μSystems Research Group

School of Engineering

Hardware Synthesis from High-level Scenario Specifications

Alessandro de Gennaro

Thesis submitted for the degree of *Doctor of Philosophy*

July 2019

Abstract

The behaviour of many systems can be partitioned into *scenarios*. These facilitate engineers' understanding of the specifications, and can be composed into efficient implementations via a form of high-level synthesis. In this work, we focus on highly concurrent systems, whose scenarios are typically described using concurrency models such as partial orders, Petri nets and data-flow structures.

In this thesis, we study different aspects of hardware synthesis from high-level scenario specifications. We propose new formal models to simplify the specification of concurrent systems, and algorithms for hardware synthesis and verification of the scenario-based models of such systems. We also propose solutions for mapping scenario-based systems on silicon and evaluate their efficiency.

Our experiments show that the proposed approaches improve the design of concurrent systems. The new formalisms can break down complex specifications into significantly simpler scenarios automatically, and can be used to fully model the data-flow of operations of reconfigurable event-driven systems. The proposed heuristics for mapping the scenarios of a system to a digital circuit supports encoding constraints, unlike existing methods, and can cope with specifications comprising hundreds of scenarios at the cost of only 5% of area overhead compared to exact algorithms.

These experiments are driven by three case studies: (1) hardware synthesis of control architectures, e.g. microprocessor control units; (2) acceleration of the ordinal pattern encoding, i.e. an algorithm for detecting repetitive patterns within data streams; (3) and acceleration of computational drug discovery, i.e. computation of shortest paths in large protein-interaction networks.

Our findings are employed to design two prototypes, which have a practical value for the considered case studies. The ordinal pattern encoding accelerator is asynchronous, highly resilient to unstable voltage supply, and designed to perform a range of computations via runtime reconfiguration. The drug discovery accelerator is synchronous, and up to three orders of magnitude faster than conventional software implementations.

Acknowledgements

First and foremost, I would like to thank my first PhD supervisor *Andrey Mokhov*. He introduced me to the world of academic research with inspiring discussions and countless ideas. His infectious passion inspired me to improve and be attentive to detail. For all of this, I will be always grateful. I also would like to thank my second PhD supervisor *Alex Yakovlev* for his valuable guidance. His incredible dedication and excitement for his work is admirable and has been a source of inspiration for the whole time of my PhD studies.

I am also thankful to all people in my research team (the μSystems Research Group) that supported my work. In particular, a great acknowledgement goes to *Ghaith Tarawneh*, *Reza Ramezani*, *Jonathan Beaumont* and *Sergey Mileiko*. A special mention is for *Danil Sokolov*, who has always been supportive and ready to help when I was overwhelmed by doubts. He has been very important in my growth as researcher, and I would never be thankful enough for this. Another mention is for *Paulius Stankaitis*, my dear friend and colleague who shared with me exciting and frustrating research moments at Newcastle University.

Last but not least, part of my acknowledgements goes to those who have been close to me emotionally, and helped sustain the encountered difficulties: *my family* and *friends*. A special acknowledgement is dedicated to my partner *Elisa Passaretti*, who supported me emotionally and practically by making my dissertation more pleasant to read.

Contents

List of Publications

Journal Article

1. A. de Gennaro, P. Stankaitis, A. Mokhov. "Efficient Composition of Scenario-based Hardware Specifications". In *IET Computers & Digital Techniques*, vol. 13, no. 2, pp. 57-69, 2019.

Book Chapters

1. A. Mokhov, A. de Gennaro, G. Tarawneh, G. Lukyanov, S. Mileiko, J. Scott, A. Yakovlev, A. Brown. "Language and Hardware Acceleration Backend for Graph Processing". In *Languages, Design Methods, and Tools for Electronic System Design - Selected Contributions from FDL 2017*, Springer, 2018, in Press.
2. A. Brown, D. Thomas, J. Reeve, G. Tarawneh, A. de Gennaro, A. Mokhov, M. Naylor, T. Kazmierski. "Distributed Event-based Computing". In *Parallel Computing is Everywhere*, IOS press, 2018, pp. 583-592.

Conference Papers

1. D. Sokolov, A. de Gennaro, A. Mokhov. "Reconfigurable Asynchronous Pipelines: from Formal Models to Silicon". In *Design, Automation and Test in Europe (DATE)*, March 2018, Dresden, Germany.
2. A. Mokhov, J. Cortadella and A. de Gennaro. "Process Windows". In *International Conference on Application of Concurrency to System Design*, 2017, Zaragoza, Spain. IEEE.

3. A. Mokhov, A. de Gennaro, G. Tarawneh, G. Lukyanov, S. Mileiko, J. Scott, A. Yakovlev and A. Brown. "Language and Hardware Acceleration Backend for Graph Processing". In *Forum on specification & Design Languages*, 2017, Verona, Italy.

4. G. Lukyanov, A. de Gennaro, A. Mokhov, P. Stankaitis, M. Rykunov. "Prototyping Resilient Processing Cores in Workcraft". In *2nd International Workshop on Resiliency in Embedded Electronic Systems*, 2017, Lausanne, Switzerland. IEEE.

5. A. de Gennaro, P. Stankaitis and A. Mokhov. "A Heuristic Algorithm for Deriving Compact Models of Processor Instruction Sets". In *15th International Conference on Application of Concurrency to System Design*, 2015, Brussels, Belgium. IEEE.

List of Public Presentations

1. FANTASI: FAst NeTwork Analysis in SIlicon. Winning project of the *Xilinx Open Hardware* competition 2018. Xilinx, Dublin (Ireland).
2. Prototyping Resilient Processing Cores in Workcraft. *2nd International Workshop on Resiliency in Embedded Electronic Systems*. 2017, Lausanne (Switzerland).
3. Adopting a Systematic Approach to the Design of Processor Instruction Sets. *ARM Research Summit*, 2016, Cambridge (UK).
4. Modelling Concurrency in Processor Instruction Sets. *12th LASER Summer School on Software Engineering - Concurrency: the next frontiers*. 2015, Elba Island (Italy).
5. A Heuristic Algorithm for Deriving Compact Models of Processor Instruction Sets. *15th International Conference on Application of Concurrency to System Design*, 2015, Brussels (Belgium).

Technical Demos

1. FPGA-based Hardware Accelerator for Drug Discovery. *Design Automation & Test in Europe* (DATE) 2018, University Booth. Dresden (Germany).
2. Reconfigurable Self-timed Dataflow Accelerator. *Design Automation & Test in Europe* (DATE) 2017, University Booth. Lausanne (Switzerland).

List of Figures

List of Tables

Chapter 1

Introduction

Electronic Design Automation (EDA) has acquired increasingly more importance in the fields of computing science and electronic engineering since the 70s, when Gordon Moore predicted the steady growth of the number of transistors per square inch in integrated circuits (IC) [1]. EDA has contributed to the expansion of the Very Large Scale Integration (VLSI) industry by allowing designers to use automated and verified approaches in response to the growth of IC complexity. Its knowledge has also been applied to solve problems in the fields of chemistry, biology and physics [2].

In the context of microelectronics, EDA encloses the set of mathematical models, algorithms and software tools used for the design, synthesis and fabrication of integrated circuits. It encompasses methodologies to abstract the complexity of electronic systems, and refines their high-level description to a more accurate and physical low-level implementation. It also covers the verification of such systems at all levels of abstraction, from model-checking of mathematical representations, to formal verification and test of physical systems.

Before EDA, integrated circuits were designed by hand. In the 70s, designers started to rely on geometry software for the generation of tapes, used by photoplotters for the production of ICs. `Calma`, a known company of those years, introduced the Graphic Data System (`GDS`) format in 1971, still used nowadays in its second version (`GDSII`) released in 1978 (see its sixth documentation release [3]). In the 80s, Mead and Conway, authors of

the book *"Introduction to VLSI systems"* [4], introduced the first programming languages synthesisable to silicon, allowing designers to produce more sophisticated and complex electronic devices faster. In the following years, the need for a higher production efficiency led to the birth of the business methodology of outsourcing the fabrication of electronic devices to specialised companies, known as "Fabless Semiconductor Industry" and pioneered by TSMC (Taiwan Semiconductor Manufacturing Company). This allowed EDA to become an independent field, whose most relevant companies are now Cadence, Synopsys and Mentor Graphics.

Research in the EDA field has been very intense since its birth. In the report *"The Tides of EDA"* [5], published for the 40th edition of the Design Automation Conference[1], Sangiovanni Vincentelli highlights the most relevant discoveries in the field, such as the ones in the context of circuit simulation, hardware description languages and high-level design. The report also shows the steady growth of the number of papers published by academia, industry and vendors from 1964 to 2003, which marks the relevance of the field.

In this work, we deal with digital design techniques based on high-level behavioural modelling. Raising the level of abstraction is crucial when dealing with complex systems for multiple reasons:

- *Abstraction*: it separates the physical from theoretical complexity of a problem. As an example, a digital hardware component is easier to describe with the usage of logic gates rather than with transistors and voltage/current levels.
- *Verification*: a system implementation has to be checked against its initial specification to make sure that the former meets the latter at all times. The verification phase is instrumental to obtain a "bug-free" system, and it needs to be carefully planned as its duration affects the cost of a final product. Engineers have different options to verify a system. One could, for example, check that the system gives the right answer (outputs) for any possible questions (inputs) – this approach is unfeasible for large systems. Another approach would be to describe the system relying on a set of mathematical rules that enables one to prove that errors cannot happen under certain conditions – this approach is named *formal verification*. Formal verification

[1]The Design Automation Conference (DAC) is one of the most relevant conferences in the field of design automation.

is a powerful tool that is used for avoiding subtle design mistakes, e.g. see the flaw affecting the Floating-Point Unit of the Intel Pentium microprocessor in 1994 [6].

- *Behavioural description*: it is arguably easier to describe the behaviour of a system rather than its hardware structure. From the former, it is possible to derive efficient implementations.

- *Portability & Reusability*: an abstract description of a system can be implemented by using different implementation libraries. This enables companies to reuse previously designed components into newer devices, and to commercialise them as intellectual properties (IP).

The above reasons lead to an increased design productivity [5].

Motivated by the above, we focus on a narrow research area within the field of *high-level behavioural modelling: the design methodologies based on behavioural scenarios*. This chapter is divided as follows. Section 1.1 discusses the new challenges of modern digital design. Section 1.2 introduces the idea of scenario-based design and describes the methodologies that we discuss in the thesis. Finally, Sections 1.3 and 1.4 outline the thesis contributions and organisation.

1.1 New design challenges

Design complexity

Hardware systems become more complex every year. Processors, for instance, are requested to handle diverse types of applications: ranging from video and audio processing to management of high-speed connections. They support new features and application-specific instructions [7], and integrate a growing number of processing cores and IP components following the need for IP reuse [8].

Since the 70s, Intel has been one of the leading manufacturers of general purpose microprocessors. Its products represent an example of how integrated circuits and processor architectures evolved since the birth of EDA [9].

- In 1971, the company's first microprocessor was the four-bit 4004, which was meant

to work in conjunction with three other chips: the 4001 (Read Only Memory), 4002 (Random Access Memory) and the 4003 (Shift register).

- In 1978, Intel released the 8086, the first microprocessor based on the x86 Instruction Set 16-bit Architecture. This processor was followed by the 80186 in 1982, which was one of the first processors to have a built-in clock-generator and interrupt controllers.
- In 1985, Intel's first 32-bit RISC processor was released.
- In 1989, the newly released 80486 integrated a Floating-Point Unit and was the first processor to benefit from a cache memory.
- In the 90s, processors hit increasingly higher clock frequencies and performance by relying on smaller transistors and clever architectures for the execution of instructions.
- In 2005, Intel's first multi-core processor was the PENTIUM D, followed by the families of CORE 2 DUO and CORE 2 QUAD processors.
- The most recent families of microprocessors released by the company are the I3, i5, I7 and I9, which integrate an internal Graphics Processing Unit (GPU), from 2 to 18 internal cores depending on the model, three levels of cache memory, and the possibility to work at difference frequencies depending on the workload.

Such progresses have been made possible by the improvements of the design techniques led by EDA, which is continuously challenged by growingly better and more complex circuits.

Growing usage of asynchronous devices

Asynchronous architectures are slowly emerging in the semiconductor industry, which has always been mostly dominated by synchronous design. There are two solid reasons behind this growth. First, the wide growth of the Internet of Things (IoT) market. IoTs are indeed event-driven either at the node level [10], and at the global level due to distributed-nature of their communication protocol. The IoT market is pushing a

growing number of companies to invest and focus on this typology of design. Second, self-timed circuits have very different properties than synchronous circuits that allow them to be preferred in a number of applications. We shortly summarise few of the most relevant below.

