signal graphs: From self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–206, 1985.

[88] Richard L. Rudell and Alberto L. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[89] Mark Schaefer, Dominic Wist, and Ralf Wollowski. Desij - enabling decomposition-based synthesis of complex asynchronous controllers. In *Proc. of 9th International Conference on Applicatioon of Concurrency to System Design (ACSD'09)*, pages 186–190, 2009. ISBN: 1550-4808.

[90] Louis Scheffer, Luciano Lavagno, and Grant Martin. *EDA for IC Systems Design, Verification, and Testing*. Taylor and Francis Group, 2006. ISBN: 0849379237.

[91] J. Shutt and R. Rubinstein. Self-modifying finite automata: An introduction. *Information Processing Letters*, 56(4 (24 November)):185–190, 1995.

[92] Danil Sokolov. *Automated synthesis of asynchronous circuits using direct mapping for control and data paths*. PhD thesis, Newcastle University, 2005.

[93] Danil Sokolov, Ivan Poliakov, and Alex Yakovlev. Analysis of static data flow structures. *Fundamenta Informaticae*, 88(4):581–610, 2008.

[94] Danil Sokolov and Alex Yakovlev. Clock-less circuits and system synthesis. In *IEE Proceedings, Computers and Digital Techniques*, volume 152, pages 298–316, 2005. ISBN: 1350-2387.

[95] Jens Sparsø and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001. ISBN: 0792376137.

[96] Alexander Taubin, Jordi Cortadella, Luciano Lavagno, Alex Kondratyev, and Ad M. G. Peeters. Design automation of real-life asynchronous devices and systems. *Foundations and Trends in Electronic Design Automation*, 2(1):1–133, 2007.

[97] W. B. Toms, D. A. Edwards, and A Bardsley. Synthesising heterogeneously encoded systems. In *ASYNC '06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, page 138. IEEE Computer Society, 2006. ISBN: 0-7695-2498-2.

[98] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Eindhoven University of Technology, Department of Computing Science, 1984.

[99] Stephen H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley and Sons, Inc., New York, 1969. ISBN: 0471896322.

[100] K. v. Berkel, H. van Gadeldonk, J. Kessels, C. Niessen, A.Peeters, M. Roncken, and R. van de Wiel. Asynchronous does not imply low power, but... In *Low Power CMOS Design. IEEE Computer Society*, 1998.

[101] Antti Valmari. The state explosion problem. *Lecture Notes in Computer Science. Lectures on Petri Nets I: Basic Models (Edited by W. Reisig and G. Rozenberg)*, 1491:429–528, 1998. ISBN: 3-540-65306-6.

[102] Kees van Berkel. *Handshake circuits: an asynchronous architecture for VLSI programming*. Cambridge University Press, 1993. ISBN: 0521617154.

[103] Kees van Berkel, Mark Josephs, and Steven Nowick. Scanning the technology: applications of asynchronous circuits. In *Proceedings of the IEEE*, pages 223–233, 1999.

[104] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384 – 389, 1991. ISBN: 0-8186-2130-3.

[105] Tom Verhoeff. Delay-insensitive codes - an overview. *Distributed Computing*, 3(1):1–8, 1988.

[106] Jiacun Wang. *Timed Petri nets: theory and application*. Springer, 1998. ISBN: 0792382706.

[107] Ingo Wegener. *The Complexity of Boolean Functions*. Johann Wolfgang Goethe-Universitat, 1987. ISBN: 0471915556.

[108] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, pages 189–234, 1996.

[109] Alexandre Yakovlev, Albert Koelmans, and Luciano Lavagno. High-Level Modeling and Design of Asynchronous Interface Logic. *IEEE Design & Test of Computers*, 12(1):32–40, 1995.

[110] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3D Asynchronous State Machines. In *In Proc. International Conf. Computer Design (ICCD*, pages 346–350. IEEE Computer Society Press, 1992.

[111] Qin Zhao, Bart Mesman, and Twan Basten. Practical instruction set design and compiler retargetability using static resource models. In *Proc. Design, Automation and Test in Europe (DATE)*, page 1021, 2002.

[112] Yu Zhou, Danil Sokolov, and Alex Yakovlev. Cost-aware synthesis of asynchronous circuits based on partial acknowledgement. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD'06)*, pages 158–163. ACM, 2006.

Index