
School of Electrical and Electronic Engineering



Compositional Approach to Design of Digital Circuits

Arseniy Alekseyev

Technical Report Series

NCL-EEE-MICRO-TR-2014-191

June 2014

Contact:

`rotsor@gmail.com`

Supported by EPSRC grants EP/G037809/1 and EP/K001698/1

NCL-EEE-MICRO-TR-2014-191

Copyright © 2014 University of Newcastle upon Tyne

School of Electrical and Electronic Engineering,

Merz Court,

University of Newcastle upon Tyne,

Newcastle upon Tyne, NE1 7RU, UK

<http://async.org.uk/>

University of Newcastle upon Tyne
School of Electrical and Electronic Engineering



Compositional Approach to Design of Digital Circuits

by Arseniy Alekseyev

PhD Thesis

June 2013

Contents

1	Introduction	1
1.1	Contributions	4
1.2	Overview	4
2	Background	8
2.1	Handshake circuits	8
2.2	Petri nets	11
2.3	Conditional Partial Order Graphs	17
2.4	Agda	19
2.4.1	Function definitions and algebraic data types	20
2.4.2	Inductive types and recursion	21
2.4.3	Indexed types and propositions	21
3	Improved Parallel Composition	25
3.1	Introduction	25
3.2	Improved parallel composition	29
3.3	Discussion	32
3.4	Proof of Proposition 1	33
3.5	Experiments	34
3.6	Balsa workflow optimisation through STG resynthesis	37
3.6.1	Support of Breeze handshake circuits as interpreted graph model in WORKCRAFT	40
3.6.2	STG specifications of individual handshake components	41
3.6.3	An example: GCD controller	42
3.6.4	Experimental results	45

3.7	Summary	46
4	Theory of Parametrised Graphs	48
4.1	Introduction	48
4.2	Parametrised Graphs	50
4.2.1	Specification and composition of instructions	57
4.3	Algebraic structure of Parametrised Graphs	58
4.4	Transitive Parametrised Graphs and their algebra	61
4.5	Case study	64
4.5.1	Phase encoders	65
4.6	Machine-assisted formalisation of Parametrised Graph theory	67
4.6.1	Graph Algebra	68
4.6.2	Parametrised Graphs	70
4.6.3	Parametrised Graph formulae	71
4.6.4	Formula equivalence	72
4.6.5	Normal form	73
4.6.6	Normalisation algorithm	74
4.7	Summary	77
5	Processor instruction set encoding	79
5.1	Introduction	79
5.2	Problem statement	82
5.2.1	Overview	84
5.2.2	Globally optimal encoding	86
5.3	SAT formulation	86
5.3.1	Weakly optimal encoding	87
5.3.2	Globally optimal encoding	88
5.3.3	Support for dynamic variables	91
5.4	Processor design example	92
5.4.1	Instructions encoding	98
5.4.2	Microcontroller synthesis	100
5.5	Summary	102

6	Conclusions	103
6.1	Improved parallel composition	103
6.2	Parametrised Graphs theory	104
6.3	Processor instruction set encoding	105
6.4	Future work	105
A	Formal proof of PG Algebra properties	107

List of Figures

1.1	Design productivity gap	1
1.2	Composition basics	3
1.3	Thesis structure	6
2.1	Handshake components	11
2.2	Parallel composition example	16
2.3	An example of a transition contraction.	16
2.4	Graphical representation of CPOGs and their projections	18
3.1	Example of standard STG composition.	26
3.2	Improved STG composition	27
3.3	Equivalence preservation by improved parallel composition	30
3.4	Example of invalid place removal	31
3.5	Example of enforcing injective labelling in an STG.	32
3.6	Scalable Balsa controllers used in experiments.	35
3.7	Number of non-contractible dummy transitions, normalized to the best value achieved.	36
3.8	Balsa design workflow	38
3.9	Modified Balsa workflow	39
3.10	Pure control path handshake components and their respective STGs .	41
3.11	Data path control components and their respective STGs	43
3.12	Data-control interface components and their respective STGs	44
3.13	Breeze Handshake Circuit model of a GCD block	45
4.1	Overlay and sequence example (no common vertices)	51
4.2	Overlay and sequence example (common vertices)	51

4.3	PG specialisations: $H _x$ and $H _{\bar{x}}$	56
4.4	Simplifying expression (4.2) using the Closure axiom	61
4.5	The PG from Fig. 4.3 simplified using the Closure axiom, together with its specialisations	65
4.6	Multiple rail phase encoding	66
4.7	PGs related to matrix phase encoder specification	67
5.1	The optimality frontier	81
5.2	Architecture of an example processor	94
5.3	TPG specifications of instruction classes	94
5.4	Optimal 3-bit instruction opcodes and the corresponding TPG spe- cification of the microcontroller	96
5.5	Synthesised CPOGs	99
5.6	Comparison of different gate bases	100

List of Tables

3.1	Costs of individual components	46
3.2	Cost of optimally split full GCD circuit	47
4.1	Two instructions specified as partial orders	58
5.1	Synthesised instruction codes	98
5.2	Encoding of conditions with dynamic variable <i>ge</i>	101

List of Abbreviations

ALU - Arithmetic Logic Unit
CAD - Computer Aided Design
CNF - Conjunctive Normal Form
CSC - Complete State Coding
CSP - Communicating Sequential Processes
CPOG - Conditional Partial Order Graph
CPU - Central Processing Unit
DAG - Directed Acyclic Graph
EDA - Electronic Design Automation
GALS - Globally Asynchronous Locally Synchronous
HOL - Higher Order Logic
IC - Integrated Circuit
IFU - Instruction Fetch Unit
IP - Intellectual Property
IR - Instruction Register
ISA - Instruction Set Architecture
MAU - Memory Access Unit
NP - Nondeterministic Polynomial
PC - Program Counter
PCIU - Program Counter Increment Unit
PG - Parameterised Graph
PN - Petri Net
QBF - Quantified Boolean Formula
SAT - Boolean Satisfiability Problem
SoC - System on Chip
STG - Signal Transition Graph
TPG - Transitive Parameterised Graph
VLSI - Very Large-Scale Integration

Abstract

In this work we explore compositional methods for design of digital circuits with the aim of improving existing methodologies for design reuse. We address compositionality techniques looking from both structural and behavioural perspectives.

First we consider the existing method of handshake circuit optimisation via control path resynthesis using Petri nets, an approach using structural composition. In that approach labelled Petri net parallel composition plays an important role and we introduce an improvement to the parallel composition algorithm, reducing the number of redundant places in the resulting Petri net representations. The proposed algorithm applies to labelled Petri nets in general and can be applied outside of the handshake circuit optimisation use case.

Next we look at the conditional partial order graph (CPOG) formalism, an approach that allows for a convenient representation of systems consisting of multiple alternative system behaviours, a phenomenon we call behavioural composition. We generalise the notion of CPOG and identify an algebraic structure on a more general notion of parameterised graph. This allows us to do equivalence-preserving manipulation of graphs in symbolic form, which simplifies specification and reasoning about systems defined in this way, as displayed by two case studies.

As a third contribution we build upon the previous work of CPOG synthesis used to generate binary encoding of microcontroller instruction sets and design the corresponding instruction decoder logic. The proposed CPOG synthesis technique solves the optimisation problem for the general case, reducing it to Boolean satisfiability problem and uses existing SAT solving tools to obtain the result.

Acknowledgements

I am grateful to everyone who made the completion of this work a reality by sharing their knowledge, assisting directly and supporting me emotionally.

My supervisor, Alex Yakovlev, guided me throughout the research programme and supported me at all times in many ways. I am thankful to Andrey Mokhov, Victor Khomenko, Ivan Poliakov and my other colleagues for their contributions to the work being presented, their participation in discussions and criticism. Special thanks go to Alexei Iliasov and Danil Sokolov for providing immense help in reviewing and restructuring the thesis into its present form. Without them this would not have been possible.

Separate thanks go to the organisations supporting me financially. This work was supported by a studentship from Newcastle University EECE school, EPSRC grant EP/G037809/1 (VERDAD) and EPSRC grant EP/K001698/1 (UNCOVER).

Chapter 1

Introduction

One of the major challenges for the future of semiconductor industry is the problem of overcoming the design productivity gap. This gap is caused by the exponential complexity growth of electronic systems [46] while the capability of design tools cannot cope with this pace, see Fig. 1.1. The only way to deal with the increasing complexity of Integrated Circuits (ICs) is to improve the efficiency of the design process, in particular, by heavily reusing system components and by advancing the design automation methods.

It has been predicted by ITRS [2] that in order to address the productivity gap challenge, by 2020 at least 90% of the complex circuits should be built of previously designed components. This rises the need for compositional (or modular) design principles where the timing of individual modules is independent of the rest of the system and therefore requires delay insensitive communication between the modules. This communication discipline is natural for asynchronous circuits where the data transfer is accompanied by request-acknowledgement handshaking between the

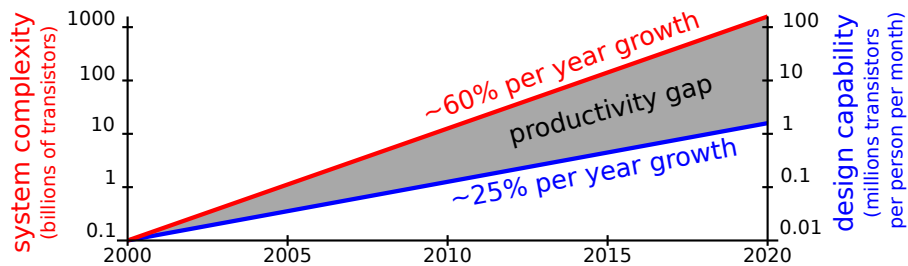


Figure 1.1: Design productivity gap

sending and the receiving counterparts. On this pathway the previously designed Intellectual Property (IP) cores will need to be adapted to the new modular architectures. The least intrusive is the Globally Asynchronous Locally Synchronous (GALS) approach [10] where special wrappers [24, 47] are built around synchronous modules to convert their communication into asynchronous handshake style. Another alternative is desynchronisation techniques [15] where the global clock is replaced by a distributed control which determines when the computation is complete and the output result is ready to be consumed. This control may take different forms, from a delay line matching the critical path of the module [16] to explicitly introduced completion detection logic [34].

The remaining 10% of the IC components, as well as the interface and control logic to support the interconnect and communication flexibility, will still need to be designed from scratch. One way is to design those components in traditional synchronous way and then apply the previously discussed techniques to comply with the delay insensitive interface requirements. This, however, may result in suboptimal solutions in terms of circuit area, computation speed and energy consumption. Better results can be achieved if the components are designed and implemented with their asynchronous environment in mind [36]. However, the logic synthesis of asynchronous circuits is computationally expensive and not applicable to large modules. This is due to high level of concurrency in truly asynchronous systems which results in a state space explosion. The computation complexity problem has been successfully addressed in the syntax-driven translation [22] approach which is based on direct mapping of a specification into hardware components without going through the state space exploration (it is assumed that there is one-to-one correspondence between the specification language constructs and the library of available components).

The major drawback of the circuits obtained by the syntax-driven translation is the suboptimal performance of their control structures [53]. In order to resolve this issue the control models of all the components need to be composed together and resynthesised exploiting the benefits of their joint optimisation. Existing resynthesis methods are based on parallel composition of component models expressed in form of Petri nets [25]. However, the efficient parallel composition of the component models is still an open question and is one of the primary goals of this thesis.

The composition of circuit components is of structural nature - they are combined

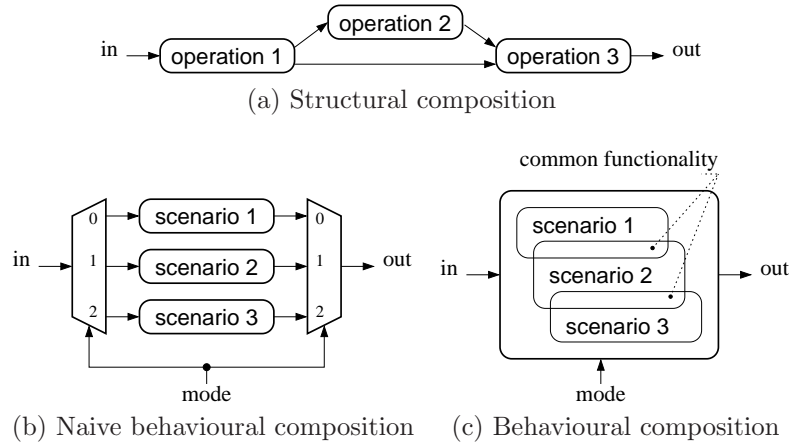


Figure 1.2: Composition basics

via input-output interfaces according to the casual dependency between the operation they perform, as shown in Fig. 1.2a. Another compositional aspect is a combination of several mutually exclusive behaviours in the same circuit. A naive way to build such a circuit is to implement the different behaviour scenarios in separate modules and structurally compose them with the use of multiplexers and demultiplexers, as shown in Fig. 1.2b. A mode selection code on the (de)multiplexors determines the current scenario. While this is a valid implementation of multi-modal functionality, it ignores the mutually exclusive feature of the implemented behaviours and ignores a possibility of partial hardware reuse for common functionality, as shown in Fig. 1.2c.

A model which naturally captures the structural and behavioural aspects of composition in a single formalism is Conditional Partial Order Graphs (CPOGs) [39]. This graph-based model is capable of expressing the structural composition by means of causality arcs (similar to Petri nets) and the behavioural composition by means of Boolean "visibility" conditions on its vertices. While CPOGs is a convenient tool for reasoning on small benchmarks, it lacks the means for capturing and transformation of large systems. The first goal of this thesis is to generalise the CPOGs model and transition from the acyclic graphs representing partial orders into a universal Parameterised Graphs (PGs). The second goal is to introduce a theory for PG manipulation in algebraic form, which enables equivalence-preserving manipulation of graphs in symbolic form and simplifies specification and reasoning about complex systems.

The Boolean conditions on graph vertices can be expressed in various forms targeting different optimisation criteria. In the context of digital circuit design these conditions are subsequently implemented as the hardware control logic, therefore such optimisation targets as minimising the number of control variables and/or reducing the complexity of logical expressions, is of paramount importance. The ambitious goal of this thesis is to solve the optimisation problem for the general case, to express it in terms of Boolean satisfiability problem and to employ the existing SAT solving tools for obtaining the best result.

1.1 Contributions

The main contributions of the thesis are as follows:

- **Improved parallel composition:** a novel method for composition of models specified with labelled Petri Nets.
- **PG theory:** CPOG generalisation to Parametrised Graph formalism and mechanised proof of its algebraic properties.
- **PG Synthesis:** a technique for synthesis of processor instruction decoder using instruction sets specified with Parametrised Graphs.
- **CAD tool support:** automation for the above, including improved parallel composition and encoding of TPG specifications using Workcraft framework.

1.2 Overview

Handshake circuits [7] are widely applied in the design and synthesis of real-life hardware. One prominent problem is obtaining an efficient implementation from a *structural* compositional specification. Syntax-based synthesis tools such as Balsa [22] are unable to take into account the compositional behaviour of STGs corresponding to handshake circuit components. To address this issue we propose a technique that selectively composes STGs of related components to obtain a smaller and more performant circuit without suffering state space explosion commonly associated with

Petri net based techniques [60]. This transformation, which we refer to as *resynthesis* [4, 11, 33, 52], is accomplished in three stages. First, we apply a heuristic to identify the most promising candidates for STG-level composition. Second, we perform a parallel composition of the selected component STGs and as a result obtain a new handshake circuit with custom components, functionally equivalent to a combination of elementary components. Finally, a gate-level implementation is obtained from the new handshake circuit via a component-wise synthesis of STGs.

Unfortunately, the standard definition of parallel composition almost always yields a ‘messy’ Petri net, with many implicit places, causing performance deterioration in techniques that are based on structural methods such as the resynthesis approach. To counter this, we propose an improved algorithm for computing the parallel composition. The algorithm generally produces nets with fewer implicit places that are better suited for subsequent application of structural methods [3].

In addition to purely structural composition of STGs, it is also beneficial to consider a mixture of structural and behavioural composition. Conditional Partial Order Graphs (CPOG) [39] is a graph-based notation supporting compact representation and efficient manipulation of both structural and behavioural composition styles. As one example, when developing complex circuit, it is often necessary to consider several operational modes of a circuit [44, 62]. For this, one needs methodologies and tools to exploit similarities between the individual modes and hence lift the level of discourse to behaviour families. This necessitates that behaviours are managed in a compositional way: the specification of the system must be composed from specifications of its blocks. Furthermore, since the approach is intended to be a part of a safety critical toolchain, it is essential that such a specification is amenable to mechanised reasoning and transformation.

In Chapter 4 we propose an extension of the CPOG formalism, called Parameterised Graph (PG). PGs deal with general graphs rather than just partial orders. We introduce an algebra of Parameterised Graphs by specifying the equivalence relation via a set of axioms, which we prove to be sound, minimal and complete [43]. This result allows one to manipulate a PG model as an algebraic expression applying the bi-directional rewrite rules of this algebra. This is in contrast to the CPOG formalism that does not offer a unifying algebraic structure. We demonstrate the usefulness of the developed formalism with two case studies coming from the area of

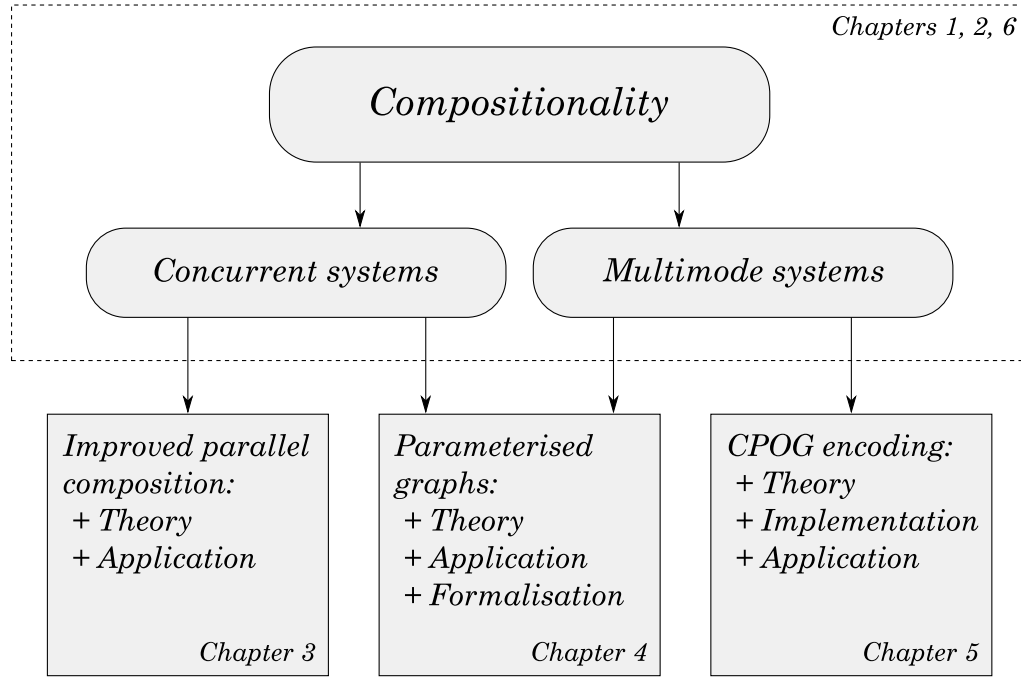


Figure 1.3: Thesis structure

microelectronics design.

The CPOG formalism can be applied to merge several distinct behaviours into a single compact CPOG [39]. As one example, this has been previously used to synthesise control logic for instruction decoding. In this thesis (Chapter 5) we improve upon this work by offering a powerful technique to automatically discover an optimal encoding and synthesise a matching optimal decoding circuit. From the outset, we consider a larger set of potential solutions which enables us to formulate the global optimality criterion. We use an automated satisfiability solving techniques to find an optimal solution [40].

To summarize the thesis structure,

- **Chapter 2** covers the basics of handshake circuits, signal transition graphs and conditional partial order graphs.
- **Chapter 3** describes the proposed improved parallel composition algorithm. The contents of this chapter is based on the results published previously in [3].
- **Chapter 4** introduces Parametrised Graph (PG) theory, defining and study-

ing an algebraic structure that generalises Conditional Partial Order Graph formalism. This chapter is based on the results previously published in [43].

- **Chapter 5** describes a technique for optimal encoding of processor instruction sets defined using PG formalism. This chapter is based on the results previously published in [40]. An earlier version of this paper has qualified for a Best Paper Award at the ACSD conference.
- **Chapter 6** summarises the achieved results and proposes ideas for future research.
- **Appendix** contains formal proofs in form of Agda source code for the PG Algebra properties discussed in Chapter 4.

The relationship between chapters is illustrated in Figure 1.3.

Chapter 2

Background

This chapter introduces a brief overview of the major techniques and models used throughout the thesis. In particular, handshake circuits – a specification formalism for synthesis of self-timed hardware; Petri nets – a graph-based notation for reasoning about concurrent behaviour; conditional partial order graphs (CPOG) – a versatile notation for describing a family of partial orders.

2.1 Handshake circuits

One of the approaches to design of asynchronous circuits is syntax-directed mapping with handshake circuits as an intermediate format. The parse tree of a program source code written in a CSP-style [26] language can be interpreted as a graph of components, connected with communication links called handshake channels. The components can then be individually mapped to gate-level implementations with complete circuit derived by implementing the handshake channels with wires.

This approach has been first used by Philips in their Tangram [29] design tool and later made publicly available when the similar free Balsa [22] system has been released.

This thesis will be working with Balsa handshake components.

A handshake activation h is said to *enclose* a process p if p can only start after h gets a request and h can get an acknowledgement only after p gets finished.

A *handshake circuit* consists of handshake components which interact by request/acknowledgment handshaking over communication channels. Each *handshake com-*

ponent is specified by a set of ports and a process communicating over those ports. A *protocol* is assigned to each port, which specifies whether the process initiates the handshakes over an *active port* or awaits for the other party over a *passive port*. It also specifies the direction and size of data transferred during the handshakes. Each *channel* connects two ports of the same data size with one port being active and the other being passive. Active input ports and passive output ports are called *pull* ports while the pasive input and active output ports are called *push* ports.

On diagrams used in this thesis we display handshake components with large circles with a process symbol inside and handshake ports with small circles where filled circle stands for active port and hollow circle stands for passive port. Channels are displayed as lines between the corresponding ports with the direction of the arrow corresponding to the direction of data flow.

The defining feature of a handshake component is the process associated with it. In Balsa there are about fifty types of processes with each having its own behaviour.

An important notion used to describe behaviours is channel *activation*. Activation is a process starting with a request being sent from the active port to the passive port and ending with an acknowledgement being sent back. The behaviour of a channel can be described as activation repeated indefinitely. For data channels activation additionally determines the period of time the values on the data wires remain valid.

Processes, including activations, are subject to a notion of *enclosure* to describe temporal relationship between them. It is said that a process p is enclosed into a process q when the beginning of p comes after the beginning of q while the end of p comes before the end end of q .

- *Sequence* (Fig. 2.1a) is a component with three control ports: a passive port s and two active ports t_1 and t_2 . The behaviour of the component is as follows: each activation on s encloses the process consisting of sequential activation on t_1 followed by t_2 .
- *Concur* (Fig. 2.1b) is a component with a similar external interface: it has a passive port s and two active ports t_1 and t_2 . The behaviour is different though: each activation on s for this component encloses activation of t_1 and t_2 concurrently.

- *Sync* (Fig. 2.1c) is a component with three control ports: two passive ports s_1 and s_2 and an active port t . This component ensures enclosure of t into both s_1 and s_2 by the means of synchronisation between s_1 and s_2 . This component is dual to *Concur* in the sense that they form a no-op when their corresponding ports are connected.
- *Call* (Fig. 2.1d) is another component with three control ports: two passive ports s_1 and s_2 and an active port t . It differs from *Sync* in that instead of expecting concurrent activation of both s_1 and s_2 it only expects one of them to be activated at a time and encloses t into the one which happens.
- *BinaryFunc* (Fig. 2.1e) is a component with three pull data ports: a passive port o and two active ports i_1 and i_2 . It is parametrised on width of those ports and on a function it computes. Control-wise it is similar to *Concur*: it encloses the concurrent activation of i_1 and i_2 into the activation on o .
- *CallMux* (Fig. 2.1f) is a component similar to *Call* with the difference that all of its ports are further extended to w -bit pull ports. Its behaviour is identical from the control point of view, with the additional data being received through t and sent through the activated output port.
- *Variable* (Fig. 2.1g) is a component with a single input data port w , called the write port and a set of passive output data ports r , called read ports. The component is parameterised on the bit width of the variable w , coinciding with the width of the data ports. It is also parameterised on the number of read ports n . The behaviour of *Variable* is to remember the data written with the latest activation of w and to output that data on any activation of a read port r_i . The behaviour is trivial from the control point of view: the only way the ports interact is through the data. However, there is a requirement that the write port activation does not overlap with any of the read port activations.
- *While* (Fig. 2.1h) is a component with a passive control port s , an active control port t and an active one-bit input data port c . Its behaviour is to enclose into the activation on s the following process: complete a handshake over c to obtain the one-bit data indicating the process activity condition; if this condition holds then activate t and repeat the operations.

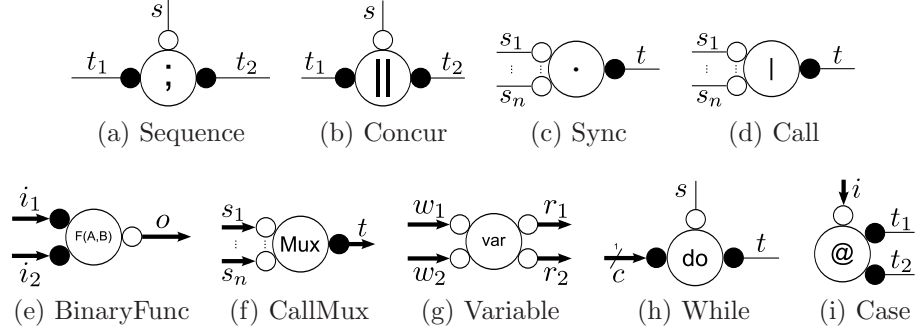


Figure 2.1: Handshake components

- *Case* (Fig. 2.1i) is a component with a passive input data port i and a set of active control ports t . It is parameterised by a number of ports n , the size of the data transferred by i and by a function f mapping $2^{\text{cl}} \rightarrow n$. The behaviour is to enclose into activation on i with the input code word d an activation of the port $t_{f(d)}$.

2.2 Petri nets

A *Petri net* is a 4-tuple $N = (P, T, W, M_N)$ where P is a finite set of *places* and T is a finite set of *transitions* with $P \cap T = \emptyset$, $W : P \times T \cup T \times P \rightarrow \mathbb{N}_0$ is the *weight function*, and M_N is the *initial marking*, where a *marking* is a multiset of places, i.e. a function $P \rightarrow \mathbb{N}_0$ which assigns a number of *tokens* to each place. A Petri net can be considered as a bipartite graph with weighted arcs between places and transitions. If necessary, we write P_N etc. for the components of N or P' (P_i) etc. for the net N' (N_i) etc.

The *preset* of a place or transition x is denoted as $\bullet x$ and defined by $\bullet x \stackrel{\text{df}}{=} \{y \in P \cup T \mid W(y, x) > 0\}$, the *postset* of x is denoted as x^\bullet and defined by $x^\bullet \stackrel{\text{df}}{=} \{y \in P \cup T \mid W(x, y) > 0\}$. These notions are extended to sets as usual. We say that there is an *arc* from each $y \in \bullet x$ to x .

A transition t is *enabled under a marking* M if $\forall p \in \bullet t : M(p) \geq W(p, t)$, which is denoted by $M[t]$. An enabled transition t can *fire* yielding a new marking M' , written as $M[t]M'$, where $M'(p) = M(p) - W(p, t) + W(t, p)$, for all $p \in P$. A transition sequence $\sigma = t_1 \dots t_n$ is *enabled under a marking* M (yielding M') if

$M[t_1]M_1[t_2] \dots M_{n-1}[t_n]M_n = M'$, and we write $M[\sigma]$, $M[\sigma]M'$ resp.; σ is called *execution of N* if $M_N[\sigma]$. The empty transition sequence λ is enabled under every marking. M is called *reachable* if a transition sequence σ with $M_N[\sigma]M$ exists.

N is called *bounded* if, for every reachable marking M and every place p , $M(p) \leq k$ for some constant $k \in \mathbb{N}$; if $k = 1$, N is called *safe*. N is bounded if and only if the set $[M_N]$ of reachable markings is finite. In this thesis, we are mostly concerned with bounded Petri nets.

A place p is *implicit* if it can be deleted from the net without changing the set of executions, and so an implicit place can be removed from the net without affecting its behaviour.¹ Unfortunately, detecting implicit places is expensive: the problem is PSPACE-complete for safe and EXSPACE-complete for general Petri nets. A place p is *duplicate* if there is another place p' with the same pre- and postsets whose initial marking does not exceed that of p . Duplicate places are implicit, and are cheap to detect.

An *STG* is a tuple $N = (P, T, W, M_N, In, Out, \ell)$ where (P, T, W, M_N) is a Petri net and In and Out are disjoint sets of *input* and *output signals*. For $Sig = In \cup Out$ being the set of all signals, $\ell : T \rightarrow Sig \times \{+, -\} \cup \{\lambda\}$ is the *labelling* function. $Sig \times \{+, -\}$ or short Sig^\pm is the set of *signal transitions*; its elements are denoted as s^+ , s^- resp. instead of $(s, +)$, $(s, -)$ resp. A plus sign denotes that a signal value changes from *logical low* (written as 0) to *logical high* (written as 1), and a minus sign denotes the opposite direction. We write s^\pm if it is not important or unknown which direction takes place.

An STG can contain transitions labelled with λ , called *dummy* transitions, which do not correspond to any signal change. *Hiding a signal s* means to change the label of all transitions labelled with s^\pm to λ . (The idea of re-synthesis approach is to hide the signals used for communication between components, which results in an STG with fewer signals that often has a simpler implementation as a circuit.) The labelling of an STG is called *injective* if for each pair of distinct non-dummy transitions t and t' , $\ell(t) \neq \ell(t')$.

Examples of STGs are shown in Figs. 3.1 and 3.2. Places are drawn as circles containing a number of tokens corresponding to the initial marking. Unmarked places which have only one transition in their presets and postsets are not drawn if

¹Note that an implicit place can cease to be implicit if another implicit place is removed first.

the corresponding arcs have the weight 1; they are implicitly given by an arc between these two transitions (and if such a place contains tokens, they are drawn on the arc itself). Transitions are drawn simply as their labels, and the weight function is drawn as directed arcs (x, y) whenever $W(x, y) \neq 0$ (and labelled with $W(x, y)$ if $W(x, y) > 1$).

We lift the notion of enabledness to transition labels: we write $M[\ell(t)]\rangle M'$ if $M[t]\rangle M'$. This is extended to sequences as usual – deleting λ -labels automatically since λ is the empty word; i.e. $M[s^\pm]\rangle M'$ means that a sequence of transitions fires, where one of them is labelled s^\pm while the others (if any) are λ -labelled. A sequence $\nu \in (Sig^\pm)^*$ is called a *trace of a marking* M if $M[\nu]\rangle$, and a *trace of* N if $M = M_N$. The *language* $L(N)$ of N is the set of all traces of N .

The *reachability graph* $RG(N)$ of an STG N is an arc-labelled directed graph on the reachable markings of N with M_N as the root; there is an arc from M to M' labelled $\ell(t)$ whenever $M[t]\rangle M'$. For bounded Petri nets and STGs, $RG(N)$ can be seen as a finite automaton (where all states are accepting), and $L(N)$ is the language of this automaton. Observe that automata with accepting states only can be regarded as STGs (with the states as places, the initial state being the only marked place, etc.); hence, all definitions for STGs also apply to automata.

N is *deterministic* if $RG(N)$ is a deterministic automaton: it contains no λ -labelled transitions and there are no *dynamic auto-conflicts*, i.e. for each reachable marking M and each signal transition s^\pm there is at most one M' with $M[s^\pm]\rangle M'$. (Note that a deterministic STG can have choices between different outputs, e.g. an STG modelling the standard arbiter is deterministic).

An STG with a set of all markings S_2 is said to *simulate* another STG with a set of all markings S_1 iff there exist an $R \subseteq S_1 \times S_2$ such that $(M_{N_1}, M_{N_2}) \in R$ and for any pair of markings $(M_1, M_2) \in R$, a label l and a marking M'_1 , $M_1[l]\rangle M'_1$ implies $M_2[l]\rangle M'_2$ for some M'_2 . We call R the witness of simulation. Now we can say that STGs N_1 and N_2 are *bisimilar* iff N_2 simulates N_1 with witness R and N_1 simulates N_2 with witness R^{-1} .

For deterministic STGs, language equivalence and bisimulation coincide, and the language can be taken as the semantics of such a specification. Unfortunately, the class of deterministic STGs is too restrictive in practice [31], e.g.:

- using dummy transitions is often convenient in manual design;
- modelling OR-causality [63] as a safe STG requires non-determinism;
- hiding internal communication (and thus introducing dummy transitions) is a crucial step in re-synthesis.

Hence, one has to deal with non-deterministic STGs as well.

One might think that if $\text{RG}(N)$ is non-deterministic, it can be *determinised* (using well-known automata-theoretic methods), i.e. turned into a language-equivalent deterministic automaton with accepting states only; in particular, the resulting automaton will have no λ -arcs. Unfortunately, this is a bad idea, as shown in [31], where the semantics of non-deterministic STGs was developed. It is based on the concept of *output-determinacy*, which is a relaxation of determinism: An STG N is *output-determinate (OD)* if $M_N[\nu]\rangle M_1$ and $M_N[\nu]\rangle M_2$ implies for every $x \in \text{Out}_N$ that $M_1[x^\pm]\rangle$ iff $M_2[x^\pm]\rangle$. It turns out that OD STGs are exactly the STGs which have correct implementations according to the implementation relation introduced in [31]. Hence, non-OD STGs are ill-formed, and in particular cannot be correctly implemented as circuits. This shows that in general, *the language is not a satisfactory semantics of non-deterministic STGs*; in particular, *synthesising the determinised reachability graph of a non-OD STG will either fail or result in an incorrect circuit*. On the other hand, for the class of OD STGs [31] shows that their language is an adequate semantics, and implementation relation can be formulated purely in terms of the language. An important property of OD STGs is that in them the enabledness of an output signal is a function of the trace, i.e. given a trace ν , the set of outputs by which ν can be extended is uniquely determined, even though there could be multiple executions corresponding to ν .

In the following definition of *parallel composition* \parallel , see e.g. [61], we will have to consider the distinction between input and output signals. The idea of parallel composition is that the composed systems run in parallel and synchronise on common actions – corresponding to circuits that are connected on the wires corresponding to the signals. Since a system controls its outputs, we cannot allow a signal to be an output of more than one component; input signals, on the other hand, can be shared. An output signal of a component may be an input of other components, and in any case it is an output of the composition.