- The lack of timing constraints enable *asynchronous devices to be bounded by the average case performance* rather than the worst case performance, as in the case of synchronous devices. This allows some architectures to be faster than their synchronous counterparts – e.g. see the Speedster FPGA by Achronix Semiconductor, which is claimed to be one of the fastest available in the market [11].

- Also, the absence of timing assumptions enables self-timed devices to *operate reliably at a wide range of speeds (voltages)*. This is a great advantage for those devices that are placed in critical environmental conditions, such as those in the area of biomedicine, which can thus function regardless of voltage supply variations or available energy (e.g. battery). A number of developed prototypes confirm this point, e.g. see the asynchronous ASIC prototype of the Intel 8051 [12], and the reconfigurable processor that we present in Chapter 5.

- *Lower latency* and *energy consumption*. The first because no time is wasted on waiting for next edge of the clock – data can propagate trough a circuit with virtually zero delay. The second because no energy is wasted when there is no data to process – the circuit goes to sleep automatically. Concrete examples showing these capabilities can be also found in the market: the UltraSPARC IIIi processor by Sun Microsystems with its asynchronous memory controller [13], and the asynchronous controller for a power buck converter [14] produced by the partnership of Newcastle University (μSystems Group) and Dialog Semiconductor.

In the light of the above, it is likely that asynchronous devices will cut a consistent portion of the semiconductor market in the next decades. This type of circuits, thus, needs to be understood and handled well by nowadays and future engineers. This work goes towards this direction, providing formal models and EDA tools for facilitating the design of asynchronous devices at different phases of the flow.

System concurrency

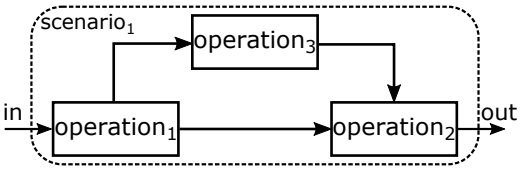
Last but not least, the need of performance driven by several applications (e.g. entertainment, real-time and big-data) has led to the development of a diverse type of architectural solutions for running software concurrently. As an example: (1) *Multi-core* architectures, consisting of a few highly-sophisticated processors communicating via shared resources (e.g. cache memories). (2) *Many-core* architectures, constituted by a large number of simple processors that communicate via network-based protocols (e.g. message passing), see [15]. (3) *In-memory processing* architectures, where the data is stored in RAM or flash memories, and it is processed in loco for avoiding data transfer delays imposed by the memory bandwidth, see [16].

Dealing with concurrency is known to be hard for humans. Thus, techniques for abstracting away the complexity derived by concurrency, or for minimising the human efforts and avoiding design mistakes are increasingly popular.

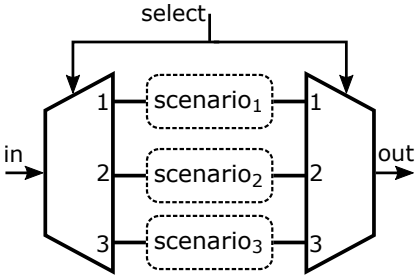
Which tools do designers have to tackle these challenges?

In response to the above challenges, designers can rely on an increased level of abstraction for hardware description. In the last decades, many languages have been extended with new statements and constructs for enhancing their expressive capabilities [17]. We report below some examples.

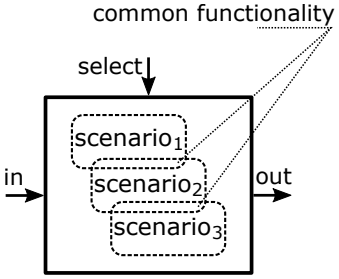
- *Verilog* and *VHDL*, with the possibility to describe a hardware component by its behaviour. They also embody `generate` statements, which provide the means for complex structural description. The latter allows engineers to use programming language constructs (e.g. if-then-else, for-loop) for describing hardware. This is important both for deriving compact and more readable hardware description, and for improving code reusability, i.e. a register made of n number of flip-flops can be described as a *generic* entity, which allows one to instantiate registers of any length.
- *SystemC*, which benefits from the high expressiveness of the C++ language, and represents an established approach for the high-level hardware description [18].
- *Bluespec SystemVerilog* and *Cλash*, which are two functional hardware description



(a) Structural composition.



(b) Inefficient scenario composition.



(c) Efficient scenario composition.

Figure 1.1: Different forms of hardware composition.

languages with their own syntax and parser [19].

- Domain-specific languages hosted by functional programming languages, e.g. *Chisel* hosted by Scala, *HardCaml* by OCaml, *Lava* and *Concepts* [20] by Haskell. These languages benefit from ad hoc functions for easier hardware description [19].

These languages approach the design of complex systems hierarchically, relying on the concept of *composition*. The latter allows engineers to specify a system by describing its constituent internal components separately, and then composing them to synthesise the final hardware implementation. In the next section, we expand this concept with the idea of behavioural scenario composition, and introduce the scenario-based methodologies that we deal with in this work.

1.2 Scenario-based design

Hardware composition is inherently structural. Transistors, logic gates, hardware modules, or more abstract operations can be interconnected to each other according to their input/output interfaces and causality dependencies, as shown in Figure 1.1a[2].

[2]Figure 1.1 has been inspired by [21].

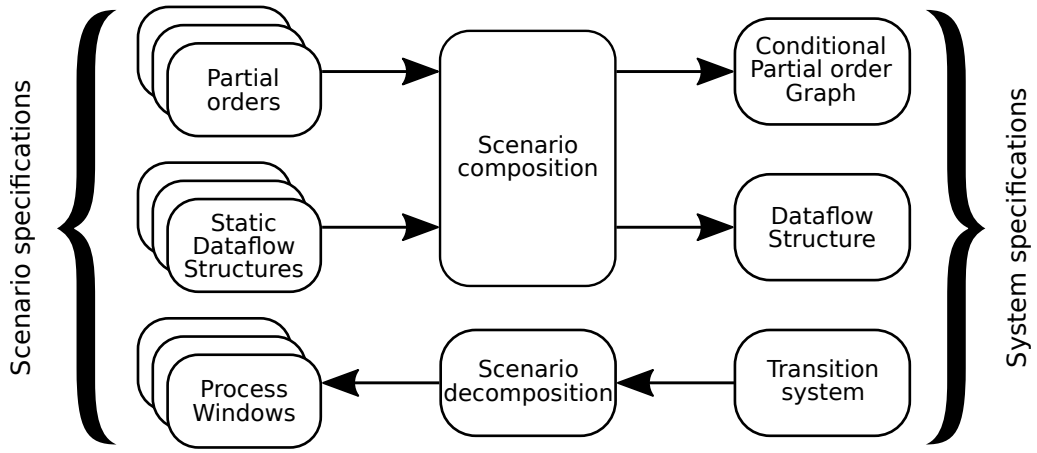


Figure 1.2: The methodologies that we study in this dissertation.

A different type of composition occurs when a system is described by the set of its constituent behaviours, which we denote as *scenarios*. Each scenario describes a possible trace of the system as a sequence of operations and their dependencies. Such scenarios can be composed using the following two approaches. (1) An inefficient approach to composition consists of synthesising every scenario in isolation, and using (de)multiplexors to run the right scenario, see Figure 1.1b. (2) On the other hand, a more efficient approach consists of deriving an implementation where common resources and functionalities are shared between scenarios, see Figure 1.1c. Here, we refer to efficiency as measure of *model compactness* – in the next chapters we show empirically that from more efficient models one can derive more optimised (e.g. in terms of area, power) hardware implementations.

In this work, we study three scenario-based methodologies that use different formal models as underlying structure. Throughout their description, we use the following **naming convention**. A set of scenarios that characterises all constituent behaviours of a system is denoted as *scenario specification*. A set of scenarios is composed into a *system specification*, which models the behaviour of a system, and comprises its scenarios and the interface for their selection, see `select` in Figures 1.1b and 1.1c.

Figure 1.2 shows the formal models of the considered scenario-based methodologies. In the following paragraphs, we briefly describe the reasons that led their investigation.
Methodology based on Conditional Partial Order Graphs - *Partial Orders* (PO) [22] are

used as formal model for scenario specifications, and are composed into a *Conditional Partial Order Graph* (CPOG) using efficient scenario composition approaches [23]. This methodology, originally conceived for the design of processor instruction sets (ISA), is important as it is supported by automated hardware synthesis flow, and by algorithms for deriving efficient implementations [24]. However, previously published algorithms do not scale to a high number of scenarios, nor support *composition constraints*, which allow to restrict certain aspects of the composition and reuse legacy IP blocks. In this work, we present a new scenario composition algorithm to overcome the above issues, and validate it on different case studies.

Methodology based on Dataflow Structures - *Static Dataflow Structures* (SDFS) [25] is a known model used for describing the data flow of operations of asynchronous circuits. However, according to its formal definition [26], SDFS cannot model dynamic reconfigurability, where one or more data paths are executed conditionally. To bridge this gap, we present the *Dataflow Structures* (DFS) formalism, which is the composition of a set of SDFSs and enables to fully model the behaviour of reconfigurable asynchronous circuits. We validate this methodology by prototyping a self-timed circuit on ASIC.

Methodology based on Process Windows - a system specification, described as a *Labelled Transition system* (LTS), is decomposed into a set of scenarios in the form of the new *Process Windows* (PW). The decomposition is performed to simplify the understanding of the system specification, as scenarios characterise its constituent (and simpler) parts.

1.3 Research contributions

Figure 1.3 describes the organisation of the core content of the thesis: the topics explored, their dependencies, and the sections where they are presented. In Chapter 3, we describe the existing behavioural models that we use in this work (see blue shadow). In Chapter 4, we present our contributions to the area of high-level scenario-based design (see green shadow). In Chapter 5, we describe the case study considered as validation of the contributions, (see red shadow). The contributions are also summarised below.

- **Efficient composition of scenarios** [27,28]. We present a new scenario composition algorithm and apply it to the CPOG methodology. Unlike existing methods, it

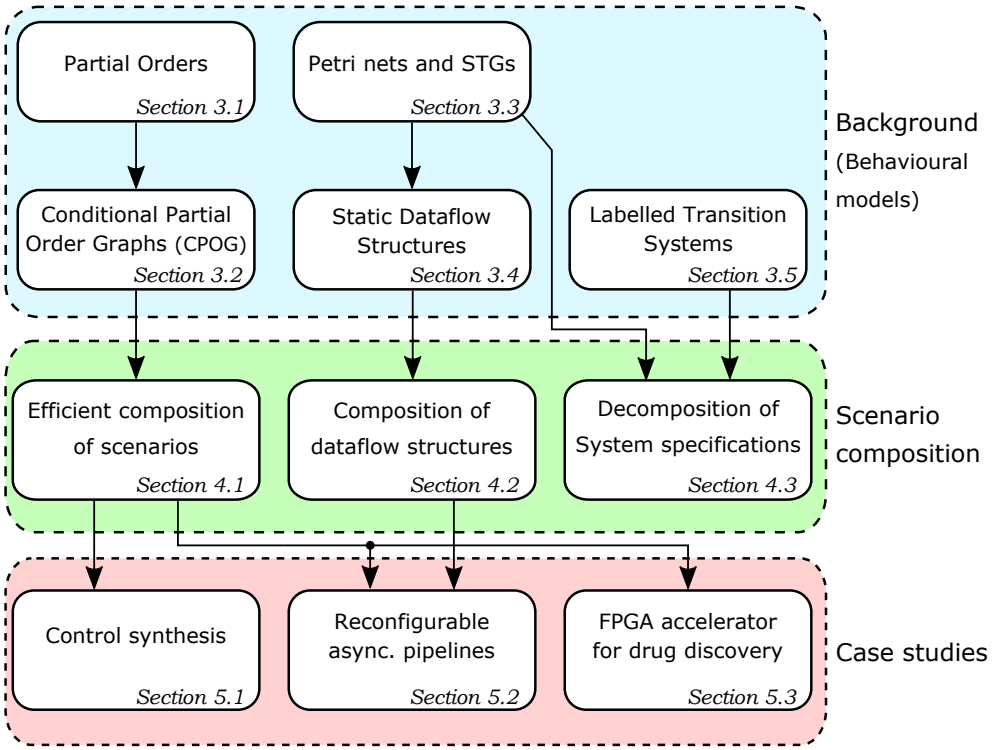


Figure 1.3: The topics of the thesis, their dependencies and the sections where they are presented.

scales to systems comprising hundreds of scenarios and supports composition constraints, which are important in real-life systems that heavily reuse IP blocks.