The parallel composition of STGs N_1 and N_2 is defined if $Out_1 \cap Out_2 = \emptyset$. If we drop this requirement, the definition gives the *synchronous product* $N_1 \times N_2$, which is often useful. The place set of the composition is the disjoint union of the place sets of the components; therefore, we can consider markings of the composition (regarded as multisets) as the disjoint union of markings of the components, and we will also write such a marking $M_1 \dot{\cup} M_2$ of the composition as (M_1, M_2) . To define the transitions, let $A = (In_1 \cup Out_1) \cap (In_2 \cup Out_2)$ be the set of common signals. If e.g. s is an output of N_1 and an input of N_2 , then firing of s^\pm in N_1 is ‘seen’ by N_2 , i.e. it must be accompanied by firing of s^\pm in N_2 . Since we do not know a priori which s^\pm -labelled transition of N_2 will fire together with some s^\pm -labelled transition of N_1 , we have to allow for each possible pairing. Thus, the *parallel composition* $N = N_1 \parallel N_2$ is obtained from the disjoint union of N_1 and N_2 by fusing each s^\pm -labelled transition t_1 of N_1 with each s^\pm -labelled transition t_2 from N_2 if $s \in A$. Such transitions are pairs and the firing $(M_1, M_2)[(t_1, t_2)](M'_1, M'_2)$ of N corresponds to the firings $M_i[t_i]M'_i$ in N_i , $i = 1, 2$; for an example of a parallel composition, see Fig. 2.2. More generally, we have $(M_1, M_2)[\nu](M'_1, M'_2)$ iff $M_i[\nu|_{N_i}](M'_i)$ for $i \in \{1, 2\}$, where $\nu|_{N_i}$ denotes the projection of the trace ν onto the signals of the STG N_i . Hence, all reachable markings of N have the form (M_1, M_2) , where M_i is a reachable marking of N_i , $i = 1, 2$.

Obviously, one can extend the notion of the parallel composition to a finite family (or collection) $(C_i)_{i \in I}$ of STGs as $\parallel_{i \in I} C_i$, provided that no signal is an output signal of more than one of the C_i . We will also denote the markings of such a composition by (M_1, \dots, M_n) if M_i is a marking of C_i for $i \in I = \{1, \dots, n\}$. As above, $(M_1, M_2, \dots, M_n)[\nu](M'_1, M'_2, \dots, M'_n)$ iff $M_i[\nu|_{C_i}](M'_i)$ for all $i \in \{1, \dots, n\}$. It is easy to see that C is deterministic if all C_i are. However, this is not true for a composition of OD STGs, as the result, in general, can be non-OD in such a case.

A composition can also be ill-defined due to *computation interference*, see e.g. [20]. Let $C \stackrel{\text{df}}{=} \parallel_{i \in I} C_i$ be a composition of STGs. It is *free from computation interference (FCI)* if for every trace ν of C the following holds: if $\nu|_{C_j} x^\pm$ is a trace of C_j for some output x of C_j , then $\nu|_C x^\pm$ is a trace of C .

Transition contraction [61] is an important operation in circuit re-synthesis. It removes a dummy transition from an STG and combines each place of its preset with each place of its postset to ‘simulate’ the firing of the deleted transition, see

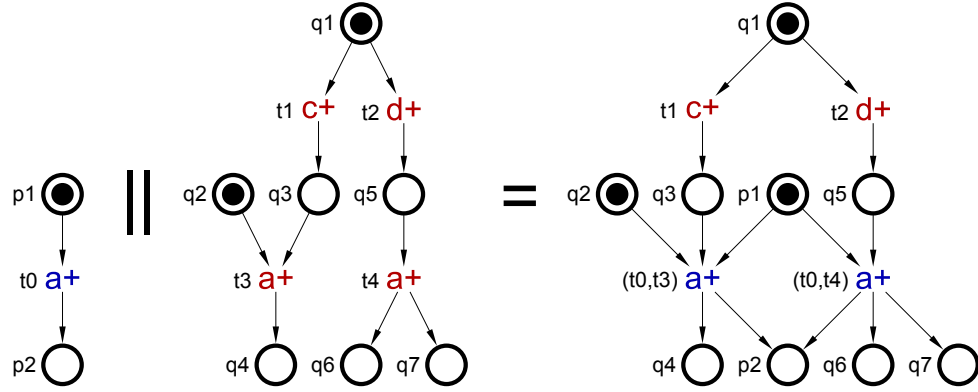


Figure 2.2: Parallel composition example. In the net fragment on the left hand side, signal a is an output, and in the fragment in the middle it is an input. Hence, in their parallel composition (right) it is an output. In this example, there is *computation interference*: the left component activates a^+ but the middle one is not ready to receive it.

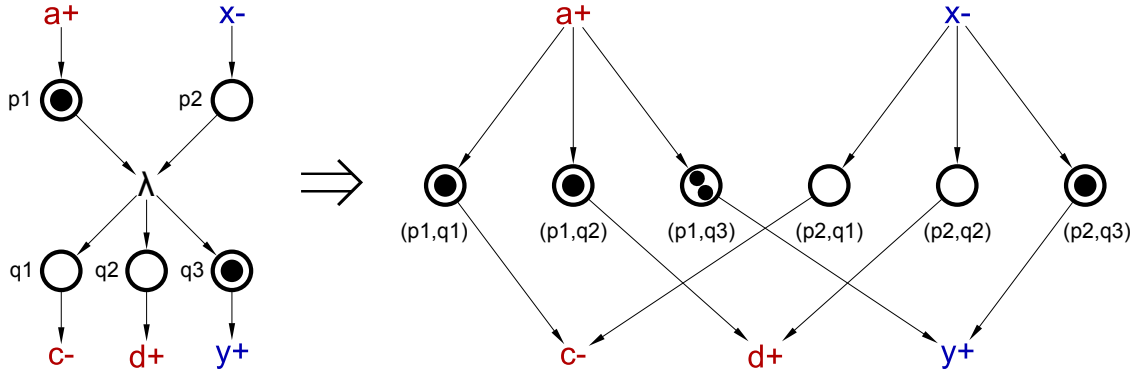


Figure 2.3: An example of a transition contraction.

Fig. 2.3. Unfortunately, transition contractions are sometimes undefined (e.g. in case the transition has a self-loop, i.e. some place occurs in both its preset and postset); moreover, even when a contraction is defined, it might change the semantics of the STG. Hence, [61] uses the notion of *secure* contractions, that preserve the semantics.

Transition contractions preserve boundedness, but in general, can turn a safe net into a non-safe one, as well as introduce weighted arcs. In practice, it is often convenient to work with safe nets, and for this [32] introduced *safeness-preserving* contractions, i.e. ones which guarantee that the transformed STG is safe if the initial one was. (Note that the transitions with weighted arcs must be dead in a safe Petri net, and so we can assume that the initial and all the intermediate STGs contain no

such arcs.) Also, [32] developed a sufficient structural condition for a contraction to be safeness-preserving.

From the point of view of this thesis, it is important to remark that implicit places can adversely affect the (secure) contractibility of a transition, i.e. it is possible to have a situation when a transition is not contractible (or not securely contractible), but becomes securely contractible after some implicit place is removed from the STG. As detecting implicit places is expensive, it is very desirable to reduce their number by some other means, in particular the approach proposed in this thesis reduces the number of such places in STGs obtained by parallel composition. This has a direct effect on re-synthesis: if the composed STG has fewer implicit places, more dummy transitions in it can be contracted, and so it will be easier to synthesise the result.

2.3 Conditional Partial Order Graphs

A *Conditional Partial Order Graph* (CPOG) [39][45] is a quintuple $H = (V, E, X, \rho, \phi)$, where V is a finite set of *vertices*, $E \subseteq V \times V$ is a set of *arcs* between them, and X is a finite set of *operational variables*. An *opcode* is an assignment $(x_1, x_2, \dots, x_{|X|}) \in \{0, 1\}^{|X|}$ of these variables; X can be assigned only those opcodes which satisfy the *restriction function* ρ of the graph, i.e. $\rho(x_1, x_2, \dots, x_{|X|}) = 1$. Function ϕ assigns a Boolean *condition* $\phi(z)$ to every vertex and arc $z \in V \cup E$ of the graph.

Figure 2.4(a) shows an example of a CPOG containing $|V| = 5$ vertices and $|E| = 7$ arcs. There is a single operational variable x ; the restriction function is $\rho(x) = 1$, hence both opcodes $x = 0$ and $x = 1$ are allowed. 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 $\phi(e) = \bar{x}$ respectively. Arcs also fall into two classes: unconditional (arc $c \rightarrow d$) and conditional (all the rest). As CPOGs tend to have many unconditional vertices and arcs we use a simplified notation in which conditions equal to 1 are not depicted in the graph. This is demonstrated in Figure 2.4(b).

The purpose of conditions ϕ is to ‘switch off’ some vertices and/or arcs in the graph according to the given opcode. This makes CPOGs capable of specifying multiple partial orders or instructions (a partial order is a form of behavioural description of an instruction). Figure 2.4(c) shows a graph and its two *projections*. The

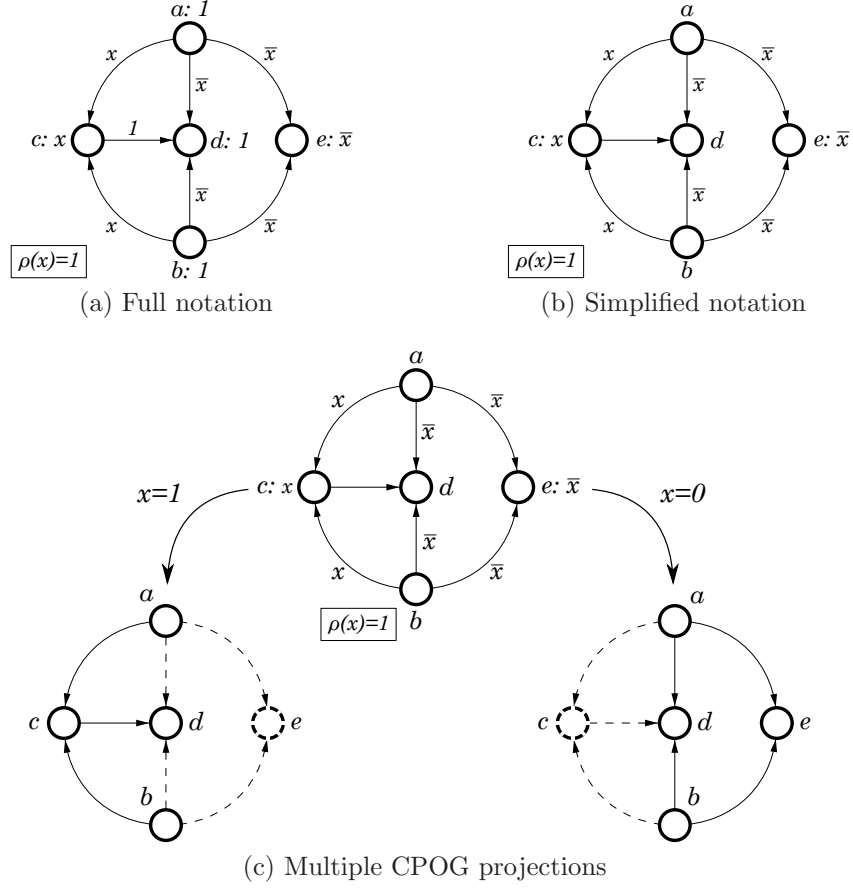


Figure 2.4: Graphical representation of CPOGs and their projections

leftmost projection is obtained by keeping in the graph only those vertices and arcs whose conditions evaluate to 1 after substitution of the operational variable x with 1. Hence, vertex e disappears, because its condition evaluates to 0: $\phi(e) = \bar{x} = \bar{1} = 0$. Arcs $\{a \rightarrow d, a \rightarrow e, b \rightarrow d, b \rightarrow e\}$ disappear for the same reason. 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 $c \rightarrow d$ 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. Each of the obtained projections can be treated as a specification of 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.

A CPOG is *well-defined* if all its projections allowed by ρ are acyclic. We consider only well-defined CPOGs in this thesis, because a cyclic projection has no natural execution semantics, in particular it is not clear which event can be executed first unless some form of a ‘token’ is introduced as in the Petri Net model [14].

To summarise, a CPOG is a structure to represent a set of encoded partial orders in a compact form. Synthesis and optimisation methods presented in [45] provide a way to obtain such a representation given a set of partial orders and their opcodes. For example, the CPOG in Figure 2.4(c) can be synthesised automatically from the two partial orders below it and the corresponding opcodes $x = 1$ and $x = 0$. The next section shows that a particular assignment of opcodes to the partial orders has a strong impact on the final CPOG, therefore in order to obtain the most compact CPOG representation one has to search for the best opcode assignment.

Note that partial orders is not the only formalism for formal specification of instructions. In particular, there is an alternative approach [6] based on automata, which treats every instruction as a burst-mode state machine and defines an operation of composition on them. While benefiting from a direct correspondence between flowcharts of algorithms and automata, the approach cannot model true concurrency: a set of causally independent events can only be executed as a ‘burst’ in the same step/clock cycle. Also, it requires explicit memory to track the current state of the automaton. We believe that partial orders are better suited for modelling instruction sets of processing units built on heterogeneous platforms, i.e. exhibiting both asynchronous and synchronous interactions [41].

2.4 Agda

Agda [49] is a system serving both as a programming language and as a proof assistant simultaneously. When viewed as a programming language, it is a purely functional language with its syntax largely inspired by Haskell. Its main distinguishing features are totality (the fact that every function is defined on every possible input) and dependent typing system that allows for types to depend on the values.

2.4.1 Function definitions and algebraic data types

A very common language construct in Agda is a function definition. Function definition must consist of type signature followed by its defining equations. As a simple example, consider this Boolean exclusive OR function:

$$\begin{aligned} xor & : Bool \rightarrow Bool \rightarrow Bool \\ xor\ x\ y & = x \wedge (\neg y) \vee y \wedge (\neg x) \end{aligned}$$

Here we define a function called *xor* of type $Bool \rightarrow Bool \rightarrow Bool$ with two arguments *x* and *y* defined in terms of $_ \wedge _$, $_ \vee _$ and $\neg _$.

Functions can have more than one defining equation when each equation defines the function for a particular shape, or *pattern* of arguments. This is called *pattern matching*. For example, the Boolean negation function can be defined the following way:

$$\begin{aligned} \neg _ & : Bool \rightarrow Bool \\ \neg\ false & = true \\ \neg\ true & = false \end{aligned}$$

A more interesting example would be a Boolean AND:

$$\begin{aligned} _ \wedge _ & : Bool \rightarrow Bool \rightarrow Bool \\ true \wedge true & = true \\ _ \wedge _ & = false \end{aligned}$$

This example demonstrates an additional feature of pattern matching: equations are ordered and the earlier ones take precedence.

In the above we used the *Bool* data type to represent Boolean values. This data type is not a built-in language construct of Agda, but can be defined using the **data** keyword:

$$\begin{aligned} \mathbf{data}\ Bool & : Set\ \mathbf{where} \\ true & : Bool \\ false & : Bool \end{aligned}$$

Here definition gives the name for the type and lists *constructors* of its values: *true* and *false*. In this case constructors have no arguments, thus corresponding

to individual values, but in general a single constructor can correspond to multiple values, as will be shown later.

2.4.2 Inductive types and recursion

We often want to reason about data types with infinite number of values, such as natural numbers. To represent them we use inductive type definitions:

data \mathbb{N} : *Set* **where**

zero : \mathbb{N}

suc : $\mathbb{N} \rightarrow \mathbb{N}$

Here we define natural numbers as something that has two forms: it is either a *zero*, or a successor *suc* x where x is another natural number. These two constructors allow us to construct an arbitrary number of values of type \mathbb{N} by successive application of *suc* to *zero*: *zero* corresponds to 0, *suc zero* corresponds to 1, *suc (suc zero)* to 2, etc.

To manipulate the values of inductive data types we use recursive functions:

$_ + _$: $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

zero + y = y

suc x + y = *suc* (x + y)

Here we define natural number addition recursively by considering the cases for the first argument: the base case of $0 + y$ must evaluate to y and for the recursive case $(1 + x) + y$ must evaluate to $1 + (x + y)$. The Agda compiler checks that the arguments to the function become smaller on recursive calls, thus ensuring that only well-behaved (terminating) definitions are admitted.

2.4.3 Indexed types and propositions

There is a close correspondence between types and logic, a phenomenon known as Curry-Howard correspondence. Specifically, types can be thought of as propositions where the type is inhabited if and only if the corresponding proposition holds true. Similarly, well-typed terms can be thought of as proofs of the corresponding propositions. Consequently, the type-checker can be used as a proof checker.

With the language features described so far we can construct types \top and \perp corresponding to true (logical tautology) and false (logical contradiction). These are not to be confused with *true* and *false* values of type *Bool* that can not be used as types.

```
data  $\top$  : Set where
  tt :  $\top$ 
data  $\perp$  : Set where
```

Here the type \top has a constructor *tt*, which makes it inhabited, thus corresponding to the logic value of true. The type \perp has no constructors, which makes it uninhabited, thus corresponding to false.

To construct more complex propositions Agda provides parameterised types, dependent function types and indexed inductive data families.

Type parameters are a basic way to allow for generic data types or logic operators. Consider the following example:

```
data Both (A : Set) (B : Set) : Set where
  both : A  $\rightarrow$  B  $\rightarrow$  Both A B
data Either (A : Set) (B : Set) : Set where
  left : A  $\rightarrow$  Either A B
  right : B  $\rightarrow$  Either A B
```

Here, *Both* *A* *B* can be thought of as a type of tuples of the form *both* *x* *y* with *x* : *A* and *y* : *B*. At the same time, for propositions *P* and *Q*, *Both* *P* *Q* can be thought of as their conjunction so that the proof *both* *p* *q* can be constructed if and only if both *p* and *q* (proofs of *P* and *Q*) can be constructed.

Similarly, *Either* is playing a dual role of taking the disjoint union of its type parameters and the logical disjunction operator. Type *Either* *P* *Q* is inhabited if and only if types *P* or *Q* or both are inhabited.

Another way to define parameterised types is to compute them as a result of a function.

Consider some examples:

```
Not : Set  $\rightarrow$  Set
Not P = P  $\rightarrow$   $\perp$ 
```

$$\begin{aligned} \text{IsTrue} &: \text{Bool} \rightarrow \text{Set} \\ \text{IsTrue true} &= \top \\ \text{IsTrue false} &= \perp \end{aligned}$$

Not takes a type P and computes another type $\text{Not } P$, which is inhabited if and only if contradiction is derivable from P . It is useful to think of $\text{Not } P$ as being inhabited if and only if P is not.

$\text{IsTrue } x$ is a type that is inhabited if and only if a Boolean value x happens to equal *true*.

Dependent function types have the form $(x : X) \rightarrow Y$ where x can be free in Y . This lets the type of the function result to depend on the value passed in as the argument. In the case when Y is a logical proposition, the dependent type can be thought of as universal quantification over x . Indeed, let us construct a value that can serve as a proof that for any boolean value x one of x and $\neg x$ must be *true*:

$$\begin{aligned} \text{lemma}_1 &: (x : \text{Bool}) \rightarrow \text{Either } (\text{IsTrue } x) (\text{IsTrue } (\neg x)) \\ \text{lemma}_1 \text{ true} &= \text{left } tt \\ \text{lemma}_1 \text{ false} &= \text{right } tt \end{aligned}$$

Finally, indexed inductive type families give you more flexibility by having the constructor choose the values for type parameter instead of having to construct a type for a given parameter value:

$$\begin{aligned} \mathbf{data} \text{ IsEven} &: \mathbb{N} \rightarrow \text{Set} \mathbf{where} \\ \text{zero} &: \text{IsEven } 0 \\ \text{suc} &: (n : \mathbb{N}) \rightarrow \text{IsEven } n \rightarrow \text{IsEven } (\text{suc } n) \end{aligned}$$

Here $\text{IsEven } n$ is inhabited if and only if n is an even number.

With dependent functions it is often useful to omit some of the arguments because their values are uniquely determined by the types of the arguments that follow. To be able to do that you can mark the corresponding parameter as implicit by putting it in curly braces:

$$\begin{aligned} \text{lemma}_2 &: \{x : \mathbb{N}\} \rightarrow \text{IsEven } x \rightarrow \text{Not } (\text{IsEven } (\text{suc } x)) \\ \text{lemma}_2 \text{ zero } () & \\ \text{lemma}_2 (\text{suc } e) (\text{suc } z) &= \text{lemma}_2 e z \end{aligned}$$

```

3 not even : Not (IsEven (suc (suc (suc zero))))
3 not even = lemma2 (suc zero)

```

Here the special syntax of `()` is used to indicate that we are doing a pattern matching on a given parameter with no cases to choose from. This situation allows you to complete the definition without having to give a right-hand side.

Of special importance is the equality type, $_ \equiv _ : \{A : Set\} \rightarrow (x : A) \rightarrow (y : A) \rightarrow Set$. We have $x \equiv y$ inhabited if and only if x and y are the same value. It is useful for equational reasoning that is often more natural than inductive proofs.

Chapter 3

Improved Parallel Composition

The contents of this chapter is based on the results published previously in [3]. The chapter covers in detail the proposed modification to the labelled Petri nets parallel composition algorithm allowing for simpler resulting nets while preserving result equivalence up to bisimulation.

3.1 Introduction

Parallel composition (synchronous product) of labelled Petri nets is a fundamental operation in modular hardware design. It is often used to combine models of subsystems into a model of the whole system. In particular, there is a direct correspondence between parallel composition of Signal Transition Graphs (STGs), a class of labelled Petri nets used for modelling asynchronous circuits, and connecting circuits by wires. Hence performing this operation efficiently is important in practice.

Unfortunately, the standard definition of parallel composition almost always yields a ‘messy’ Petri net, with many implicit places even when the component Petri nets did not have them. Some of these places are computationally cheap to remove (e.g. duplicate places – places with identical pre- and postsets). In general, however, for removing remaining implicit places one needs full-blown model checking, something infeasible if the resulting composition is large [57]. Although implicit places do not have noticeable effect on tools based on state space exploration, such as PETRIFY [13], the performance of tools that are based on structural methods, such as DESIJ [56], often deteriorates.

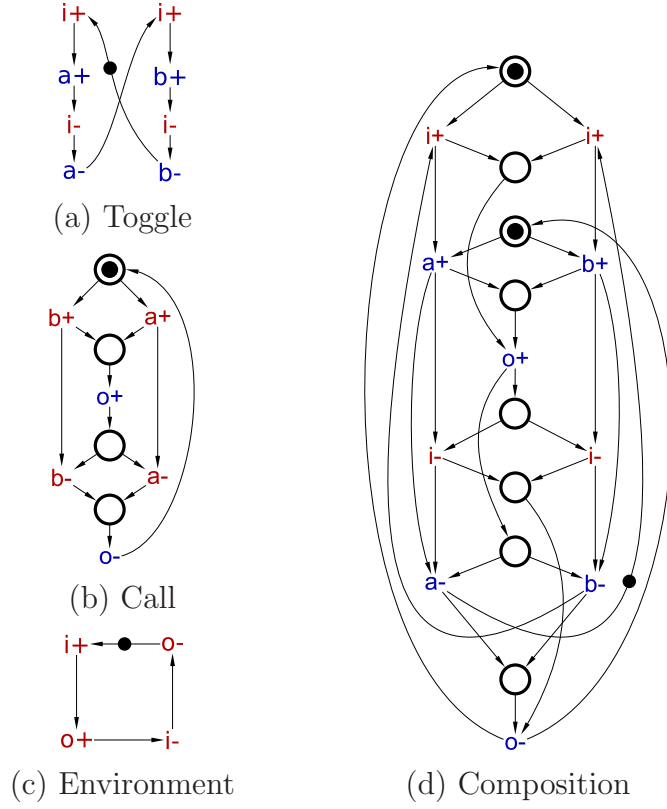


Figure 3.1: Example of standard STG composition.

Consider the example in Fig. 3.1. It depicts an STG specifications of two components (a,b) and the specification of the environment (c). The used short-hand drawing notation for STGs is explained in Sect. 2.2. The model of the behaviour of the entire system can be obtained by constructing a parallel composition of these three STGs, as shown in part (d) of the figure. It contains a few implicit places that are not duplicate places; intuitively, they appear due to repeated causality specifications for every signal: the one coming from the component where this signal is an output, and others from the components where it is an input. Removing these places yields a much ‘cleaner’ STG, as shown in Fig. 3.2(d).

One operation where implicit places matter is *transition contraction* [61] – a crucial part of the resynthesis approach. The idea is to hide the internal communication between the components by labelling the corresponding transitions as ‘dummy’ (they correspond to signals a and b in our example), contract as many of these dummy transitions as possible, thus reducing the size of the STG, and resynthesise the ob-

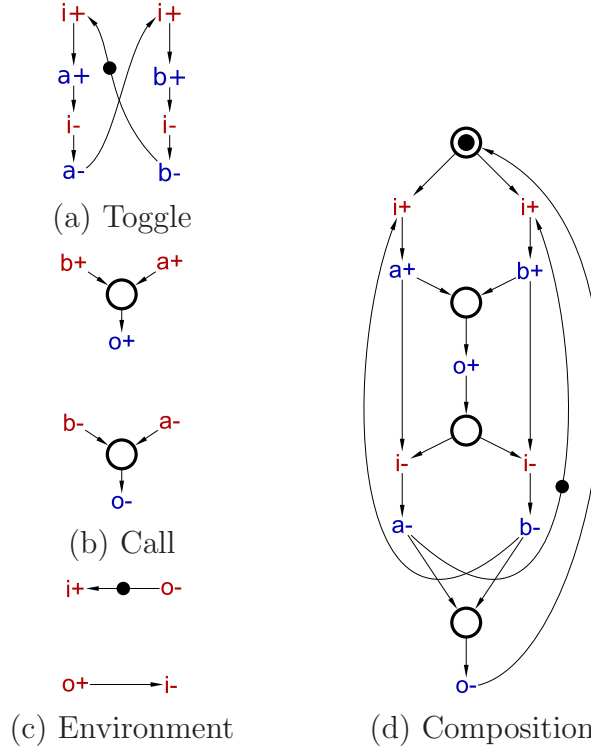


Figure 3.2: Example of improved STG composition: the components are obtained from the corresponding ones in Fig. 3.1 by removing some places, and then the standard parallel composition is applied to these modified components.

tained STG as a circuit. The result is often smaller than the original circuit due to removal of some signals. Transition contraction is normally performed on very large STGs, such as those corresponding to the whole control path of the circuit, and so, for efficiency, it has to be a structural operation. However, such structural contractions are not always possible (see Sect. 2.2), and implicit places in the pre-set and/or post-set of a transition can prevent contracting it, even if a contraction is possible after removing these implicit places. In the example in Fig. 3.1(d), DESIJ [56] cannot contract any of the dummy transitions, even though it performs some structural tests for place redundancy. Yet, it is able to contract all the dummy transitions if the implicit places are removed, i.e. when applied to the STG in Fig. 3.2(d).

In Chapter 3 we present a new method for computing the parallel composition of labelled Petri nets that generates fewer implicit places. It uses the *freeness from computation interference (FCI)* assumption [20] stating that it is impossible that

when one component wants to produce an output it is prevented from doing so by another component not ready to receive it. Violation of the FCI assumption means that the behaviour of the composition does not correspond to that of an implied physical system. For example, an output of a circuit component cannot be physically disabled by another component that is not ready to receive this signal, and so producing this output will lead to a malfunction. However, the composition will be oblivious to the presence of the malfunction and behave as if such an output could not be produced. Hence FCI is a basic correctness requirement – whenever it is violated there is no point in computing a parallel composition since it would not characterise an intended behaviour. In practice, FCI is often guaranteed by construction, e.g. its satisfaction is guaranteed for the control path of a Balsa [21] or HASTE/TANGRAM [8, 51] specification of an asynchronous circuit. The idea of using the FCI condition is reminiscent of the method of input/output exposure in the synthesis by direct mapping described in [59] and of the correct by construction composition of Petri nets for circuit components and the environment used in the DI2PN tool [28].

The essence of the proposed method is illustrated by the example in Fig. 3.2. Before computing a parallel composition, one can remove some of the places in the components (see (a–c) in Fig. 3.2) and then compose the modified STGs. The precise conditions that allow us to remove a particular place will be stated in Sect. 3.2. At this point we shall only mention that they are structural and thus can be efficiently checked. The method guarantees that the number of places in the resulting Petri net is not larger and often much smaller (as the number of places in the composition is the total number of places in all the components), and, under the FCI assumption, the resulting behaviour is the same up to bisimilarity. In the example, composing the modified components yields the STG in Fig. 3.2(d), containing no implicit places. The modified components may include unconstrained transitions and unbounded places and thus be non-implementable. Fortunately, this does not matter, as they are never used on their own, but only in composition with other components, and the resulting behaviour of the composition is guaranteed to correspond to that of the standard composition.

Resynthesis (Section 3.6) of asynchronous circuits is the intended application of the proposed method. However, we envisage that it may find a much wider

applicability since composition of labelled Petri nets is a fundamental operation and the FCI assumption is often known to hold for practically important examples.

3.2 Improved parallel composition

The improved parallel composition algorithm extends the conventional one by adding a pre-processing step, where some places are removed from the components, as they are guaranteed to be implicit in the result. To identify these places, one can note that a place is required in the final composition only if under some reachable marking it can be the place that disables some transition in its postset.

For simplicity, consider the parallel composition $C = C_1 \parallel C_2$, whose components synchronise on a single signal s which is an output of C_1 and an input of C_2 . Let (M_1, M_2) be a reachable marking of C , where M_1 and M_2 are some reachable markings of C_1 and C_2 , respectively. Furthermore, suppose that M_1 enables, say, s^+ in C_1 , where s is an output. Now, if M_2 does not enable s^+ in C_2 , where s is an input, then there is computation interference. Therefore, if the FCI assumption holds, M_2 has to enable s^+ in C_2 , i.e. whenever s^+ is enabled in C_1 , it is also enabled in C_2 . In other words, the firing of s^+ in C is fully controlled by C_1 , and so *the constraints on firing of s that are present in C_2 can be ignored*. This means that the places in the preset of an s^+ -labelled transition in C_2 will be implicit in the composition (subject to some technical conditions formulated below), and so can be removed before the composition is performed.

The above is true for the simple case of STGs with injective labelling and no dummies. However, the general picture is more complicated. In case of non-injective labelling, there can be multiple transitions corresponding to the same input signal transition, and the FCI assumption only guarantees the enabledness of one of them. Hence, some ‘memory’ (in the form of places) is required to trace which of these transitions has to be fired, which prohibits the removal of places from their presets. Furthermore, if the STG contains dummies, removing places from their postsets introduces some undesirable effects explained later. These considerations lead to the following conditions of applicability of the proposed optimisation.

Proposition 1.¹ *Let $C \stackrel{\text{df}}{=} \parallel_{i \in I} C_i$ be a composition of STGs that satisfies the FCI*

¹Note that in our previous work [3] we claimed a stronger equivalence (viz. isomorphism of

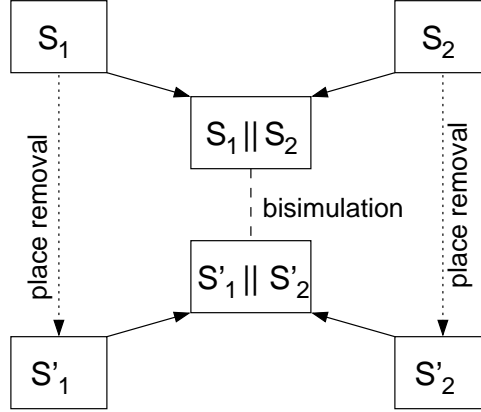


Figure 3.3: Equivalence preserved by place removal in improved parallel composition.

property and yields an output-determinate STG, and, for each $i \in I$, C'_i be the STG obtained from C_i by deleting all places p such that:

1. each transition $t \in p^\bullet$ is labelled with a signal, say s , and:
 - a) s is an input;
 - b) there is an STG C_j for which s is an output;
 - c) there are at most one s^+ - and at most one s^- -labelled transition in C_i ;
2. $\bullet p$ does not contain dummy transitions.

Then $C' \stackrel{\text{df}}{=} \parallel_{i \in I} C'_i$ and C are bisimilar.

The proposition can be depicted schematically by a diagram in Fig. 3.3. Here the boxes named S_1 and S_2 represent the original STGs, S'_1 and S'_2 are represent STGs obtained from S_1 and S_2 by removing places according to the rules detailed above, and $S_1 \parallel S_2$ and $S'_1 \parallel S'_2$ represent S_1 composed with S_2 and S'_1 composed with S'_2 respectively. We use a dashed line to signify the bisimulation relation between $S_1 \parallel S_2$ and $S'_1 \parallel S'_2$.

The conditions 1a and 1b are intrinsic to the proposed method, and essentially state that due to the FCI assumption, firing of an input signal in a component can

reachability graphs), but there was a subtle problem in the proof discovered by Walter Vogler from University of Augsburg, who found a counter-example. Here an updated version of the theorem is presented that avoids the discovered problem.

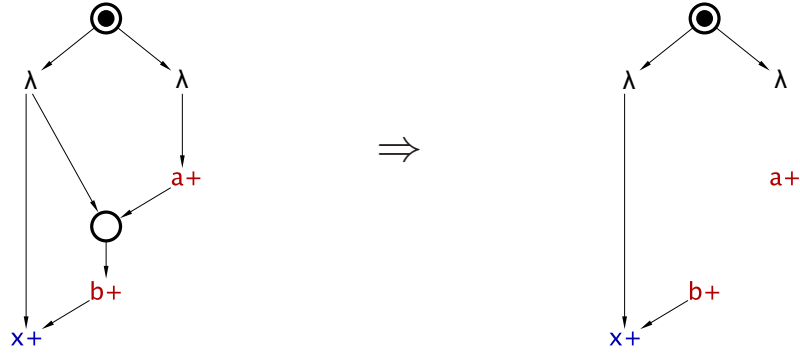


Figure 3.4: Example of an STG where removal of places in the postset of dummy transitions results in a wrong behaviour.

be controlled from the outside (viz. by the component controlling the corresponding output — whose existence is ensured by 1b), and so the component itself can get rid of the places controlling it.

The conditions 1c and 2 are technical restrictions on application of our method. If condition 1c is violated, there are several transitions that have the same label, say s^+ (where s is an input) in the component. When the corresponding output s^+ is produced by some other component, only one of these transitions should fire to match it — but to know which one, the component needs to control their firing, and so the places in their presets cannot be removed.

The necessity of condition 2 is illustrated by Fig. 3.4. Intuitively, the original STG on the left either receives a^+ followed by b^+ without outputting anything, or receives b^+ and produces x^+ in response. However, if the places in front of a^+ and b^+ are removed (which would be possible without condition 2), as shown on the right, then it might produce the unexpected x^+ after the trace $a^+ b^+$. Intuitively, in the initial STG firing of a^+ acts as an evidence that the dummy transition in the right branch has fired, while in the modified one the postset of this dummy transition has been removed, and so it is not possible anymore to guarantee that it has fired when a^+ fires.

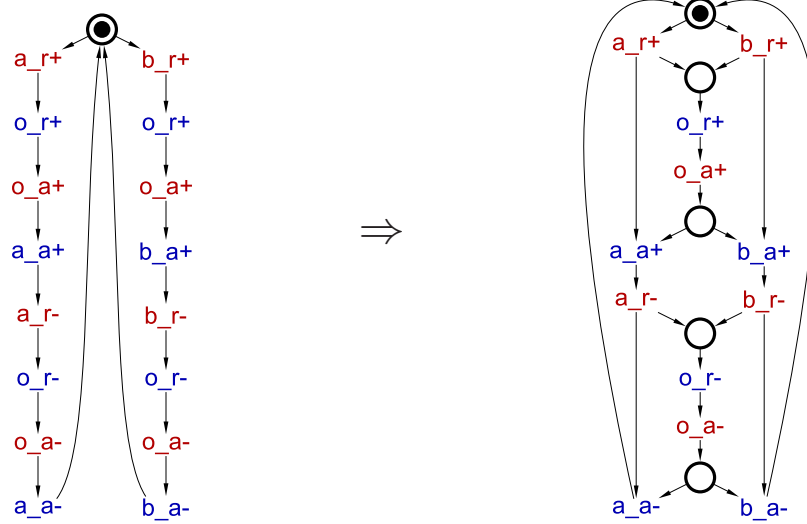


Figure 3.5: Example of enforcing injective labelling in an STG.

3.3 Discussion

In practice, when performing the parallel composition, one would like as few implicit places as possible in the result, and so it would be desirable to weaken the conditions in Prop. 1, so that as many places as possible are removed. As the conditions 1a and 1b are intrinsic, it is unlikely that they can be relaxed. However, the technical conditions 1c and 2 can be dealt with — by ensuring that the components always satisfy them. Indeed, as mentioned in Sect. 2.2, for output-determinate STGs the language is the semantics, and so one often can remove dummy transitions and enforce injective labelling without changing the language, e.g. using the PETRIFY tool [13]; this will ensure that conditions 1c and 2 hold. An example of such a transformation for the Balsa standard component Call is shown in Fig. 3.5. This operation is performed on (small) components rather than the (large) composition, and so is usually cheap. Moreover, in some applications, in particular circuit re-synthesis, the components are taken from a fixed library of component types, and so the transformation can be performed only once for each component type, and subsequently incur no runtime penalty at all.

3.4 Proof of Proposition 1

We begin by defining a relation R between markings of C and C' , which we will show to be a bisimulation. We say that $(M, M') \in R$ iff M is reachable in C and $M(p) = M'(p)$ for each $p \in P'$. Note that $M(p)$ is well-defined, as $P' \subseteq P$ because C' was obtained from C by removing places. Also note that for any given reachable marking M there is exactly one M' such that $(M, M') \in R$, obtained by restricting the domain of M , so R is a function and we employ the notation $M' = R(M)$ to use it as such.

Now we need to prove that the introduced relation R is indeed bisimulation. We start by proving that C' simulates C with R . For that, we first need to show $M'_N = R(M_N)$. That follows from both parallel composition and place removal preserving initial markings of individual places. Now given a reachable marking M^1 with an enabled transition $M^1[l] \gg M^2$ we must show that $R(M^1)[l] \gg R(M^2)$. To prove that we note that any transition enabled in M^1 must also be enabled in $R(M^1)$ because the enabledness condition gets weakened with removal of places. Using that fact we show that whichever transition t labelled l was enabled to allow $M^1[t] \gg M^2$ it is also enabled in $R(M^1)$. The marking after firing must coincide with $R(M^2)$ because the arcs to existing places and their weights are preserved by place removal. This shows $R(M^1)[l] \gg R(M^2)$ and concludes the proof that C' simulates C .

In the second part of the proof we show that C simulates C' with R^{-1} . Here, given $R(M^1)[l] \gg M'^2$ we need to show that there exists an M^2 such that $M^1[l] \gg M^2$ and $M'^2 = R(M^2)$. Again, we note that there must be a transition t' labelled l such that $R(M^1)[t'] \gg M'^2$. If the corresponding transition t is enabled $M^1[t] \gg M^2$ then the proof is complete since we already showed that M'^2 must be equal to $R(M^2)$. For the sake of contradiction, suppose t is not enabled in marking M^1 . Then there is necessarily one of the deleted places p in $\bullet t$ in some of the component STGs C_i , and the number of tokens in this place at marking M^1 is smaller than the weight of the arc (p, t) in C_i (*). Since by condition 1a p^\bullet can contain only input transitions, t must be labelled by s^\pm , where s is an input signal of C_i ; wlog., we assume that the label is s^+ . By condition 1b there is also a component STG C_j where s is an output signal.

Let σ be an execution of C terminating at marking M^1 , and ν be the trace

corresponding to σ (note that such a σ always exists as we restrict R to only include reachable markings). We proceed by showing that (i) $\nu|_{C_j}s^+$ is a trace of C_j and (ii) νs^+ is not a trace of C ; these would mean that there is a violation of FCI in the original composition, leading to a contradiction.

(i) Since s^+ -labelled transition t' is enabled by $R(M^1)$ of C' , its component t_j must also be enabled in C'_j and labelled by the same signal s^+ . Since s is an output in C_j , no places were removed from $\bullet t_j$ when building C'_j due to condition 1a, which means that t_j is also enabled by the marking M_j^1 , and so $\nu|_{C_j}s^+$ is a trace of C_j .

(ii) For the sake of contradiction, suppose νs^+ is a trace of C . Due to the output-determinacy of C , the set of outputs by which ν can be extended is uniquely determined, and so s^+ must be enabled by M^1 (perhaps, after firing several dummy transitions). By condition 1c there is only one s^+ -labelled transition in C_i (viz. t), and so each s^+ -labelled transition in C has p in its preset with the arc from p to this transition having the same weight as the arc (p, t) in C_i . Consequently, each s^+ -transition in C is blocked at marking M^1 because by (*) the number of tokens in p is smaller than the weight of the corresponding arc. Moreover, firing only dummy transitions cannot increase the number of tokens in p and thus enable an s^+ -labelled transition, as by condition 2 $\bullet p$ contains no dummy transitions, a contradiction. Hence νs^+ is not a trace of C .

As explained above, (i) and (ii) imply a violation of FCI and so lead to a contradiction, which means that C and C' must be bisimilar.

3.5 Experiments

The proposed parallel composition algorithm has been evaluated on three series of scalable benchmarks (available from [50]), see Fig. 3.6. They are built of a subset of standard Balsa components [21]: Paralleliser (\parallel), Sequencer ($;$), Call ($()$), Synchroniser (\cdot) and Arbiter (Arb). These controllers are considered to be of size 1; a controller of size $k > 1$ is constructed by replacing the dotted lines with the controllers of size $k - 1$. Each basic component is described by an individual STG; then these STGs are composed using four different techniques:

std the standard parallel composition;

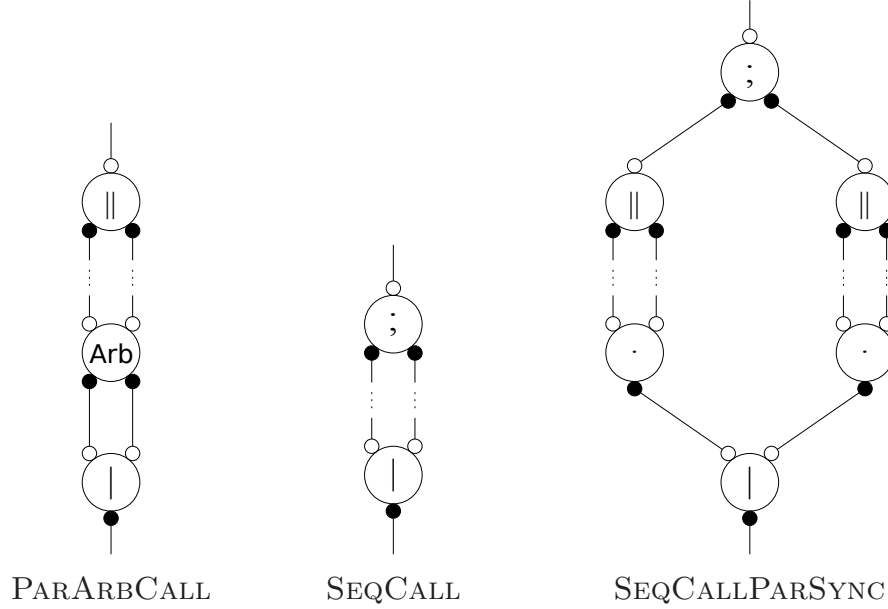


Figure 3.6: Scalable Balsa controllers used in experiments.

opt the optimised parallel composition presented here;

inj the standard parallel composition of the components with enforced injective labelling;

opt+inj the optimised parallel composition of the components with enforced injective labelling.

Note that all the used Balsa components except Call initially had injective labelling, so only the STG for Call was changed in the inj and opt+inj series. Both the standard and optimised parallel composition algorithms have been implemented in PComp tool [50]. The tool automatically deletes duplicate places in all compositions, so all the experimental results are subject to this simplification. The runtimes of PComp were negligible and so not reported.

For each composed STG, the internal signals of the composition were hidden (i.e. turned into dummies), and the DESIJ tool [56] was used to structurally eliminate as many dummies as possible, using either secure or safeness-preserving secure contractions.

The results of our experiments are summarised by the charts in Fig. 3.7. There are six charts altogether, for each of the three benchmark series and each of the

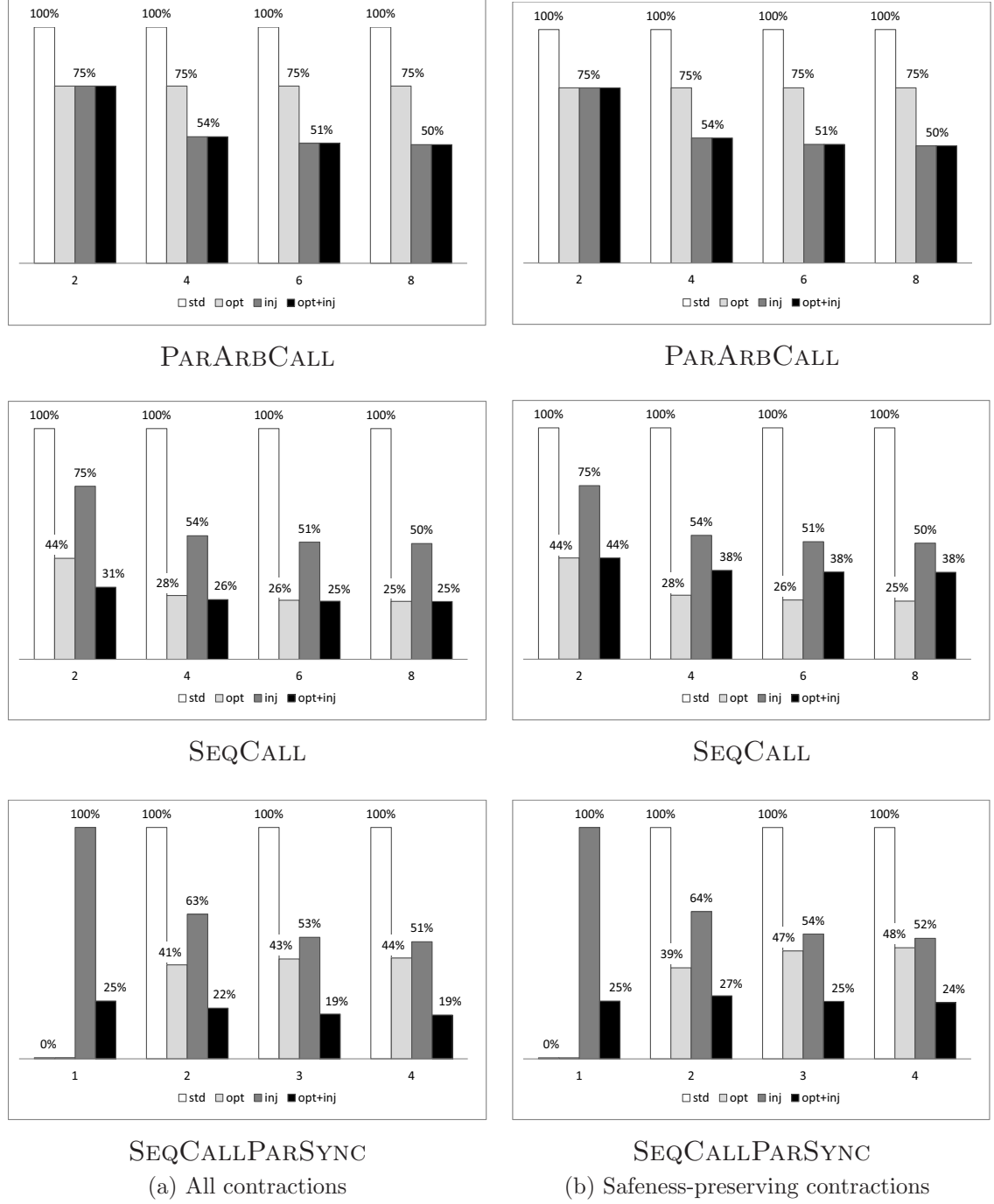


Figure 3.7: Number of non-contractible dummy transitions, normalized to the best value achieved.

contraction modes (secure or safeness-preserving secure). Each chart reports, for each benchmark size within the corresponding series, the numbers of non-contractible dummy transitions (normalised w.r.t. the worst result) remaining in the STG for each of the four composition methods described above.

The experiments demonstrate that the optimised parallel composition is never worse than the standard one in terms of the STG structure that is used for removing dummies (the opt bars are never longer than the std bars, and the opt+inj bars are never longer than the inj bars), and is significantly better in some cases (e.g. for the SEQCALLPARSYNC (4) benchmark there is a factor of five improvement). Moreover, using the optimised technique in conjunction with injective labelling is usually advantageous (the fourth bar is the shortest in almost all cases).

3.6 Balsa workflow optimisation through STG re-synthesis

The main obstacle for the wider acceptance of asynchronous systems is the inherent complexity of their design. Several solutions are accepted by the industry to help to simplify the design process through abstraction of predesigned asynchronous circuit parts as standardised high level components. A designer is able to use these components as “building blocks”, and then obtain the final gate-level design through an automated mapping process. Some of the well-known asynchronous design automation packages, such as Tangram [8], and Balsa [22], define a high-level programming language that is used to describe systems. The language constructs are then directly translated into a network of *handshake components*— blocks with predefined functionality that use *handshakes* to interface with other components, which are in turn mapped into a gate netlist [7].

Although this method greatly enhances the designer’s productivity, it has several important drawbacks. Of these, the control-path overhead is the most decisive. The controllers obtained by syntax-directed mapping are usually far from optimal because the predesigned components are required to implement their declared protocols fully and correctly in order to be reusable in all possible circuit configurations. However, it is often the case that a significant part of their functionality becomes redundant due

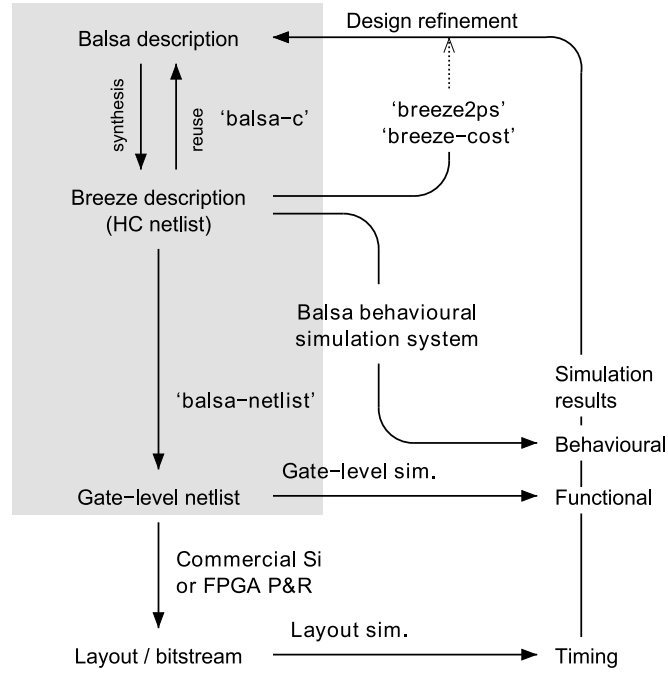


Figure 3.8: Balsa design workflow

to the peculiarities of the specific configuration, e.g. in many cases full handshaking between the components can be avoided.

This redundancy may be eliminated by replacing a manually designed gate-level implementation of the high level components with an equivalent STG (signal transition graph) specification [64]. The STGs of individual components are then composed together to form a STG representation of the whole system STG [52] and is optimised with PETRIFY [13]. An optimal gate-level implementation is then automatically produced from the STG using tools such as PETRIFY [13], SIS [58] and MPSAT [30]. Automatic synthesis becomes problematic when the size of a STG becomes large: modern synthesis tools can handle STGs of no more than 100 signals. The impact of this problem can be lessened by including STG decomposition tools [56] into the workflow. They break a large, optimised STG down into several smaller STGs that are synthesisable in reasonable time. Alternatively, the decomposition step is carried out at the level of handshake circuits, dividing a circuit into several smaller blocks of components.

The standard Balsa design workflow is comprised of several stages (Figure 3.8). The designer writes the system specification in Balsa language. It is passed to

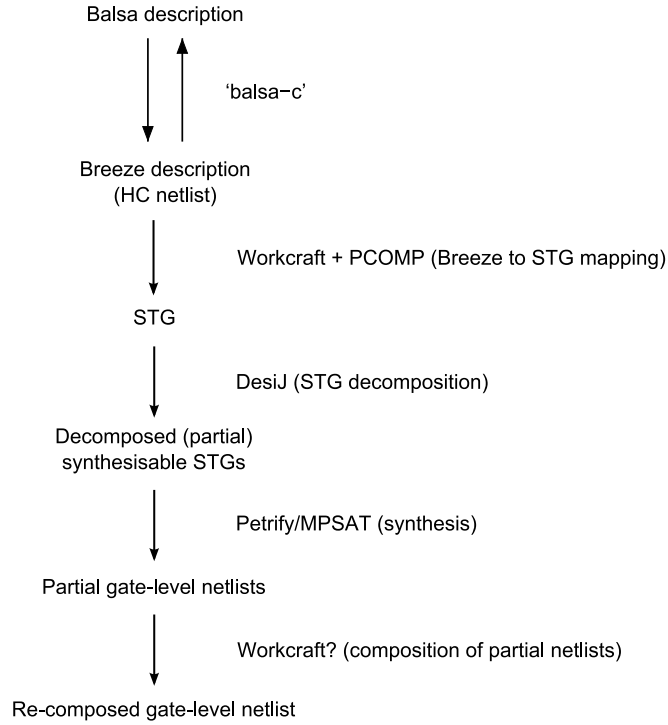


Figure 3.9: Modified Balsa workflow

the Balsa compiler, which generates a handshake component netlist (produced in a language called Breeze) using syntax-directed mapping on the source code. Syntax-directed mapping in this context means that there is a predefined handshake component construct for every syntactic structure. The Breeze netlist is then translated into a gate-level netlist using direct mapping, this time from individual handshake components to their gate-level implementation, which is defined beforehand.

The proposed modification of this workflow is shown in Figure 3.9. The translation from Balsa language into Breeze netlist is retained (and is still done by the Balsa compiler), but the Breeze-netlist to gate-level-netlist mapping is replaced with the STG resynthesis flow as introduced above. Instead of using Balsa tools to produce a gate-level netlist, the Breeze netlist is read by a special interpreted graph model plugin to WORKCRAFT tool [54], which replaces the handshake components with their STG specifications and produces a composition of those STGs using PCOMP tool. If the resulting STG is small enough, the gate-level implementation may immediately be synthesised using any of the available synthesis tools.

However, for many practical cases the composed STG will become quite large. In this case, to synthesise the implementation it is necessary to insert an additional decomposition step, which may be either STG decomposition (implemented using a tool called DESIJ [56] that is automatically called from the plug-in), or handshake circuit (HC) decomposition which is supported by the plug-in directly. Therefore, the whole process is automated in the WORKCRAFT framework.

The technique allows to synthesise more efficient control circuits while at the same time preserving the benefit of rapid design methodology fundamental to Balsa. It should be noted, however, that full modelling of all Breeze components with STGs is not practical. The behaviour of most data components would be too complex to synthesise from an STG. Circuit resynthesis for such components would take too much time and would often be less effective than an already existing gate-level implementation done by an experienced designer. Subsequently, all data-related functionality in HCs is modelled outside of STG composition framework: the STG models include only control signals for the data path elements. These control signals are to be connected after the gate-level generation step to the data-path circuit that is assembled separately (its components are specified by a structural Verilog netlist). The data path is generated automatically side-by-side with the STG behaviour model.

3.6.1 Support of Breeze handshake circuits as interpreted graph model in Workcraft

For the purpose of implementation of the design flow discussed in this section the WORKCRAFT framework was extended with a plug-in that introduces support for Breeze HCs. The new HC model allows WORKCRAFT's convenient visual editing tools to be applied for creation and editing of Breeze netlists. The same plug-in also performs generation of the STG behaviour model for the specified HC. The STG generation algorithm is designed to be highly customisable, with support of multiple handshake protocols and various STG implementations for each type of component. At the moment of writing, STG generation was implemented for a limited set of components using early 4-phase handshake protocol. The library of components is being expanded and will include all Breeze components with support for different handshake protocols.

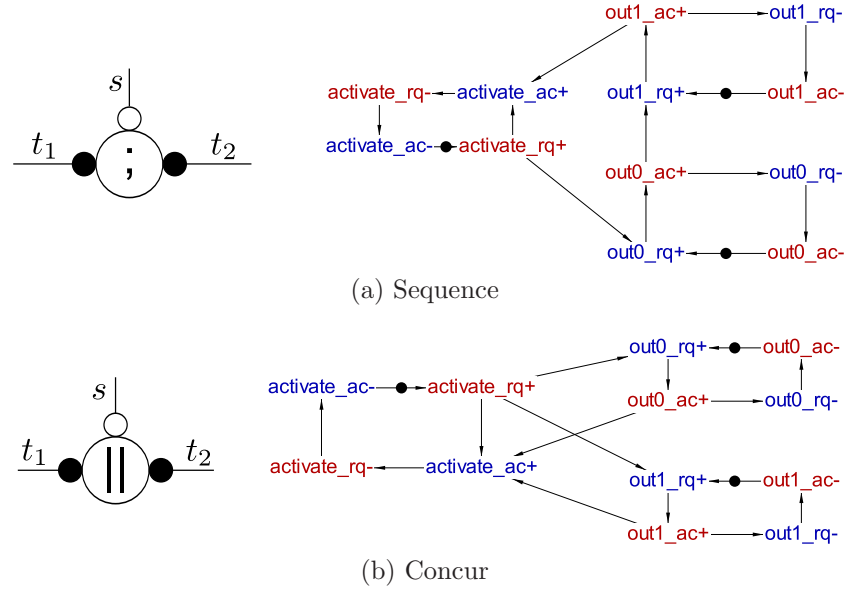


Figure 3.10: Pure control path handshake components and their respective STGs

3.6.2 STG specifications of individual handshake components

Balsa components can be roughly divided in three groups: pure control components, data path control components and data-control interface components. We will review each group separately.

Pure control path components

Pure control components only control the behaviour of other components and do not carry out any data operations. These components are expected to gain the most from the new design workflow because all of their handshakes are inside the control path and such handshaking does not have to always strictly correspond to the general protocol.

The examples are Concur (Figure 3.10b) and SequenceOptimised (Figure 3.10a) components. The STGs in those figures are highly parallel specifications of these components. However, experimental results show that although such implementation might look better on paper, in practise it is sometimes better to specify traditional, more sequential behaviour. This significantly simplifies the task for synthesis tools,

particularly those based on state space exploration techniques, because high parallelism often leads to early state space explosion problem. Besides that, a parallel specification suffers more from CSC (complete state coding) problems: a significant number of auxiliary signals have to be introduced to achieve CSC.

Data path control components

This group of components is used to control the corresponding data path components that execute predefined operations on data. These operations are far too complex for automated synthesis, but the control path part can still be optimised using STG resynthesis, which makes it reasonable to separate data and control signals. The signals that control the data path are in this case specified as the input and output signals of the component's STG. Because the data path blocks are outside this specification, their handshake protocols must be implemented strictly and thus cannot be optimised. This, however, does not prevent the optimisation of handshakes that belong to the same component but interface with other control path components.

BinaryFunc (Figure 3.11a), CallMux (Figure 3.11b), Variable (Figure 3.11c) are good examples of the data path control components.

Data-control interface components

Data-control interface components provide conversion of data to control signals or vice versa. For example, the While component (Figure 3.12a) analyses the input data to decide whether it should end its operation and conclude the activation handshake, or to continue activating the output handshake. Case component (Figure 3.12b) handles the data in a very similar way, however it has an arbitrary bus width, so for bus widths of more than one bit a decoder that resides in the data path could be used to reduce the STG complexity. These components STGs can become quite complex and the strict behaviour of their data-path handshakes must be preserved.

3.6.3 An example: GCD controller

We have chosen the GCD controller (Figure 3.13) to demonstrate the proposed technique. The GCD controller is a good research example because it has components

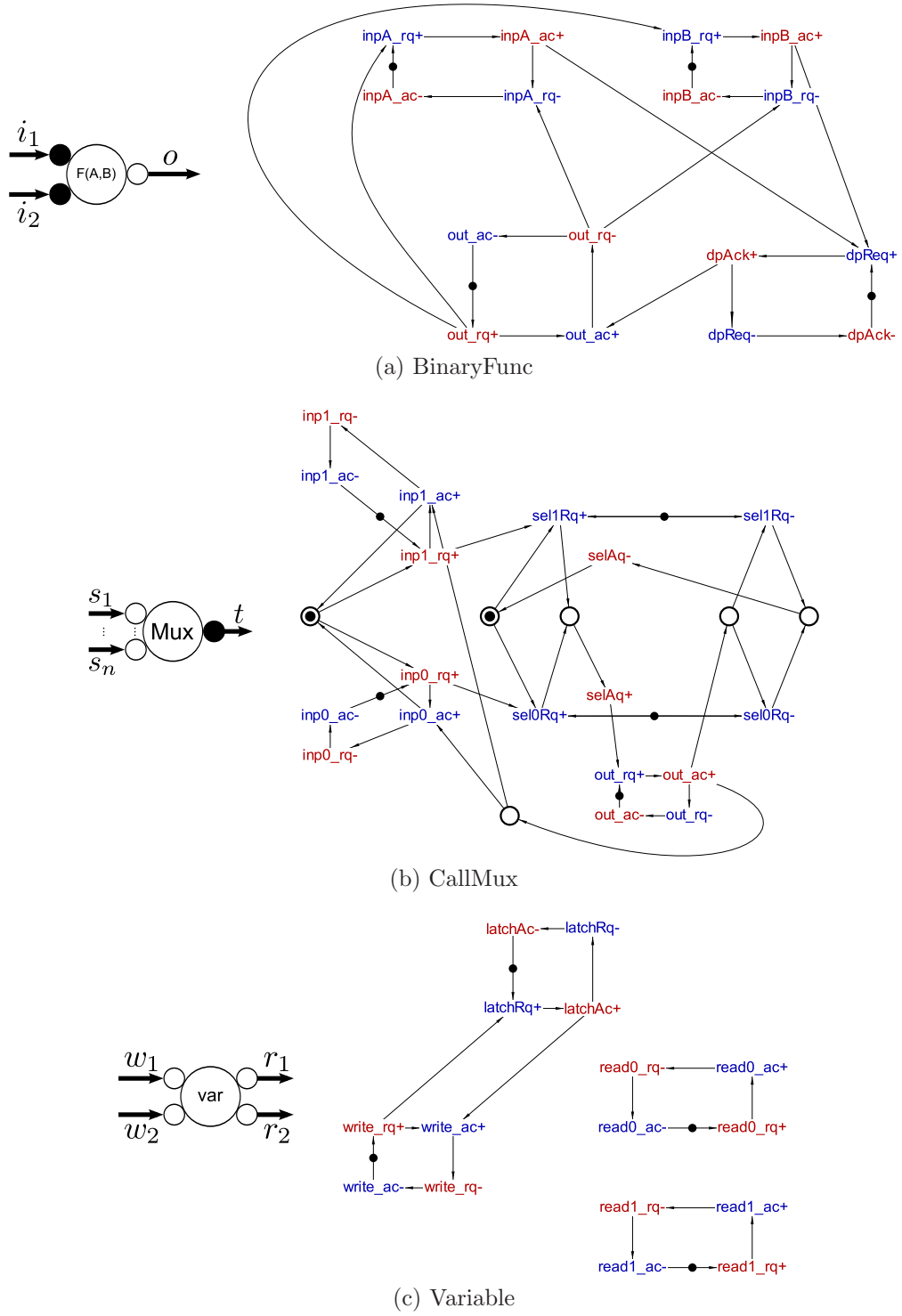


Figure 3.11: Data path control components and their respective STGs

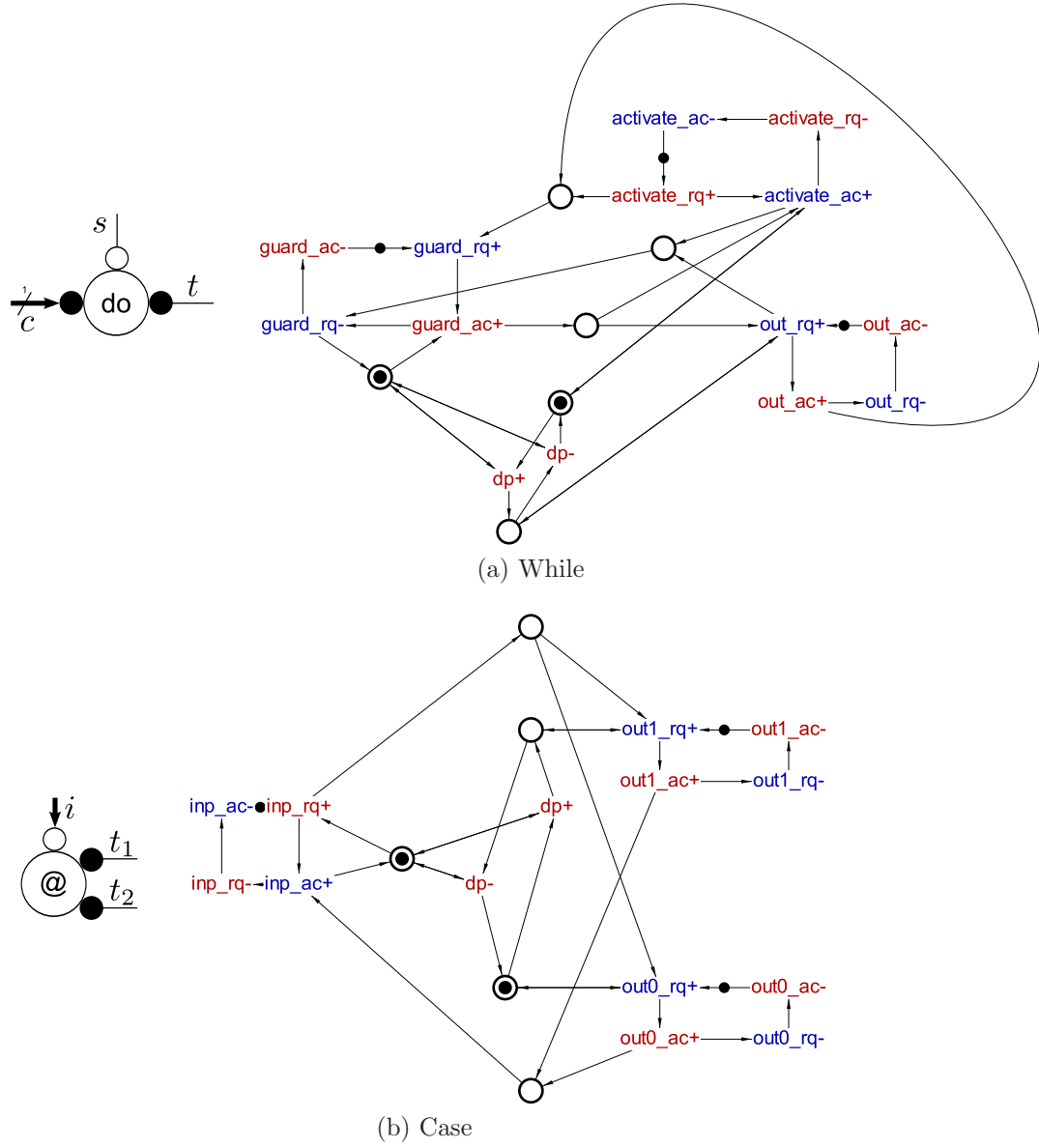


Figure 3.12: Data-control interface components and their respective STGs

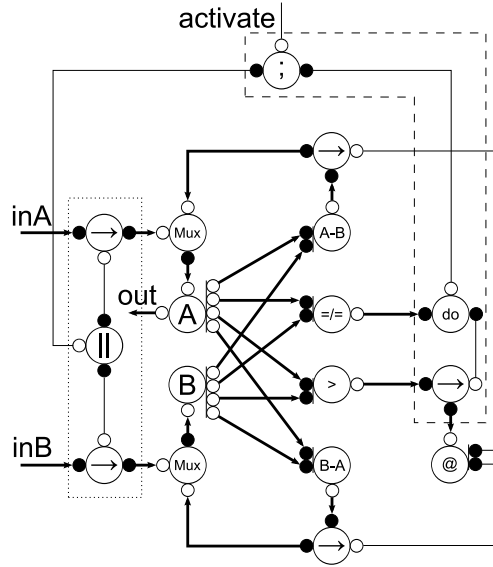


Figure 3.13: Breeze Handshake Circuit model of a GCD block

from every group described in section 3.6.2 and its complexity does not allow omitting of the STG decomposition step, which is an important part of the proposed workflow. All available synthesis tools failed to synthesise a circuit from the fully composed STG model of GCD controller. This proves that the decomposition is a necessary step lacking which the synthesis of a practical circuit is not likely to succeed.

Decomposition on the level of STG can be replaced with decomposition on the level of handshake components. Such decomposition can be done simply by partitioning the input handshake circuit into blocks, trying to minimise the number of handshakes between blocks, and applying the synthesis process to each block separately. While working with the GCD example it was found that decomposition on the level of handshake components can be done easier and is guaranteed to be successful, whereas decomposition on the STG level is a complex task, which requires additional third-party tools.

3.6.4 Experimental results

For the evaluation of the proposed method effectiveness, each individual handshake component was synthesised separately and its cost (in logic equation literals) es-

Component type	Multiplicity	MPSAT cost	PETRIFY cost	Best
BinaryFunc	4	21	27	21
Case	1	13	13	13
Fetch	5	17	13	13
Concur	1	16	16	16
Variable	2	13	18	13
Sequence	1	13	13	13
CallMux	2	25	33	25
While	1	17	17	17
TOTAL		304	334	284

Table 3.1: Costs of individual components

timated. Then, parts of the GCD handshake circuit were synthesised from the STG composition, and the cost of this implementation was compared to the sum of costs of individual components implementations. For synthesis, two tools were used: MPSAT and PETRIFY.

The process of circuit synthesis using this approach was completely automated.

In Table 3.1, the costs of each standalone handshake component, synthesised from the STG specifications, are shown. The results are shown for both applied synthesis tools. The total circuit cost, computed as a sum of individual component costs taking their multiplicity into account.

In Table 3.2, the cost of fully synthesised GCD controller is shown. The cost was derived for synthesis carried out by each tool individually, and for the best mix of HC parts produced by both tools, selected on lowest total cost basis (in Figure 3.13, two such parts are highlighted).

It can be seen from the tables that the cost improvement for the GCD circuit was approximately 28%.

3.7 Summary

We have presented an improved algorithm for computing the parallel composition of STGs or labelled Petri nets. Under the FCI assumptions, it allows to produce nets with fewer implicit places, which aids the subsequent structural algorithms like dummy contraction. It uses only simple structural checks and thus is very efficient

MPSAT		PETRIFY	
Synthesised block	Cost	Synthesised block	Cost
seq+concur+2xfetch	35	var+2xBF	52
fetch+var+2xBF	49	var+2xBF	52
fetch+var+2xBF	49	2xfetch+case	29
fetch+case	23	fetch+while	29
while	17	seq+concur+fetch	29
callmux	25	callmux	33
callmux	25	fetch	13
		callmux	33
TOTAL	223	TOTAL	270

BEST CHOICE	
Synthesised block	Cost
var+callmux+2xBF	63
fetch+while	29
fetch+case	21
seq+concur+2xfetch	35
callmux	25
fetch+var+2xBF	47
TOTAL	220

Table 3.2: Cost of optimally split full GCD circuit

even for large compositions, so the improvement comes at negligible cost.

The algorithm was implemented in the PCOMP tool and evaluated on a set of scalable benchmarks. The experiments proved its efficiency, which increases even more when the components are pre-processed to remove dummies and ensure injective labelling (this is usually cheap, as the components are small; moreover, if the components come from a standard library of component types, this step can be completely eliminated).

Another important advantage is that the improved algorithm places almost no additional effort on the user: the only requirement is to pass an additional command-line option to PCOMP so that it can assume the FCI property and apply the proposed optimisation.

Chapter 4

Theory of Parametrised Graphs

This chapter is based on the results previously published in [43]. It introduces a new formalism called Parametrised Graphs (PG) and tries to capture its most important characteristics in an algebraic structure we call PG-algebra. We introduce some of the theorems in the theory of PG-algebra and provide their proofs. Many of the proofs are written in Agda [49], a formal proof language, so they can be machine-checked instead of being manually inspected.