- **Composition of dataflow structures** [29, 30]. We present the Dataflow Structures formalism, whose models are derived by composing sets of Static Dataflow Structures. The new formalism enables to formally capture the behaviour of dynamically reconfigurable systems, thereby expanding the classes of problems that can be solved in the area of asynchronous formal modelling.

- **Decomposition of system specifications** [31]. We present the new Process Windows formalism, which allows to extract sets of scenarios from concurrent system specifications. In this area, I focused on elaborating an approach to synthesise the Boolean equations needed to orchestrate the behaviour of extracted scenarios. This feature has been automated in an open-source EDA tool [32].

- **Design automation of the open-source WORKCRAFT toolkit** [33]. The above

novel methodologies and design automation tools have been integrated into WORKCRAFT, which is an open-source [34] EDA tool for modelling, design, simulation and formal verification of many types of (microelectronic) systems. The developed design automation is described in the sections of the *Scenario composition* chapter.

- **Control synthesis** [28]. The new scenario composition algorithm is evaluated on a set of benchmarks that include various control hardware architectures. It is also compared to existing approaches for scenario composition, and to methodologies that use behavioural synthesis features to derive control architectures.

- **Reconfigurable asynchronous pipelines** [29, 30]. We describe a methodology based on the Dataflow Structures for designing and implementing reconfigurable asynchronous pipelines. The methodology is validated by designing, fabricating and testing an ASIC prototype that implements a reconfigurable asynchronous accelerator for the *Ordinal Pattern Encoding* [35]. We also compare our reconfigurable accelerator to a static implementation on silicon for characterising area, speed and power overhead of asynchronous dynamic reconfigurability.

- **FPGA accelerator for drug discovery** [36–38]. We use scenarios to design a synchronous accelerator for processing large scale *protein-interaction networks* [39], and prototype it on FPGA. In our approach, networks are not stored in memories, but are hardware components that can be simulated for collecting data. The accelerator is designed to compute shortest paths between proteins, and is up to three orders of magnitude faster than conventional software implementations.

1.4 Organisation and collaboration

The rest of the thesis is divided as follows. **Chapter 2** presents the application areas and ideas that motivated our work. We discuss that scenarios are beneficial for designing complex control units such as those driving microprocessors, and for designing highly resilient architectures with tight speed or power constraints. We also present our ideas of using scenarios to facilitate engineers' understanding of complex system specifications,

and to process scenarios of very large networks.

Chapter 3 reviews the formal models used across the dissertation – it provides the basic knowledge for understanding the presented results.

Chapters 4 and **5** are the core of our research (see Section 1.3), and include results that have been published previously. The work on the new scenario composition approach, and its application to the area of control synthesis is the result of a collaboration with my colleague P. Stankaitis and my advisor A. Mokhov. We published the initial idea and preliminary results at the *Application of Concurrency to System Design* conference in 2014 [27], and the algorithm and final results in the *IET Computers & Design Techniques* journal [28] recently. The work on the Dataflow Structures formalism, and its application to the ordinal pattern encoding case study is the result of a collaboration with D. Sokolov and my advisor. This work was published at the *Design Automation and Test in Europe* conference in 2018 [29], and will be extended for a journal [30]. The work on the Process Windows formalism has been led by A. Mokhov and J. Cortadella, and was published in 2017 at the *Application of Concurrency to System Design* conference [31]. Finally, the work on the FPGA accelerator for computational drug discovery originates from a collaboration of our research team and e-Therapeutics, which is the company that pioneered this experimental approach. We published our idea and hardware prototype to accelerate drug discovery at the *Forum on specification & Design Languages* conference in 2017 [36]. This conference paper was selected for being included as chapter of a book published by Springer [37], and part of our results was also included in a chapter of the recently published book entitled *Parallel Computing is Everywhere* [38].

Chapter 6, finally, summarises the findings of the research, highlights its limitations, and suggests a number of directions for future research.

Chapter 2

Motivation

In the first chapter, we introduced the idea of scenario-based design and the three methodologies that we discuss in this dissertation, which are based on Conditional Partial Order Graphs, Dataflow Structures, and Process Windows.

This chapter motivates the research by questioning: *"Why are scenario-based methodologies worth to be studied?"*. To answer this question, we will present some examples where scenarios and the idea of scenario composition (and decomposition) are already or would be useful.

2.1 Processor instruction sets

Our first motivating example comes from the domain of microprocessor design. In [24], the high-level methodology based on the Conditional Partial Order Graphs formalism has been applied to the design of such systems. The main idea was to represent instructions of a processor as behavioural scenarios for an easier analysis of the system from a higher level perspective, and for the synthesis of efficient processor control units. The importance of this application is also highlighted in [12], where this scenario-based methodology has been used to design the control unit of an asynchronous version of the Intel 8051 processor.

In this domain, behavioural scenarios describe the functionality and data flow of each

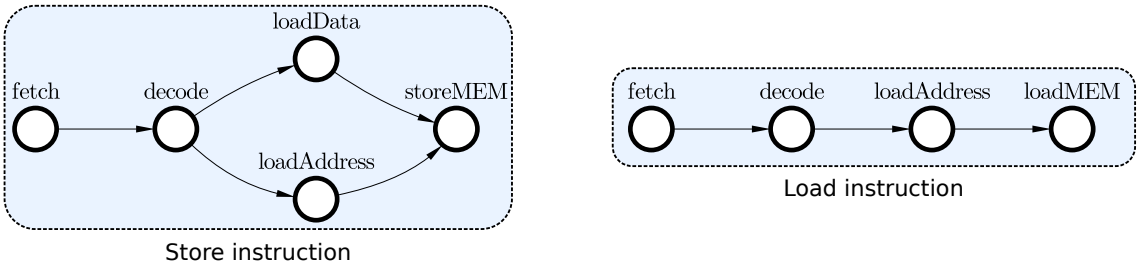


Figure 2.1: Two scenarios representing the store and load general instructions.

instruction. As an example, Figure 2.1 shows the scenario specification, in the form of partial orders, of the *load* and *store* instructions. Graphically, vertices represent datapath operations and arcs represent data dependencies between operations:

- the **Store instruction** scenario fetches (`fetch` operation) and decodes (`decode`) an instruction from the program memory, loads a data item $\mathcal{D}$ and a memory address $\mathcal{A}$ concurrently (`loadData` $\|$ `loadAddress`), and finally stores $\mathcal{D}$ into the memory via the `storeMem` operation, i.e. $\texttt{MEM}(\mathcal{A}) \leftarrow \mathcal{D}$.
- the **Load instruction** scenario, on the other hand, fetches (`fetch` operation) and decodes (`decode`) an instruction from the program memory, loads a memory address $\mathcal{A}$ and uses it to load a data item $\mathcal{D}$ from the memory, i.e. $\mathcal{D} \leftarrow \texttt{MEM}(\mathcal{A})$.

Such a formal specification is supported by automated hardware synthesis flow [23], and by algorithms for generating efficient hardware implementations [40]. As an example, the scenario composition process of the CPOG methodology takes as input the scenario specification in Figure 2.1, and generates the system specification on top of Figure 2.2, which is in the form of a CPOG. The CPOG models the processor control unit, which orchestrates the datapath modules and executes each instruction relying on the Boolean variable b (i.e. the opcode). As an example, if $b = 0$, the processor runs the store instruction by disabling the operation `loadMEM`, see the bottom-left scenario in Figure 2.2 where this operation is dashed.

Research has thus shown that CPOGs are effective for designing microprocessor control units. However, in this work, we show that the *state-of-the-art algorithms for scenario composition [23, 40] do not support systems composed of hundreds of scenarios, and lack of a mechanism for specifying composition constraints*, which are instrumental in this application where IP blocks are heavily employed. This is what motivates our work

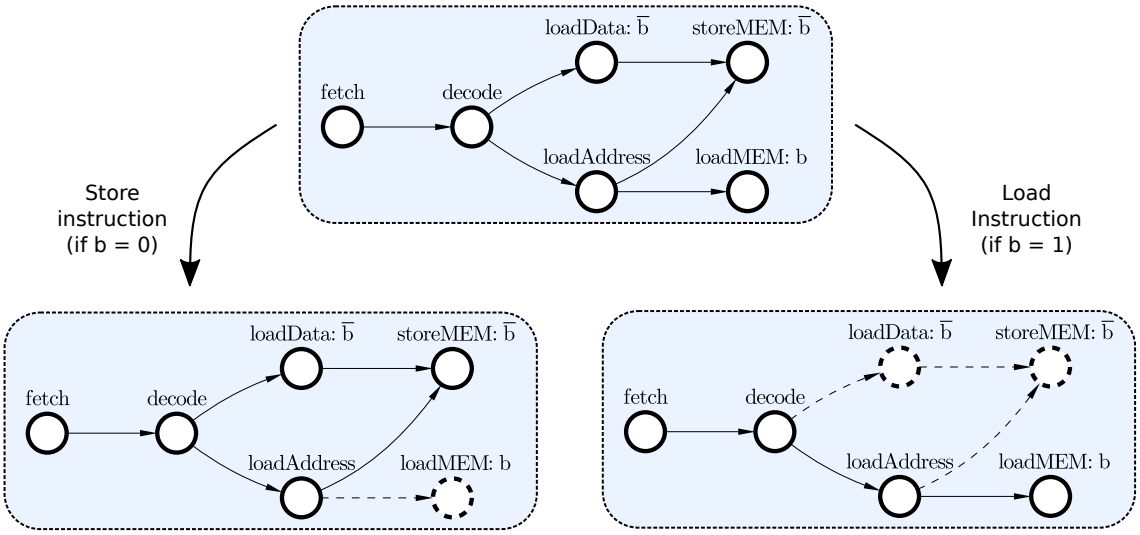


Figure 2.2: The system specification **(on top)**, in the form of a Conditional Partial Order Graphs, generated by the scenario specification in Figure 2.1. It selectively activates the two processor instructions **(at the bottom)** according to the value of the Boolean variable b, produced by the scenario composition process. As an example: if $b = 0$, the Store instruction will be executed.

on the development of a novel composition algorithm for the CPOG methodology, presented in Chapter 4.

2.2 Reconfigurable architectures

Previously, we introduced *specification scenarios*, which describe a set of functionally different tasks that a system might need to perform, e.g. instructions of a processor. In this section, on the other hand, we introduce *implementation scenarios*, which describe a set of different approaches for achieving a functionally unique task. Having a set of approaches to perform a single task might be important for at least two reasons:

- **Fault tolerance -** fault tolerant systems must continue functioning regardless of hardware failures [41]. For such systems, it is important to have *backup* approaches to rely on in case of malfunction to the primary computation system.
- **Energy-quality (EQ) scalability -** it is a recent design direction [42] in which the approaches for achieving a task are ranked in terms of their energy consumption and result quality. EQ systems are designed to consume the least amount of energy for obtaining a desired result, e.g. approximate computing [43].