4.1 Introduction

We continue the work started in [45] where a formal model, called Conditional Partial Order Graphs (CPOGs), was introduced. Using CPOGs as a foundation allowed us to represent individual system configurations and operational modes as annotated graphs and to efficiently overlay them by exploiting their similarities. However, the CPOG formalism lacks the compositionality and the ability to compare and transform specifications in a rigorous manner [43]. In particular, CPOGs always represent a specification as a ‘flat’ structure, similar to the canonical form defined in Section 4.2, hence a hierarchical representation of a system as a composition of its components is not possible. We extend this formalism in several ways:

- We transition from the graphs representing partial orders to general graphs and lift the assumption of graph acyclicity. Nevertheless, if a partial orders is the most natural way to represent a certain aspect of system, this still can be

handled.

- The new formalism is fully compositional – it adds algebraic operations for combining existing specifications.
- We describe the equivalence relation between the specifications as a set of axioms, obtaining an algebra of parametrised graphs. This set of axioms is proved to be sound, minimal and complete [43].
- We have defined equivalence preserving transformations; this permits one to use the algebra to safely manipulate PG specifications. This can be viewed as adding a syntactic level to the semantic representation of specifications, and is reminiscent of the relationship between digital circuits and Boolean algebra.

Since parametrised graphs are likely to be applied in a safety-critical toolchain, it is imperative to attain a degree of confidence in the properties of the PG formalism. Equally important is to convince prospective users that the technique is sound and lives up to its promises. To fulfil this goal, it was decided to construct in a strict and controlled manner a complete formalisation of the PG formalism. The Agda system [49] was chosen for its expressive notation language and extensive support for machine-checked formal inference. Agda has enjoyed a notable success as the basis for the formalisation of wide range of problems in the domain of programming language research [5, 27].

We will show that, in contrast to more expressive formalisms such as process algebrae, many useful properties of PGs can be efficiently verified, for example equality is decidable and, moreover, decidable in NP-time. It therefore lies in the complexity class below Petri nets (whose interesting properties are PSPACE-hard) and Turing-powerful process algebrae. Despite their lower expressive power, PGs are sufficient for our purposes as demonstrated in case studies.

We demonstrate the usefulness of the developed formalism on the basis of two case studies. The first one (Section 4.5.1) is concerned with development of a phase encoding controller that represents information by the order of arrival of signals on n wires. As there are $n!$ possible arrival orders, it is a challenge to specify the set of corresponding behavioural scenarios in a compact way. The proposed formalism not only allows us to solve this problem but also does it in a compositional manner.

The final specification is obtained through the composition of fixed-size fragments describing the behaviours of a pair of wires (the latter is impossible with the CPOG formalism).

4.2 Parametrised Graphs

A *Parametrised Graph* (PG) is a model which has evolved from Conditional Partial Order Graphs (CPOG) [45]. We consider directed graphs $G = (V, E)$ whose vertices are picked from the fixed alphabet of *actions* $\mathcal{A} = \{a, b, \dots\}$. Hence the vertices of G would usually model actions (or *events*) of the system being designed, while the arcs would usually model the *precedence* or *causality* relation: if there is an arc going from a to b then action a precedes action b . We will denote the *empty graph* (\emptyset, \emptyset) by ε and the *singleton graphs* $(\{a\}, \emptyset)$ simply by a , for any $a \in \mathcal{A}$.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs, where V_1 and V_2 as well as E_1 and E_2 are not necessarily disjoint. We define the following operations on graphs (in the order of increasing precedence):

Overlay: $G_1 + G_2 \stackrel{\text{df}}{=} (V_1 \cup V_2, E_1 \cup E_2)$.

Sequence: $G_1 \rightarrow G_2 \stackrel{\text{df}}{=} (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$.

Condition: $[1]G \stackrel{\text{df}}{=} G$ and $[0]G \stackrel{\text{df}}{=} \varepsilon$.

In other words, the *overlay* $+$ and *sequence* \rightarrow are binary operations on graphs with the following semantics: $G_1 + G_2$ is a graph obtained by *overlaying* graphs G_1 and G_2 , i.e. it contains the union of their vertices and arcs, while graph $G_1 \rightarrow G_2$ contains the union plus the arcs connecting every vertex from graph G_1 to every vertex from graph G_2 (self-loops can be formed in this way if V_1 and V_2 are not disjoint). From the behavioural point of view, if graphs G_1 and G_2 correspond to two systems then $G_1 + G_2$ corresponds to their *parallel composition* and $G_1 \rightarrow G_2$ corresponds to their *sequential composition*. One can observe that any non-empty graph can be obtained by successively applying the operations $+$ and \rightarrow to the singleton graphs.

To make notation cleaner we are going to use the following operator precedence rules: $[0]$ and $[1]$ bind more tightly than \rightarrow and \rightarrow binds more tightly than $+$.

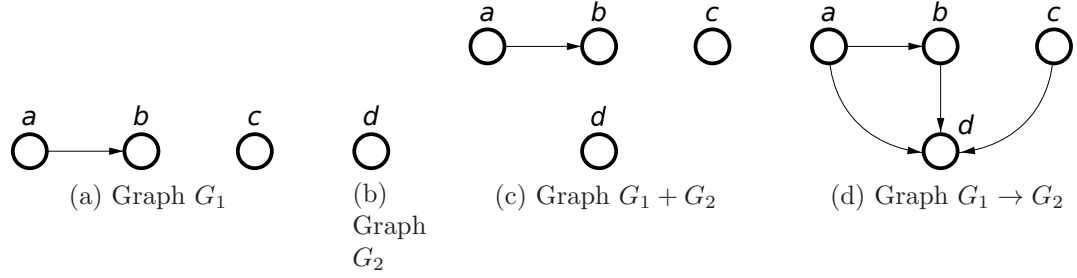


Figure 4.1: Overlay and sequence example (no common vertices)

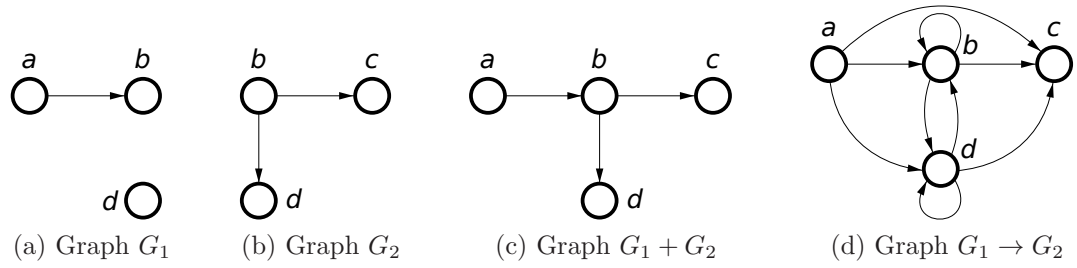


Figure 4.2: Overlay and sequence example (common vertices)

Fig. 4.1 shows an example of two graphs together with their overlay and sequence. One can see that the overlay does not introduce any dependencies between the actions coming from different graphs, therefore they can be executed concurrently. On the other hand, the sequence operation imposes the order on the actions by introducing new dependencies between actions a, b and c coming from graph G_1 and action d coming from graph G_2 . Hence, the resulting system behaviour is interpreted as the behaviour specified by graph G_1 followed by the behaviour specified by graph G_2 . Another example of system composition is shown in Fig. 4.2. Since the graphs have common vertices, their compositions are more complicated, in particular, their sequence contains the self-dependencies (b, b) and (d, d) which lead to a *deadlock* in the resulting system: action a can occur, but all the remaining actions are locked.

Given a graph G , the unary *condition* operations can either preserve it (*true condition* $[1]G$) or nullify it (*false condition* $[0]G$). They should be considered as a family $\{[b]\}_{b \in \mathbb{B}}$ of operations parametrised by a Boolean value b .

Having defined the basic operations on the graphs, one can build graph expressions using these operations, the empty graph ε , the singleton graphs $a \in \mathcal{A}$, and

the Boolean constants 0 and 1 (as the parameters of the conditional operations) — much like the usual arithmetical expressions. We now consider replacing the Boolean constants with Boolean variables or general predicates (this step is akin going from arithmetic to algebraic expressions). The value of such an expression depends on the values of its parameters, and so we call such an expression a *parametrised graph* (PG).

One can easily prove the following properties of the operations introduced above.

- Properties of overlay:

Identity: $G + \varepsilon = G$

Commutativity: $G_1 + G_2 = G_2 + G_1$

Associativity: $(G_1 + G_2) + G_3 = G_1 + (G_2 + G_3)$

- Properties of sequence:

Left identity: $\varepsilon \rightarrow G = G$

Right identity: $G \rightarrow \varepsilon = G$

Associativity: $(G_1 \rightarrow G_2) \rightarrow G_3 = G_1 \rightarrow (G_2 \rightarrow G_3)$

- Other properties:

Left/right distributivity:

$$G_1 \rightarrow (G_2 + G_3) = G_1 \rightarrow G_2 + G_1 \rightarrow G_3$$

$$(G_1 + G_2) \rightarrow G_3 = G_1 \rightarrow G_3 + G_2 \rightarrow G_3$$

Decomposition:

$$G_1 \rightarrow G_2 \rightarrow G_3 = G_1 \rightarrow G_2 + G_1 \rightarrow G_3 + G_2 \rightarrow G_3$$

- Properties involving conditions:

Conditional ε : $[b]\varepsilon = \varepsilon$

Conditional overlay: $[b](G_1 + G_2) = [b]G_1 + [b]G_2$

Conditional sequence: $[b](G_1 \rightarrow G_2) = [b]G_1 \rightarrow [b]G_2$

AND-condition: $[b_1 \wedge b_2]G = [b_1][b_2]G$

OR-condition: $[b_1 \vee b_2]G = [b_1]G + [b_2]G$

Condition regularisation:

$$[b_1]G_1 \rightarrow [b_2]G_2 = [b_1]G_1 + [b_2]G_2 + [b_1 \wedge b_2](G_1 \rightarrow G_2)$$

Now, due to the above properties of the operators, it is possible to define the following canonical form of a PG. In the proof below, we call a singleton graph, possibly prefixed with a condition, a *literal*.

Proposition 1 (Canonical form of a PG). *Any PG can be rewritten in the following canonical form:*

$$\left(\sum_{v \in V} [b_v]v \right) + \left(\sum_{u, v \in V} [b_{uv}](u \rightarrow v) \right), \quad (4.1)$$

where:

- V is a subset of singleton graphs that appear in the original PG;
- for all $v \in V$, b_v are canonical forms of Boolean expressions and are distinct from 0;
- for all $u, v \in V$, b_{uv} are canonical forms of Boolean expressions such that $b_{uv} \Rightarrow b_u \wedge b_v$.

Proof. (i) First we prove that any PG can be converted to the form (4.1).

All the occurrences of ε in the expression can be eliminated by the identity and conditional ε properties (unless the whole PG equals to ε , in which case we take $V = \emptyset$). To avoid unconditional subexpressions, we prefix the resulting expression with $[1]$, and then by the conditional overlay/sequence properties we propagate all the conditions that appear in the expression down to the singleton graphs (compound conditions can be always reduced to a single one by the AND-condition property). By the decomposition and distributivity properties, the expression can be rewritten as an overlay of literals and subexpressions of the form $l_1 \rightarrow l_2$, where l_1 and l_2 are

literals. The latter subexpressions can be rewritten using the condition regularisation rule:

$$[b_1]u \rightarrow [b_2]v = [b_1]u + [b_2]v + [b_1 \wedge b_2](u \rightarrow v)$$

Now, literals corresponding to the same singleton graphs, as well as subexpressions of the form $[b](u \rightarrow v)$ that correspond to the same pair of singleton graphs u and v , are combined using the OR-condition property. Then the literals prefixed with 0 conditions can be dropped. Now the set V consists of all the singleton graphs occurring in the literals. To turn the overall expression into the required form it only remains to add missing subexpressions of the form $[0](u \rightarrow v)$ for every $u, v \in V$ such that the expression does not contain the subexpression of the form $[b](u \rightarrow v)$. Note that the property $b_{uv} \Rightarrow b_u \wedge b_v$ is always enforced by this construction:

- condition regularisation ensures this property;
- combining literals using the OR-condition property can only strengthen the right hand side of this implication, and so cannot violate it;
- adding $[0](u \rightarrow v)$ does not violate the property as it trivially holds when $b_{uv} = 0$.

(ii) We now show that (4.1) is a canonical form, i.e. if $L = R$ then their canonical forms $\text{can}(L)$ and $\text{can}(R)$ coincide.

For the sake of contradiction, assume this is not the case. Then we consider two cases (all possible cases are symmetric to one of these two):

1. $\text{can}(L)$ contains a literal $[b_v]v$ whereas $\text{can}(R)$ either contains a literal $[b'_v]v$ with $b'_v \neq b_v$ or does not contain any literal corresponding to v , in which case we say that it contains a literal $[b'_v]v$ with $b'_v = 0$. Then for some values of parameters one of the graphs will contain vertex v while the other will not.
2. $\text{can}(L)$ and $\text{can}(R)$ have the same set V of vertices, but $\text{can}(L)$ contains a subexpression $[b_{uv}](u \rightarrow v)$ whereas $\text{can}(R)$ contains a subexpression $[b'_{uv}](u \rightarrow v)$ with $b'_{uv} \neq b_{uv}$. Then for some values of parameters one of the graphs will contain the arc (u, v) (note that due to $b_{uv} \Rightarrow b_u \wedge b_v$ and $b'_{uv} \Rightarrow b_u \wedge b_v$ vertices u and v are present), while the other will not.

In both cases there is a contradiction with $L = R$. □

This canonical form allows one to lift the notion of *adjacency matrix* of a graph to PGs. Recall that the adjacency matrix (b_{uv}) of a graph (V, E) is a $|V| \times |V|$ Boolean matrix such that $b_{uv} = 1$ if $(u, v) \in E$ and $b_{uv} = 0$ otherwise. The adjacency matrix of a PG is obtained from the canonical form (4.1) by gathering the predicates b_{uv} into a matrix. The adjacency matrix of a PG is similar to that of a graph, but it contains predicates rather than Boolean values. It does not uniquely determine a PG, as the predicates of the vertices cannot be derived from it; to fully specify a PG one also has to provide predicates b_v from the canonical form (4.1).

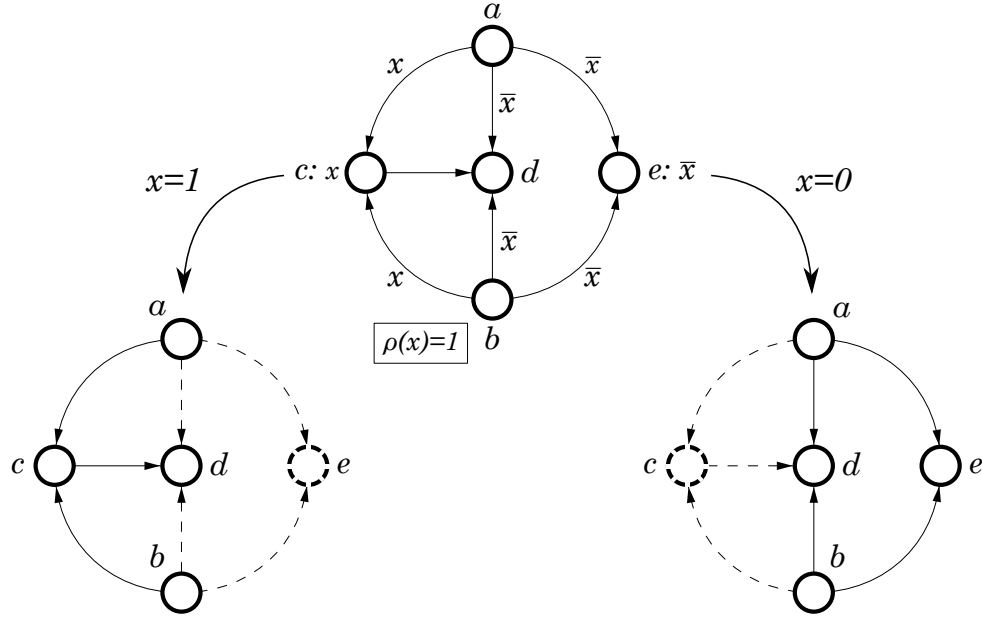
Another advantage of this canonical form is that it provides a graphical notation for PGs. The vertices occurring in the canonical form (set V) can be represented by circles, and the subexpressions of the form $u \rightarrow v$ by arcs. The label of a vertex v consists of the vertex name, colon and the predicate b_v , while every arc (u, v) is labelled with the corresponding predicate b_{uv} . As adjacency matrices of PGs tend to have many constant elements, we use a simplified notation in which the arcs with constant 0 predicates are not drawn, and constant 1 predicates are dropped; moreover, it is convenient to assume that the predicates on arcs are implicitly ANDed with those on incident vertices (to enforce the invariant $b_{uv} \Rightarrow b_u \wedge b_v$), which often allows one to simplify predicates on arcs. This can be justified by introducing the ternary operator, called *conditional sequence*:

$$u \xrightarrow{b} v \stackrel{\text{df}}{=} [b](u \rightarrow v) + u + v$$

Intuitively, PG $u \xrightarrow{b} v$ consists of two unconditional vertices connected by an arc with the condition b . By case analysis on b_1 and b_2 one can easily prove the following properties of the conditional sequence that allow simplifying the predicates on arcs:

$$\begin{aligned} [b_1]u \xrightarrow{b_1 \wedge b_2} v &= [b_1]u \xrightarrow{b_2} v \\ u \xrightarrow{b_1 \wedge b_2} [b_2]v &= u \xrightarrow{b_1} [b_2]v \end{aligned}$$

Fig. 4.3(top) shows an example of a PG. The predicates depend on a Boolean variable x . The predicates of vertices a , b and d are constants 1; such vertices are called *unconditional*. Vertices c and e are *conditional*, and their predicates are x


 Figure 4.3: PG specialisations: $H|_x$ and $H|_{\bar{x}}$

and \bar{x} , respectively. Arcs also fall into two classes: *unconditional*, i.e. those whose predicate and the predicates of their incident vertices are constants 1, and *conditional* (in this example, all the arcs are conditional).

A *specialisation* $H|_p$ of a PG H under predicate p is a PG, whose predicates are simplified under the assumption that p holds. If H specifies the behaviour of the whole system, $H|_p$ specifies the part of the behaviour that can be realised under condition p . An example of a graph and its two specialisations is presented in Fig. 4.3. The leftmost specialisation $H|_x$ is obtained by removing from the graph those vertices and arcs whose predicates evaluate to 0 under condition x , and simplifying the other predicates. Hence, vertex e and arcs (a,d) , (a,e) , (b,d) and (b,e) disappear, and all the other vertices and arcs become unconditional. The rightmost specialisation $H|_{\bar{x}}$ is obtained analogously. Each of the obtained specialisations can be regarded as a specification of a particular behavioural scenario of the modelled system, e.g. as specification of a processor instruction.

4.2.1 Specification and composition of instructions

Consider a processing unit that has two registers A and B , and can perform two different instructions: *addition* and *exchange* of two variables stored in memory. The processor contains five datapath components (denoted by $a \dots e$) that can perform the following atomic actions:

- a) Load register A from memory;
- b) Load register B from memory;
- c) Compute the sum of the numbers stored in registers A and B , and store it in A ;
- d) Save register A into memory;
- e) Save register B into memory.

Table 4.1 describes the addition and exchange instructions in terms of usage of these atomic actions.

The addition instruction consists of loading the two operands from memory (causally independent actions a and b), their addition (action c), and saving the result (action d). Let us assume for simplicity that in this example all causally independent actions are always performed concurrently, see the corresponding scenario *ADD* in the table.

The operation of exchange consists of loading the operands (causally independent actions a and b), and saving them into swapped memory locations (causally independent actions d and e), as captured by the *XCHG* scenario. Note that in order to start saving one of the registers it is necessary to wait until both of them have been loaded to avoid overwriting one of the values.

One can see that the two scenarios in Table 4.1 appear to be the two specialisations of the PG shown in Fig. 4.3, thus this PG can be considered as a joint specification of both instructions. Two important characteristics of such a specification are that the common events $\{a, b, d\}$ are overlaid, and the choice between the two operations is modelled by the Boolean predicates associated with the vertices and arcs of the PG. As a result, in our model there is no need for a ‘nodal point’ of choice, which tend to appear in alternative specification models: a Petri Net (resp. Finite

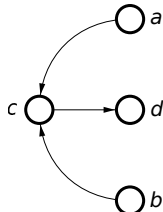
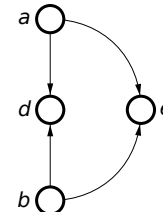
Instruction	Addition	Exchange
Action sequence	a) Load A b) Load B c) Add B to A d) Save A	a) Load A b) Load B d) Save A e) Save B
Execution scenario with maximum concurrency	 <p style="text-align: center;">ADD</p>	 <p style="text-align: center;">$XCHG$</p>

Table 4.1: Two instructions specified as partial orders

State Machine) would have an explicit choice place (resp. state), and a specification written in a Hardware Description Language would describe the two instructions by two separate branches of a conditional statement **if** or **case** [19]).

The PG operations introduced above allow for a natural specification of the system as a collection of its behavioural scenarios, which can share some common parts. For example, in this case the overall system is composed as

$$\begin{aligned}
 H &= [x]ADD + [\bar{x}]XCHG = \\
 &= [x]((a+b) \rightarrow c + c \rightarrow d) + [\bar{x}]((a+b) \rightarrow (d+e)).
 \end{aligned} \tag{4.2}$$

Such specifications can often be simplified using the properties of graph operations. The next section describes the equivalence relation between the PGs with a set of axioms, thus obtaining an algebraic structure.

4.3 Algebraic structure of Parametrised Graphs

A parametrised graph is a quintuple (V, E, C, D, X) where $E \subseteq V \times V$ and $C \in V \rightarrow \text{FORMULA}(X)$, $D \in E \rightarrow \text{FORMULA}(X)$ provide vertex and edge conditions; X

is the set of identifiers occurring free in parametrisation conditions. Some φ from $\text{FORMULA}(X)$ is a Boolean predicate with free variables from X . We consider a predicate equal to another predicate whenever they are extensionally equal. For any $x \in X$, we say that the formula $x \in \text{FORMULA}(X)$ is an atomic occurrence of x .

A PG-algebra is a tuple $\langle \mathcal{G}, +, \rightarrow, [0], [1] \rangle$, where \mathcal{G} is a set of graphs whose vertices are picked from the alphabet \mathcal{A} and the operations match those defined for graphs above. The equivalence relation is given by the following axioms.

- $+$ is commutative and associative
- \rightarrow is associative
- ε is a left and right identity of \rightarrow
- \rightarrow distributes over $+$:

$$\begin{aligned} p \rightarrow (q + r) &= p \rightarrow q + p \rightarrow r \\ (p + q) \rightarrow r &= p \rightarrow r + q \rightarrow r \end{aligned}$$

- Decomposition:

$$p \rightarrow q \rightarrow r = p \rightarrow q + p \rightarrow r + q \rightarrow r$$

- Condition: $[0]p = \varepsilon$ and $[1]p = p$

The following derived equalities can be proved from PG-algebra axioms [42, Prop. 2, 3]:

- ε is an identity of $+$: $p + \varepsilon = p$
- $+$ is idempotent: $p + p = p$
- Left and right absorption:

$$\begin{aligned} p + p \rightarrow q &= p \rightarrow q \\ q + p \rightarrow q &= p \rightarrow q \end{aligned}$$

- Conditional ε : $[b]\varepsilon = \varepsilon$
- Conditional overlay: $[b](p + q) = [b]p + [b]q$

- Conditional sequence: $[b](p \rightarrow q) = [b]p \rightarrow [b]q$
- AND-condition: $[b_1 \wedge b_2]p = [b_1][b_2]p$
- OR-condition: $[b_1 \vee b_2]p = [b_1]p + [b_2]p$
- Choice propagation:

$$\begin{aligned} [b](p \rightarrow q) + [\bar{b}](p \rightarrow r) &= p \rightarrow ([b]q + [\bar{b}]r) \\ [b](p \rightarrow r) + [\bar{b}](q \rightarrow r) &= ([b]p + [\bar{b}]q) \rightarrow r \end{aligned}$$

- Condition regularisation:

$$[b_1]p \rightarrow [b_2]q = [b_1]p + [b_2]q + [b_1 \wedge b_2](p \rightarrow q)$$

Note that as ε is a left and right identity of \rightarrow and $+$, there can be no other identities for these operations. Interestingly, unlike with many other algebraic structures, the two main operations in the PG-algebra have the same identity.

It is easy to see that PGs are a model of PG-algebra, as all the axioms of PG-algebra are satisfied by PGs; in particular, this means that PG-algebra is *sound*. Moreover, any PG-algebra expression has the canonical form (4.1), as the proof of Prop. 1 can be directly imported:

- It is always possible to translate a PG-algebra expression to this canonical form, as part (i) of the proof relies only on the properties of PGs that correspond to either PG-algebra axioms or equalities above.
- If $L = R$ holds in PG-algebra then $L = R$ holds also for PGs (as PGs are a model of PG-algebra), and so the PGs $can(L)$ and $can(R)$ coincide, see part (ii) of the proof. Since PGs $can(L)$ and $can(R)$ are in fact the same objects as the expressions $can(L)$ and $can(R)$ of the PG-algebra, (4.1) is a canonical form of a PG-algebra expression.

This also means that PG-algebra is *complete* w.r.t. PGs, i.e. any PG equality can be either proved or disproved using the axioms of PG-algebra (by converting to the canonical form).

$$\begin{aligned}
 [x]((a + b) \rightarrow c + c \rightarrow d) + [\bar{x}]((a + b) \rightarrow (d + e)) &= \text{(closure)} \\
 [x]((a + b) \rightarrow c + (a + b) \rightarrow d + c \rightarrow d) + \\
 + [\bar{x}]((a + b) \rightarrow (d + e)) &= \text{(decomposition)} \\
 [x]((a + b) \rightarrow c \rightarrow d) + [\bar{x}]((a + b) \rightarrow (d + e)) &= \text{(choice propagation)} \\
 (a + b) \rightarrow ([x](c \rightarrow d) + [\bar{x}](d + e)) &= \text{(conditional overlay)} \\
 (a + b) \rightarrow ([x](c \rightarrow d) + [\bar{x}]d + [\bar{x}]e) &= (\rightarrow -\text{identity}) \\
 (a + b) \rightarrow ([x](c \rightarrow d) + [\bar{x}](\varepsilon \rightarrow d) + [\bar{x}]e) &= \text{(choice propagation)} \\
 (a + b) \rightarrow (([x]c + [\bar{x}]\varepsilon) \rightarrow d + [\bar{x}]e) &= \text{(conditional } \varepsilon, +\text{-identity)} \\
 (a + b) \rightarrow ([x]c \rightarrow d + [\bar{x}]e). &
 \end{aligned}$$

Figure 4.4: Simplifying expression (4.2) using the Closure axiom

The provided set of axioms of PG-algebra is *minimal*, i.e. no axiom from this set can be derived from the others. The minimality was checked by enumerating the fixed-size models of PG-algebra with the help of the ALG tool [9]: It turns out that removing any of the axioms leads to a different number of non-isomorphic models of a particular size, implying that all the axioms are necessary.

Hence, the following result holds:

Theorem 2 (Soundness, Minimality and Completeness). *The set of axioms of PG-algebra is sound, minimal and complete w.r.t. PGs.*

4.4 Transitive Parametrised Graphs and their algebra

In many cases the arcs of the graphs are interpreted as the causality relation, and so the graph itself is a partial order. However, in practice it is convenient to drop some or all of the transitive arcs, i.e. two graphs should be considered equal whenever their transitive closures are equal. E.g. in this case the graphs specified by the expressions $a \rightarrow b + b \rightarrow c$ and $a \rightarrow b + a \rightarrow c + b \rightarrow c$ are considered as equal. PGs with this equality relation are called *Transitive Parametrised Graphs* (TPG). To capture this

algebraically, we augment the PG-algebra with the *Closure* axiom:

$$\text{if } q \neq \varepsilon \text{ then } p \rightarrow q + q \rightarrow r = p \rightarrow q + p \rightarrow r + q \rightarrow r.$$

One can see that by repeated application of this axiom one can obtain the transitive closure of any graph, including those with cycles. The resulting algebraic structure is called Transitive Parametrised Graphs Algebra (TPG-algebra).

Note that the condition $q \neq \varepsilon$ in the Closure axiom is necessary, as otherwise

$$a + b = a \rightarrow \varepsilon + \varepsilon \rightarrow b = a \rightarrow \varepsilon + a \rightarrow b + \varepsilon \rightarrow b = a \rightarrow b,$$

and the operations $+$ and \rightarrow become identical, which is clearly undesirable.

The Closure axiom helps to simplify specifications by reducing the number of arcs and/or simplifying their conditions. For example, consider the PG expression (4.2). As the scenarios of this PG are interpreted as the orders of execution of actions, it is natural to use the Closure axiom. Note that the expression cannot be simplified in PG-algebra; however, in the TPG-algebra it can be considerably simplified, as shown in Fig. 4.4.

The corresponding TPG is shown in Fig. 4.5. Note that it has fewer conditional elements than the PG in Fig. 4.3; though the specialisations are now different, they have the same transitive closures.

We now lift the canonical form (4.1) to TPGs and TPG-algebra. Note that the only difference is the last requirement.

Proposition 3 (Canonical form of a TPG). *Any TPG can be rewritten in the following canonical form:*

$$\left(\sum_{v \in V} [b_v]v \right) + \left(\sum_{u, v \in V} [b_{uv}](u \rightarrow v) \right), \quad (4.3)$$

where:

1. V is a subset of singleton graphs that appear in the original TPG;
2. for all $v \in V$, b_v are canonical forms of Boolean expressions and are distinct from 0;

3. for all $u, v \in V$, b_{uv} are canonical forms of Boolean expressions such that $b_{uv} \Rightarrow b_u \wedge b_v$;

4. for all $u, v, w \in V$, $b_{uv} \wedge b_{vw} \Rightarrow b_{uw}$.

Proof. (i) First we prove that any TPG can be converted to the form (4.3).

We can convert the expression into the canonical form (4.1), which satisfies the requirements 1–3. Then we iteratively apply the following transformation, while possible: If for some $u, v, w \in V$, $b_{uv} \wedge b_{vw} \Rightarrow b_{uw}$ does not hold (i.e. requirement 4 is violated), we replace the subexpression $[b_{uw}](u \rightarrow w)$ with $[b_{uw}^{new}](u \rightarrow w)$ where $b_{uw}^{new} \stackrel{\text{df}}{=} b_{uw} \vee (b_{uv} \wedge b_{vw})$. Observe that after this the requirement 4 will hold for u, v and w , and the requirement 3 remains satisfied, i.e. $b_{uw}^{new} \Rightarrow b_u \wedge b_w$ due to $b_{uv} \Rightarrow b_u \wedge b_v$, $b_{vw} \Rightarrow b_v \wedge b_w$ and $b_{uw} \Rightarrow b_u \wedge b_w$. Moreover, the resulting expression will be equivalent to the one before this transformation due to the following equality (see [42] for the proof):

$$\begin{aligned} & \text{If } v \neq \varepsilon \text{ then } [b_{uv}](u \rightarrow v) + [b_{vw}](v \rightarrow w) = \\ & = [b_{uv}](u \rightarrow v) + [b_{vw}](v \rightarrow w) + [b_{uv} \wedge b_{vw}](u \rightarrow w). \end{aligned}$$

This iterative process converges, as there can be only finitely many expressions of the form (4.3) (recall that we assume that the predicates within the conditional operators are always in some canonical form), and each iteration replaces some predicate b_{uw} with a greater one b_{uw}^{new} , in the sense that b_{uv} strictly subsumes b_{uw}^{new} (i.e. $b_{uv} \Rightarrow b_{uw}^{new}$ and $b_{uv} \not\equiv b_{uw}^{new}$ always hold), i.e. no predicate can be repeated during these iterations.

(ii) We now show that (4.3) is a canonical form, i.e. if $L = R$ then their canonical forms $\text{can}(L)$ and $\text{can}(R)$ coincide.

For the sake of contradiction, assume this is not the case. Then we consider two cases (all possible cases are symmetric to one of these two).

1. $\text{can}(L)$ contains a literal $[b_v]v$ whereas $\text{can}(R)$ either contains a literal $[b'_v]v$ with $b'_v \neq b_v$ or does not contain any literal corresponding to v , in which case we say that it contains a literal $[b'_v]v$ with $b'_v = 0$. Then for some values of parameters one of the graphs will contain vertex v while the other will not.
2. $\text{can}(L)$ and $\text{can}(R)$ have the same set V of vertices, but $\text{can}(L)$ contains a

subexpression $[b_{uv}](u \rightarrow v)$ and $\text{can}(R)$ contains a subexpression $[b'_{uv}](u \rightarrow v)$ with $b'_{uv} \neq b_{uv}$. Then for some values of parameters one of the graphs will contain the arc (u, v) while the other will not. Since the transitive closures of the graphs must be the same due to $\text{can}(L) = L = R = \text{can}(R)$, the other graph must contain a path $t_1 t_2 \dots t_n$ where $u = t_1$, $v = t_n$ and $n \geq 3$; w.l.o.g., we assume that $t_1 t_2 \dots t_n$ is a shortest such path. Hence, the canonical form (4.1) would contain the subexpressions $[b_{t_i t_{i+1}}](t_i \rightarrow t_{i+1})$, $i = 1 \dots n-1$, and moreover $\bigwedge_{i=1}^{n-1} b_{t_i t_{i+1}} \neq 0$ for the chosen values of the parameters, and so $\bigwedge_{i=1}^{n-1} b_{t_i t_{i+1}} \neq 0$. But then the iterative process above would have added to the canonical form the missing subexpression $[b_{t_1 t_2} \wedge b_{t_2 t_3}](t_1 \rightarrow t_3)$, as the corresponding predicates $\neq 0$. Hence, for the chosen values of the parameters, there is an arc (t_1, t_3) , contradicting the assumption that $t_1 t_2 \dots t_n$ is a shortest path between u and v .

In both cases there is a contradiction with $L = R$. \square

The process of constructing the canonical form (4.3) of a TPG from the canonical form (4.1) of a PG corresponds to computing the transitive closure of the adjacency matrix. As the entries of this matrix are predicates rather than Boolean values, this has to be done symbolically. This is always possible, as each entry of the resulting matrix can be represented as a finite Boolean expression depending on the entries of the original matrix only.

By the same reasoning as in the previous section, we can conclude that the following result holds.

Theorem 4 (Soundness, Minimality and Completeness). *The set of axioms of TPG-algebra is sound, minimal and complete w.r.t. TPGs.*

4.5 Case study

In this section we consider a practical case study from the domain of hardware synthesis. We apply PG-algebra for a formal and compositional approach to system design. It also allows us to rigorously manipulate specifications, in particular, algebraically simplify them.

In Chapter 5 we use the transitive version of PG-algebra – TPG-algebra – in application to the problem of instruction set design and encoding.

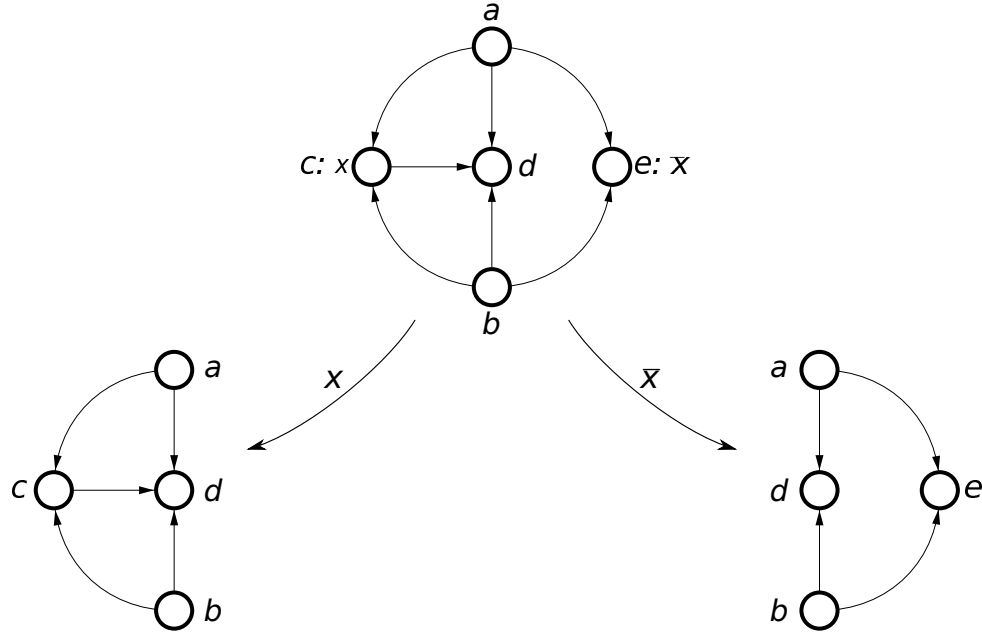


Figure 4.5: The PG from Fig. 4.3 simplified using the Closure axiom, together with its specialisations

4.5.1 Phase encoders

This section demonstrates the application of PG-algebra to designing the *multiple rail phase encoding* controllers [17]. They use several wires for communication, and data is encoded by the order of occurrence of transitions in the communication lines. Fig. 4.6(a) shows an example of a data packet transmission over a 4-wire phase encoding communication channel. The order of rising signals on wires indicates that permutation $abdc$ is being transmitted. In total it is possible to transmit any of the $n!$ different permutations over an n -wire channel in one communication cycle. This makes the multiple rail phase encoding protocol very attractive for its information efficiency [45].

Phase encoding controllers contain an exponential number of behavioural scenarios w.r.t. the number of wires, and are very difficult for specification and synthesis using conventional approaches. In this section we apply PG-algebra to specification of an n -wire *matrix phase encoder* – a basic phase encoding controller that generates a permutation of signal events given a matrix representing the order of the events in the permutation.

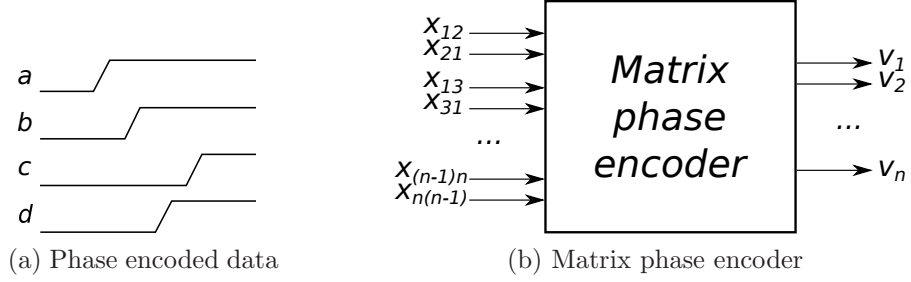


Figure 4.6: Multiple rail phase encoding

Fig. 4.6(b) shows the top-level view of the controller's structure. Its inputs are $\binom{n}{2}$ dual-rail ports that specify the order of signals to be produced at the controller's n output wires. The inputs of the controller can be viewed as an $n \times n$ Boolean matrix (x_{ij}) with diagonal elements being 0. The outputs of the controller will be modelled by n actions $v_i \in \mathcal{A}$. Whenever $x_{ij} = 1$, event v_i must happen before event v_j . It is guaranteed that x_{ij} and x_{ji} cannot be 1 at the same time, however, they can be simultaneously 0, meaning that the relative order of the events is not known yet and the controller has to wait until $x_{ij} = 1$ or $x_{ji} = 1$ is satisfied (other outputs for which the order is already known can be generated meanwhile).

The overall specification of the controller is obtained as the overlay $\sum_{1 \leq i < j \leq n} H_{ij}$ of fixed-size expressions H_{ij} , modelling the behaviour of each pair of outputs. In turn, each H_{ij} is an overlay of three possible scenarios:

1. If $x_{ij} = 1$ (and so $x_{ji} = 0$) then there is a causal dependency between v_i and v_j , described using the PG-algebra sequence operator: $v_i \rightarrow v_j$.
2. If $x_{ji} = 1$ (and so $x_{ij} = 0$) then there is a causal dependency between v_j and v_i : $v_j \rightarrow v_i$.
3. If $x_{ij} = x_{ji} = 0$ then neither v_i nor v_j can be produced yet; this is expressed by a circular wait condition between v_i and v_j : $v_i \rightarrow v_j + v_j \rightarrow v_i$.¹

We prefix each of the scenarios with its precondition and overlay the results:

$$H_{ij} = [x_{ij} \wedge \overline{x_{ji}}](v_i \rightarrow v_j) + [x_{ji} \wedge \overline{x_{ij}}](v_j \rightarrow v_i) + [\overline{x_{ij}} \wedge \overline{x_{ji}}](v_i \rightarrow v_j + v_j \rightarrow v_i).$$

¹There are other ways to describe this scenario, e.g. by creating self-loops $v_i \rightarrow v_i + v_j \rightarrow v_j$.

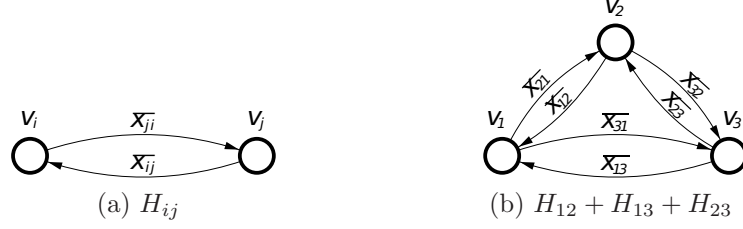


Figure 4.7: PGs related to matrix phase encoder specification

Using the rules of PG-algebra, we can simplify this expression to

$$[\overline{x_{ji}}](v_i \rightarrow v_j) + [\overline{x_{ij}}](v_j \rightarrow v_i),$$

or, using the conditional sequence operator, to

$$[\overline{x_{ij}} \vee \overline{x_{ji}}](v_i \xrightarrow{\overline{x_{ji}}} v_j + v_j \xrightarrow{\overline{x_{ij}}} v_i).$$

Now, bearing in mind that condition $[\overline{x_{ij}} \vee \overline{x_{ji}}]$ is assumed to hold in the proper controller environment (x_{ij} and x_{ji} cannot be 1 simultaneously), we can replace it with [1] and drop it. The resulting expression can be graphically represented as shown in Fig. 4.7(a). An example of an overall controller specification $\sum_{1 \leq i < j \leq n} H_{ij}$ for the case when $n = 3$ is shown in Fig. 4.7(b). The synthesis of this specification to a digital circuit can be performed in a way similar to [45].

4.6 Machine-assisted formalisation of Parametrised Graph theory

While developing mathematical theories and proofs it is important to maintain logical soundness. Even if the proof correctness may be obvious to its author, the peer researchers are often unable (because the proof is not detailed enough) or not willing (because the proof is too involved) to verify it rigorously. To avoid such problems we have decided to encode the theory in a formal system so that only definitions would require careful inspection, with proofs being checked automatically.

This section uses Agda [49] – a programming language and proof assistant based on the Martin-Löf type theory – for formalization of Parametrised Graphs theory.

The section additionally describes the algorithm for conversion of PG formulae to normal form and shows that the correctness of the algorithm has been verified.

The section extensively uses the syntax of Agda and references several definitions from the Agda standard library [18].

4.6.1 Graph Algebra

We start with defining an algebraic structure of non-parametrised graphs, to extend them with conditions later.

We define graph algebra as an algebraic structure over a set \mathbf{G} with an equivalence relation \approx supporting the following operations:

- An empty graph, denoting no actions.

$$\varepsilon : \mathbf{G}$$

- Graph overlay, denoting the parallel composition of actions from both graphs.

$$+_{} : \mathbf{G} \rightarrow \mathbf{G} \rightarrow \mathbf{G}$$

- Graph sequencing, denoting the causal dependency between actions in the first graph and in the second graph.

$$_{\gg} : \mathbf{G} \rightarrow \mathbf{G} \rightarrow \mathbf{G}$$

Additionally, the operations must satisfy the following properties:

- Overlay is commutative and associative.

$$+\text{assoc} : \forall p q r \rightarrow (p + q) + r \approx p + (q + r)$$

$$+\text{comm} : \forall p q \rightarrow p + q \approx q + p$$

- Sequencing is associative.

$$\begin{aligned} \gg \text{assoc} : \forall p q r \rightarrow \\ (p \gg q) \gg r \approx p \gg (q \gg r) \end{aligned}$$

- Empty graph is a no-op in relation to sequencing.

$$\gg \text{identity}^l : \forall p \rightarrow \varepsilon \gg p \approx p$$

$$\gg \text{identity}^r : \forall p \rightarrow p \gg \varepsilon \approx p$$

- Sequencing distributes over overlay.

$$\text{distrib}^l : \forall p \ q \ r \rightarrow$$

$$p \gg (q + r) \approx p \gg q + p \gg r$$

$$\text{distrib}^r : \forall p \ q \ r \rightarrow$$

$$(p + q) \gg r \approx p \gg r + q \gg r$$

- Sequence of more than two actions may be decomposed into shorter sequences, forming the original sequence with overlay.

$$\text{decomposition} : \forall p \ q \ r \rightarrow$$

$$p \gg q \gg r \approx p \gg q + p \gg r + q \gg r$$

Derived theorems

The following theorems has been derived from the axioms:

- Empty graph is a no-op in relation to overlay.

$$+\text{identity} : \forall p \rightarrow p + \varepsilon \approx p$$

- Overlay is idempotent.

$$+\text{idempotence} : \forall p \rightarrow p + p \approx p$$

- Absorption.

$$\text{absorption}^l : \forall p \ q \rightarrow p \gg q + p \approx p \gg q$$

$$\text{absorption}^r : \forall p \ q \rightarrow p \gg q + q \approx p \gg q$$

4.6.2 Parametrised Graphs

The graph algebra introduced in the previous subsection can only describe static event dependencies. To describe complex dynamic systems one has to consider the conditional behaviour as well. To do this, we have extended the graph algebra by annotating the graphs with conditions. Given a set \mathbf{G} of the parametrised graphs and a set \mathbf{B} of all the possible Boolean conditions, together with the following operations:

$$_ \vee _ : \mathbf{B} \rightarrow \mathbf{B} \rightarrow \mathbf{B}$$

$$_ \wedge _ : \mathbf{B} \rightarrow \mathbf{B} \rightarrow \mathbf{B}$$

$$\neg : \mathbf{B} \rightarrow \mathbf{B}$$

$$\top : \mathbf{B}$$

$$\perp : \mathbf{B}$$

we require a new operation called *condition*:

$$[_]_ : \mathbf{B} \rightarrow \mathbf{G} \rightarrow \mathbf{G}$$

The condition operation must have the following properties:

$$\text{true-condition} : \forall x \rightarrow [\top] x \approx x$$

$$\text{false-condition} : \forall x \rightarrow [\perp] x \approx \varepsilon$$

$$\text{and-condition} : \forall f g x \rightarrow [f \wedge g] x \approx [f] [g] x$$

$$\text{or-condition} : \forall f g x \rightarrow [f \vee g] x \approx [f] x + [g] x$$

$$\text{conditional+} : \forall f x y \rightarrow [f] (x + y) \approx [f] x + [f] y$$

$$\text{conditional}\gg : \forall f x y \rightarrow [f] (x \gg y) \approx [f] x \gg [f] y$$

We say that there is a *parametrised graph algebra* on a set \mathbf{G} with a condition set \mathbf{B} if there is a graph algebra on \mathbf{G} , a Boolean algebra on \mathbf{B} and a condition operator satisfying the requirements above.

Derived Theorems

The following theorems has been derived for the Parametrised Graph algebra.

Choice propagation. If we have a choice between similar subgraphs, we can factor out the similarity and propagate choice onto the differing parts.

$$\begin{aligned}
\text{choice-propagation}_1 & : \forall b \, p \, q \, r \rightarrow \\
& [b] (p \gg q) + [\neg b] (p \gg r) \approx p \gg ([b] q + [\neg b] r) \\
\text{choice-propagation}_2 & : \forall b \, p \, q \, r \rightarrow \\
& [b] (p \gg r) + [\neg b] (q \gg r) \approx ([b] p + [\neg b] q) \gg r
\end{aligned}$$

Condition regularisation. A sequence of conditional events can be rewritten as an overlay of simpler terms.

$$\begin{aligned}
\text{condition-regularisation} & : \forall f \, g \, p \, q \rightarrow \\
& [f] p \gg [g] q \approx [f] p + [g] q + [f \wedge g] (p \gg q)
\end{aligned}$$

Strengthened condition regularisation. This generalizes the regularisation theorem by allowing any z containing all the edges between p and q to be used instead of $p \gg q$.

$$\begin{aligned}
\text{condition-regularisation}_s & : \forall f \, g \, p \, q \, z \\
& \rightarrow p \gg q \approx p + q + z \\
& \rightarrow [f] p \gg [g] q \approx [f] p + [g] q + [f \wedge g] z
\end{aligned}$$

4.6.3 Parametrised Graph formulae

To perform automated manipulations of PG algebra formulae, we describe the formulae as an algebraic data type in the following way.

$$\begin{aligned}
\text{data PGFormula} & : \text{Set where} \\
\text{+}_- & : (x \, y : \text{PGFormula}) \rightarrow \text{PGFormula} \\
\text{>>}_- & : (x \, y : \text{PGFormula}) \rightarrow \text{PGFormula} \\
\varepsilon & : \text{PGFormula} \\
\text{var} & : (a : \mathbf{A}) \rightarrow \text{PGFormula} \\
[_]_- & : (c : \mathbf{B}) \rightarrow \text{PGFormula} \rightarrow \text{PGFormula}
\end{aligned}$$

Here \mathbf{A} is a set of graph variables and \mathbf{B} is a set of condition variables. We also have a constructor of **PGFormula** corresponding to each of the algebra operations and an additional constructor to reference the free variables. This way we can construct the formulae in a straightforward way:

var "x" + var "y" \gg var "z"

Formula evaluation then is catamorphism of PGFormula, replacing constructor applications with the corresponding algebra operations and **var** constructors with the actual variable values.

```
pg-eval : { A B G : Set }
  → ( _+_s_ _>>_s_ : G → G → G )
  → ( ε_s : G )
  → ( [_]_s_ : B → G → G )
  → ( var_s : A → G )
  → PGFormula A B
  → G
```

We use the same technique to define the **BoolFormula** data structure, with constructors $_ \wedge _$, $_ \vee _$, $\neg _$, \top , \perp and **var**.

4.6.4 Formula equivalence

Naturally, it is possible to write the same mathematical function in many structurally different, but logically equivalent ways. Here we define a notion of PG formula equivalence. We say that two formula are equivalent iff they can be structurally transformed one into the other by the set of rules corresponding to the equality rules of PG algebra. We express this with an indexed inductive data family by explicitly enumerating all the important constructors.

```
data _≈_ : PGFormula (BoolFormula B) V
  → PGFormula (BoolFormula B) V → Set where
  +assoc : ∀ p q r → (p + q) + r ≈ p + (q + r)
  +comm : ∀ p q → p + q ≈ q + p
  >> assoc : ∀ p q r → (p >> q) >> r ≈ p >> (q >> r)
  ...
```

This definition allows for convenient formula manipulation, without mentioning its semantics. However, the meaning of this definition is dubious because it was constructed manually without any mention of PG Algebra. To connect the formulae

equivalence with an algebra object equivalence, we have defined the proper equivalence relation on formulae, in terms of their semantics. We say that equivalent formulae must give equivalent results for any algebra they are evaluated in.

$$\begin{aligned}
 f1 \approx^s f2 &= \\
 \forall G \rightarrow (\text{algebra} : \text{PGAlgebra } G) \rightarrow (f : V \rightarrow G) \rightarrow \\
 \text{eval algebra } f f1 &\approx \text{eval algebra } f f2
 \end{aligned}$$

Here we assume that `eval algebra` applies `pg-eval` to all of the `algebra` operations. Now we can show that our easier to use equivalence relation is equivalent to the semantics-based definition:

$$\begin{aligned}
 \approx \rightarrow \approx^s &: \forall f g \rightarrow f \approx g \rightarrow f \approx^s g \\
 \approx^s \rightarrow \approx &: \forall f g \rightarrow f \approx^s g \rightarrow f \approx g
 \end{aligned}$$

4.6.5 Normal form

We say that a normal form (NF) of PG formula (PG) is an overlay of literals (Lit) where each literal is a **Node** annotated with a condition and each node is either a variable (V) or two variables connected with a sequence operator. We encode these definitions assuming Boolean formulae (BF) as conditions.

$$\begin{aligned}
 \text{Node} &= V \uplus V \times V \\
 \text{Lit} &= \text{Node} \times \text{BF} \\
 \text{NF} &= \text{List Lit}
 \end{aligned}$$

So far we have defined the structure of those types without formally saying anything about their semantics. We define the semantics for them by providing a corresponding Parametrised Graph Formulae (PG).

A **Node**, depending on its constructor, corresponds to either a single variable or two variables connected via the sequence operator.

$$\begin{aligned}
 \text{fromNode} &: \text{Node} \rightarrow \text{PG} \\
 \text{fromNode } (\text{inj}_1 x) &= \text{var } x \\
 \text{fromNode } (\text{inj}_2 (x, y)) &= \text{var } x \gg \text{var } y
 \end{aligned}$$

A **Lit** of the form $(node, condition)$ corresponds to the formula $[condition] \text{ node}$.

```
fromLit : Lit → PG
fromLit (node, cond) = [cond] fromNode node
```

NF corresponds to the overlay of all of its literals.

```
fromNF : NF → PG
fromNF = foldr _+_ ε ∘ map fromLit
```

4.6.6 Normalisation algorithm

To automate the translation of formulae to normal form we have developed the algorithm presented in this subsection.

The top-level normalisation function traverses the PG formula recursively, normalising all of the subformulae and combining them with the appropriate functions ($_{-+NF}$ for $+$, $_{->>NF}$ for \gg , etc.).

```
normalise : PG → NF
normalise = pg-eval
  _+_NF_
  _->>NF_
  []
  addCondition
  fromVar
```

The individual functions manipulating normal forms are implemented in the following way.

- The normal form of ε is empty list.
- The normal form of a variable literal x is a singleton list containing $[T] \ x$.

```
fromVar : V → NF
fromVar x = (inj1 x, T) :: []
```

- Overlay of two normal forms is concatenation of their literals.

$$\begin{aligned} _+_{\text{NF}} _ : \text{NF} &\rightarrow \text{NF} \rightarrow \text{NF} \\ a +_{\text{NF}} b &= a \text{ ++ } b \end{aligned}$$

- Sequence of two normal forms can be defined by applying the distributivity rules as a sum of pairwise sequencing of their literals.

$$\begin{aligned} _ \gg_r _ : \text{Lit} &\rightarrow \text{NF} \rightarrow \text{NF} \\ lit \gg_r [] &= lit :: [] \\ lit \gg_r (x :: xs) &= (lit \gg_1 x) + (lit \gg_r xs) \\ _ \gg_{\text{NF}} _ : \text{NF} &\rightarrow \text{NF} \rightarrow \text{NF} \\ [] \gg_{\text{NF}} b &= b \\ (h :: t) \gg_{\text{NF}} b &= (h \gg_r b) + (t \gg_{\text{NF}} b) \end{aligned}$$

- Sequence of two literals $[f] p \gg [g] q$ then can be defined as $[f] p + [g] q + [f \wedge g] r$ where $r = \text{newArrows } p \ q$ is the set of new arc nodes formed by sequencing the nodes p and q .

$$\begin{aligned} \text{vertices} : \text{Node} &\rightarrow \text{List V} \\ \text{vertices } (\text{inj}_1 x) &= x :: [] \\ \text{vertices } (\text{inj}_2 (x,y)) &= x :: y :: [] \\ \text{newArrows} : \text{Node} &\rightarrow \text{Node} \rightarrow \text{List Node} \\ \text{newArrows } p \ q &= \\ &\quad \text{map } \text{inj}_2 (\text{vertices } p \otimes \text{vertices } q) \\ _ \gg_1 _ : \text{Lit} &\rightarrow \text{Lit} \rightarrow \text{List Lit} \\ (p,f) \gg_1 (q,g) &= (p,f) :: (q,g) \\ &\quad :: (\text{map } (\text{flip } _ _ (f \wedge g)) (\text{newArrows } p \ q)) \end{aligned}$$

Here **vertices** n is the list of graph vertices contained in node n – one vertex when n is a vertex node and two vertices when n is an arc node.

newArrows $a \ b$ then is a set of arc nodes connecting each of the vertices in a to each of the vertices in b .

Algorithm Correctness

We define the correctness of normalisation by saying that the semantics of the resulting normal form must be equivalent to the original formula.

$$\text{normalise-correct} : \forall f \rightarrow f \approx \text{fromNF} (\text{normalise } f)$$

To prove this theorem we had to prove several simpler statements.

Normal form overlay is correct. That is, the semantics of concatenated normal forms is the overlay of their individual semantics.

$$\begin{aligned} +\text{correct} : \forall x \ y \rightarrow \\ \text{fromNF } x + \text{fromNF } y \approx \text{fromNF } (x +_{\text{NF}} y) \end{aligned}$$

This follows from the monoid structure of overlay.

The normal form sequencing functions are correct.

$$\begin{aligned} \gg \text{correct} : \forall x \ y \rightarrow \\ \text{fromNF } x \gg \text{fromNF } y \approx \text{fromNF } (x \gg_{\text{NF}} y) \end{aligned}$$

This relies on the right distributivity and the correctness of \gg_r .

$$\begin{aligned} \gg_r \text{correct} : \forall x \ y \rightarrow \\ \text{fromLit } x \gg \text{fromNF } y \approx \text{fromNF } (x \gg_r y) \end{aligned}$$

This relies on the left distributivity and the correctness of \gg_1 .

$$\begin{aligned} \gg_1 \text{correct} : \forall x \ y \rightarrow \\ \text{fromLit } x \gg \text{fromLit } y \approx \text{fromNF } (x \gg_1 y) \end{aligned}$$

The correctness of \gg_1 is proven by the following chain of reasoning.

$$\begin{aligned} & \text{fromLit } (x,f) \gg \text{fromLit } (y,g) \\ & \approx \langle \text{condition-regularisation}_s; \text{newArrows-correct} \rangle \\ & \text{fromLit } (x,f) + \text{fromLit } (y,g) \\ & + [f \wedge g] \text{sumNodes } (\text{newArrows } x \ y) \\ & \approx \langle \text{propagating the condition to the literals} \rangle \\ & \text{fromLit } (x,f) + \text{fromLit } (y,g) \end{aligned}$$

$$\begin{aligned}
& + \text{fromNF } (\text{map } (\text{flip } _ _ (f \wedge g)) (\text{newArrows } x \ y)) \\
& \approx \langle \text{by } +\text{assoc and definitions} \rangle \\
& \text{fromNF } ((x, f) \gg_1 (y, g))
\end{aligned}$$

The desired properties of the **newArrows** function are not as obvious as the properties of the other functions. We have formulated them as follows.

$$\begin{aligned}
& \text{newArrows-correct} : \forall x \ y \rightarrow \\
& \text{fromNode } x \gg \text{fromNode } y \approx \\
& \text{fromNode } x + \text{fromNode } y \\
& + \text{sumNodes } (\text{newArrows } x \ y)
\end{aligned}$$

where **sumNodes** = **foldr** $_ + _ \varepsilon \circ \text{map fromNode}$. Our proof of this property is less than elegant. We manually enumerate all four cases (vertex and vertex, vertex and arc, arc and vertex, arc and arc) and prove four theorems individually, using the decomposition, commutativity and associativity axioms. It's likely possible to simplify the proof by treating the nodes as lists of sequenced vertices and prove by induction on those lists, instead of enumerating all the possible cases.

4.7 Summary

This chapter has introduced the Parametrised Graphs (PG) formalism which extends the CPOG notation in a number of ways. First, it allows one to consider general directed graphs in place of simple partial orders. Second, PGs possesses an algebraic structure (PG algebra), suited for formal and machine-assisted manipulation of complex graphs.

We have identified a sub-algebra of PG algebra: transitive PG algebra. This algebra has transitively closed PG graphs as a model and is an efficient basis for defining and analysing causal dependencies.

The PG algebra is defined by a set of axioms. To establish the consistency of the axioms we have made a formal juxtaposition of PG graphs and the axiom set of the PG algebra showing that PG graphs is a valid model of PG algebra. To demonstrate the minimality of the axiomatization we have shown, with the help of the ALG tool [9] that removal of either axiom results in a differing set of models.

We have devised a procedure to automatically construct a normal form of a PG expression and proven it to be sound and terminating using the Agda proof checker [49]. The normal form is used to simplify PG expressions and can be used as a foundation for a wide range of graph manipulation techniques.

The next chapter presents an application of transitive parametrised graphs to encoding and decoding of microcontroller instruction opcodes.

Chapter 5

Processor instruction set encoding

Main contributions of this chapter are: firstly, it formulates several instruction set encoding problems in terms of the TPG model; secondly, it establishes that the problems can be reduced to the corresponding Boolean satisfiability (SAT) problem instances leading to their automated solution; thirdly, it demonstrates application of the TPG methodology at different stages of a processor design flow — from architectural-level specification, design and behavioural description of an instruction set to its encoding and synthesis of the physical implementation of the microcontroller. The chapter is organised as follows. Section 5.2 introduces the TPG encoding problem, overviews the existing encoding techniques and gives a brief introduction to the new technique of globally optimal encoding. A method for automated translation of the problems into SAT instances is explained in Section 5.3. It is followed by a processor design and synthesis example in Section 5.4 and conclusions.

The chapter is based on the results published in [40]. It shows how to represent processor instruction sets using TPG formalism and provides a ground for a concise formulation of several encoding problems, which are reducible to the well-known SAT problem and can be efficiently solved by modern SAT solvers. Application of all the presented techniques is demonstrated on a processor design example.

5.1 Introduction

Automated design and synthesis of both application-specific and general-purpose instruction-set processors is an active area of research [38] with instruction set ar-

chitecture (ISA) design seeing new techniques and improvements [65].

Synthesis of instruction sets is a particularly active research area. There are methods of automated derivation of ISA for a given platform according to available system components and for given software requirements. These methods eventually produce a set of instructions satisfying certain properties (orthogonality, completeness, regularity, etc.); instructions are grouped into categories and each category is allocated a certain opcode range within the code space [48]. At this point automation is typically stopped or becomes trivial: the instructions are given arbitrary codes within the allocated ranges. This limits performance due to instruction decoder circuitry overheads. The problem is usually approached by ad-hoc heuristics or application-specific optimisation techniques (see, for example, [35]).

The TPG notation has an application in CPU design with a natural hardware correspondence: it describes synchrous or asynchronous control logic over the units defined by graph vertices. Such logic is commonly called a *decoder* – a circuit that takes an instruction code word and produces control signals for CPU components.

Note that the graph we are interested in when studying asynchronous control logic is the *causality graph*, the graph where the nodes correspond to events (CPU component activations) and arcs correspond to causal dependencies. Causality relation is naturally transitive, and the causality graph is parameterised on code word, exactly the kind of graphs you can describe with the TPG formalism.

We are going to address the problem of TPG *encoding* (related to CPOG encoding [39]) – the process of identification of an efficient TPG from which a given set of partial orders can be obtained via the TPG specialisation mechanism. An efficient encoding makes it possible to describe systems in a compact functional form and apply structural synthesis methods which significantly improve performance of the whole design flow. These features make the model very efficient for representation and management of processor instruction sets in hardware and EDA software.

We discuss the known TPG encoding techniques:

- *binary encoding*, synthesising a TPG with the minimal number of variables;
- *matrix encoding*, where each vertex and arc are assigned a unique control variable;
- *one hot encoding*, attributing a variable to each unique original scenario;

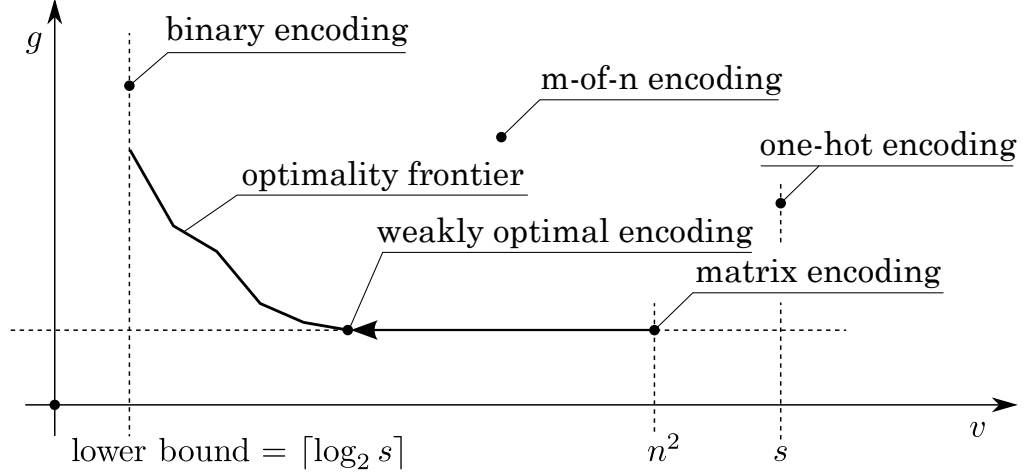


Figure 5.1: The relationship between the optimality frontier, as found by our algorithm, and the solutions proposed in [39]. On the axis g we plot the number of logic gates needed to describe the instruction decoder and the axis v corresponds to the number of variables used to encode the instructions.

- *weakly optimal encoding*¹ finds a solution with the minimal total number of variables among the solutions with fewest literals (ground terms); like the matrix encoding, it yields a TPG with single variable conditions; however, generally, it uses fewer variables in total.

We proceed to discuss how we improve upon this work by introducing a new algorithm. The algorithm is able to identify a globally optimal solution for a given fixed set of optimality criteria. The techniques discussed above are only able to identify various forms of local optimums. More precisely, the four existing techniques define just four points in the *solution space* constructed by our algorithm. Only one of these points is located on the Pareto optimality frontier, constructed by our algorithm (see Fig. 5.1).

The gist of the approach lies in a combination of binary search and Boolean satisfiability solving (SAT). Binary search travels through a binary tree of potential solutions using a choice function of the form $f \in D \rightarrow \mathbb{B}$ where D is domain limiting the space of explored solutions. In our case, such D may define either a set of gate counts of a decoder circuit or, alternatively, the set of number of variables used in TPG conditions. In a general case, f takes the following form:

¹In [39], this is referred to simply as *optimal*.

$$f(x) = \begin{cases} 1, & \text{when there exists a solution } S \text{ such that its cost is } x \\ 0, & \text{otherwise} \end{cases}$$

Such function f may be challenging to compute since the set from which solutions are drawn is typically large. Let v and g signify the maximum number of variables and gates; also let s be the number of scenarios encoded in the original TPG. Then the size of the solutions space is in the order of

$$(2(v + g))^{2g} (2^v)^s$$

For instance, for $v = 5$, $g = 5$ and $s = 5$ the formula above gives circa 10^{21} potential solutions to consider. Clearly, computing f with a brute-force approach is infeasible. Note, however, that f has a form of an existential quantifier binding identifiers to finite domains. This form of statement is perfectly suited for SAT solving. To bridge to the SAT-level notation we have to do binary vector encoding of scenario and variable indices, gate identifiers and cost indicators. The result is processed by a SAT solver and the output is a binary representation of a decoding circuit. In particular, we have used MINISAT [23] and CLASP [12] SAT solvers. As one example, we have been able to obtain an optimal solution for $v = 8$, $s = 8$ and $g \in 3..12$.

5.2 Problem statement

In microcontroller design, the encoding of an instruction set is a common reoccurring problem. A form of encoding optimality is essential to achieve a more compact instruction decoder logic, while, at the same time, offering higher decoding performance. The first step is to represent each instruction microcode as a TPG *scenario*. Such scenario is a program each instruction of each is a reference to a microcontroller unit. The directed graph semantics underlying PGs expresses the casual dependencies between events of unit firings; informally, this means that instruction execution steps may be ordered sequentially or concurrently.

An encoding scheme must ensure that every scenario is adequately represented in the composed graph. This property, called the *encoding correctness condition*, is formulated by the following statement

$$S_i = G \mid_{X=\theta_i}$$

here X is the set of free variables of PG G (Chapter 4), S_i is a scenario from a scenario set S and $X = \theta_i$ stands for $\bigwedge_{x \in X} x = \theta_i(x)$. The condition states that a scenario is properly decoded by applying an opcode-defined graph specialisation θ . Every encoding schema may be characterised by a specific choice of set X and encoding function θ . We refer to a tuple of (X, θ) as encoding of a scenario set S .

To uniquely determine, upto PG equivalence, and for encoding and scenario set a compositional graph G one also needs to consider the *property of composition minimality*:

$$G = \sum_{i < n} [X = \theta_i](G \mid_{X=\theta_i})$$

Informally, the condition states that G does not contain anything in addition to the encoding of scenarios S .

To give an intuition behind encoding we shall consider a simple case of encoding for two scenarios S_1, S_2 . This encoding is constructed by putting together, in a certain manner, graphs S_1 and S_2 . More precisely, we overlay the graphs of S_1 and S_2 using the $+$ operator:

$$G \stackrel{\text{df}}{=} [X = \theta_1]S_1 + [X = \theta_2]S_2$$

The overlaid graphs are conditioned by their opcodes. In the above, θ_1, θ_2 is a unique value of vector X identifying a given scenario.

In a general case, for n scenarios, this statement takes the following form:

$$G \stackrel{\text{df}}{=} \sum_{i < n} [X = \theta_i]S_i$$

Such encoding constructing preserves the property of composition minimality provided the correctness property also holds. Let us now briefly consider one particular encoding scheme - the *one hot* encoding - to the composition of scenarios S_1, S_2 :

$$G \stackrel{\text{df}}{=} [x_1 = 1 \wedge x_2 = 0]S_1 + [x_1 = 0 \wedge x_2 = 1]S_2$$

It is easy to check that one hot encoding satisfies the correctness condition. For instance, for the case of scenario S_1 , we can prove that the scenario can be obtained via graph G specialisation as follows:

$$S_1 = ([x_1 = 1 \wedge x_2 = 0]S_1 + [x_1 = 0 \wedge x_2 = 1]S_2) \mid_{x_1=1 \wedge x_2=0}$$

which is obviously correct by the definition of the specialisation operator $H|_p$.

A common measure of complexity of a Boolean formula f is denoted as $C(f)$ and is defined to be the total count of literals in it, e.g. $C(x \cdot z + y \cdot \bar{z}) = 4$, $C(1) = 0$, etc. This is the metric used in [39] to assess the produced encodings. Indeed, when applied to a single formula this metric often works well: the more literals the formula has, the larger the circuit to evaluate this formula. However, when used to assess multiple similar formulae or a single formula with repeating sub-formulae, this metric fails to account for the fact that the outputs of logic gates evaluating the subformulae can be reused. To take advantage of this fact we use the new metric $G(C)$, showing the total number of gates in the circuit needed to compute all of the conditions found in C . We use two versions of this metric, differing in type of gates allowed: one with AND/OR gates with all possible input and output inversions and another one with NAND-gates only.

5.2.1 Overview

In this section we briefly discuss the encoding approaches developed in [39]. We implement as SAT-based solution for the pre-existing techniques (weakly optimal encoding) and introduce a new, more powerful technique for identifying globally optimal solutions.