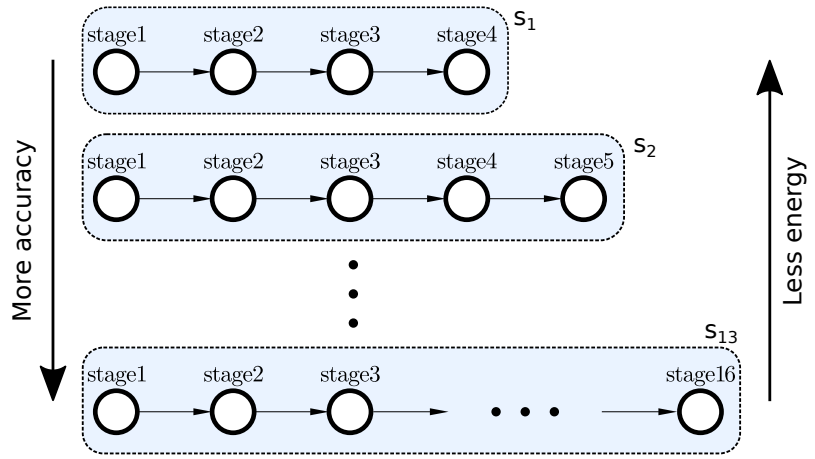
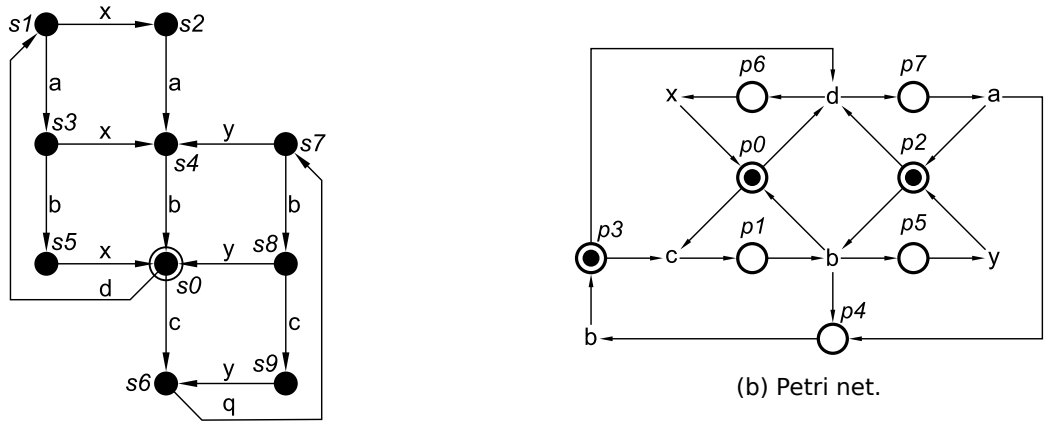


Figure 2.3: Energy-Quality tradeoff of the implementations scenarios of our prototype.

Our idea is to use implementation scenarios to design dynamically reconfigurable hardware circuits, which are heavily employed in the above areas [44–46]. Thus, we develop a scenario-based methodology that targets reconfigurable asynchronous circuits, and we validate it by designing an asynchronous EQ scalable accelerator for computing the Ordinal Pattern Encoding (OPE) [35]. Our implementation is said *dynamically reconfigurable* because the pipeline depth of the accelerator can be reconfigured at runtime. We focus on the asynchronous domain for taking advantage of the higher resilience to process and voltage variation.

Here, we show and describe the specification of the controller of our research prototype to give the reader an idea of the expressive capabilities of scenarios. The implementation scenarios of the OPE pipeline represent the operating modes of the prototype, which are selected depending on the desired EQ level. Figure 2.3 shows 3 of the 13 implementation scenarios in the form of partial orders. Each stage takes as input the result of the previous one and produces a more accurate result. The first scenario (s_1) is the least accurate and the most energy efficient, while the latest scenario (s_{13}) is the most accurate and the least energy efficient operating mode.

The shown specification can be used to derive the control unit of the accelerator, but a further approach is needed to derive the datapath. To fill in this gap, in Chapter 4, we propose the *Dataflow Structures* formalism for modelling the datapath of reconfigurable asynchronous circuits. In Chapter 5, we describe a methodology for designing and implementing reconfigurable asynchronous pipelines that relies on scenarios.



(a) Transition system.

(b) Petri net.

Figure 2.4: System specification of a concurrent system.

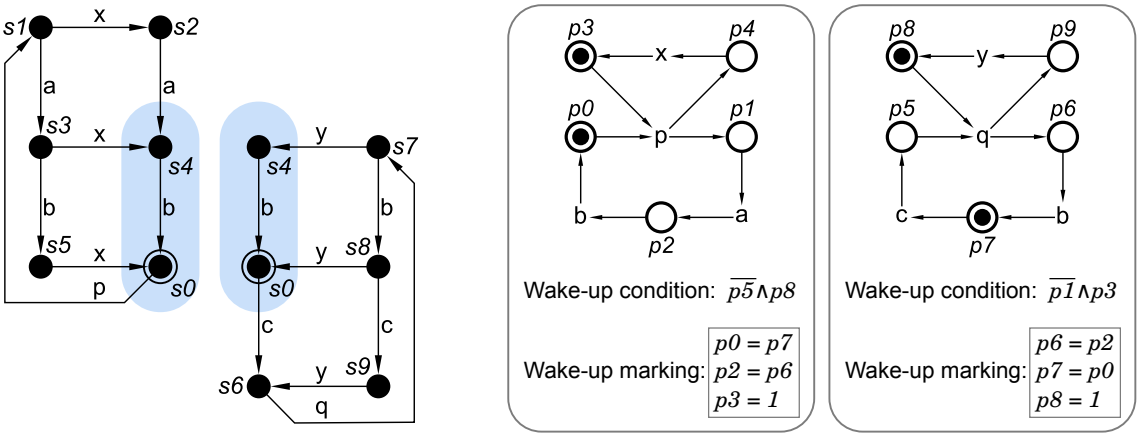
2.3 Understanding complex systems

In the previous sections, we discussed the usage of scenarios for designing processor control units and reconfigurable architectures. In this section, we show that scenarios are also important at a different phase of the process: when engineers need to fully understand the specification of a system.

It is difficult to fully understand highly concurrent system specifications that incorporate many internal behaviours. *A possible solution for fostering an easier understanding is to decompose such systems into their scenario formulation* [47]. Transition systems and Petri nets are two widely-used formalisms for the representation of concurrent systems, which would benefit from an automated approach to their simplification through *scenario decomposition*.

As an example, consider the Transition system in Figure 2.4a and its corresponding Petri net in Figure 2.4b. Arguably, the behaviour of the represented system is difficult to understand, as it is made of a mix of concurrency and of *non-deterministic choices* (i.e. the behaviour the system cannot be predicted statically, e.g. see the transitions **c** and **d** that are enabled and can fire at the same time) that are clearly visible in the Petri net.

The transition system can be decomposed into its constituent scenarios shown in Figure 2.5a, which are simpler to understand as they do not contain non-deterministic choices. The two scenarios, whose corresponding Petri nets are shown in Figure 2.5b,



(a) Decomposed Transition system.

(b) Windows w_1 (left) and w_2 (right).

Figure 2.5: Scenario specification of the concurrent systems in Figure 2.4.

are denoted as *windows* in the formalism that we present in this work, and reproduce the functionality of the system by their alternate execution. The states $s0$ and $s4$ are both covered by the two windows, and represent the "bridges" between their execution. The Boolean conditions generated during scenario decomposition process, namely *Wake-up* and *Wake-up marking* conditions, orchestrate the activation and deactivation of the windows for modelling the complete system functionality.

In Chapter 4, we present the novel *Process windows* formalism, which aims to simplify the understanding of complex concurrent systems by partitioning the specification into its constituent simpler scenarios.

2.4 Network analysis

Specification and implementation scenarios are of interest both for the design of various types of architectures, and as a way to simplify the understanding of concurrent systems. As last motivating example, we introduce the idea of *activation scenarios*, which are important for identifying and activating "*key areas*" of a system. This work falls into the domain of *network science*, i.e. an area of research where graphs are the main actors.

Network science grows steadily, as the usage of graph as underlying representation is expanding across a large class of applications, e.g. telecommunication, social media, biology [48]. In addition to the representation of complex data structures, graphs can

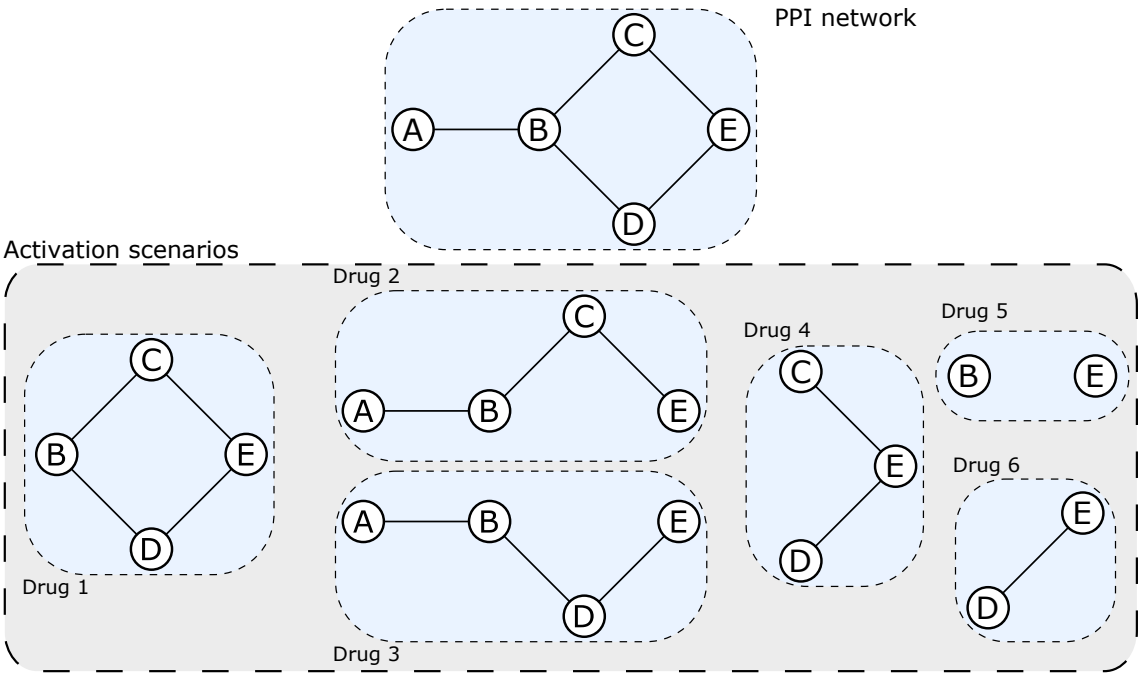


Figure 2.6: A biological system (PPI network) represented as an undirected graph **(on top)**, and its activation scenarios **(at the bottom)** that identify the areas of interest (proteins/vertices and bonds/arcs) of the PPI.

be used to unveil unapparent system properties. In the domain of *computational drug discovery* [39], for example, complex biological systems are modelled through protein-protein interaction networks (PPI) [49], and their analysis aims at developing new drugs.

In this work, protein-protein interaction networks are modelled by undirected graphs, i.e. vertices represent proteins that interact to each other due to their bonds represented as arcs. Drugs, when injected into such biological systems, disable some of their proteins and inhibit their properties. This process is emulated via a computer-based model: software algorithms use large libraries of drugs to inhibit the data-structures representing the PPI networks, and collecting statistics necessary to catalogue the most promising drug candidates, i.e. which have a high chance to be effective biologically. The latter are finally analysed in pharmacological laboratories for further verifying their suitability against certain diseases. This experimental approach has been pioneered by e-Therapeutics [50], which has collaborated with us throughout this research providing real-life PPI data-sets.

Activation scenarios are important in this application for their ability to identify

subsets of the biological system induced by drug perturbation. As an example, Figure 2.6 (on top) shows a biological system represented as an undirected graph, where `A`, `B`, `C`, `D` and `E` are its proteins. The graphs at the bottom of the Figure are the activation scenarios induced by some drugs under test. Such scenarios represent the proteins and bonds that are not disabled by the injection of a particular drug – e.g. the scenario containing the proteins `B` and `E` is derived by injecting the *drug 5*, which disables the proteins `A`, `C` and `D` and corresponding bonds, leaving the remaining proteins `B` and `E` unconnected. Activation scenarios can be used to derive a model (i.e. based on CPOGs) of the PPI network, where internal key areas can be activated and analysed in isolation from the rest of the system.

However, the number of scenarios explodes exponentially *when the size of a protein-interaction network increases*. Practically, *scenarios can neither be specified exhaustively nor used to synthesise optimal implementations, yet an approach for analysing them is needed*. To bridge this gap, in Chapter 5, we present our prototyped FPGA accelerator for computational drug discovery, developed during the EPSRC programme grant *POETS* [51] in partnership with e-Therapeutics. The accelerator is much faster than conventional software implementations, and can potentially process any scenarios of a given protein-interaction network.

2.5 Summary

We used the described motivating examples to identify three different types of scenarios. (1) *Specification* scenarios describe sets of functionally different tasks of a system. (2) *Implementation* scenarios denote sets of possible approaches (implementations) to achieve a unique task. (3) *Activation* scenarios denote subsets of a system that need to be activated and handled in isolation from the whole system.

Motivated by the above discussion, in the *Scenario composition* chapter, we present our contributions to the field of high-level scenario-based hardware design. In the *Case studies* chapter, subsequently, we validate the presented discoveries by presenting our real-life case studies.

Chapter 3

Background

This Chapter reviews the existing behavioural models that we employ in our research (see boxes in the blue shadow in Figure 1.3). Before proceeding with their description, we provide an overview of the existing behavioural models that are described in this chapter, and whose names are highlighted in italics below.