One hot

The one hot encoding scheme associates a unique indicator variable (a hot wire) with each scenario. The set of free variables is exactly that of the scenarios being encoded:

$$X = \mathbf{S}$$

where \mathbf{S} is the scenario set. The encoding is very simple: one sets i -th bit to 1 to select an i -th scenario and the rest must be reset to zero:

$$\theta_i(s) = (S_i = s), \quad \text{where } s, S_i \in \mathbf{S}$$

Matrix encoding

The matrix encoding allocates a unique variable for each edge and vertex of scenario graphs S_i . Hence, the set of free variables is given as

$$X = \{0\} \times V \cup \{1\} \times E$$

An encoding bit is set depending on whether a given edge or vertex are included in a given scenario s :

$$\begin{aligned} \theta_i(0, v) &= C_i(v), \quad \text{where } v \in V_i \\ \theta_i(1, e) &= D_i(e), \quad \text{where } e \in E_i \end{aligned}$$

where scenario S_i is defined by a PG of the form $(V_i, E_i, C_i, D_i, X_i)$.

Binary encoding

Binary encoding associates a scenario index with the value of binary number coded by an instruction opcode. One specific form of X can be given as

$$X = 1..\lceil \log |\mathbf{S}| \rceil$$

where \mathbf{S} is the scenario set.

The encoding function computes the k -th bit of i -th scenario opcode as a binary encoding of a natural number representing the scenario index:

$$\theta_i(k) = \left\lfloor \frac{i-1}{2^{k-1}} \right\rfloor \bmod 2, \quad \text{where } k \text{ is the bit position}$$

Weakly optimal

The optimal encoding with unconstrained code length, which we call weakly optimal encoding, tries to minimize the size of set X within the universe of solutions with

1-restricted conditions. We say that a graph is 1-restricted if all the edge and vertex conditions contain at most one literal. It has been shown in [39] that the problem of computing a weakly optimal encoding is NP-complete thus barring any attempts at brute force solutions. It is challenging to give a compact characterisation of optimal encoding in the terms of free variable set X and coding function θ . In the continuation of the Chapter we shall give a SAT formulation of this encoding approach.

5.2.2 Globally optimal encoding

The weakly optimal encoding method presented above generates the smallest PG description of a set of partial orders but the number of used variables cannot be controlled; in many practical cases it will use more variables than is affordable under the design or technology constraints. In this section we briefly describe a method for generating the smallest PG given a limit on the number of variables, i.e. given the required length of the instruction codes.

Let L be the given limit on the number of variables. We generate all non-trivial encoding constraints and try to satisfy them with opcode variables. Only L of them are free variables $X = \{x_1, x_2, \dots, x_L\}$; other variables $F = \{f_1, f_2, \dots, f_m\}$ are not free — they are expressed in terms of variables from $X \cup F$ using Boolean binary functions, e.g. $f_1 = x_1 + \overline{x_3}$, $f_2 = f_1 \cdot x_2$, etc. As L is fixed all we have to do is to minimise the number of non-free variables m . This minimisation problem requires exploration of large search space; fortunately, it still belongs to the NP complexity class and we can reduce it to the SAT problem: the solver has to ‘guess’ all the opcodes, formulae of variables in F , and allocation of all variables $X \cup F$ to the non-trivial encoding constraints.

5.3 SAT formulation

This section presents SAT formulations of the optimal encoding problems described in the previous section.

The Boolean satisfiability problem (SAT) is to decide whether a given Boolean formula $F(x_1, x_2, \dots, x_n)$ is satisfiable or not, i.e. if it is possible to find an assign-

ment of Boolean values $(\alpha_1, \alpha_2, \dots, \alpha_n) \in \{0, 1\}^n$ to the variables (x_1, x_2, \dots, x_n) which makes the formula true: $F(\alpha_1, \alpha_2, \dots, \alpha_n) = 1$. As SAT is a decision (not optimisation) problem, we define a cost function and use a binary search to minimise its value by calling the SAT solver with different cost constraints.

We have implemented all the techniques presented in this section in an automated software tool which uses MINISAT [23] and CLASP [12] as SAT solver engines. They operate on CNF (conjunctive normal form) representations of Boolean formulae. Since our SAT-instances are not necessarily given in CNF, we implemented their automated conversion to CNF formulae. This conversion introduces intermediate variables but the overall size of the obtained formula is linear with respect to the size of the given SAT-instance.

5.3.1 Weakly optimal encoding

To solve the weakly optimal encoding problem described in Subsection 5.2 we minimise the number of colours used for a conflict graph colouring. Minimisation is performed by solving a series of instances of the following decision problem.

Let $G = (V, E)$ be an extended conflict graph, where vertices V correspond to encoding constraints and edges $E \subseteq V \times V$ to conflicts between them. V contains both original $V_o = \{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$ and inverted $V_i = \{\bar{\mathbf{e}}_1, \bar{\mathbf{e}}_2, \dots, \bar{\mathbf{e}}_n\}$ constraints, such that $V = V_o \cup V_i$. The problem is to find a colouring of G which uses no more than m colours.

For every pair of vertices $(\mathbf{e}_k, \bar{\mathbf{e}}_k)$ we introduce a Boolean variable p_k and an integer number c_k whose values have to be found by the SAT solver: p_k indicates which of the two vertices is coloured – if $p_k = 1$ (resp. $p_k = 0$) then \mathbf{e}_k (resp. $\bar{\mathbf{e}}_k$) is coloured, while c_k represents the colour of the chosen vertex.

The SAT problem \mathcal{ENCODE} consists of four constraints:

$$\mathcal{ENCODE} = \mathcal{NUM} \cdot \mathcal{COL}_{oo} \cdot \mathcal{COL}_{oi} \cdot \mathcal{COL}_{ii}$$

where \mathcal{NUM} restricts colours such that $0 \leq c_k < m$. Encoding of numbers c_k in Boolean domain can be different, for example, if we use binary encoding we need $\lceil \log_2 m \rceil$ bits for each c_k . Implementation of \mathcal{NUM} depends on the chosen encoding;

its general form is:

$$\mathcal{NUM} = \prod_{1 \leq k \leq n} (0 \leq c_k) \cdot (c_k < m)$$

Constraints \mathcal{COL} check that adjacent vertices are assigned different colours:

$$\mathcal{COL}_{oo} = \prod_{(\mathbf{e}_j, \mathbf{e}_k) \in E \cap (V_o \times V_o)} (p_j \cdot p_k) \Rightarrow (c_j \neq c_k)$$

$$\mathcal{COL}_{oi} = \prod_{(\mathbf{e}_j, \bar{\mathbf{e}}_k) \in E \cap (V_o \times V_i)} (p_j \cdot \bar{p}_k) \Rightarrow (c_j \neq c_k)$$

$$\mathcal{COL}_{ii} = \prod_{(\bar{\mathbf{e}}_j, \bar{\mathbf{e}}_k) \in E \cap (V_i \times V_i)} (\bar{p}_j \cdot \bar{p}_k) \Rightarrow (c_j \neq c_k)$$

If we assume that complexity of comparison operations over numbers c_k is C , then the overall complexity of \mathcal{ENCODE} is $\Theta((|V| + |E|) \cdot C)$. In particular, in case of binary encodings² the complexity is $\Theta((|V| + |E|) \cdot \log m)$. Depending on the chosen number encodings, there are from $\Theta(|V| \cdot \log m)$ to $\Theta(|V| \cdot m)$ free variables.

If L is the minimum value of m for which formula \mathcal{ENCODE} is satisfiable then the optimal encoding uses L variables $X = \{x_1, x_2, \dots, x_L\}$. Values p_k and c_k which satisfy the formula are used to resolve encoding constraints \mathbf{e}_k in the following way: if $p_k = 1$ then \mathbf{e}_k is resolved by x_{c_k} , otherwise it is resolved by \bar{x}_{c_k} .

5.3.2 Globally optimal encoding

The version of the optimal encoding problem with constrained code length is significantly more complicated and computationally intensive. It requires finding a set of Boolean functions of L arguments (where L is the specified code length) and there are 2^{2^L} of them – it is impossible to explore search spaces of such magnitudes, e.g. 2^{2^8} roughly equals to the number of atoms in the universe. To cope with this, we reduce the search space to 2-argument Boolean functions only. From the practical point of view this is justified by the fact that most modern technology libraries contain only 2- or 3-input logic gates anyway. Importantly, every complex function can

²Formula $(a < b)$ for binary numbers comparison is $(a < b) = \bar{a}_0 \cdot b_0 + (a_0 = b_0) \cdot (a' < b')$ where a' and b' are obtained from a and b by removal of their most significant digits (a_0 and b_0); the formula is linear with respect to the lengths of a and b .

be represented as a composition of simpler ones, therefore our approach can find any function, albeit at the cost of introducing intermediate variables. This is similar to what actually happens during technology mapping and logic decomposition of functional components into hardware gates [14][19].

Formally, let $(\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n)$ be a set of encoding constraints defined in Subsection 5.2, S be the number of scenarios, and L be the required code length. We are looking for such a vector $(\psi_1, \psi_2, \dots, \psi_S)$ of L -bit encodings and n functions F_j , $1 \leq j \leq n$ such that $F_j(\psi_k) = \mathbf{e}_j[k]$ for every scenario $1 \leq k \leq S$ (unless $\mathbf{e}_j[k]$ is a don't care value).

We represent a set of functions F as a combinational circuit consisting of G 2-input Boolean gates, where G is the value to be minimised. An output of the circuit can be taken directly from one of its inputs or be produced by a gate. In addition, any output can be inverted:

$$F_j(\psi) = \text{select}(\text{Signals}(\psi), o\text{Selector}_j) \oplus \text{Inv}_j$$

Here $\text{Signals}(\psi)$ is a function computing all circuit signals including both circuit inputs (given by parameter ψ) and gate outputs, $o\text{Selector}_j$ is the number indicating which circuit signal is 'connected' to the j -th circuit output, function $\text{select}(V, k)$ selects k -th element from a given vector V , and $\text{Inv}_j = 1$ iff j -th circuit output is inverted. Implementation of function $\text{select}(V, k)$ depends on the encoding of k (we used one-hot encoding in this case, which allows for simpler implementation). Circuit signals are computed as follows:

$$\begin{cases} \text{Signals}(\psi) &= \text{Wires}_G(\psi) \\ \text{Wires}_k(\psi) &= \begin{cases} \psi & \text{if } k = 0 \\ \text{Wires}_{k-1}(\psi) \circ \text{Gate}_k(\psi) & \text{if } 0 < k \leq G \end{cases} \\ \text{Gate}_k(\psi) &= \text{arg}_{1,k}(\psi) \cdot \text{arg}_{2,k}(\psi) \\ \text{arg}_{j,k}(\psi) &= \text{select}(\text{Wires}_{k-1}(\psi), a\text{Selector}_{j,k}) \oplus \text{InvArg}_{j,k} \end{cases}$$

In other words, a set of wires is initially equal to the set of circuit inputs ($\text{Wires}_0(\psi) = \psi$) and then is iteratively extended by appending Gate_k to the previously computed set of wires Wires_{k-1} . Eventually, after G iterations we obtain the set of all signals

$Signals(\psi) = Wires_G(\psi)$. Every $Gate_k$ corresponds to an AND gate with possible input inversions (indicated by $InvArg_{0,k}$ and $InvArg_{1,k}$). Its arguments are selected by $aSelector_{0,k}$ and $aSelector_{1,k}$ from the set of wires computed in the previous iteration. This guarantees the absence of combinational loops.

As every signal in the circuit can be optionally inverted (by setting $Inv_j = 1$ or $InvArg_{j,k} = 1$), the resultant gate basis includes 8 logic gates: AND, OR, NAND, NOR, plus 4 other gates, obtained from these by inversion of exactly one of their inputs (they do not have commonly adopted names, apart, perhaps, from Boolean implication $x \Rightarrow y$ which corresponds to $OR(\bar{x}, y)$). We have also investigated a simpler basis, consisting of only NAND gates with no optional input inversions. The basis leads to smaller search space and works faster, but, as expected, produces larger circuits (see Figure 5.6 for a comparison of two bases on a processor example). If the NAND basis is used then free variables $InvArg_{j,k}$ can be dropped and the formulae for $Gate_k(\psi)$ and $arg_{j,k}(\psi)$ should be modified as follows:

$$\begin{cases} Gate_k(\psi) &= \overline{arg_{1,k}(\psi) \cdot arg_{2,k}(\psi)} \\ arg_{j,k}(\psi) &= select(Wires_{k-1}(\psi), aSelector_{j,k}) \end{cases}$$

We tried to extend the 8-gate basis by addition of gates XOR and XNOR but on practical examples it did not bring any benefit in terms of the number of used gates, while significantly increasing the computation time (due to additional free variables $IsXor_k$ and more complex $Gate_k(\psi)$ functions).

In case of the standard 8-gate basis the SAT solver has to assign the following free variables: $\psi_j[k]$ ($1 \leq j \leq S$, $1 \leq k \leq L$), Inv_j ($1 \leq j \leq n$), $InvArg_{0,j}$ and $InvArg_{1,j}$ ($1 \leq j \leq G$). Also it has to find numbers $oSelector_j$ ($1 \leq j \leq n$), $aSelector_{0,j}$ and $aSelector_{1,j}$ ($1 \leq j \leq G$). All other variables are derived.

The SAT problem \mathcal{ENCODE} consists of two constraints:

$$\mathcal{ENCODE} = \mathcal{NUM} \cdot \mathcal{EVAL}$$

Constraint \mathcal{NUM} restricts all the selectors to their domains:

$$\begin{aligned} \mathcal{NUM} = & \prod_{1 \leq j \leq n} (1 \leq oSelector_j) \cdot (oSelector_j \leq L + G) \cdot \\ & \cdot \prod_{\substack{1 \leq j \leq 2 \\ 1 \leq k \leq G}} (1 \leq aSelector_{j,k}) \cdot (aSelector_{j,k} < L + k) \end{aligned} \quad (5.1)$$

Constraint \mathcal{EVAL} checks that the circuit outputs satisfy the encoding constraints:

$$\prod_{\substack{1 \leq j \leq n \\ 1 \leq k \leq S}} (\mathbf{e}_j[k] \neq -) \Rightarrow (F_j(\psi_k) \Leftrightarrow \mathbf{e}_j[k])$$

If G_{min} is the minimum value of G for which formula \mathcal{ENCODE} is satisfiable then the optimal encoding is obtained in vectors ψ_j ($1 \leq j \leq S$), an encoding constraint \mathbf{e}_k is resolved by function $F_j(\psi)$, and the circuit which produces these functions contains G_{min} gates.

We tried binary and one-hot number encodings in our implementation. In both cases the complexity of the formula is $\Theta(S \cdot G \cdot (G + L))$. The number of free variables is $\Theta(S \cdot L + (G + n) \cdot C)$, where C is $\log(G + L)$ and $G + L$ for binary and one-hot encodings, respectively. In practice, one-hot encoding proved to be more efficient despite significantly larger number of free variables. This can be explained by the fact that one-hot encoding leads to simpler constraints.

5.3.3 Support for dynamic variables

A lot of practical applications require the use of *dynamic variables*, i.e. such variables that can change their values during execution of a partial order and affect its further execution flow [39]. An example of such application, a processor microcontroller, is discussed in the next section.

Dynamic variables manifest themselves as encoding constraints with non-constant elements, e.g. $\mathbf{e} = 110y1\bar{y}$, which means that in the fourth scenario the corresponding condition has to evaluate to some dynamic variable y and in the sixth scenario it has to evaluate to \bar{y} . To compute the optimal encoding with such non-constant constraints we have to modify the method from the previous subsection in the following way.

Let y be a dynamic variable. We generate formula \mathcal{ENCODE}_0 (resp. \mathcal{ENCODE}_1) using encoding constraints where y is replaced by 0 (resp. 1). Note that the free variables in both formulae have to be the same and we have to add y into the set of circuit inputs. Then we use the SAT solver to find an assignment that satisfies $\mathcal{ENCODE}_0 \cdot \mathcal{ENCODE}_1$. Interpretation of the resulting assignment is the same apart from the added input signal y . In case of more than one dynamic variable, the process should be repeated for each of them. Potentially this leads to an exponential explosion of the formula. Fortunately, the number of dynamic variables is rather small in practice, thus the explosion is not dramatic.

It is still possible to avoid the explosion of the formula by conversion of the problem into an instance of 2-QBF problem (a quantified Boolean formula with two quantifiers [55]):

$$\exists X \forall Y \mathcal{ENCODE}$$

where X represents the set of all free variables, and Y stands for the set of dynamic variables. However, conversion of a formula into 2-QBF does not necessarily reduce the computation time needed to find its satisfying assignment. Implementation of a tool based on a 2-QBF solver is a subject of future work.

The next section demonstrates application of this technique in a processor microcontroller design.

5.4 Processor design example

This section demonstrates application of TPG-algebra to designing processor microcontrollers. Specification of such a complex system as a processor has to start at the architectural level, which helps to manage the system complexity by structural abstraction [19].

Fig. 5.2 shows the architecture of an example processor. Separate *Program memory* and *Data memory* blocks are accessed via the *Instruction fetch* (IFU) and *Memory access* (MAU) units, respectively. The other two operational units are: *Arithmetic logic unit* (ALU) and *Program counter increment unit* (PCIU). The units are controlled using request-acknowledgement interfaces (depicted as bidirectional arrows) by the *Central microcontroller*, which is our primary design objective.

The processor has four registers: two general purpose registers A and B , *Program counter* (PC) storing the address of the current instruction in the program memory, and the *Instruction register* (IR) storing the *opcode* (operation code) of the current instruction. For the purpose of this chapter, the actual width of the registers (the number of bits they can store) is not important. ALU has access to all the registers via the register bus; MAU has access to general purpose registers only; IFU, given the address of the next instruction in PC, reads its opcode into IR; and PCIU is responsible for incrementing PC (moving to the next instruction). The microcontroller has access to the IR and ALU *flags* (information about the current state of ALU which is used in branching instructions).

Now we define the set of instructions of the processor. Rather than listing all the instructions, we describe classes of instructions with the same *addressing mode* [1] and the same execution scenario. As the scenarios here are partial orders of actions, we use TPG-algebra, and the corresponding TPGs are shown in Fig. 5.3.

ALU operation Rn to Rn An instruction from this class takes two operands stored in the general purpose registers (A and B), performs an operation, and writes the result back into one of the registers (so called *register direct addressing mode*). Examples: *ADD A, B* – addition $A := A + B$; *MOV B, A* – assignment $B := A$. ALU works concurrently with PCIU and IFU, which is captured by the expression $ALU + PCIU \rightarrow IFU$; the corresponding PG is shown in Fig. 5.3(a). 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*, *MOV*, or any other instruction from this class) because the scenario is the same from its point of view (it is the responsibility of ALU to detect which operation it has to perform according to the current opcode).

ALU operation #123 to Rn In this class of instructions one of the operands is a register and the other is a constant which is given immediately after the instruction opcode (e.g. *SUB A, #5* – subtraction $A := A - 5$), so called *immediate addressing mode*. At first, the constant has to be fetched into IR, modelled as $PCIU \rightarrow IFU$. Then ALU is executed concurrently with another increment of PC: $ALU + PCIU'$ (we use $'$ to distinguish the different occurrences of actions of the same unit). Finally, it is possible to fetch the next instruction into IR: IFU' . The overall scenario is then $PCIU \rightarrow IFU \rightarrow (ALU + PCIU') \rightarrow IFU'$.

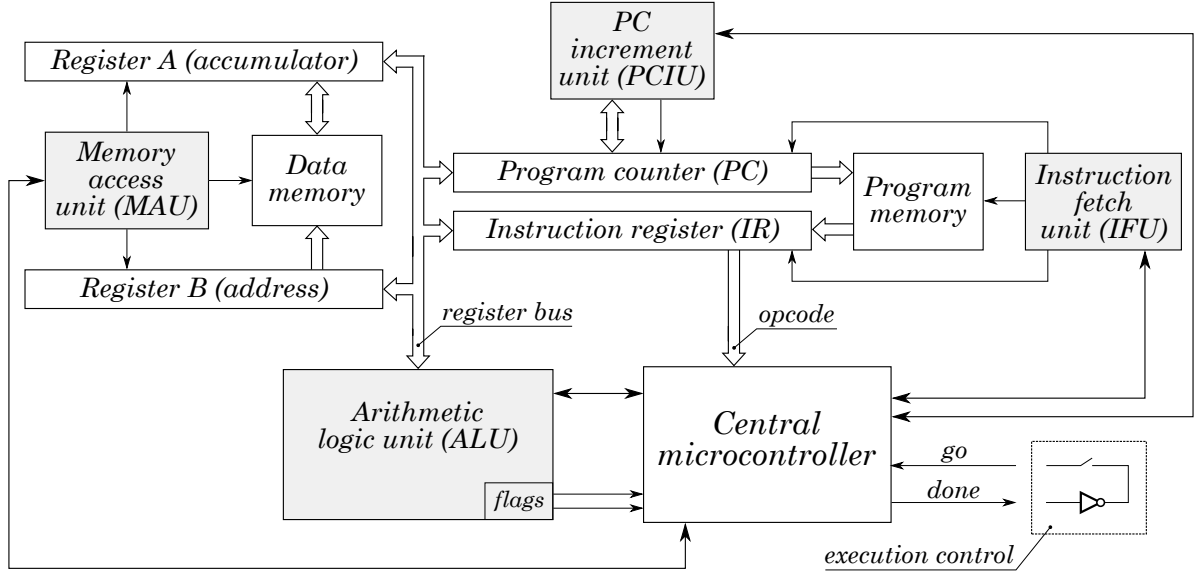


Figure 5.2: Architecture of an example processor

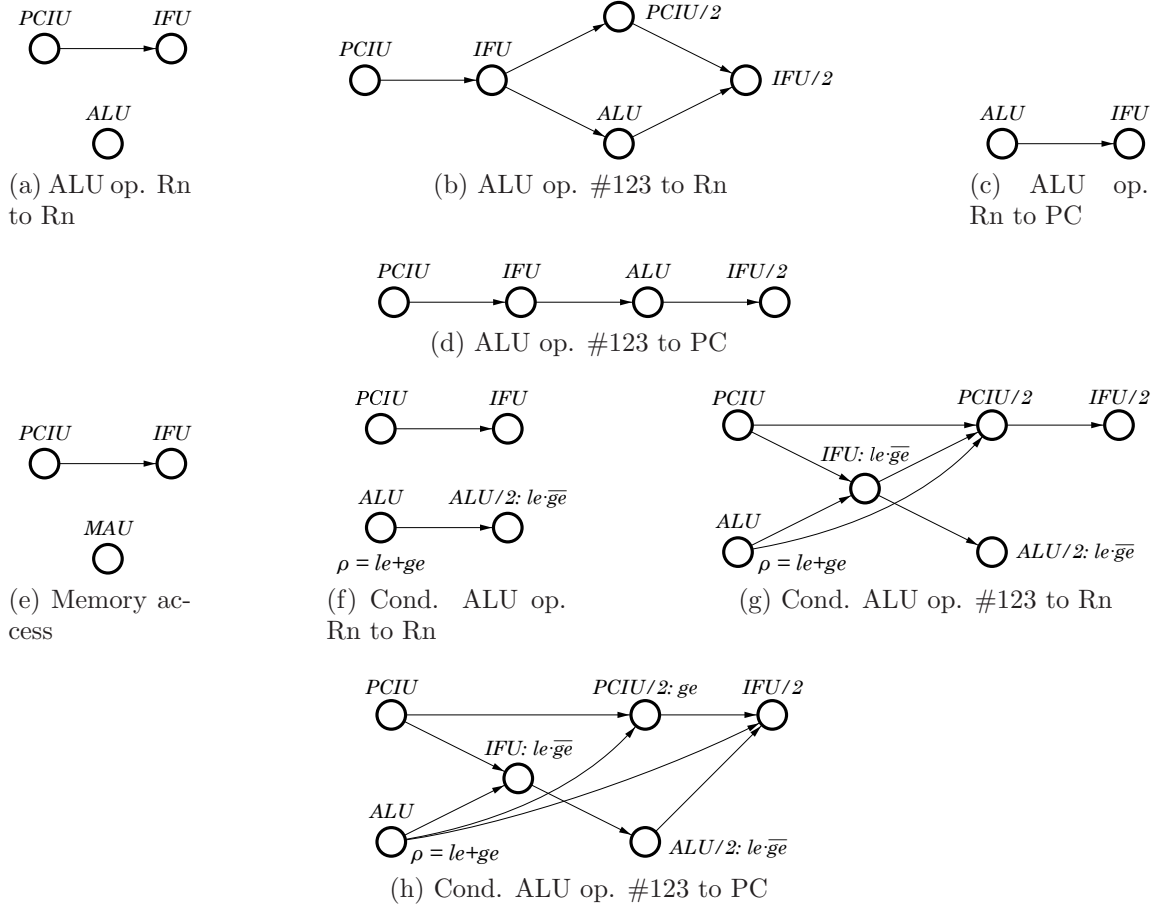


Figure 5.3: TPG specifications of instruction classes

ALU operation Rn to PC This class contains operations for unconditional branching, in which PC register is modified. Branching can be absolute or relative: *MOV PC, A* – absolute branch to address stored in register *A*, $PC := A$; *ADD PC, B* – relative branch to the address *B* instructions ahead of the current address, $PC := PC + B$. The scenario is very simple in this case: $ALU \rightarrow IFU$.

ALU operation #123 to PC Instructions in this class are similar to those above, with the exception that the branch address or offset is specified explicitly as a constant. The execution scenario is composed of : $PCIU \rightarrow IFU$ (to fetch the constant), followed by an ALU operation, and finally by another IFU operation, IFU' . Hence, the overall scenario is $PCIU \rightarrow IFU \rightarrow ALU \rightarrow IFU'$.

Memory access There are two instructions in this class: *MOV A, [B]* and *MOV [B], A*. They load/save register *A* from/to memory location with address stored in register *B*. Due to the presence of separate program and data memory access blocks, this memory access can be performed concurrently with the next instruction fetch: $PCIU \rightarrow IFU + MAU$.

Conditional instructions These three classes of instructions are similar to their unconditional versions above with the difference that they are performed only if the condition $A < B$ holds. The first ALU action compares registers *A* and *B*, setting the ALU flag *lt* (less than) according to the result of the comparison. This flag is then checked by the microcontroller in order to decide on the further scheduling of actions.

Rn to Rn This instruction conditionally performs an ALU operation with the registers (if the condition does not hold, the instruction has no effect, except changing the ALU flags). The operation starts with an ALU operation comparing *A* with *B*; depending on the result of this comparison, i.e. the status of the flag *lt*, the second ALU operation may be performed. This is captured by the expression $ALU \rightarrow [lt]ALU'$. Concurrently with this, the next instruction is fetched: $PCIU \rightarrow IFU$. Hence, the overall scenario is $PCIU \rightarrow IFU + ALU \rightarrow [lt]ALU'$.

#123 to Rn This instruction conditionally performs an ALU operation with a register and a constant which is given immediately after the instruction opcode (if the condition does not hold, the instruction has no effect, except changing the ALU flags). We consider the two possible scenarios:

- $A < B$ holds: First, ALU compares *A* and *B* concurrently with a PC incre-

Instructions class	Opcode: xyz
ALU Rn to Rn	000
ALU #123 to Rn	110
ALU Rn to PC	101
ALU #123 to PC	010
Memory access	100
C/ALU Rn to Rn	001
C/ALU #123 to Rn	111
C/ALU #123 to PC	011

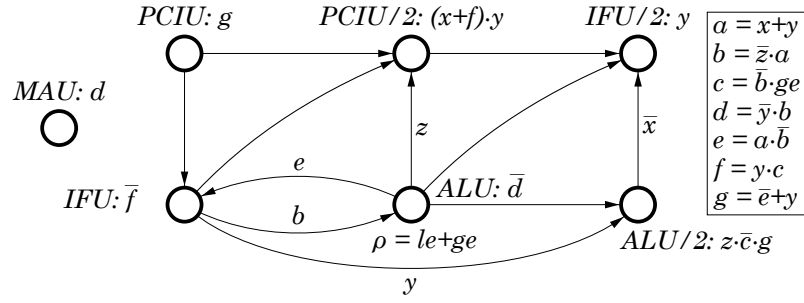


Figure 5.4: Optimal 3-bit instruction opcodes and the corresponding TPG specification of the microcontroller

ment; since $A < B$ holds, the ALU sets flag lt and the constant is fetched to the instruction register: $(ALU + PCIU) \rightarrow IFU$. After that PC has to be incremented again, $PCIU'$, and ALU performs the operation, ALU' . Finally, the next instruction is fetched (it cannot be fetched concurrently with ALU' as ALU is using the constant in IR): $(ALU' + PCIU') \rightarrow IFU'$.

- $A < B$ does not hold: First, ALU compares A and B concurrently with a PC increment; since $A < B$ does not hold, the ALU resets flag lt and the constant that follows the instruction opcode is skipped by incrementing the PC: $(ALU + PCIU) \rightarrow PCIU'$. Finally, the next instruction is fetched: IFU' .

Hence, the overall scenario is the overlay of the two subscenarios above prefixed with appropriate conditions (here we denote the predicate $A < B$ by lt):

$$[lt]((ALU + PCIU) \rightarrow IFU \rightarrow (ALU' + PCIU') \rightarrow IFU') + \\ + [\overline{lt}]((ALU + PCIU) \rightarrow PCIU' \rightarrow IFU').$$

This expression can be simplified using the rules of TPG-algebra:³

$$(ALU + PCIU) \rightarrow [lt]IFU \rightarrow (PCIU' + [lt]ALU') \rightarrow IFU'.$$

#123 to PC This instruction performs a conditional branching in which the branch address or offset is specified explicitly as a constant. We consider the two possible scenarios:

- $A < B$ holds: First, ALU compares A and B concurrently with a PC increment; since $A < B$ holds, the ALU sets flag lt and the constant is fetched to the instruction register: $(ALU + PCIU) \rightarrow IFU$. After that ALU performs the branching operation by modifying PC, ALU' . After PC is changed, the next instruction is fetched, IFU' .
- $A < B$ does not hold: the scenario is exactly the same as in the **#123 to Rn** case when $A < B$ does not hold.

Hence, the overall scenario is the overlay of the two subscenarios above prefixed with appropriate conditions (here we denote the predicate $A < B$ by lt):

$$[lt]((ALU + PCIU) \rightarrow IFU \rightarrow ALU' \rightarrow IFU') + \\ + [\overline{lt}]((ALU + PCIU) \rightarrow PCIU' \rightarrow IFU').$$

This expression can be simplified using the rules of TPG-algebra:

$$(ALU + PCIU) \rightarrow ([\overline{lt}]PCIU' + [lt](IFU \rightarrow ALU')) \rightarrow IFU'.$$

The overall specification of the microcontroller can now be obtained by prefixing the scenarios with appropriate conditions and overlaying them. These conditions can be naturally derived from the instruction opcodes. The opcodes can be either imposed externally or chosen with the view to optimise the microcontroller. In the

³This case illustrates the advantage of using the new hierarchical approach that allows to specify the system as a composition of scenarios and formally manipulate them in an algebraic fashion. In our previous work [40], the CPOG for this class of instruction was designed monolithically, and because of this the arc between ALU' and IFU' was missed. Adding this arc not only fixes the dangerous race between these two blocks, but also leads to a smaller microcontroller due to the additional similarity between TPGs for this class of instructions and for the one described below.

Instructions class	Trivial encoding	Optimal encoding		
		$L = 8$	$L = 3$	$L = 5$
ALU R_n to R_n	000	00000000	000	00000
ALU #123 to R_n	001	01001010	110	01001
ALU R_n to PC	010	00010001	101	00010
ALU #123 to PC	011	01000010	010	01000
Memory access	100	01000100	100	00100
C/ALU R_n to R_n	101	00100000	001	10000
C/ALU #123 to R_n	110	10111010	111	11001
C/ALU #123 to PC	111	10110010	011	11000

Table 5.1: Synthesised instruction codes

latter case, TPG-algebra and TPGs allow for a formal statement of this optimisation problem and aid in its solving; in particular, the sizes of the TPG-algebra expression or TPG are useful measures of microcontroller complexity (there is a compositional translation from a TPG-algebra expression into a linear-size circuit). Note that it is natural to use three bits for opcodes as there are eight classes of instructions, and give an example of optimal 3-bit encoding in the table in Fig. 5.4; the TPG specification of the corresponding microcontroller is shown in the right part of this figure (the TPG-algebra expression is not shown because of its size).

5.4.1 Instructions encoding

Now the instructions have to be encoded. The simplest way to do this is to use the binary encoding scheme, i.e. assign opcodes $\{000, \dots, 111\}$ to the instructions in arbitrary order as shown in Table 5.1 (column ‘Trivial encoding’). This is not optimal in terms of area and latency of the final microcontroller implementation. To obtain the smallest possible TPG specification one has to apply the optimal encoding procedure from Subsection 5.2.1. Generated opcodes have 8 bits instead of 3 (shown in column ‘Optimal encoding’ of the same table). Whether 8 bit opcodes are affordable or not depends on the chosen width of instruction register IR and other design parameters. If it is not possible to use 8 bit opcodes one can try to apply the constrained synthesis problem from Subsection 5.2.2 and generate instruction codes of required length $3 \leq L < 8$ (it is not possible to use less than 3 bits, and there is no sense in setting $L \geq 8$ because the optimal encoding uses 8 bits). We show

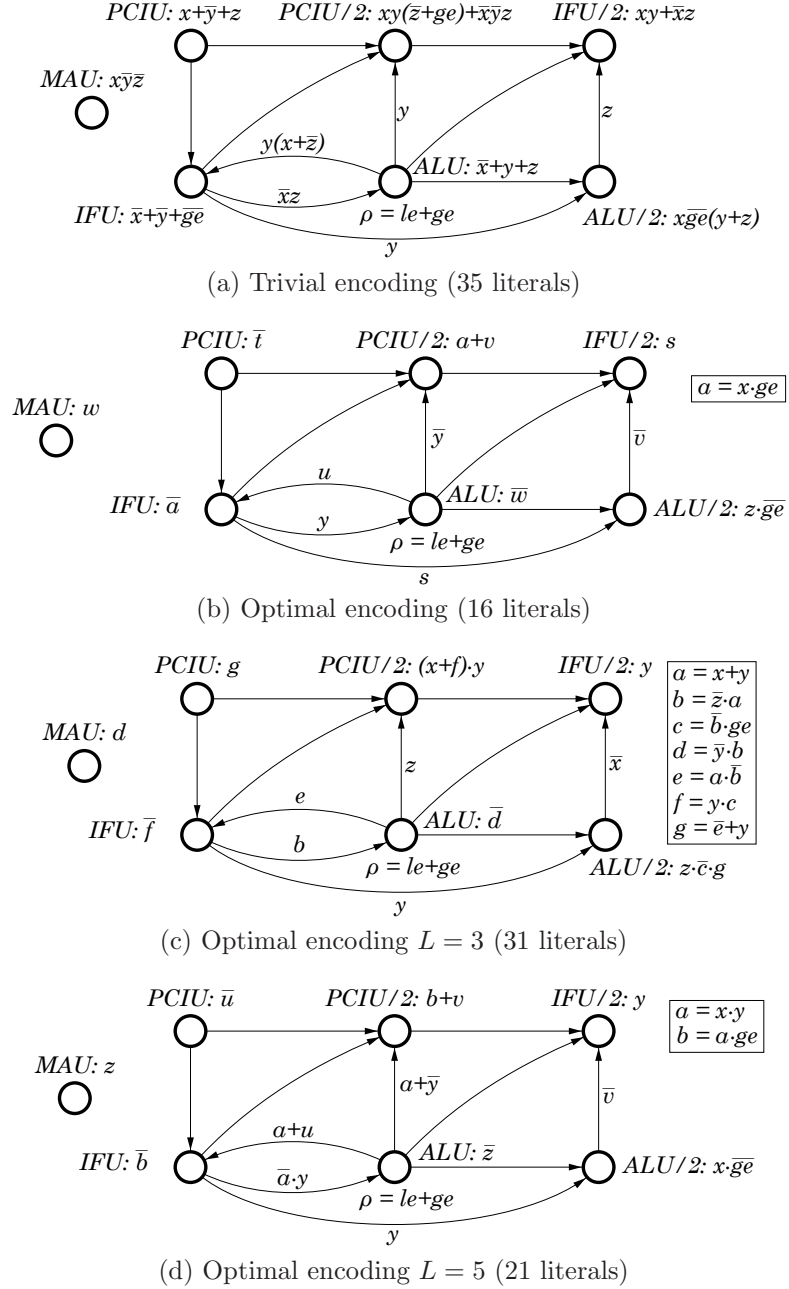


Figure 5.5: Synthesised CPOGs

the generated opcodes for cases $L = 3$ and $L = 5$ – see the corresponding columns of Table 5.1. Note that the optimal 3-bit opcodes are very different from the trivial 000 – 111 sequential encoding.