Partial Orders (PO) constitute the internal scenarios of the *Conditional Partial Order Graphs* (CPOG) formalism. The latter is important as it is supported by automated scenario composition and hardware synthesis features. In this work, we present a new scenario composition algorithm.

Signal Transition Graphs (STG) are labelled *Petri nets* (PN), where labels are associated to signal state changes. These two models are well-known to the asynchronous community, and are supported by a number of back-end tools that provide several design automation features (e.g. hardware synthesis, verification). To make our new Dataflow Structures formalism attractive, we present an automated approach to convert DFS into PN for reusing the existing tool-set. DFS models are obtained by composing sets of *Static Dataflow Structures* (SDFS).

Lastly, *Labelled Transition Systems* (LTS) is a well-known model that is used for representing distributed systems. In our work, we propose an automated technique to decompose LTSs into Process Windows.

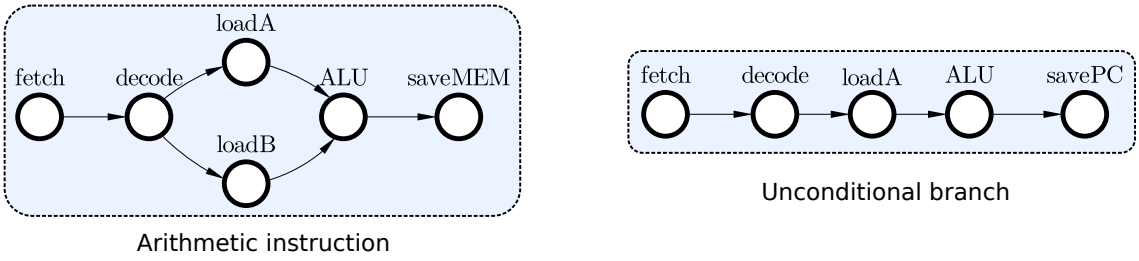


Figure 3.1: Two instructions represented as Partial Order graphs.

3.1 Partial orders

Partial Orders (POs) represent ordered sets of operations (or events) [22]. Typically, they are used for representing the behaviour of highly concurrent systems, where the causality dependencies between operations are more relevant than other types of relations (e.g. temporal). In this work, for example, we employ POs for describing the data flow of operations in processor instructions, where the functional dependencies between internal modules are the primary concern. In the context of scenario-based methodologies, POs are important as they constitute the internal scenarios of Conditional Partial Order Graphs, reviewed in the next section. Formally, a Partial order is defined below.

Definition 3.1. ***(Strict partial order) -*** *A strict partial order $P(\mathcal{O}, \prec)$ is a binary precedence relation $\prec$, describing dependencies between a set of operations (or events) $\mathcal{O}$, which satisfies two properties:*

- *Irreflexivity:* $\forall a \in \mathcal{O}, \neg(a \prec a)$
- *Transitivity:* $\forall a, b, c \in \mathcal{O}, (a \prec b) \wedge (b \prec c) \Rightarrow (a \prec c)$

This work uses strict partial orders only, hence we further omit "strict" for brevity. The *Hasse diagram* [22] of a partial order is a directed acyclic graph (DAG) [52] that is obtained by removing all transitive dependencies from the PO. Hasse diagrams preserve all immediate dependencies, and contain the minimum number of them, and are therefore a convenient and widely-used notation for specifying POs.

As an example, Fig. 3.1 shows Hasse diagrams of two Partial Orders. The two POs describe the order of operations of two instructions of a general processor: the

arithmetic and the unconditional branch instructions. The unconditional branch PO is a simple sequence of events, and the order in which they occur is fully determined by the dependencies. The arithmetic instruction PO, on the other hand, contains concurrency. For example, operations `loadA` and `loadB` are not connected by any arc, therefore the specification allows them to occur either in sequence (i.e. `loadA` $\rightarrow$ `loadB` or `loadB` $\rightarrow$ `loadA`), or concurrently (i.e. `loadA` $\parallel$ `loadB`). The two scenarios, in the form of partial orders, are functionally described below:

- the **Arithmetic instruction** scenario fetches (`fetch` operation) and decodes (`decode`) an instruction from the program memory, loads two operands $\{\mathcal{A}, \mathcal{B}\}$ concurrently (`loadA` $\parallel$ `loadB`), and uses them to perform an arithmetic operation (`ALU`). Subsequently, the result is saved into the memory via the `saveMEM` operation.
- the **Unconditional branch** scenario fetches (`fetch` operation) and decodes (`decode`) an instruction from the program memory, loads one operand ($\mathcal{A}$) and adds it to the program counter register (`PC`) to compute the branch address (`ALU`). Finally, the result is saved into program counter register (`savePC`) for the branch, i.e. `PC` $=$ `PC` $+ \mathcal{A}$.

3.2 Conditional partial order graphs

The Conditional Partial Order Graphs (CPOG) formalism was introduced by Mokhov in his PhD dissertation [23]. This behavioural model, originally conceived to support the design of processor instruction sets (see *Motivation* chapter), is important in the context of scenario-based methodologies as it is supported by automated behavioural composition, and as it allows to synthesise hardware implementations automatically. Intuitively, it is a collection of scenarios in the form of Partial orders. Formally, it is defined below.

Definition 3.2. ***(Conditional partial order graph) -*** *a conditional partial order graph [53] is a tuple* $H = (V, E, B, \phi)$[1]*:*

- $\boldsymbol{V}$ *is a set of vertices which correspond to operations (or events) in a modelled system.*
- $\boldsymbol{E} \subseteq V \times V$ *is a set of arcs representing dependencies between the operations.*

[1] A CPOG is $H = (V, E, B, \phi, \rho)$ in [53], ρ is not described here as it is not used.

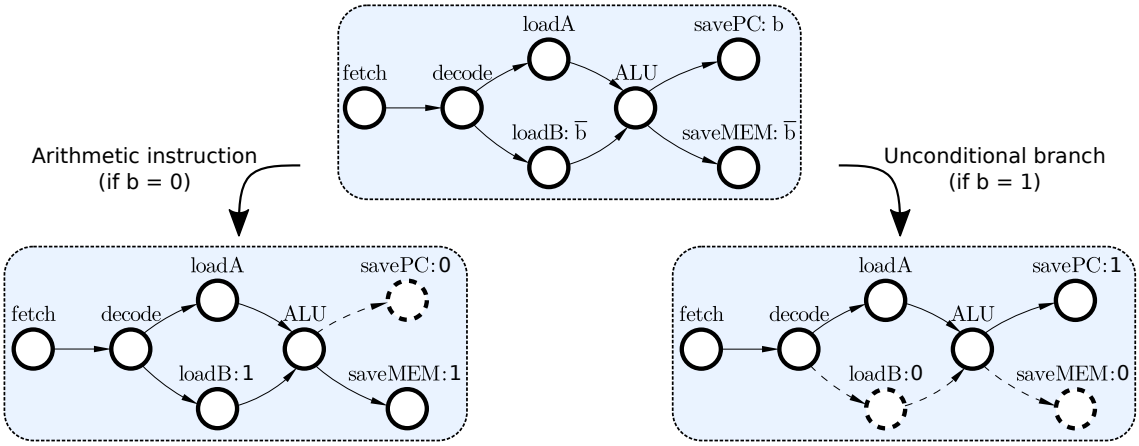


Figure 3.2: Example of CPOG with 2 projections: $H|_{b=0}$ on the left side models the functionality of the *arithmetic instruction* scenario, $H|_{b=1}$ on the right models the *unconditional branch* scenario.

- ***B*** *is a set of Boolean variables* $\{b_1, b_2, ..., b_{|B|}\}$. *A code is an assignment* $c : B \to \{0, 1\}$ *of these variables, e.g.* $B = \{b_1, b_2\}$, $c(b_1) = 0$ *and* $c(b_2) = 1$ *that will be further denoted as* $c = 01$ *for brevity. A code selects a particular PO from those contained in the CPOG.*
- *Function* $\boldsymbol{\phi}$: $(V \cup E) \to F(B)$, *with* $F(B)$ *being the set of all Boolean functions over variables in B, assigns a Boolean condition* $\phi(z) \in F(B)$ *to every vertex and arc* $z \in V \cup E$.

Graphically, CPOG vertices are depicted as circles ◯, and arcs are depicted as arrows →. Vertices and arcs $z \in V \cup E$ are labelled with conditions $\phi(z)$, which are formed by the Boolean variables B and have the purpose to switch vertices and arcs on (off) when the conditions on them are (not) satisfied.

As an example, Figure 3.2 (on top) shows the CPOG derived by composing the two scenarios in Figure 3.1. The CPOG manages to represent the two initial partial order graphs compactly by overlaying the common elements of the two graphs. The functionality of constituent scenarios is preserved, as the functions $\phi = \{b, \overline{b}\}$ are in charge of reproducing the functionality of the internal scenarios:

- when $b = 0$, the `savePC` vertex is off and the CPOG reproduces the behaviour of the *Arithmetic instruction*;
- when $b = 1$, the vertices `loadB` and `saveMEM` are off, and the functionality of the *Unconditional branch* is reproduced.

The variable b, in this case, represents the opcode of the two instructions.

The example in Fig. 3.2 shows that a CPOG can be used to compactly represent multiple behavioural scenarios by overlaying their common parts. In practice CPOGs remain compact and easy to understand even when the number of scenarios increases, making the formalism suitable for representing a large class of hardware systems.

3.3 Petri nets

Petri nets (PN) or PT nets (from Place/Transition nets) is a well-known formalism for the description of concurrent behaviours, such as those of asynchronous circuits, distributed and multi-core processing systems. In this section, we recall the basics of this formalism, which is important as it is supported by extensive hardware synthesis and verification features, and by a number of established EDA tools, e.g. Petrify [54,55]. For more details about this formalism, we refer the reader to [56,57].

Definition 3.3. ***(Net)*** **-** *A net is a triple $N = (P, T, F)$, where:*

- ***P*** *is a finite set of places, which are denoted graphically as ◯.*
- ***T*** *is a finite set of transitions $(T \cap P = \emptyset)$, which are denoted graphically as □.*
- $\boldsymbol{F} : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$ *is a set of arcs, which are denoted graphically as →.*

A net describes the possible flow of actions of a system by specifying its internal events and causality dependencies. Since the initial conditions are not specified by a net, the system is modelled under all possible initial conditions. The latter are instead specified in the Petri nets, defined below.

Definition 3.4. ***(Petri net)*** **-** *A Petri net is a pair $P = (N, m_0)$, where:*

- ***N*** *is a net as defined in Definition 3.3.*
- $\boldsymbol{m_0} : P \rightarrow \mathbb{N}$ *is the marking of the net, i.e. the initial conditions of a system. Formally, it is a mapping function of the places P over a finite number of data tokens, which represents the state of a system.*

As an example, Figure 3.3 shows a Petri net that is composed of 6 places $P = \{p0, p1, p2, p3, p4, p5, p6\}$, and 4 transitions $T = \{t0, t1, t2, t3\}$. The initial marking m_0 of the Petri net is $\{p0 = 1, p1 = 1, p2 = 0, p3 = 0, p4 = 0, p5 = 0, p6 = 0\}$ (the places can

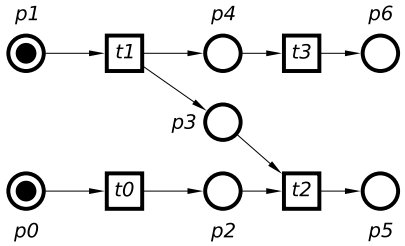


Figure 3.3: Example of a Petri Net.

be omitted for brevity: $m_0 = \{1, 1, 0, 0, 0, 0, 0\}$), which identifies the places that contain a *token* and are filled with a • in the graphical representation, e.g. $p0$ and $p1$ contain a token, while $p2 \cdots p6$ do not. Intuitively, transitions represent the events that might arise in a system (e.g. the change of a state of a digital signal in a circuit, or the assignment of a data-query to one of the core in a GPU), while places represent the resources/conditions needed for the transitions to happen (e.g. the availability of one of the core in a GPU). A marking represents the global state of the represented system. In this work, we only focus on *safe* Petri nets, i.e. the internal places contain one token at most. Hence we further omit the word "safe" for brevity.

The behaviour of a Petri Net is named *token-game*. It governs the evolution of the markings, and is ruled by the enabling and firing conditions described in Definition 3.6. The latter is formalised by means of the below definition of places preset and postset.