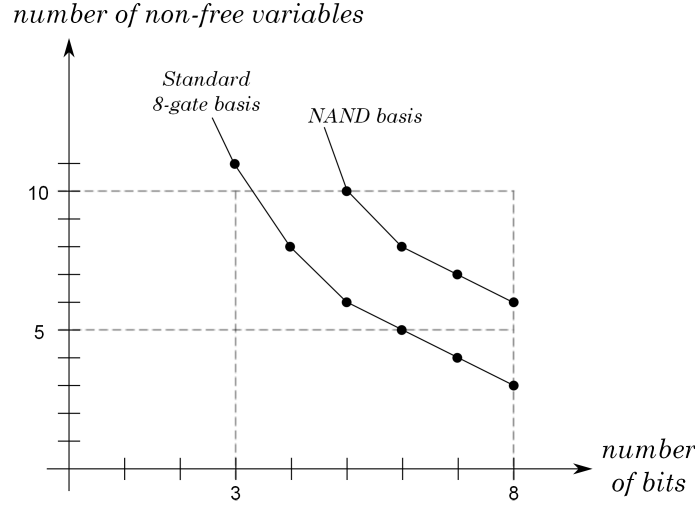


Figure 5.6: Comparison of different gate bases

5.4.2 Microcontroller synthesis

Figure 5.5 shows four CPOGs obtained using instruction encodings shown in Table 5.1. The trivial encoding results in the most complex CPOG shown in Figure 5.5(a); it uses three variables $X = \{x, y, z\}$ and contains 35 literals. The optimal encoding produces the CPOG with only 16 literals in its conditions, Figure 5.5(b), but it uses 8 opcode variables $X = \{x, y, z, u, v, w, s, t\}$. Figures 5.5(c, d) show the optimal CPOGs encoded with 3 and 5 ($X = \{x, y, z, u, v\}$) variables, respectively; derived variables (denoted by names starting from a) are shown in boxes.

Figure 5.6 illustrates dependency of the number of non-free variables on the number of free variables. As expected, the more free variables we have, the less non-free variables are needed to satisfy all the encoding constraints. It is interesting to note that if we restrict our functional basis to NAND gates only (i.e. if we allow only functions $f_k = \overline{f_i \cdot f_j}$ to be used), the number of non-free variables does not increase dramatically.

Choice of a particular scenario within every CPOG in Figure 5.5 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 flag le turned out to be redundant and was removed from all the conditions, because original condition $le \cdot \overline{ge}$ is equivalent to \overline{ge} if the restriction function of ALU ($\rho_{ALU} = le + ge$) is satisfied: $\overline{ge} \cdot (le + ge) \Leftrightarrow le \cdot \overline{ge}$. Thus it is enough to

test only one ALU flag ge to correctly schedule all the instructions.

The optimal CPOG (Figure 5.5(b)) contains only 16 literals which leads to a twice smaller and faster microcontroller than the one obtained using the trivial encoding. Note that in this case it is not possible to reduce the final result to pure 1-restricted form: the graph contains conditions depending on flag ge which cannot be mixed with other variables for optimisation purposes as it is provided by ALU and can be changed during execution of an ALU operation. Three non 1-restricted conditions are: $\phi(IFU) = \bar{a} = \overline{x \cdot \overline{ge}}$, $\phi(ALU/2) = z \cdot \overline{ge}$, and $\phi(PCIU/2) = a + v = x \cdot ge + v$. It is impossible to use fewer literals for these conditions; this is clarified in Table 5.2: depending on the instruction $\phi(IFU)$ has to evaluate either to 1 or to \overline{ge} and this choice is delegated to operational variable x such that $\phi(IFU) = \overline{x \cdot \overline{ge}}$. Condition $\phi(ALU/2)$ is similar: it must evaluate either to 0 or to \overline{ge} , hence $\phi(ALU/2) = z \cdot \overline{ge}$. 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 $\phi(PCIU/2) = x \cdot ge + v$. Optimal encoding of conditions depending on ALU flags is performed automatically together with all other conditions as explained in Subsection 5.3.3, thus the optimal result (in terms of the number of used literals) is guaranteed.

Finally, the chosen CPOG can be mapped into equations to produce the physical implementation of the microcontroller using the mapping algorithms from [39][45].

Instructions class	x	z	v	ϕ_{IFU}	$\phi_{ALU/2}$	$\phi_{PCIU/2}$
ALU Rn to Rn	0	0	0	1	0	0
ALU #123 to Rn	0	0	1	1	0	1
ALU Rn to PC	0	0	0	1	0	0
ALU #123 to PC	0	0	0	1	0	0
Memory access	0	0	0	1	0	0
Cond. ALU Rn to Rn	0	1	0	1	\overline{ge}	0
Cond. ALU #123 to Rn	1	1	1	\overline{ge}	\overline{ge}	1
Cond. ALU #123 to PC	1	1	0	\overline{ge}	\overline{ge}	ge
Optimal condition				$\overline{x \cdot \overline{ge}}$	$z \cdot \overline{ge}$	$x \cdot ge + v$

Table 5.2: Encoding of conditions with dynamic variable ge

5.5 Summary

In this Chapter we have considered the problem of microcontroller instruction design and encoding. We have shown how the application of Parametrised Graphs helps to semi- or completely automatically derive the specification of entire controller from the specification of individual instruction microcode. Crucially, the result is amenable to further transformations possibly with automated rewriting tools.

We have developed a new technique for Parametrised Graph encoding that permits the computation of the globally optimal solution to the PG encoding problem. We do translation into a SAT instance to harness the power of SAT solving tools. The proposal has important practical applications in microelectronics industry and we plan to further pursue the fruitful direction.

Chapter 6

Conclusions

In this thesis we have presented a number of solutions to the problem of compositional design of large scale digital circuits. These solutions help to synthesise efficient asynchronous control circuits, transform and analyse compositional structures suitable for subsequent generation of hardware control logic and compute an efficient encoding of micro-controller instruction set.

6.1 Improved parallel composition

In Chapter 3 we discussed an improved algorithm for computing the parallel composition of STGs or labelled Petri nets. The algorithm makes use of the FCI assumption to produce nets with fewer implicit places. This aids in the synthesis of subsequent structural algorithms like dummy contraction. It uses only simple structural checks and thus is very efficient even for large compositions, so the improvement comes at negligible cost.

The algorithm was implemented in the PCOMP tool and evaluated on a set of scalable benchmarks. The experiments proved its efficiency, which increases even more when the components are pre-processed to remove dummies and ensure injective labelling (this is usually cheap, as the components are small; moreover, if the components come from a standard library of component types, this step can be completely eliminated).

Another important advantage is that the improved algorithm places almost no additional effort on the user: the only requirement is to pass an additional command-

line option to PCOMP so that it can assume the FCI property and apply the proposed optimisation.

6.2 Parametrised Graphs theory

We introduced a new formalism called Parametrised Graphs and the corresponding algebra. The formalism allows to manage a large number of system configurations and execution modes, exploit similarities between them to simplify the specification, and to work with groups of configurations and modes rather than with individual ones. The modes and groups of modes can be managed in a compositional way, and the specifications can be manipulated (transformed and/or optimised) algebraically in a fully formal and natural way.

We develop two variants of the algebra of parametrised graphs, corresponding to the two natural graph equivalences: graph isomorphism and isomorphism of transitive closures. Both cases are specified axiomatically, and the soundness, minimality and completeness of the resulting sets of axioms are formally proved. Moreover, the canonical forms of algebraic terms are developed in each case.

We have formalised the definitions of algebra of parametrised graph in Agda, and developed the machine-checked proofs of several properties of that algebra.

The formula representation data structure was designed together with the custom structural equivalence relation on formula representations for convenient formula manipulations. The equivalence relation has been proved equivalent to the one defined using formula semantics, thus showing its adequacy.

The normal form representation data structure for PG formulae was designed with its semantics defined as translation to the corresponding general formula representations. The algorithm of finding the normal form of general formulae was developed and was proved to be correct.

The usefulness of the developed formalism has been demonstrated on two case studies, a phase encoding controller and a processor micro-controller. Both have a large number of execution scenarios, and the developed formalism allows to capture them algebraically, by composing individual scenarios and groups of scenarios. The possibility of algebraical manipulation was essential to obtain the optimised final specification in each case.

The developed formalism is also convenient for implementation in a tool, as manipulating algebraic terms is much easier than general graph manipulation; in particular, the theory of term rewriting can be naturally applied to derive the canonical forms.

6.3 Processor instruction set encoding

The Chapter 5 presented the PG model based methodology for micro architecture design and studied its application for specification and synthesis of a central processor micro-controller. The key contribution is the method for synthesis of optimal instruction op-codes; the corresponding optimisation problem is formulated in terms of PGs and reduced to the well-known Boolean satisfiability problem. The method is implemented in a software tool and can be used within the conventional micro architecture design flow.

The studied processor example is purely academic. Nonetheless, it captures many important features of real processors. It has been demonstrated that the PG model is capable of modelling concurrency between different subsystems and handling multiple choice during instruction execution. Future work focuses on specification and synthesis of a real processor and optimisation of the encoding algorithms.

Both have a large number of execution scenarios, and the developed formalism allows to capture them algebraically, by composing individual scenarios and groups of scenarios. The possibility of algebraical manipulation was essential to obtain the optimised final specification in each case.

The developed formalism is also convenient for implementation in a tool, as manipulating algebraic terms is much easier than general graph manipulation; in particular, the theory of term rewriting can be naturally applied to derive the canonical forms.

6.4 Future work

The improved parallel composition from Chapter 3 can be generalised by weakening the test used in implicit place elimination and thus taking into the consideration the non-trivial relationships between inputs and outputs. At the moment, we only

remove places directly connected to input and output transitions; a stronger version of Proposition 1 would reason not at the level of individual places, immediately connected to input/output transitions, but rather whole paths going from an output transition to an input transition.

For Parametrised Graphs, we plan to automate the algebraic manipulation of PGs, and implement automatic synthesis of PGs into digital circuits. For the latter, much of the code developed for the precursor formalism of Conditional Partial Order Graphs (CPOGs) can be re-used. One of the important problems that needs to be automated is that of simplification of (T)PG expressions, in the sense of deriving an equivalent expression with the minimum possible number of operators. Our preliminary research suggests that this problem is strongly related to modular decomposition of graphs [37].

We also plan to formalise the proof of CPOG being a model of PG Algebra and modification of the normalisation algorithm from Chapter 4 computing the canonical form (where each graph node is mentioned no more than once) instead of just a normal form. Canonical form is much more useful because its size is at most quadratic while the size of a normal form is exponential in general. The canonical form is also a way to quickly compare graphs and solve equational relations.

Appendix A

Formal proof of PG Algebra properties

What follows is the full Agda source code of formal proofs used in Chapter 4.

```
module PG.Eq where

record Eq (G : Set) : Set1 where
  constructor equality
infix 3 _ ≈ _
field
  _ ≈ _ : G → G → Set

module PG.FormulaEq where

open import PG.Formulae
open import PG.GraphAlgebra
open import PG.PGAlgebra
open import Function

pgeval' : ∀ {B V G : Set} → (V → G) → PGOps B G → PGFormula B V → G
pgeval' f ops = pgeval Ops._ + _ Ops._ >> _ Ops.ε f Ops. [-] where
  module Ops = PGOps ops

_ ≈s _ : ∀ {B V : Set} → PGFormula B V → PGFormula B V → Set1
_ ≈s _ {B} {V} f1 f2 = ∀ {G : Set} → (pgalgebra : PG B G) → (f : V → G)
  → let open PG pgalgebra in pgeval' f pgops f1 ≈ pgeval' f pgops f2

module EQC where
data EquivClosure {X : Set} (~_ : X → X → Set) : X → X → Set where
  refl : ∀ {f} → EquivClosure ~_ f f
  trans : ∀ {a b c} → a ~ b → b ~ c → EquivClosure ~_ a c
```

```

    sym :  $\forall \{a\ b\} \rightarrow a \sim b \rightarrow \text{EquivClosure } \_ \sim \_ b\ a$ 

open EQC
open EQC public using (EquivClosure; module EquivClosure)

formulagraphops :  $\forall \{B\ V : \text{Set}\} \rightarrow \text{GraphOps } (\text{PGFormula } B\ V)$ 
formulagraphops = record
    { _ + _ = _ + _
    ; _  $\gg$  _ = _  $\gg$  _
    ;  $\varepsilon$  =  $\varepsilon$  }

formulaops :  $\{B\ V : \text{Set}\} \rightarrow \text{PGOps } B\ (\text{PGFormula } B\ V)$ 
formulaops = record
    { graphops = formulagraphops
    ; [-] _ = [-] _ }

module WithBV {B : Set} {V : Set} where

infix 3 _  $\approx$  _
infix 3 _B  $\approx$  _

data _B  $\approx$  _ : BoolFormula B  $\rightarrow$  BoolFormula B  $\rightarrow$  Set where

    isEquivalence :  $\forall \{a\ b\} \rightarrow \text{EquivClosure } \_B \approx \_ a\ b \rightarrow a\ B \approx b$ 
     $\wedge$ cong :  $\forall \{p\ q\ r\ s\} \rightarrow p\ B \approx r \rightarrow q\ B \approx s \rightarrow p \wedge q\ B \approx r \wedge s$ 
     $\vee$ cong :  $\forall \{p\ q\ r\ s\} \rightarrow p\ B \approx r \rightarrow q\ B \approx s \rightarrow p \vee q\ B \approx r \vee s$ 
     $\neg$ cong :  $\forall \{p\ q\} \rightarrow p\ B \approx q \rightarrow \neg p\ B \approx \neg q$ 

     $\vee$ complementr :  $\forall \{x\} \rightarrow x \vee \neg x\ B \approx \top$ 
     $\wedge$ complementr :  $\forall \{x\} \rightarrow x \wedge \neg x\ B \approx \perp$ 

     $\vee$ comm :  $\forall \{x\ y\} \rightarrow x \vee y\ B \approx y \vee x$ 
     $\vee$ assoc :  $\forall \{x\ y\ z\} \rightarrow (x \vee y) \vee z\ B \approx x \vee (y \vee z)$ 
     $\wedge$ comm :  $\forall \{x\ y\} \rightarrow x \wedge y\ B \approx y \wedge x$ 
     $\wedge$ assoc :  $\forall \{x\ y\ z\} \rightarrow (x \wedge y) \wedge z\ B \approx x \wedge (y \wedge z)$ 
     $\vee$ absorbs $\wedge$  :  $\forall \{x\ y\} \rightarrow x \vee (x \wedge y)\ B \approx x$ 
     $\wedge$ absorbs $\vee$  :  $\forall \{x\ y\} \rightarrow x \wedge (x \vee y)\ B \approx x$ 
     $\vee$ distributes :  $\forall \{x\ y\ z\} \rightarrow (y \wedge z) \vee x\ B \approx (y \vee x) \wedge (z \vee x)$ 

data _  $\approx$  _ : PGFormula (BoolFormula B) V  $\rightarrow$  PGFormula (BoolFormula B) V  $\rightarrow$  Set where

    isEquivalence :  $\forall \{a\ b\} \rightarrow \text{EquivClosure } \_ \approx \_ a\ b \rightarrow a \approx b$ 
    +cong :  $\forall \{p\ q\ r\ s\} \rightarrow p \approx r \rightarrow q \approx s \rightarrow p + q \approx r + s$ 
     $\gg$  cong :  $\forall \{p\ q\ r\ s\} \rightarrow p \approx r \rightarrow q \approx s \rightarrow p \gg q \approx r \gg s$ 

    +assoc :  $\forall \{p\ q\ r\} \rightarrow (p + q) + r \approx p + (q + r)$ 
    +comm :  $\forall \{p\ q\} \rightarrow p + q \approx q + p$ 
     $\gg$  assoc :  $\forall \{p\ q\ r\} \rightarrow (p \gg q) \gg r \approx p \gg (q \gg r)$ 
     $\gg$  identityl :  $\forall \{p\} \rightarrow \varepsilon \gg p \approx p$ 
    
```


$\gg \text{identity}^r : \forall \{p\} \rightarrow p \gg \varepsilon \approx p$
 $\text{distrib}^l : \forall \{p \ q \ r\} \rightarrow p \gg (q + r) \approx p \gg q + p \gg r$
 $\text{distrib}^r : \forall \{p \ q \ r\} \rightarrow (p + q) \gg r \approx p \gg r + q \gg r$
 $\text{decomposition} : \forall \{p \ q \ r\} \rightarrow p \gg q \gg r \approx p \gg q + p \gg r + q \gg r$
 $\text{condcong} : \forall \{f \ g \ p \ q\} \rightarrow f \ B \approx g \rightarrow p \approx q \rightarrow [f] \ p \approx [g] \ q$
 $\text{truecondition} : \forall \{x\} \rightarrow [\top] \ x \approx x$
 $\text{falsecondition} : \forall \{x\} \rightarrow [\perp] \ x \approx \varepsilon$
 $\text{orcondition} : \forall \{f \ g \ x\} \rightarrow [f \vee g] \ x \approx [f] \ x + [g] \ x$
 $\text{andcondition} : \forall \{f \ g \ x\} \rightarrow [f \wedge g] \ x \approx [f] \ [g] \ x$
 $\text{conditional+} : \forall \{f \ x \ y\} \rightarrow [f] \ (x + y) \approx [f] \ x + [f] \ y$
 $\text{conditional} \gg : \forall \{f \ x \ y\} \rightarrow [f] \ (x \gg y) \approx [f] \ x \gg [f] \ y$
 $\text{conditional}\varepsilon : \forall \{f\} \rightarrow [f] \ \varepsilon \approx \varepsilon$

open import *Relation.Binary*

$\text{iseqbyclosure} : \forall \{X : \text{Set}\} \{ \sim_- : X \rightarrow X \rightarrow \text{Set} \}$
 $\rightarrow (\forall \{a \ b\} \rightarrow \text{EquivClosure } \sim_- \ a \ b \rightarrow a \sim b) \rightarrow \text{IsEquivalence } \sim_-$
 $\text{iseqbyclosure fromClosure} = \text{record}$
 $\{ \text{refl} = \text{fromClosure refl}$
 $;\text{sym} = \lambda \text{eq} \rightarrow \text{fromClosure (sym eq)}$
 $;\text{trans} = \lambda \text{l r} \rightarrow \text{fromClosure (trans l r)}$
 $\}$

$\approx \text{isequivalence} : \text{IsEquivalence } \approx_-$
 $\approx \text{isequivalence} = \text{iseqbyclosure isEquivalence}$
 $B \approx \text{isequivalence} : \text{IsEquivalence } B \approx_-$
 $B \approx \text{isequivalence} = \text{iseqbyclosure isEquivalence}$

$\text{boolformulaops} : \forall \{B : \text{Set}\} \rightarrow \text{BoolOps } (\text{BoolFormula } B)$
 $\text{boolformulaops} = \text{record}$
 $\{ _ \vee _ = _ \vee _$
 $;_ \wedge _ = _ \wedge _$
 $;\neg = \neg _$
 $;\top = \top$
 $;\perp = \perp \}$

open import *PG.Eq*

open import *Data.Product*

$\text{boolformulaisboolalg} : \text{IsBoolAlg } (\text{equality } B \approx _) \text{ boolformulaops}$
 $\text{boolformulaisboolalg} = \text{record}$
 $\{ \text{isDistributiveLattice} = \text{record } \{$
 $\text{isLattice} = \text{record } \{$
 $\text{isEquivalence} = B \approx \text{isequivalence};$

```

     $\vee comm = \lambda x y \rightarrow \vee comm;$ 
     $\vee assoc = \lambda x y z \rightarrow \vee assoc;$ 
     $\vee cong = \vee cong;$ 
     $\wedge comm = \lambda x y \rightarrow \wedge comm;$ 
     $\wedge assoc = \lambda x y z \rightarrow \wedge assoc;$ 
     $\wedge cong = \wedge cong;$ 
     $absorptive = (\lambda x y \rightarrow \vee absorbs \wedge), (\lambda x y \rightarrow \wedge absorbs \vee) \};$ 
     $\vee \wedge distrib^r = \lambda x y z \rightarrow \vee distributes \}$ 
;  $\vee complement^r = \lambda x \rightarrow \vee complement^r$ 
;  $\wedge complement^r = \lambda x \rightarrow \wedge complement^r$ 
;  $\neg cong = \neg cong$ 
}

pgformulaisgraph : IsGraph (equality _  $\approx$  _) formulagraphops
pgformulaisgraph = record {
    +cong = +cong;
     $\gg cong = \gg cong;$ 
     $\approx iseq = \approx isequivalence;$ 
    +assoc = +assoc;
    +comm = +comm;
     $\gg assoc = \gg assoc;$ 
     $\gg identity^l = \gg identity^l;$ 
     $\gg identity^r = \gg identity^r;$ 
     $distrib^l = distrib^l;$ 
     $distrib^r = distrib^r;$ 
    decomposition = decomposition}

pgformulaispg : IsPG (equality _B  $\approx$  _) (equality _  $\approx$  _) formulaops boolformulaops
pgformulaispg = record {
    isgraph = pgformulaisgraph;
    isbool = boolformulaisboolalg;
    condcong =  $\lambda l r \rightarrow condcong l r;$ 
    truecondition =  $\lambda x \rightarrow truecondition;$ 
    falsecondition =  $\lambda x \rightarrow falsecondition;$ 
    andcondition =  $\lambda f g x \rightarrow andcondition;$ 
    orcondition =  $\lambda f g x \rightarrow orcondition;$ 
    conditional+ =  $\lambda f x y \rightarrow conditional+;$ 
    conditional  $\gg = \lambda f x y \rightarrow conditional \gg \}$ 

module PG.Formulae where
module PGF (B V : Set) where
infixl 13 _ + _

```

```

infixl 18 _  $\gg$  _

data PGFormula : Set where
  _ + _ : (x y : PGFormula)  $\rightarrow$  PGFormula
  _  $\gg$  _ : (x y : PGFormula)  $\rightarrow$  PGFormula
   $\varepsilon$  : PGFormula
  var : (a : V)  $\rightarrow$  PGFormula
  [ _ ] : (c : B)  $\rightarrow$  PGFormula  $\rightarrow$  PGFormula

open PGF public

infixl 5 _  $\vee$  _
infixl 6 _  $\wedge$  _

data BoolFormula X : Set where
  var : X  $\rightarrow$  BoolFormula X
  _  $\vee$  _  $\wedge$  _ : BoolFormula X  $\rightarrow$  BoolFormula X  $\rightarrow$  BoolFormula X
   $\neg$  _ : BoolFormula X  $\rightarrow$  BoolFormula X
   $\top \perp$  : BoolFormula X

pgeval : { A B G : Set }
   $\rightarrow$  ( _ +s _  $\gg$ s _ : G  $\rightarrow$  G  $\rightarrow$  G )
   $\rightarrow$  ( $\varepsilon^s$  : G)
   $\rightarrow$  (vars : A  $\rightarrow$  G)
   $\rightarrow$  (conds : B  $\rightarrow$  G  $\rightarrow$  G)
   $\rightarrow$  PGFormula B A  $\rightarrow$  G

pgeval { A } { B } { G } _ +s _  $\gg$ s _  $\varepsilon^s$  vars conds = go where
  go : PGFormula B A  $\rightarrow$  G
  go (x + y) = go x +s go y
  go (x  $\gg$  y) = go x  $\gg$ s go y
  go  $\varepsilon$  =  $\varepsilon^s$ 
  go (var a) = vars a
  go ([c] y) = conds c (go y)

module PG.GraphAlgebra where

open import Algebra
import Algebra.FunctionProperties
open import Algebra.Structures
open import Relation.Binary
open import Data.Product
open import PG.Eq

record GraphOps (G : Set) : Set where
infixl 13 _ + _
infixl 18 _  $\gg$  _

```

field

$- + - : (p \ q : G) \rightarrow G$

$- \gg - : (p \ q : G) \rightarrow G$

$\varepsilon : G$

record *IsGraph* { *G* : *Set* } (*Eq* : *Eq* *G*) (*Ops* : *GraphOps* *G*) : *Set* **where**

open *GraphOps* *Ops*

open *Eq* *Eq* **renaming** ($- \approx -$ *to* $- \approx -$)

field

$+cong : \forall \{p \ q \ r \ s\} \rightarrow p \approx r \rightarrow q \approx s \rightarrow p + q \approx r + s$

$\gg cong : \forall \{p \ q \ r \ s\} \rightarrow p \approx r \rightarrow q \approx s \rightarrow p \gg q \approx r \gg s$

$\approx iseq : IsEquivalence \ - \approx -$

$+assoc : \forall \{p \ q \ r\} \rightarrow (p + q) + r \approx p + (q + r)$

$+comm : \forall \{p \ q\} \rightarrow p + q \approx q + p$

$\gg assoc : \forall \{p \ q \ r\} \rightarrow (p \gg q) \gg r \approx p \gg (q \gg r)$

$\gg identity^l : \forall \{p\} \rightarrow \varepsilon \gg p \approx p$

$\gg identity^r : \forall \{p\} \rightarrow p \gg \varepsilon \approx p$

$distrib^l : \{p \ q \ r : G\} \rightarrow p \gg (q + r) \approx p \gg q + p \gg r$

$distrib^r : \{p \ q \ r : G\} \rightarrow (p + q) \gg r \approx p \gg r + q \gg r$

$decomposition : \{p \ q \ r : G\} \rightarrow p \gg q \gg r \approx p \gg q + p \gg r + q \gg r$

open *IsEquivalence* $\approx iseq$ **public**

record *Graph* (*G* : *Set*) : *Set*₁ **where**

field

eq : *Eq* *G*

ops : *GraphOps* *G*

isGraph : *IsGraph* *eq ops*

open *IsGraph* *isGraph* **public**

open *Eq* *eq* **public**

open *GraphOps* *ops* **public**

import *Level*

open *Level* **using** () **renaming** (*zero* *to* *0*)

import *Relation.Binary.EqReasoning*

open import *Function*

open import *Data.List* **hiding** ([_])

import *Relation.Binary.PropositionalEquality*

module *GraphTheory* { *P* : *Set* } (*graph* : *Graph* *P*) **where**

open *Graph* *graph*

open *Relation.Binary.EqReasoning* **record** { *isEquivalence* = $\approx iseq$ }

open import *Relation.Binary*

$$_ \approx \approx \approx _ : \{p \ q \ r : P\} \rightarrow p \approx q \rightarrow q \approx r \rightarrow p \approx r$$

$$x \approx \approx \approx y = \text{trans } x \ y$$

$$\mathbf{infixl} \ 8 \ _ \approx \approx \approx _$$

$$+cong_1 : \{p \ q \ r : P\} \rightarrow p \approx r \rightarrow p + q \approx r + q$$

$$+cong_1 \ p \approx r = p \approx r \langle +cong \rangle \text{refl}$$

$$+cong_2 : \{p \ q \ r : P\} \rightarrow q \approx r \rightarrow p + q \approx p + r$$

$$+cong_2 \ q \approx r = \text{refl} \langle +cong \rangle \ q \approx r$$

$$\gg cong_2 : \{p \ q \ r : P\} \rightarrow q \approx r \rightarrow p \gg q \approx p \gg r$$

$$\gg cong_2 \ q \approx r = \text{refl} \langle \gg cong \rangle \ q \approx r$$

$$\gg cong_1 : \{p \ q \ r : P\} \rightarrow p \approx r \rightarrow p \gg q \approx r \gg q$$

$$\gg cong_1 \ p \approx r = p \approx r \langle \gg cong \rangle \text{refl}$$

$$\text{rdco} : \forall \{a\} \rightarrow a + a + \varepsilon \approx a$$

$$\text{rdco} \ \{a\} =$$

begin

$$a + a + \varepsilon$$

$$\approx \langle \text{sym} (\gg \text{identity}^r \langle +cong \rangle \gg \text{identity}^r \langle +cong \rangle \gg \text{identity}^r) \rangle$$

$$a \gg \varepsilon + a \gg \varepsilon + \varepsilon \gg \varepsilon$$

$$\approx \langle \text{sym decomposition} \rangle$$

$$a \gg \varepsilon \gg \varepsilon$$

$$\approx \langle \gg \text{identity}^r \approx \approx \gg \text{identity}^r \rangle$$

$$a$$

■

$$+\text{identity}^r : \forall \{a\} \rightarrow a + \varepsilon \approx a$$

$$+\text{identity}^r \ \{a\} =$$

begin

$$a + \varepsilon$$

$$\approx \langle \text{sym rdco} \rangle$$

$$(a + \varepsilon) + (a + \varepsilon) + \varepsilon$$

$$\approx \langle +cong_1$$

$$(+assoc \approx \approx \approx (+cong_2 (+comm \approx \approx \approx +assoc) \approx \approx \approx \text{sym} + assoc))$$

$$\approx \approx \approx +assoc \rangle$$

$$a + a + (\varepsilon + \varepsilon + \varepsilon)$$

$$\approx \langle +cong_2 \text{rdco} \rangle$$

$$a + a + \varepsilon$$

$$\approx \langle \text{rdco} \rangle$$

$$a$$

■

$$+\text{identity}^l : \{p : P\} \rightarrow \varepsilon + p \approx p$$

$$+identity^l = +comm \approx \approx \approx +identity^r$$

$$idempotence : \forall \{p\} \rightarrow p + p \approx p$$

$$idempotence \{p\} = sym +identity^r \approx \approx \approx rdeco$$

$$absorption : \forall \{a\} \{b\} \rightarrow a \gg b + a + b \approx a \gg b$$

$$absorption \{a\} \{b\} =$$

begin

$$a \gg b + a + b$$

$$\approx \langle sym (refl \langle +cong \rangle \gg identity^r \langle +cong \rangle \gg identity^r) \rangle$$

$$(a \gg b) + (a \gg \varepsilon) + (b \gg \varepsilon)$$

$$\approx \langle sym decomposition \rangle$$

$$a \gg b \gg \varepsilon$$

$$\approx \langle \gg identity^r \rangle$$

$$a \gg b$$

■

$$absorption^l : \{p\} \{q\} \rightarrow p \gg q + p \approx p \gg q$$

$$absorption^l \{p\} \{q\} =$$

begin

$$p \gg q + p$$

$$\approx \langle +cong_1 (sym absorption) \rangle$$

$$p \gg q + p + q + p$$

$$\approx \langle +assoc \approx \approx \approx +cong_2 +comm \approx \approx \approx (sym +assoc \approx \approx \approx +cong_1 +assoc) \rangle$$

$$p \gg q + (p + p) + q$$

$$\approx \langle +cong_1 (+cong_2 idempotence) \rangle$$

$$p \gg q + p + q$$

$$\approx \langle absorption \rangle$$

$$p \gg q$$

■

$$absorption^r : \forall \{p\} \{q\} \rightarrow p \gg q + q \approx p \gg q$$

$$absorption^r \{p\} \{q\} =$$

begin

$$p \gg q + q$$

$$\approx \langle +cong_1 (sym absorption) \rangle$$

$$p \gg q + p + q + q$$

$$\approx \langle +assoc \rangle$$

$$p \gg q + p + (q + q)$$

$$\approx \langle +cong_2 idempotence \rangle$$

$$p \gg q + p + q$$

$$\approx \langle absorption \rangle$$

$$p \gg q$$

■

$- \subseteq - : (p \ q : P) \rightarrow Set$

$p \subseteq q = p + q \approx q$

infix 10 $- \subseteq -$

$leastElement : \{p : P\} \rightarrow \varepsilon \subseteq p$

$leastElement = +identity^l$

$p + q \subseteq p \gg q : \{p \ q : P\} \rightarrow p + q \subseteq p \gg q$

$p + q \subseteq p \gg q = +comm \approx \approx \approx sym + assoc \approx \approx \approx absorption$

$\subseteq refl_0 : \{p : P\} \rightarrow p \subseteq p$

$\subseteq refl_0 = idempotence$

$\subseteq refl : \{p \ q : P\} \rightarrow p \approx q \rightarrow p \subseteq q$

$\subseteq refl \ p \approx q = +cong_1 \ p \approx q \approx \approx \approx idempotence$

$\subseteq antisym : \{p \ q : P\} \rightarrow p \subseteq q \rightarrow q \subseteq p \rightarrow p \approx q$

$\subseteq antisym \ p + q \approx q \ q + p \approx p = sym (+comm \approx \approx \approx q + p \approx p) \approx \approx \approx p + q \approx q$

$\subseteq trans : \{p \ q \ r : P\} \rightarrow p \subseteq q \rightarrow q \subseteq r \rightarrow p \subseteq r$

$\subseteq trans \ \{p\} \ \{q\} \ \{r\} \ p + q \approx q \ q + r \approx r =$

begin

$p + r$

$\approx \langle sym (+assoc \approx \approx \approx +cong_2 \ q + r \approx r) \rangle$

$p + q + r$

$\approx \langle +cong_1 \ p + q \approx q \approx \approx \approx q + r \approx r \rangle$

r

■

$\subseteq isPartialOrder : IsPartialOrder \ - \approx \ - \subseteq \ -$

$\subseteq isPartialOrder = \mathbf{record}$

$\{ isPreorder = \mathbf{record}$

$\{ isEquivalence = \approx iseq$

$; trans = \subseteq trans$

$; reflexive = \subseteq refl$

$\}$

$; antisym = \subseteq antisym$

$\}$

$+isSemigroup : IsSemigroup \ - \approx \ - \ + \ -$

$+isSemigroup = \mathbf{record}$

$\{ isEquivalence = \approx iseq$

$; assoc = \lambda \ x \ y \ z \rightarrow +assoc \ \{x\} \ \{y\} \ \{z\}$

$; \bullet cong = +cong$

```

    }
+isCommutativeMonoid : IsCommutativeMonoid _ ≈ _ + _ ε
+isCommutativeMonoid = record
    { isSemigroup = +isSemigroup
    ; identityl = λ x → +identityl { x }
    ; comm = λ x y → +comm { x } { y }
    }
>> isSemigroup : IsSemigroup _ ≈ _ >> _
>> isSemigroup = record
    { isEquivalence = ≈ iseq
    ; assoc = λ x y z → >> assoc { x } { y } { z }
    ; •cong = >> cong
    }
>> isMonoid : IsMonoid _ ≈ _ >> _ ε
>> isMonoid = record
    { isSemigroup = >> isSemigroup
    ; identity = (λ x → >> identityl { x } ), (λ x → >> identityr { x } )
    }
cpogIsSemiringWithoutAnnihilatingZero
    : IsSemiringWithoutAnnihilatingZero _ ≈ _ + _ >> _ ε ε
cpogIsSemiringWithoutAnnihilatingZero = record
    { +isCommutativeMonoid = +isCommutativeMonoid
    ; *isMonoid = >> isMonoid
    ; distrib = (λ _ _ → distribl ), (λ _ _ → distribr )
    }
+monotony : { p q r s : P } → p ⊆ q → r ⊆ s → p + r ⊆ q + s
+monotony { p } { q } { r } { s } p + q ≈ q r + s ≈ s =
begin
    (p + r) + (q + s)
    ≈ ⟨ sym +assoc ≈≈≈
    +cong1 (+assoc ≈≈≈ +cong2 +comm ≈≈≈ sym +assoc)
    ≈≈≈ +assoc ⟩
    (p + q) + (r + s)
    ≈ ⟨ p + q ≈ q ⟨ +cong ⟩ r + s ≈ s ⟩
    q + s
    ■
+preserves_⊆_ : _ + _ Preserves2 _⊆_ → _⊆_ → _⊆_
+preserves_⊆_ = +monotony
    
```



```

    >> arg1Monotony : {p q r : P} → p ⊆ q → p >> r ⊆ q >> r
    >> arg1Monotony p + q ≈ q = sym distribr ≈≈≈ >> cong1 p + q ≈ q
    >> arg2Monotony : {p r s : P} → r ⊆ s → p >> r ⊆ p >> s
    >> arg2Monotony r + s ≈ s = sym distribl ≈≈≈ >> cong2 r + s ≈ s
    >> monotony : {p q r s : P} → p ⊆ q → r ⊆ s → p >> r ⊆ q >> s
    >> monotony p + q ≈ q r + s ≈ s = ⊆ trans (>> arg1Monotony p + q ≈ q) (>> arg2Monotony r + s ≈ s)
    >> preserves_⊆_ : _ >> _ Preserves2 _ ⊆ _ → _ ⊆ _ → _ ⊆ _
    >> preserves_⊆_ = >> monotony
module PG.Normalizercorrect (A B : Set) where
open import Relation.Binary
import Relation.Binary.PropositionalEquality as PropEq
open PropEq using (≡)
open import PG.Formulae
module WithOrder {_ <_ : A → A → Set} (ASTO : IsStrictTotalOrder _ ≡ _ < _) where
    open import PG.FormulaEq
    import PG.Normalizer
    open PG.Normalizer.WithOrder A B ASTO renaming (_ + _ to _ + ' _; _ >> _ to _ >> ' _)
    open PG.Normalizer A B
    import Data.List as List
    open List using ([]; _ : _; foldr; map)
    open import Function
    open WithBV {B} {A}
    import PG.GraphAlgebra as Alg
    module GT = Alg.GraphTheory (record {isGraph = pgformulaisgraph})
    module BEq = IsEquivalence B ≈ isequivalence
    open IsEquivalence ≈ isequivalence
    import Relation.Binary.EqReasoning as EqR
    open EqR record {isEquivalence = ≈ isequivalence}
    open EqR record {isEquivalence = B ≈ isequivalence} renaming
        (begin_ to Bbegin_; _ ≈ ⟨_⟩_ to _B ≈ ⟨_⟩_; ■ to _B■)
    open import Algebra.Structures
    open IsBooleanAlgebra boolformulaisboolalg using
        ()
    open import Algebra.Props.BooleanAlgebra
        (record {isBooleanAlgebra = boolformulaisboolalg}) using
        (∧ ∨ distrib; ∧ complement; ∨ complement;
         ∨ ∧ isCommutativeSemiring; ∧ idempotent; ∨ idempotent)
    open IsCommutativeSemiring ∨ ∧ isCommutativeSemiring
    
```

```

using () renaming (*identity to  $\wedge$ identity; zero to  $\wedge$ zero)

open import Data.Product using (–, –; proj1; proj2)
open import Data.Sum using (inj1; inj2)

absorption :  $\forall \{x\ y\} \rightarrow x + y + x \gg y \approx x \gg y$ 
absorption = +comm  $\langle$  trans  $\rangle$  (sym +assoc  $\langle$  trans  $\rangle$  GT.absorption)

lem :  $\forall f\ p \rightarrow p \approx [f]\ p + [\neg f]\ p$ 
lem f p =
  begin
    p
     $\approx \langle$  sym truecondition  $\rangle$ 
     $[\top]\ p$ 
     $\approx \langle$  condcong (isEquivalence (EQC.sym  $\vee$  complementr)) refl  $\rangle$ 
     $[f \vee \neg f]\ p$ 
     $\approx \langle$  orcondition  $\rangle$ 
     $[f]\ p + [\neg f]\ p$ 
    ■

caseanalyse :  $\forall \{f\ p\ q\} \rightarrow [f]\ p \approx [f]\ q \rightarrow [\neg f]\ p \approx [\neg f]\ q \rightarrow p \approx q$ 
caseanalyse {f} {p} {q} fcase  $\neg$ fcase =
  begin
    p
     $\approx \langle$  lem – –  $\rangle$ 
     $[f]\ p + [\neg f]\ p$ 
     $\approx \langle$  +cong fcase  $\neg$ fcase  $\rangle$ 
     $[f]\ q + [\neg f]\ q$ 
     $\approx \langle$  sym (lem – –)  $\rangle$ 
    q
    ■

elim+ :  $\forall \{f\ a\ b\ c\ d\}$ 
   $\rightarrow [f]\ a \approx [f]\ c \rightarrow [f]\ b \approx [f]\ d$ 
   $\rightarrow [f]\ (a + b) \approx [f]\ (c + d)$ 
elim+ a  $\approx$  c b  $\approx$  d =
  trans (trans conditional+ (+cong a  $\approx$  c b  $\approx$  d)) (sym conditional+)

elim $\gg$  :  $\forall \{f\ a\ b\ c\ d\} \rightarrow$ 
   $[f]\ a \approx [f]\ c \rightarrow [f]\ b \approx [f]\ d \rightarrow$ 
   $[f]\ (a \gg b) \approx [f]\ (c \gg d)$ 
elim $\gg$  a  $\approx$  c b  $\approx$  d =
  trans (trans conditional $\gg$  ( $\gg$  cong a  $\approx$  c b  $\approx$  d)) (sym conditional $\gg$ )

dupcond :  $\forall \{f\ x\} \rightarrow [f]\ [f]\ x \approx [f]\ x$ 

```

$$\begin{aligned}
 \text{dupcond} &= \text{trans } (\text{sym andcondition}) (\text{condcong } (\wedge \text{idempotent } _) \text{ refl}) \\
 \text{falsecond} &: \forall \{f\ x\} \rightarrow [\neg f] [f] x \approx [\neg f] \varepsilon \\
 \text{falsecond} &= \text{trans } (\text{sym dupcond}) (\text{condcong BEq.refl } (\text{trans} \\
 &\quad (\text{trans } (\text{sym andcondition}) \\
 &\quad \quad (\text{condcong } (\text{proj}_1 \wedge \text{complement } _) \text{ refl})) \\
 &\quad \text{falsecondition} \\
 &\quad) \\
 &\quad) \\
 \text{condswap} &: \forall \{f\ g\ x\} \rightarrow [f] [g] x \approx [g] [f] x \\
 \text{condswap} &= \text{trans } (\text{sym andcondition}) (\text{trans } (\text{condcong } \wedge \text{comm refl}) \text{ andcondition}) \\
 \text{side1} &: \forall \{f\ g\ p\ q\} \\
 &\quad \rightarrow [f] ([f] p \gg [g] q) \\
 &\quad \approx [f] ([f] p + [g] q + [f \wedge g] (p \gg q)) \\
 \text{side1 } \{f\} \{g\} \{p\} \{q\} &= \\
 \text{begin} & \\
 &\quad [f] ([f] p \gg [g] q) \\
 &\quad \approx \langle \text{elim } \gg \text{dupcond refl} \rangle \\
 &\quad [f] (p \gg [g] q) \\
 &\quad \approx \langle \text{condcong BEq.refl } (\text{caseanalyse } \{g\} \\
 &\quad \quad (\\
 &\quad \quad \text{begin} \\
 &\quad \quad \quad [g] (p \gg [g] q) \\
 &\quad \quad \quad \approx \langle \text{elim } \gg \text{refl dupcond} \rangle \\
 &\quad \quad \quad [g] (p \gg q) \\
 &\quad \quad \quad \approx \langle \text{sym } (\text{condcong BEq.refl absorption}) \rangle \\
 &\quad \quad \quad [g] (p + q + p \gg q) \\
 &\quad \quad \quad \approx \langle \text{sym } (\text{elim+ } (\text{elim+ refl dupcond}) \text{ dupcond}) \rangle \\
 &\quad \quad \quad [g] (p + [g] q + [g] (p \gg q)) \\
 &\quad \quad \quad \blacksquare \\
 &\quad \quad) \\
 &\quad \quad (\text{trans } (\text{elim } \gg \text{refl falsecond}) (\text{trans} \\
 &\quad \quad \quad (\text{condcong BEq.refl} \\
 &\quad \quad \quad (\text{trans } \gg \text{identity}^r (\text{sym } (\text{trans GT.} + \text{identity}^r \text{ GT.} + \text{identity}^r)))) \\
 &\quad \quad \quad (\text{sym } (\text{elim+ } (\text{elim+ refl falsecond}) \text{ falsecond})))) \rangle \\
 &\quad \quad [f] (p + [g] q + [g] (p \gg q)) \\
 &\quad \quad \approx \langle \text{sym } (\text{elim+ } (\text{elim+ dupcond refl}) \\
 &\quad \quad \quad (\text{trans } (\text{condcong BEq.refl andcondition}) \text{ dupcond})) \rangle \\
 &\quad \quad [f] ([f] p + [g] q + [f \wedge g] (p \gg q)) \\
 &\quad \quad \blacksquare
 \end{aligned}$$

$side2 : \forall \{f\ g\ p\ q\}$
 $\rightarrow [\neg f] ([f] p \gg [g] q)$
 $\approx [\neg f] ([f] p + [g] q + [f \wedge g] (p \gg q))$
 $side2 \{f\} \{g\} \{p\} \{q\} =$
 $begin$
 $[\neg f] ([f] p \gg [g] q)$
 $\approx \langle elim \gg falsecond refl \rangle$
 $[\neg f] (\varepsilon \gg [g] q)$
 $\approx \langle condcong BEq.refl \gg identity^l \rangle$
 $[\neg f] [g] q$
 $\approx \langle condcong BEq.refl (sym (trans (GT. + identity^r) (GT. + identity^l))) \rangle$
 $[\neg f] (\varepsilon + [g] q + \varepsilon)$
 $\approx \langle sym (elim+ (elim+ falsecond refl) (trans (sym dupcond)$
 $(condcong BEq.refl (trans (sym andcondition)$
 $(trans (condcong (BEq.trans (BEq.sym \wedge assoc)$
 $(BEq.trans (\wedge cong (proj_1 \wedge complement _) BEq.refl) (proj_1 \wedge zero _)))$
 $refl) falsecondition)))) \rangle$
 $[\neg f] ([f] p + [g] q + [f \wedge g] (p \gg q))$

■

$conditionregularisation : \forall \{f\ g\} \rightarrow \{p\ q : PG\}$
 $\rightarrow [f] p \gg [g] q \approx [f] p + [g] q + [f \wedge g] (p \gg q)$
 $conditionregularisation \{f\} \{g\} \{p\} \{q\} = caseanalyse \{f\}$
 $side1\ side2$

$absorbbycondition : \forall \{f\ g\ p\} \rightarrow [f] p + [f \wedge g] p \approx [f] p$
 $absorbbycondition \{f\} \{g\} \{p\} =$
 $sym orcondition \langle trans \rangle condcong \vee absorbs \wedge refl$

$conditionregularisation' : \forall f\ g\ p\ q\ z$
 $\rightarrow p \gg q \approx p + q + z$
 $\rightarrow [f] p \gg [g] q$
 $\approx [f] p + [g] q + [f \wedge g] z$

$conditionregularisation' f\ g\ p\ q\ z\ eq =$
 $begin$
 $[f] p \gg [g] q$
 $\approx \langle conditionregularisation \rangle$
 $[f] p + [g] q + [f \wedge g] (p \gg q)$
 $\approx \langle +cong refl (condcong (isEquivalence EQC.refl) eq) \rangle$
 $[f] p + [g] q + [f \wedge g] (p + q + z)$
 $\approx \langle +cong refl (conditional+ \langle trans \rangle +cong conditional+ refl) \rangle$
 $[f] p + [g] q + ([f \wedge g] p + [f \wedge g] q + [f \wedge g] z)$

$$\begin{aligned}
 &\approx \langle \text{sym} + \text{assoc} \langle \text{trans} \rangle + \text{cong} (+\text{assoc} \langle \text{trans} \rangle) \\
 &\quad (+\text{cong refl} (\text{sym} + \text{assoc} \langle \text{trans} \rangle + \text{cong} + \text{comm refl} \langle \text{trans} \rangle + \text{assoc}) \\
 &\quad \langle \text{trans} \rangle \text{sym} + \text{assoc})) \text{refl} \rangle \\
 &([f] p + [f \wedge g] p) + ([g] q + [f \wedge g] q) + [f \wedge g] z \\
 &\approx \langle +\text{cong} (+\text{cong absorbbycondition} (+\text{cong refl} (\text{condcong} \wedge \text{comm refl}) \\
 &\quad \langle \text{trans} \rangle \text{absorbbycondition})) \text{refl} \rangle \\
 &[f] p + [g] q + [f \wedge g] z
 \end{aligned}$$

■

$$\begin{aligned}
 +\text{correct}' &: \forall x y \rightarrow \text{fromNF } x + \text{fromNF } y \approx \text{fromNF } (x +' y) \\
 +\text{correct}' [] y &= \text{GT}. + \text{identity}^l (\text{fromNF } y) \\
 +\text{correct}' (x : : xs) y &= +\text{assoc} \langle \text{trans} \rangle + \text{cong refl} (+\text{correct}' xs y) \\
 \text{sumNodes} &= \text{foldr } _ + _ \varepsilon \circ \text{map fromNode} \\
 \text{absorption}^l &: \forall \{x y\} \rightarrow x + x \gg y \approx x \gg y \\
 \text{absorption}^l &= +\text{comm} \langle \text{trans} \rangle \text{GT}. \text{absorption}^l \\
 \text{absorption}^r &: \forall \{x y\} \rightarrow y + x \gg y \approx x \gg y \\
 \text{absorption}^r &= +\text{comm} \langle \text{trans} \rangle \text{GT}. \text{absorption}^r \\
 \text{newArrowsgood} &: \forall x y \rightarrow \text{fromNode } x \gg \text{fromNode } y \\
 &\approx \text{fromNode } x + \text{fromNode } y + \text{sumNodes} (\text{newArrows } x y) \\
 \text{newArrowsgood} (\text{inj}_1 x) (\text{inj}_1 y) &= \\
 &\quad \text{sym} (+\text{cong refl GT}. + \text{identity}^r \langle \text{trans} \rangle \text{absorption}) \\
 \text{newArrowsgood} (\text{inj}_1 x) (\text{inj}_2 (y_1, y_2)) &= \\
 &\quad \text{sym} (+\text{cong} + \text{comm} (+\text{cong refl GT}. + \text{identity}^r) \\
 &\quad \langle \text{trans} \rangle (+\text{assoc} \langle \text{trans} \rangle + \text{comm} \langle \text{trans} \rangle \\
 &\quad + \text{cong} (\text{sym} + \text{assoc} \langle \text{trans} \rangle + \text{cong absorption}^l \text{refl}) \text{refl}) \\
 &\quad \langle \text{trans} \rangle \text{sym decomposition} \\
 &\quad \langle \text{trans} \rangle \gg \text{assoc}) \\
 \text{newArrowsgood} (\text{inj}_2 (x_1, x_2)) (\text{inj}_1 y) &= \\
 &\quad \text{sym} (+\text{assoc} \langle \text{trans} \rangle \\
 &\quad + \text{cong refl} (\text{sym} + \text{assoc} \langle \text{trans} \rangle + \text{cong absorption}^r \text{GT}. + \text{identity}^r) \\
 &\quad \langle \text{trans} \rangle \text{sym} + \text{assoc} \langle \text{trans} \rangle \\
 &\quad \text{sym decomposition}) \\
 \text{newArrowsgood} (\text{inj}_2 (x_1, x_2)) (\text{inj}_2 (y_1, y_2)) &= \\
 &\quad \text{sym} (+\text{assoc} \langle \text{trans} \rangle + \text{cong refl} \\
 &\quad (+\text{cong} (\text{sym GT}. \text{idempotence}) (+\text{cong refl} (+\text{cong refl} \\
 &\quad (+\text{cong refl GT}. + \text{identity}^r)) \langle \text{trans} \rangle \text{sym} + \text{assoc} \langle \text{trans} \rangle + \text{comm}) \\
 &\quad \langle \text{trans} \rangle \text{sym} + \text{assoc} \\
 &\quad \langle \text{trans} \rangle + \text{cong} (+\text{assoc} \langle \text{trans} \rangle + \text{cong refl} + \text{comm}) \text{refl} \\
 &\quad \langle \text{trans} \rangle + \text{comm} \\
 &\quad \langle \text{trans} \rangle \text{sym} + \text{assoc}
 \end{aligned}$$

$$\begin{aligned}
 & \langle \text{trans} \rangle + \text{cong} (\text{sym decomposition}) (\text{sym decomposition}) \\
 & \langle \text{trans} \rangle + \text{cong} \gg \text{assoc} \gg \text{assoc} \\
 &) \\
 & \langle \text{trans} \rangle \text{sym} + \text{assoc} \\
 & \langle \text{trans} \rangle \text{sym decomposition} \\
 \text{allmult} : \forall l \ c \rightarrow \text{fromNF} (\text{List.map} (\text{flip } -, - \ c) \ l) \approx [c] \ \text{sumNodes} \ l \\
 \text{allmult} [] \ c = \text{sym conditional} \varepsilon \\
 \text{allmult} (x : : xs) \ c = + \text{cong refl} (\text{allmult} \ xs \ c) \langle \text{trans} \rangle \text{sym conditional} + \\
 \gg_1 \text{preserves} : \forall x \ y \rightarrow \text{fromLit} \ x \gg \text{fromLit} \ y \approx \text{fromNF} (x \gg_1 y) \\
 \gg_1 \text{preserves} (x, f) (y, g) = \\
 \begin{aligned}
 & \text{begin} \\
 & \text{fromLit} (x, f) \gg \text{fromLit} (y, g) \\
 & \approx \langle \text{conditionregularisation}' f \ g \ - \ - \ - (\text{newArrowsgood} \ x \ y) \rangle \\
 & \text{fromLit} (x, f) + \text{fromLit} (y, g) + [f \wedge g] \ \text{sumNodes} (\text{newArrows} \ x \ y) \\
 & \approx \langle + \text{assoc} \langle \text{trans} \rangle + \text{cong refl} \\
 & \quad (+ \text{cong refl} (\text{sym} (\text{allmult} (\text{newArrows} \ x \ y) (f \wedge g)))) \rangle \\
 & \text{fromNF} ((x, f) : : (y, g) : : (\text{map} (\text{flip } -, - \ (f \wedge g)) (\text{newArrows} \ x \ y))) \\
 & \approx \langle \text{refl} \rangle \\
 & \text{fromNF} ((x, f) \gg_1 (y, g)) \\
 & \blacksquare
 \end{aligned} \\
 \gg^r \text{preserves} : \forall x \ y \rightarrow \text{fromLit} \ x \gg \text{fromNF} \ y \approx \text{fromNF} (x \gg^r y) \\
 \gg^r \text{preserves} x [] = \gg \text{identity}^r \langle \text{trans} \rangle \text{sym GT.} + \text{identity}^r \\
 \gg^r \text{preserves} x (y : : ys) = \\
 \begin{aligned}
 & \text{distrib}^l \langle \text{trans} \rangle + \text{cong} (\gg_1 \text{preserves} \ x \ y) (\gg^r \text{preserves} \ x \ ys) \\
 & \langle \text{trans} \rangle + \text{correct}' (x \gg_1 y) (x \gg^r ys)
 \end{aligned} \\
 \gg \text{correct}' : \forall x \ y \rightarrow \text{fromNF} \ x \gg \text{fromNF} \ y \approx \text{fromNF} (x \gg' y) \\
 \gg \text{correct}' [] \ y = \gg \text{identity}^l \\
 \gg \text{correct}' (x : : xs) \ y = \\
 \begin{aligned}
 & \text{distrib}^r \langle \text{trans} \rangle (+ \text{cong} (\gg^r \text{preserves} \ x \ y) (\gg \text{correct}' \ xs \ y) \\
 & \langle \text{trans} \rangle + \text{correct}' (x \gg^r y) (xs \gg' y))
 \end{aligned} \\
 + \text{correct} : \forall x \ y \rightarrow x + y \approx \text{fromNF} (\text{normalize} \ x \ +' \ \text{normalize} \ y) \\
 \gg \text{correct} : \forall x \ y \rightarrow x \gg y \approx \text{fromNF} (\text{normalize} \ x \ \gg' \ \text{normalize} \ y) \\
 \text{normalisecorrect} : \forall f \rightarrow f \approx \text{fromNF} (\text{normalize} \ f) \\
 + \text{correct} \ x \ y = \\
 \begin{aligned}
 & + \text{cong} (\text{normalisecorrect} \ x) (\text{normalisecorrect} \ y) \langle \text{trans} \rangle \\
 & + \text{correct}' (\text{normalize} \ x) (\text{normalize} \ y)
 \end{aligned} \\
 \gg \text{correct} \ x \ y = \\
 \gg \text{cong} (\text{normalisecorrect} \ x) (\text{normalisecorrect} \ y) \langle \text{trans} \rangle
 \end{aligned}$$

```

    >> correct' (normalize x) (normalize y)
mapConditionscorrect : ∀ c x → [c] fromNF x ≈ fromNF (mapConditions (λ _ & c) x)
mapConditionscorrect c [] = conditionale
mapConditionscorrect c (x :: xs) =
    conditional + ⟨ trans ⟩ + cong (sym andcondition) (mapConditionscorrect c xs)

normalisecorrect (x + y) = +correct x y
normalisecorrect (x >> y) = >>correct x y
normalisecorrect ε = refl
normalisecorrect (var a) = sym truecondition ⟨ trans ⟩ sym GT. + identityr
normalisecorrect ([c] y) = condcong BEq.refl (normalisecorrect y) ⟨ trans ⟩
    mapConditionscorrect c (normalize y)

module PG.Normalizer (V : Set) (B : Set) where

open import Data.Empty
open import Data.Sum using (λ ⊕ _; inj1; inj2)
open import Function using (id; flip; λ _ ∘ _)
import Data.Product as Product
open Product using (λ _ , _; λ _ × _)
open import Level using () renaming (zero to 0)

open import PG.Formulae using
    (BoolFormula; λ _ ∧ _; λ _ ∨ _; ⊤; var; PGFormula; module PGFormula; pgeval)
open import Relation.Binary
open import Relation.Binary.PropositionalEquality using (λ _ ≡ _)
import Data.List as List
open List using (foldr; List; concat; λ _ ++ _; λ _ : _; []; map)

BF = BoolFormula B
PG = PGFormula BF V
Node = V ⊕ (V × V)
Lit = Node × BF
NF = List Lit

module WithOrder {λ _ < _ : V → V → Set} (ASTO : IsStrictTotalOrder λ _ ≡ λ _ < _) where

    module Semantics where
        open PGFormula

        fromNode : Node → PG
        fromNode (inj1 x) = var x
        fromNode (inj2 (x, y)) = var x >> var y

        fromLit : Lit → PG
        fromLit (node, cond) = [cond] fromNode node

```

```

    fromNF : NF → PG
    fromNF = foldr _+_ ε ∘ map fromLit
open Semantics public
open import Category.Monad
open RawMonad (List.monad {o})

    _+_ : NF → NF → NF
    a + b = a ++ b

    vertices : Node → List V
    vertices (inj1 x) = x : []
    vertices (inj2 (x, y)) = x :: y :: []

    newArrows : Node → Node → List Node
    newArrows p q = map inj2 (vertices p ⊗ vertices q)

    _≫1 _ : Lit → Lit → List Lit
    (p, f) ≻1 (q, g) =
        (p, f) :: (q, g) :: (map (flip _-, _ (f ∧ g)) (newArrows p q))

    _≫r _ : Lit → NF → NF
    lit ≻r [] = lit :: []
    lit ≻r (x :: xs) = (lit ≻1 x) + (lit ≻r xs)

    _≫ _ : NF → NF → NF
    [] ≻ b = b
    (h :: t) ≻ b = (h ≻r b) + (t ≻ b)

    mapConditions : (BF → BF) → NF → NF
    mapConditions f = map (Product.map id f)

    fromVar : V → NF
    fromVar x = (inj1 x, ⊤) :: []

    addCondition : BF → NF → NF
    addCondition = mapConditions ∘ _ ∧ _

    normalize : PG → NF
    normalize = pgeval

    _+_
    _≫ _
    []
    fromVar
    addCondition

open import PG.PGAlgebra using (BoolOps)
module PG.PGAlgebra where

```



```

open import PG.GraphAlgebra
open import PG.Eq
open import Algebra.Structures

record BoolOps (B : Set) : Set where
infix 8 ¬
infixr 7 _ ∧ _
infixr 6 _ ∨ _
field
  _ ∨ _ : B → B → B
  _ ∧ _ : B → B → B
  ¬ : B → B
  ⊤ : B
  ⊥ : B

record PGOps (B G : Set) : Set where
infix 20 [-] _
field
  graphops : GraphOps G
  [-] _ : B → G → G
open GraphOps graphops public

IsBoolAlg : {B : Set} → (Beq : Eq B) → (boolops : BoolOps B) → Set
IsBoolAlg (equality eq) ops = let open BoolOps ops in
  IsBooleanAlgebra eq _ ∨ _ _ ∧ _ ¬ ⊤ ⊥

record IsPG {B G : Set}
  (Beq : Eq B) (Geq : Eq G)
  (PGOps : PGOps B G) (Bops : BoolOps B) : Set where
open PGOps PGOps
open Eq Geq
open Eq Beq renaming (_ ≈ _ to _B ≈ _)
open BoolOps Bops
field
  isgraph : IsGraph Geq graphops
  isbool : IsBoolAlg Beq Bops
  condcong : ∀ {f g x y} → f B ≈ g → x ≈ y → [f] x ≈ [g] y
  truecondition : ∀ x → [⊤] x ≈ x
  falsecondition : ∀ x → [⊥] x ≈ ε
  andcondition : ∀ f g x → [f ∧ g] x ≈ [f] [g] x
  orcondition : ∀ f g x → [f ∨ g] x ≈ [f] x + [g] x
  conditional+ : ∀ f x y → [f] (x + y) ≈ [f] x + [f] y

```

```

    conditional  $\gg$  :  $\forall f\ x\ y \rightarrow [f]\ (x \gg y) \approx [f]\ x \gg [f]\ y$ 
open PGOps PGops public
open Eq Geq public
open IsGraph isgraph public
open IsBooleanAlgebra isbool public using ()
open BoolOps Bops public
record PG (B G : Set) : Set1 where
field
    Beq : Eq B
    Geq : Eq G
    pgops : PGOps B G
    Bops : BoolOps B
    isPg : IsPG Beq Geq pgops Bops
open IsPG isPg public

```

Bibliography

- [1] *MSP430x4xx Family User's Guide*.
- [2] International technology roadmap for semiconductors: 2011 edition. <http://www.itrs.net/Links/2011ITRS/Home2011.htm>, 2011.
- [3] Arseniy Alekseyev, Victor Khomenko, Andrey Mokhov, Dominic Wist, and Alex Yakovlev. Improved parallel composition of labelled Petri nets. In Benoît Cailaud, Josep Carmona, and Kunihiro Hiraishi, editors, *ACSD*, pages 131–140. IEEE, 2011.
- [4] Arseniy Alekseyev, Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Optimisation of Balsa control path using STG resynthesis. In *UK Asynchronous Forum*, 2009.
- [5] Thorsten Altenkirch and Peter Morris. Indexed containers. In *LICS*, pages 277–285. IEEE Computer Society, 2009.
- [6] Samary I. Baranov. *Logic Synthesis for Control Automata*. Kluwer Academic Publishers, 1994.
- [7] Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge International Series on Parallel Computation. Cambridge University Press, 2004.
- [8] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalijs. The VLSI-programming language TANGRAM and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, 1991.

- [9] Aleš Bizjak and Andrej Bauer. *ALG User Manual*, Faculty of Mathematics and Physics, University of Ljubljana, 2011.
- [10] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous systems*. PhD thesis, Stanford University, October 1984.
- [11] Tiberiu Chelcea and Steven M. Nowick. Resynthesis and peephole transformations for the optimization of large-scale asynchronous systems. In *Proc. DAC'02*, pages 405–410, 2002.
- [12] Clasp tool. URL: <http://www.cs.uni-potsdam.de/clasp/>.
- [13] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. PETRIFY: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. Information and Systems*, E80-D, 3:315–325, 1997.
- [14] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002.
- [15] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos Sotiriou. Desynchronisation: synthesis of asynchronous circuits from synchronous specifications. *IEEE Transaction on Computer Aided Design*, 25(10):1904–1921, October 2006.
- [16] Jordi Cortadella, Luciano Lavagno, Djavad Amiri, Jon‘as Casanova, Carlos Macián, Ferran Martorell, Juan A. Moya, Luca Necchi, Danil Sokolov, and Emre Tuncer. Narrowing the margins with elastic clocks. In *IEEE International Conference on Integrated Circuit Design and Technology (ICICDT)*, pages 146–150, 2010.
- [17] Crescenzo D’Alessandro, Andrey Mokhov, Alexander Bystrov, and Alex Yakovlev. Delay/phase regeneration circuits. In *13th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 2007*, pages 105–116, March 2007.

- [18] Nils Anders Danielsson. Agda standard library. <http://wiki.portal.chalmers.se/agda/agda.php?n=Libraries.StandardLibrary>, 2011.
- [19] Giovanni de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.
- [20] Jo C. Ebergen. Arbiters: an exercise in specifying and decomposing asynchronously communicating components. *Sci. of Computer Programming*, 18:223–245, 1992.
- [21] Doug A. Edwards and Andrew Bardsley. Balsa: an Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, 2002.
- [22] Doug A. Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45 (1):12–18, jan 2002.
- [23] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, pages 333–336, 2004.
- [24] Xin Fan, Miloš Krstić, and Eckhard Grass. Analysis and optimization of pausable clocking based gals design. In *IEEE International Conference on Computer Design (ICCD)*, pages 358–365, 2009.
- [25] Francisco Fernández-Nogueira and Josep Carmona. Logic synthesis of handshake components using structural clustering techniques. In Lars Svensson and José Monteiro, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 5349 of *Lecture Notes in Computer Science*, pages 188–198. Springer Berlin Heidelberg, 2009.
- [26] Charles Antony Richard Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- [27] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Koen Claessen and Nikhil Swamy, editors, *PLPV*, pages 49–60. ACM, 2012.
- [28] Mark B. Josephs and Dennis Furey. Delay-insensitive interface specification and synthesis. In *Proc. DATE’00*, pages 169–173, 2000.

- [29] Joep Kessels and Ad Peeters. The tangram framework: asynchronous circuits for low power. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 255–260, 2001.
- [30] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting state encoding conflicts in STG unfoldings using SAT. *Fundam. Inf.*, 62(2):221–241, 2004.
- [31] Victor Khomenko, Mark Schäfer, and Walter Vogler. Output-determinacy and asynchronous circuit synthesis. *Fundamenta Informaticae*, 88(4):541–579, 2008. Special Issue on Best Papers from ACSD’07.
- [32] Victor Khomenko, Mark Schäfer, Walter Vogler, and Ralf Wollowski. STG decomposition strategies in combination with unfolding. *Acta Informatica*, 46(6):433–474, 2009.
- [33] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. *Proc. ASYNC’96*, pages 233–243, 1996.
- [34] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous cad tools. *IEEE Design & Test of Computers*, 19(4):107–117, 2002.
- [35] Jong-eun Lee, Kiyoun Choi, and Nikil Dutt. Efficient instruction encoding for automatic instruction set design of configurable asips. In *Proc. of the International Conference on Computer-aided Design (ICCAD)*, 2002.
- [36] Alain J. Martin and Mika Nyström. Asynchronous techniques for system-on-chip design. *IEEE Proceedings*, 94:1089–1120, 2006.
- [37] Ross M. McConnell and Fabien de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Mathematics*, 145(2):198–209, 2005.
- [38] Prabhat Mishra and Nikil 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.
- [39] Andrey Mokhov. *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, 2009.

- [40] Andrey Mokhov, Arseniy Alekseyev, and Alex Yakovlev. Encoding of processor instruction sets with explicit concurrency control. *IET Computers & Digital Techniques*, 5(6):427–439, 2011.
- [41] Andrey Mokhov, Alexei Iliasov, Danil Sokolov, Maxim Rykunov, Alex Yakovlev, and Alexander Romanovsky. Synthesis of processor instruction sets from high-level ISA specifications. *IEEE Transactions on Computers*, 2014 (in print).
- [42] Andrey Mokhov, Victor Khomenko, Arseniy Alekseyev, and Alex Yakovlev. Algebra of Parametrised Graphs. Technical Report CS-TR-1307, School of Computing Science, Newcastle University, 2011. URL: <http://www.cs.ncl.ac.uk/publications/trs/abstract/1307>.
- [43] Andrey Mokhov, Victor Khomenko, Arseniy Alekseyev, and Alex Yakovlev. Algebra of parameterised graphs. In Jens Brandt and Keijo Heljanko, editors, *ACSD*, pages 22–31. IEEE, 2012.
- [44] Andrey Mokhov, Danil Sokolov, and Alex Yakovlev. Adapting asynchronous circuits to operating conditions by logic parametrisation. In *18th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 17–24, 2012.
- [45] Andrey Mokhov and Alex Yakovlev. Conditional Partial Order Graphs: Model, Synthesis and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.
- [46] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):68–70, April 1965.
- [47] Robert Mullins and Simon Moore. Demystifying data-driven and pausable clocking schemes. In *Proc. International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 175–185, 2007.
- [48] Achim Nohl, Volker Greive, Gunnar Braun, Andreas Hoffman, Rainer Leupers, Oliver Schliebusch, and Heinrich 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.

- [49] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- [50] PCOMP tool. URL: <http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/pcomp>.
- [51] Ad Peeters and Mark de Wit. *HASTE Manual, v. 3.0*. Handshake Solutions, 2005. URL: <http://handshakesolutions.com/Technology/Haste/Article-14902.html>.
- [52] Marco A. Peña and Jordi Cortadella. Combining process algebras and Petri nets for the specification and synthesis of asynchronous circuits. In *Proc. ASYNC'96*, pages 222–232. IEEE Computer Society, 1996.
- [53] Luis A. Plana, Sam Taylor, and Doug A. Edwards. Attacking control overhead to improve synthesised asynchronous circuit performance. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 703–710, 2005.
- [54] Ivan Poliakov, Victor Khomenko, and Alex Yakovlev. Workcraft - a framework for interpreted graph models. In *Petri Nets*, pages 333–342, 2009.
- [55] Darsh Ranjan, Daijue Tang, and Sharad Malik. A Comparative Study of 2QBF Algorithms. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*. ACM, 2004.
- [56] Mark Schäfer. *DesiJ - A Tool for STG Decomposition*. Technical Report tr-11-2007, University of Augsburg, October 2007.
- [57] Mark Schäfer, Walter Vogler, Ralf Wollowski, and Victor Khomenko. Strategies for optimised STG decomposition. In *In Proceedings of ACSD*, pages 123–132, 2006.
- [58] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K.

- Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.
- [59] Danil Sokolov, Alexander Bystrov, and Alex Yakovlev. Direct mapping of low-latency asynchronous controllers from STGs. *IEEE Trans. CAD*, 26(6):993–1009, 2007.
- [60] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.
- [61] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. In Jordi Cortadella et al., editors, *Concurrency and Hardware Design*, Lect. Notes Comp. Sci. 2549, 152 – 190. Springer, 2002.
- [62] Fei Xia, Andrey Mokhov, Yu Zhou, Yuan Chen, Isi Mitrani, Delong Shang, Danil Sokolov, and Alex Yakovlev. Towards power-elastic systems through concurrency management. *Computers Digital Techniques, IET*, 6(1):33–42, January 2012.
- [63] Alex Yakovlev, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Maciej Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, 9:189–233, 1996.
- [64] Alex Yakovlev and Albert M. Koelmans. Petri nets and Digital Hardware Design Lectures on Petri Nets II: Applications. *Advances in Petri Nets, Lecture Notes. Computer Science*, 1492:154–236, 1998.
- [65] 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)*, 2002.