Definition 3.5. ***(Preset and postset) -*** *The preset of a place p is denoted as $\bullet p$, and is composed of the set of transitions T' such that $F(T', p) > 0$. The postset of a place p is denoted as $p\bullet$, and is composed of the set of transitions T' such that $F(p, T') > 0$. Similarly, the preset of a transition t is denoted as $\bullet t$, and is composed of the set of places P' such that $F(P', t) > 0$. The postset of a transition t is denoted as $t\bullet$, and is composed of the set of places P' such that $F(t, P') > 0$.*

As an example, let us consider the Petri Net in Figure 3.3. The preset of the place $p4$ is composed of the transition $t1$, its postset of the transition $t3$. The place $p1$ has an empty preset, as it is not preceded by any transition. The preset of the transition $t2$ is composed of the places $\{p2, p3\}$, and its postset of the place $p5$.

Definition 3.6. ***(Enabling and firing) -*** *A transition $t \in T$ is enabled if all its resources are available, i.e. any place in its preset contains a token. When an enabling transition $t \in T$ fires,*

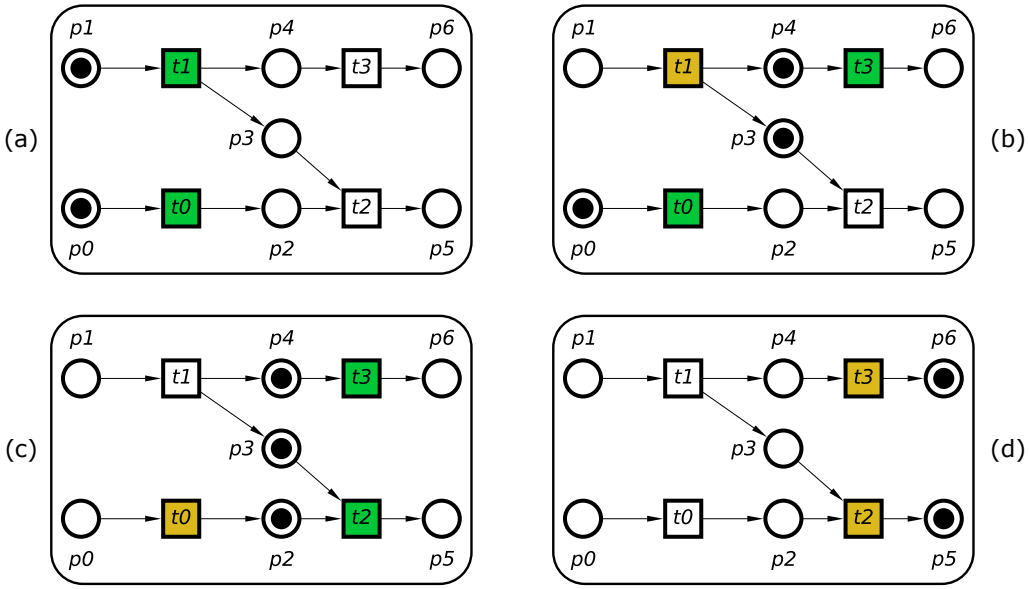


Figure 3.4: Representation of the Petri Net behaviour, namely *token game*. The initial marking of the Petri Net is m_0, while the last marking is m_4.

it modifies the current marking on the Petri net by moving the tokens from the preset places ($\bullet t$) into the postset places ($t\bullet$).

As an example, Figure 3.4 shows one of the possible traces of the Petri net in Figure 3.3 according to the enabling and firing rules previously defined. The marking changes of the PN are also described below.

(a) - The initial marking m_0 of the Petri net is shown: the places $p0$ and $p1$ contains a token, and the transitions $t0$ and $t1$ are enabled (see the green transitions).

(b) - The transition $t1$ fires (see the yellow transition), and the token in the preset of $t1$ ($p1$) moves to the places in its postset ($\{p3, p4\}$). In the current PN marking, the transition $t3$ becomes enabled. Notice that the transition $t2$ is still not enabled, as not all the places in $\bullet t2$ contain a token, see $p2$.

(c) - $t0$ fires and enables the transition $t2$.

(d) - Transitions $t2$ and $t3$ fire concurrently, and the Petri net terminates, as there are no left enabled transitions.

At every firing of one or more transitions, the Petri net reaches a new marking where new transitions are enabled. Hence, the behaviour of a Petri net can be denoted by

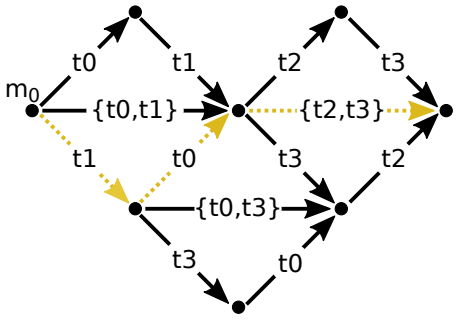


Figure 3.5: The reachability graph of the Petri net in Figure 3.3. The dotted arcs in yellow shows the trace of events shown in Figure 3.4.

its sequence of markings, which are represented in the *reachability graph* (RG), whose definition is in 3.7.

Definition 3.7. ***(Reachability graph)*** **-** *It represents all the possible behaviours of a Petri net. Formally, it is a transition system where the nodes (•) are the global states of the Petri net and represent the markings that are reachable from m_0; the arcs connect such nodes and are labelled with the transitions that need to be fired for the markings to be reached.*

Figure 3.5 depicts the reachability graph of the Petri net in Figure 3.3, where m_0 is the initial state of the transition system, and the dotted arcs in yellow depict the behaviour shown in Figure 3.4. Notice that the transitions might have fired differently, leading to different markings and paths in the reachability graph.

In this work, Petri nets are used with the *read-arcs* extension [58]. A read-arc is a special arc that only affects the enabling rules of the PN without affecting the firing. Consider the PN in Figure 3.6 (left), for example, where the place $p5$ is connected to the transition $t3$ via a read-arc, leading to $\bullet t3 = \{p4, p5\}$. The transition $t3$ is enabled if and only if any of the places in $\bullet t3$ contains a token. The firing of $t3$ moves the token from $p4$ to $p6$, and does not affect the position of the token in $p5$. Figure 3.6 (right) shows the corresponding RG further to the read-arc addition. Notice that the transition $t3$ can now only fire after the firing of $t2$.

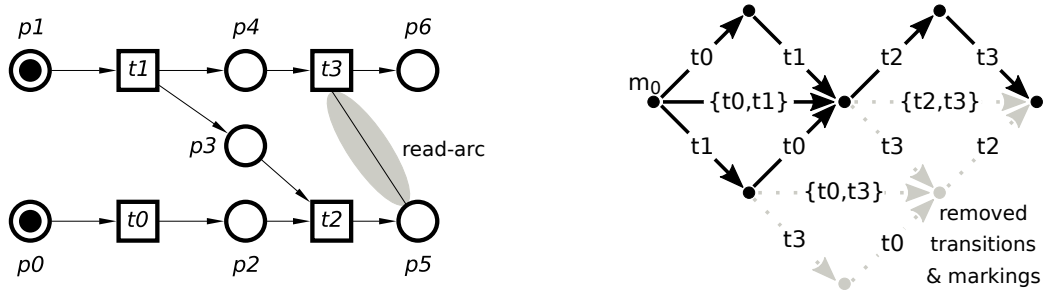


Figure 3.6: Petri net in Figure 3.3 with the additional *read-arc* on the **left**-hand side, its corresponding Reachability graph on the **right**.

3.3.1 Signal transition graphs

Signal Transition Graphs (STG) [58] is a type of labelled Petri net where transitions between states are paired with changes in the values of binary variables. In the context of our work, such variables represent input, output, or internal signals of digital circuits, which can assume the values either of a logic 0 or 1. The states of an STG, therefore, represent the values that the signals of the modelled circuit can assume. The formal definition of an STG is reported in 3.8:

Definition 3.8. ***(Signal Transition Graphs) -*** *A signal transition graph is a tuple* $G = (N, m_0, X, \lambda)$*, where:*

- $\boldsymbol{(N, m_0)}$ *is a Petri net, where* $N = (P, T, F)$*.*
- $\boldsymbol{X}$ *is a set of binary values, which generates a finite alphabet* $S^X = X \times \{+, -\}$ *of signal transitions. The transition in the state of a signal* $s \in S$ *from* 0 *to* 1 *is denoted by* `s+`*, and the one from* 1 *to* 0 *by* `s-`*. Every signal s can be either an input, output or internal signal of the digital circuit represented. Graphically, in our work, these are highlighted in red, blue and green, respectively.*
- $\boldsymbol{\lambda}$ $: T \rightarrow S^X$ *is a labelling function that pairs the transition of the net N with the changes of the binary values. The labelling function does not need to be 1-to-1 with respect to* $|T|$*.*

Graphically, we represent STGs as Petri nets where the transitions □ are substituted with changes in the binary variables S^X. For compactness, places can be omitted and the presence of tokens • can be overlaid with the arcs that connect transitions.

As an example, we refer to the formal STG specification of the asynchronous power-management controller presented in [14]. Figure 3.7(top) shows the waveforms

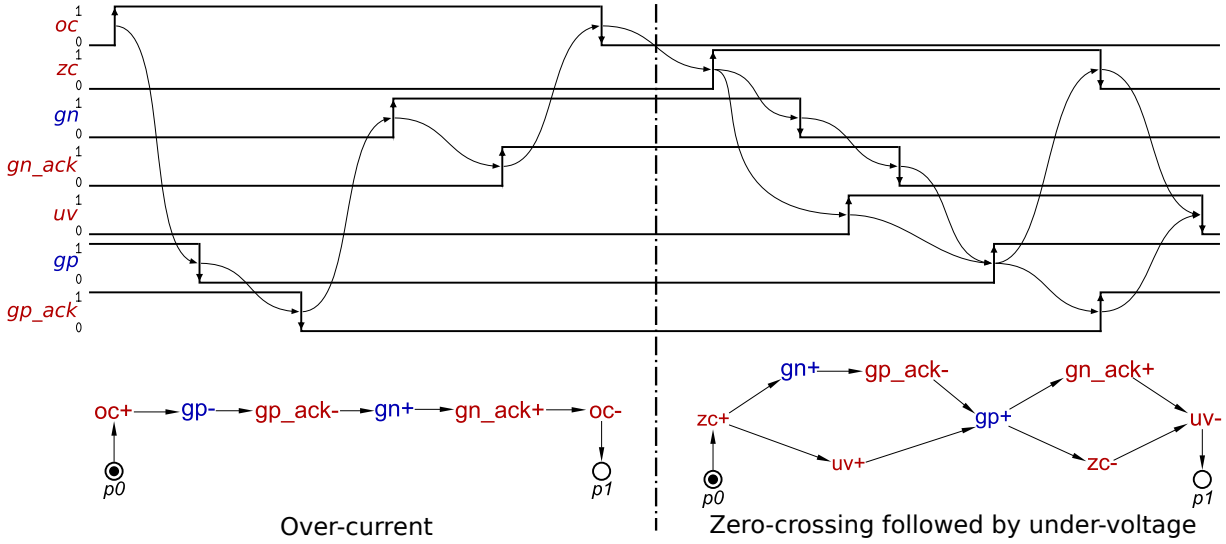


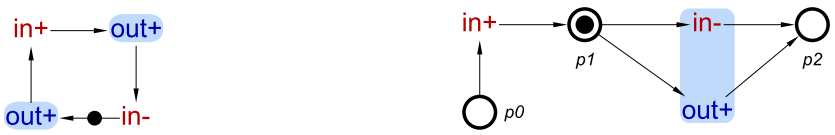
Figure 3.7: Two scenarios of a buck controller represented as waveforms **(top)** and signal transition graphs **(bottom)**.

corresponding to two behavioural scenarios of a buck converter. The system controls the power regulating PMOS (gp) and NMOS (gn) transistors in response to three signals coming from sensors within the power regulator: over-current (oc), under-voltage (uv) and zero-crossing (zc). Below, we describe these two scenarios (note that the two transistors must never be on at the same time to avoid a short circuit):

- **Over-current scenario:** when the oc condition is detected (input event oc+), the PMOS transistor must be switched off (output event gp−). Afterwards, the NMOS transistor must be switched on (output event gn+).
- **Zero-crossing followed by under-voltage scenario:** If zc is detected before uv, the NMOS transistor must be switched off (output event gn−). The two transistors must stay off until the arrival of the uv condition. Afterwards, the PMOS transistor must be switched on (output event gp+).

As shown in Figure 3.7(bottom), STGs can be used for describing the evolution of input/output values of the signals of the buck converter. Additionally, they can be used for modelling the behaviour of such systems due to the inheritance of the Petri net semantics (enabling and firing rules of the token game).

In this research, we model part of our case studies via STGs, and use these to



(a) Non-consistent STG.

(b) Non-persistent STG.

Figure 3.8: Examples of STG properties.

synthesise hardware implementations via the established EDA flow [55]. We, then, compare the resulting implementations with the ones that are derived with the EDA tools that we present alongside our findings.

Here, we provide the background that is necessary to understand STGs targeting the synthesis of speed-independent digital circuits [55]. An STG has to satisfy the following three conditions to be synthesised into a functional circuit: it has to be *consistent*, *persistent* and it has to satisfy the *Complete State Coding* condition. The first two conditions are described below, as they are important for the specification of an STG model. The third condition is not described here, as it is part of the STG synthesis process and is not necessary to understand the presented contributions. We refer, however, the reader to [55, 59] for more information about the Complete State Coding condition.

An STG is said to be consistent if for any signal `s` in X the change of the value `s+`(`s-`) can never occur twice consecutively starting from the initial marking. An STG is said to be persistent if an enabled transition can never be disabled by the firing of another enabled transition in the same marking. Figure 3.8 shows a non-consistent (left-hand side) and a non-persistent (on the right) STG. In the former, the `out` signal changes its value from 0 to 1 twice consecutively, which cannot happen in a circuit. In the latter, the `in-` and `out+` transitions disable each other in the shown marking, i.e. the firing of `in-` disables `out+` and vice versa.

3.4 Static dataflow structures

The Static Dataflow Structure (SDFS) is a high-level model for asynchronous digital circuits that has been first introduced in [25]. Similarly to the Register Transfer Level (RTL) for synchronous circuits, SDFSs model asynchronous behaviours at system-level by abstracting away the low-level implementation details of corresponding digital

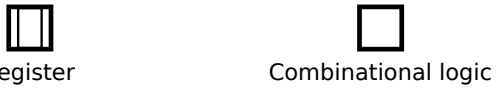


Figure 3.9: Static dataflow structures nodes.

circuits. In this work we employ SDFSs to model static asynchronous circuit scenarios, and show how these can be composed using the novel Dataflow Structure model.

In [26], D. Sokolov et al. formalise and compare three different behavioural semantics for this model, namely *atomic token*, *spread token* and *counterflow*. These dictate the way through which the nodes that constitute the model communicate. Here, we review the spread token behavioural semantics, which is the one that we use in our work.

Static dataflow structures are constituted by two basic nodes: *register* and *combinational logic*. In this work, these are denoted with their conventional graphical representation that was introduced in [25], see Figure 3.9.

- **Registers** model the behaviour of sequential circuit elements, which can store a data item (or *token*) coming from their inputs, and provide it to their outputs. These nodes implement the handshaking protocol for modelling asynchronous circuit behaviours.
- **Combinational logic** nodes model the behaviour of the combinatorial logic of the circuit. They are "transparent" to the handshaking mechanism implemented, as they can only propagate an input token to the output through combinatorial functions.

Formally, a Static Dataflow Structure is defined below:

Definition 3.9. ***(Static Dataflow Structures)*** *A static dataflow structure is a directed graph* $G = (V, E, D, m_0)$*:*

- $\boldsymbol{V} = R \cup L$ *is a set of nodes that are either registers (R) or combinational logic (L), i.e. if* $v \in V$*, then* $v \in R$ *or* $v \in L$*, with* $R \cap L = \emptyset$*.*
- $\boldsymbol{E} \subseteq V \times V$ *is a set of arcs that determine the flow relation among the internal nodes of the graph, an edge between* $v1$ *and* $v2$ *is denoted as* $(v1, v2)$*.*
- $\boldsymbol{D}$ *is a set of data values that are modelled by the data tokens flowing into the graph. Graphically, these are denoted as* $\bullet$*.*

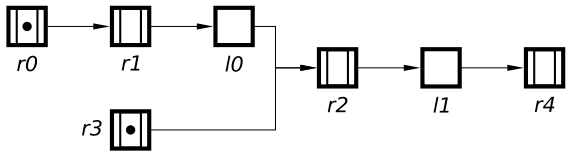


Figure 3.10: An example of static dataflow structure.

- $\boldsymbol{m_0}$ *is the initial marking of the graph, i.e. the initial conditions of the system. Formally, it maps the tokens D to the registers of the graph R.*

Figure 3.10 shows an example of a static dataflow structure composed of the five registers $\{r0, \cdots, r4\}$, and by the two combinational logic nodes $\{l0, l1\}$. The initial marking m_0 of the SDFS maps a data token to the registers $r0$ and $r3$, while the remaining registers do not to store any token. The presence of a token inside a register represents the storage of a meaningful data value for the circuit behaviour.

For formalising the behavioural semantics of the SDFS formalism, we provide the below definitions.

Definition 3.10. ***(Preset and postset) -*** *The preset of a node $v \in V$ is denoted as $\bullet v$, and identifies the nodes that directly precede v in the flow relation, i.e. $\bullet v = \{v' : (v', v) \in E\}$. The postset of a node $v \in V$ is denoted as $v\bullet$, and identifies the nodes that directly follow v in the flow relation, i.e. $v\bullet = \{v' : (v, v') \in E\}$.*

Definition 3.11. ***(L-Path) -*** *An L-path is a non-empty sequence of arcs between a source node (v_s) and a destination node (v_d), where the nodes in between v_s and v_d are combinational logic nodes, i.e. $\delta(v_s, v_d) = \{(v_{i-1}, v_i) \in E : (i \in [1 \cdots n]) \wedge (v_0 = v_s \wedge v_n = v_d) \wedge (v_i \in L \text{ for all } 0 < i < n)\}$.*

Definition 3.12. ***(R-preset and R-postset) -*** *The R-preset of a node $v \in V$ is denoted as $\star v$, and identifies the registers that precedes v in the flow relation, i.e. $\star v = \{r \in V : \exists \delta(r, v)\}$. The R-postset of a node $v \in V$ is denoted as $v\star$, and identifies the registers that follows v in the flow relation, i.e. $v\star = \{r \in V : \exists \delta(v, r)\}$.*

For an easier understanding of the above definition, we apply them to a few nodes of the SDFS shown in Figure 3.10:

- $\boldsymbol{\bullet r2 = \{l0, r4\}}$ - The preset of the node $r2$ is composed of the combinational logic node $l0$ and of the register $r4$.

$\boldsymbol{r3\bullet = \{r2\}}$ - the postset of the node $r3$ is $r2$.

$\boldsymbol{\star r2 = \{r1, r3\}}$ - the R-preset of the node $r2$ is composed of the two registers$r1$ and $r3$. The register $r0$ does not belong to the R-preset of $r2$, as the register $r1$ is contained in the nodes of the L-path $\delta(r0, r2)$.

$\boldsymbol{r1\star = l0\star = r3\star = \{r2\}}$ - the R-postset of the nodes $r1$, $l0$ and $r3$ is the same and equal to $r2$.

In the light of the above definitions, we can describe the rules that orchestrate the SDFS communication. A logic node l can be *evaluated* ($C_\uparrow$) when its preset logic nodes have been evaluated, and its preset registers are *marked* (i.e. contain a valid data item, or token). Symmetrically, a logic node l can be *reset* ($C_\downarrow$) when its preset logic nodes are reset and its preset registers are unmarked (i.e. contain no token). Intuitively, a logic node can be evaluated when all its inputs contain a valid data value, and it can be reset when its inputs contain a *spacer* (i.e. absence of a valid data). These nodes are passive to the handshake mechanism of a circuit. The state of a logic node can be checked by the function $C(l)$. See below equations from [26].

$$\begin{aligned} C(l) &= C_\uparrow(l) \vee \overline{C_\downarrow(l)} \wedge C(l) \\ C_\uparrow(l) &= \bigwedge_{k \in \bullet l \cap L} C(k) \wedge \bigwedge_{r \in \bullet l \cap R} M(r) \\ C_\downarrow(l) &= \bigwedge_{k \in \bullet l \cap L} \overline{C(k)} \wedge \bigwedge_{r \in \bullet l \cap R} \overline{M(r)} \end{aligned} \tag{3.1}$$

In turn, a register r can be *marked* ($M_\uparrow$) (and store a token) when its input logic nodes are evaluated, its R-preset are marked, and its R-postset are reset. Symmetrically, a register r can be *reset* ($M_\downarrow$) when its preset logic nodes are reset, its R-preset are reset, and its R-postset are marked. Intuitively, a register can store a new token when all its inputs are valid and when the output registers do not contain a token. It can release the token when it has been propagated through its output registers and released by its input registers. The communication works with the condition that two different tokens are separated by a spacer, as it happens in the *4-phase handshake protocol* [25] for asynchronous

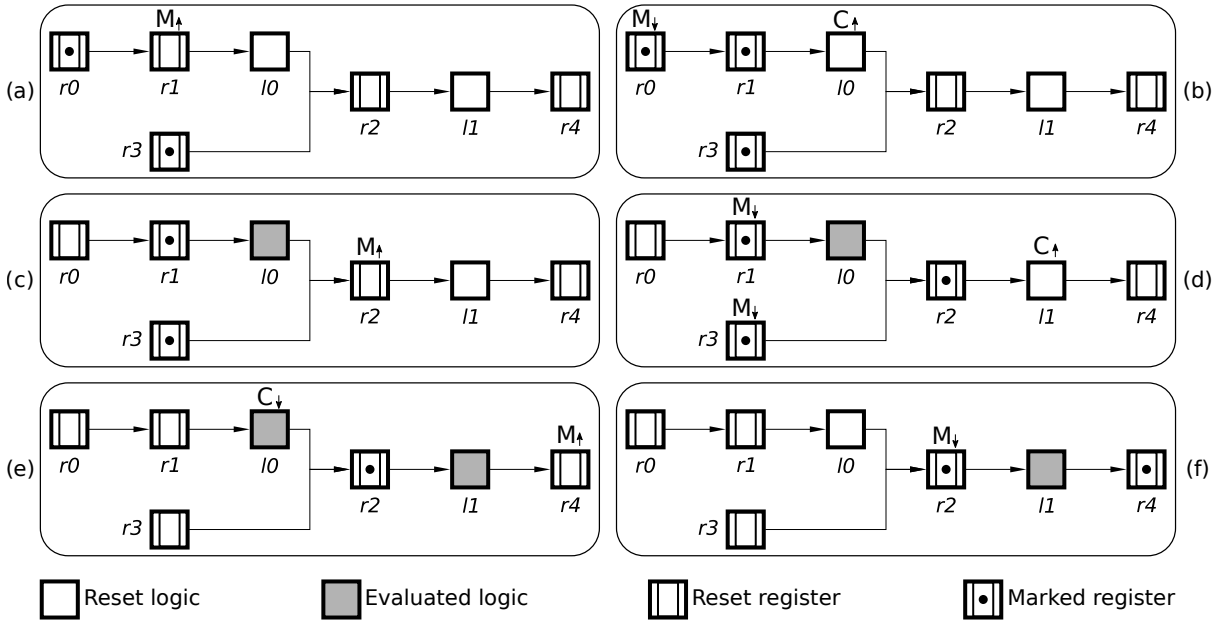


Figure 3.11: A possible simulation trace using the spread token behavioural semantics.

circuits. See below equations from [26].

$$
\begin{aligned}
M(r) &= M_{\uparrow}(r) \vee \overline{M_{\downarrow}(r)} \wedge M(r) \\
M_{\uparrow}(r) &= \bigwedge_{l \in \bullet r \cap L} C(l) \wedge \bigwedge_{q \in \star r} M(q) \wedge \bigwedge_{q \in r \star} \overline{M(q)} \\
M_{\downarrow}(r) &= \bigwedge_{l \in \bullet r \cap L} \overline{C(l)} \wedge \bigwedge_{q \in \star r} \overline{M(q)} \wedge \bigwedge_{q \in r \star} M(q)
\end{aligned}
\tag{3.2}
$$

Figure 3.11 shows a possible simulation trace of the SDFS in Figure 3.10. **(a)** In the initial marking of the SDFS, the register $r1$ is ready to be marked and to receive the token from $r0$. **(b)** The token is stored by $r1$, and can be propagated to $l0$, while $r0$ can be reset. **(c)** The nodes $l0$ and $r3$ are ready to propagate their tokens, thus $r2$ is ready to be marked. **(d)** Registers $r1$ and $r3$ can be reset as their data token is now safely stored by $r2$. The logic node $l1$ can now be evaluated. **(e)** The values of registers $r1$ and $r3$ become invalid, and $l0$ can be reset. Register $r4$ can be marked, and can store the token hold by $r2$ and processed by $l1$. **(f)** The register $r4$ stores the token, and $r2$ can be reset.

3.5 Labelled transition systems

Labelled Transition Systems (LTS) are used to model discrete systems, i.e. which can be described by a finite number of states and their dependencies. Here, we provide a brief definition of LTSs, but we refer the reader to [60] for a more complete description. The formal definition of an LTS is below.

Definition 3.13. ***(Labelled Transition Systems) -*** *a labelled transition system is a tuple* (S, T, L, s_0)*, where:*

- $\boldsymbol{S}$ *is a set of states;*
- $\boldsymbol{L}$ *is a dictionary of labels for the transitions* T*;*
- $\boldsymbol{T}$ *is a set of labelled transitions between pairs of states:* $T \subseteq S \times L \times S$*;*
- $\boldsymbol{s_0}$ *is the initial state.*

We represent LTSs graphically as *directed graphs*, where states are denoted with filled circles (•) and arcs with arrows (→). A transition from a state s_1 to another state s_2 via the a labelled transition is denoted as (s_0, a, s_2), or $s_0 \xrightarrow{a} s_2$. The dictionary of labels depends on the application. For example, labels can be used to highlight the effect of transitions over an input/output of a circuit (as for STGs), or represent mathematical conditions that need to be satisfied for enabling some change in a state of a system (as for CPOGs).

In this work, we present the *Process Windows* formalism and an algorithm for extracting sets of scenarios from labelled transition systems. The below definition will be used for describing this contribution.

Definition 3.14. ***(Enabling and Backward Enabling Sets) -*** *Let* $t \in T$ *be a transition of a labelled transition system. The Enabling Set (ES) of* t *is the set of states where the transition* t *is enabled, i.e.* $ES(t) = \{s \in S \text{ such that } : \exists s' \in S : s \xrightarrow{t} s'\}$*. Symmetrically, the Backward Enabling Set (BES) of* t *is the set of states that are enabled by the transition* t*, i.e.* $BES(t) = \{s \in S \text{ such that } \exists s' \in S : s' \xrightarrow{t} s\}$*.*

To conclude the description of the LTS formalism, we give an example of two labelled transition systems (in Figure 3.12) and describe them below.

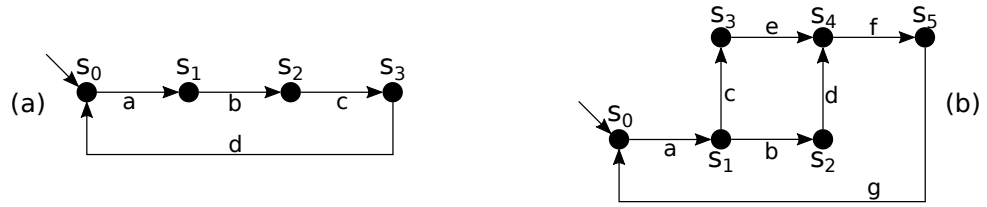


Figure 3.12: Two labelled transition systems.

- **Figure 3.12a** has as set of states $\{s_0, s_1, s_2, s_3\}$, the initial state is s_0 and is denoted by a dangling arrow. Its transitions are labelled by the dictionary of labels $\{a, b, c, d\}$. The transition from s_0 to s_1, for instance, is depicted as $s_0 \xrightarrow{a} s_1$. The Enabling Set of the transition c is $\{s_2\}$, while the Backward Enabling Set of the transition a is $\{s_1\}$.

- **Figure 3.12b** has as set of states $\{s_0, s_1, s_2, s_3, s_4, s_5\}$, the initial state is s_0. Its transitions are labelled by the dictionary of labels $\{a, b, c, d, e, f, g\}$. The transition from s_5 to s_0, for instance, is depicted as $s_5 \xrightarrow{g} s_0$. The Enabling Set of the transition e is $\{s_3\}$, while the Backward Enabling Set of the transition g is $\{s_0\}$.

Chapter 4

Scenario composition

So far, we introduced the idea of behavioural composition with high-level scenarios, and showed that it can be useful for a number of applications. We also described the formal behavioural models and existing theories that this research is based on.

In this chapter, we present our contributions to the field of high-level scenario-based design. In Section 4.1, we describe a new approach for composing scenarios of a system efficiently and deriving efficient implementations – we apply our technique to the composition of partial orders into conditional partial order graphs. In Section 4.2, we show how to compose static asynchronous circuit behaviours by means of Dataflow Structures. In Section 4.3, we present an automated approach to the decomposition of complex specifications in the form of labelled transition systems.

4.1 Efficient composition of scenarios

We described the meaning of efficient composition of scenarios in Section 1.2. In this section, we present one possible solution to the following question:

Problem: *Given a set of scenarios that describe the behaviour of a system, find the most efficient implementation (for example in terms of area) capable of executing each of the scenarios in response to an external set of inputs.*

To answer this question, we rely on the Conditional Partial Order Graphs for-

malism (reviewed in Section 3.2), whose models are obtained by composing sets of scenarios in the form of partial orders and can be used to derive efficient hardware implementations via an automated flow (as will be illustrated shortly). A set of scenarios can be composed into a CPOG in many ways, affecting both the compactness of the model and the size of the final hardware implementation. In this section, we present a novel approach to scenario composition that minimises the area of the resulting implementation and supports *composition constraints*.

This section is divided as follows. Section 4.1.1 reviews in detail the CPOG-based methodology. Section 4.1.2 reviews the previously published algorithms for efficient scenario composition. Section 4.1.3 describes our proposed scenario composition algorithm. Section 4.1.4 describes the developed tool that implements the CPOG methodology and comprises all existing scenario composition approaches. The content of this section has been published in [27, 28].

4.1.1 Background

This section reviews the design methodology based on the CPOGs [24], see Figure 4.1. The scenarios of a system are formally specified by a *scenario specification* (in the form of a set of partial orders). Scenarios are composed into a *system specification* (in the form of a CPOG). The latter is used to synthesise a *hardware controller* (in the form of gate-level description in Verilog). The presented approach supports the specification of *composition constraints* (in the form of codes). The controller is then automatically interfaced to the specified *datapath modules* in the final *system implementation*.

Below, we review the basics of the CPOG methodology, using as running example the scenario specification in Figure 4.2b, which has been introduced in Section 3.1.

4.1.1.1 Scenario specification

A hardware system is described by a collection of scenarios, each in the form of a partial order. Vertices and arcs constitute the basic elements of these graphs, where vertices represent system operations (or events), and arcs represent dependencies between them.

System scenarios can be specified either graphically (see Section 4.1) or textually in

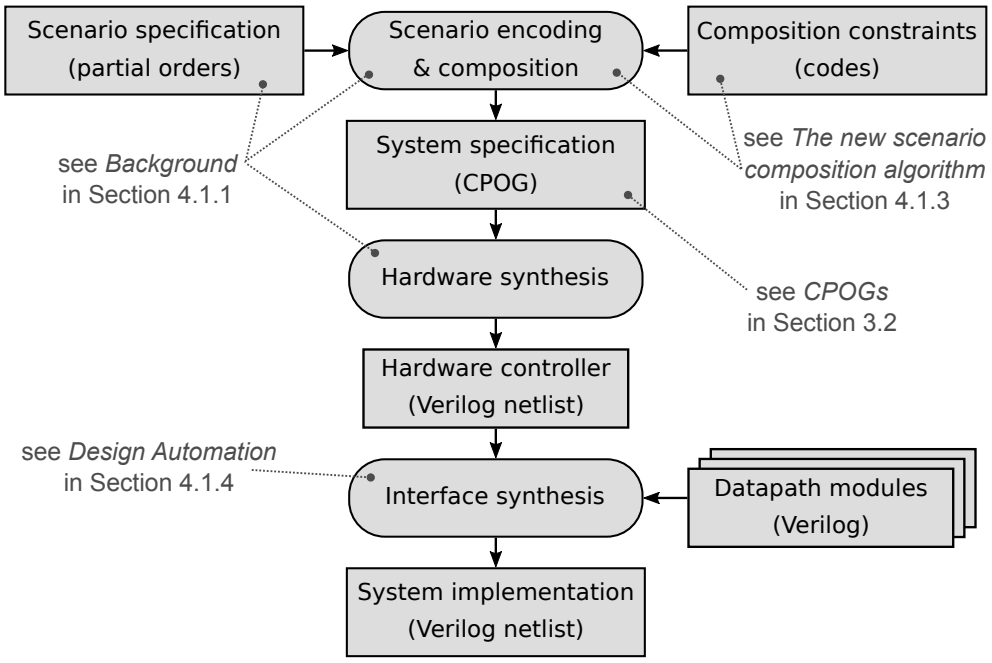


Figure 4.1: The design methodology based on the CPOGs.

a file using the *algebra of graphs* [61]. Text files containing scenarios are parsed, and each scenario is converted into a graph. As an example, Figure 4.2 shows the scenario specification of a processor instruction set composed of two instructions: s_1 is the arithmetic instruction scenario, and s_2 is the unconditional branch scenario. The textual description specification of such scenarios is in Fig. 4.2a, while the graphical specification based on partial orders is in Fig. 4.2b.

The effort required by engineers to produce such scenario specifications is high, and it is desirable to extract scenarios from higher-level descriptions. There are several examples of high-level specification languages targeting processor architectures, e.g. see Arm's Architecture Specification Language [62] and Sail [63]. This aspect of automation is outside the scope of this work; we refer the reader to [64] for a relevant example.

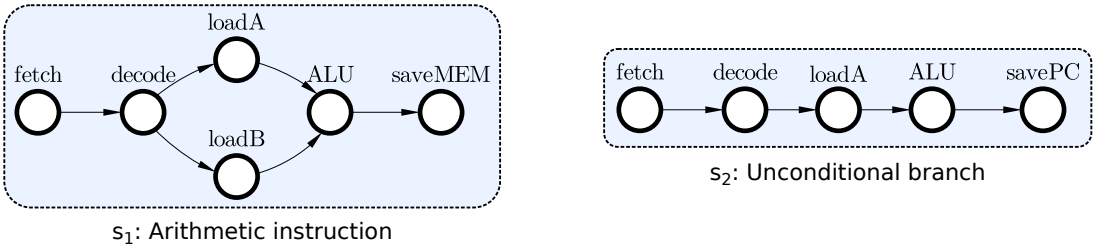
4.1.1.2 Scenario encoding

Scenario encoding is the process of finding an injective function between a set of scenarios and a set of codes. Let n be the number of scenarios. The following definitions will be used to formally state the *CPOG encoding problem*.

$$s_1 = \texttt{fetch} \rightarrow \texttt{decode} \rightarrow (\texttt{loadA} + \texttt{loadB}) \rightarrow \texttt{ALU} \rightarrow \texttt{saveMEM}$$

$$s_2 = \texttt{fetch} \rightarrow \texttt{decode} \rightarrow \texttt{loadA} \rightarrow \texttt{ALU} \rightarrow \texttt{savePC}$$

(a) Textual specification.



(b) Graphical specification.

Figure 4.2: A scenario specification comprising two processor instructions.

- S is the set of scenarios $\{s_1, s_2, ..., s_n\}$ described as POs.
- C is the universe of codes $\{c_1, c_2..., c_{|C|}\}$ satisfying the following two properties:
  1. $|C| = 2^{|B|}$;
  2. $c_i \neq c_j$ for $1 \leq i < j \leq |C|$;

  e.g. given a set of Boolean variables $B = \{b_1, b_2\}$, the corresponding code universe is $C(B) = \{00, 01, 10, 11\}$.
- *Encoding* is a set of n pairs $\{(s_1, c_1), ..., (s_n, c_n)\}$, where each scenario s_i is encoded by the code c_i, such that: $s_i \neq s_j \ \wedge \ c_i \neq c_j$ for all $1 \leq i < j \leq n$;

The arithmetic instruction scenario s_1 and the unconditional branch scenario s_2 in Figure 4.2b can be encoded by one Boolean variable $B = \{b\}$, with the code universe $C(B) = \{0, 1\}$. In this example, the two scenarios have the following encoding: $\{(s_1, 0), (s_2, 1)\}$.

Different encodings lead to different CPOGs, and consequently to different hardware implementations. In our process, we aim at finding an encoding that minimises the size of derived implementations. Reducing the size of a circuit is important as it impacts also other characteristics, e.g. static power consumption, gate-level synthesis complexity, latency.

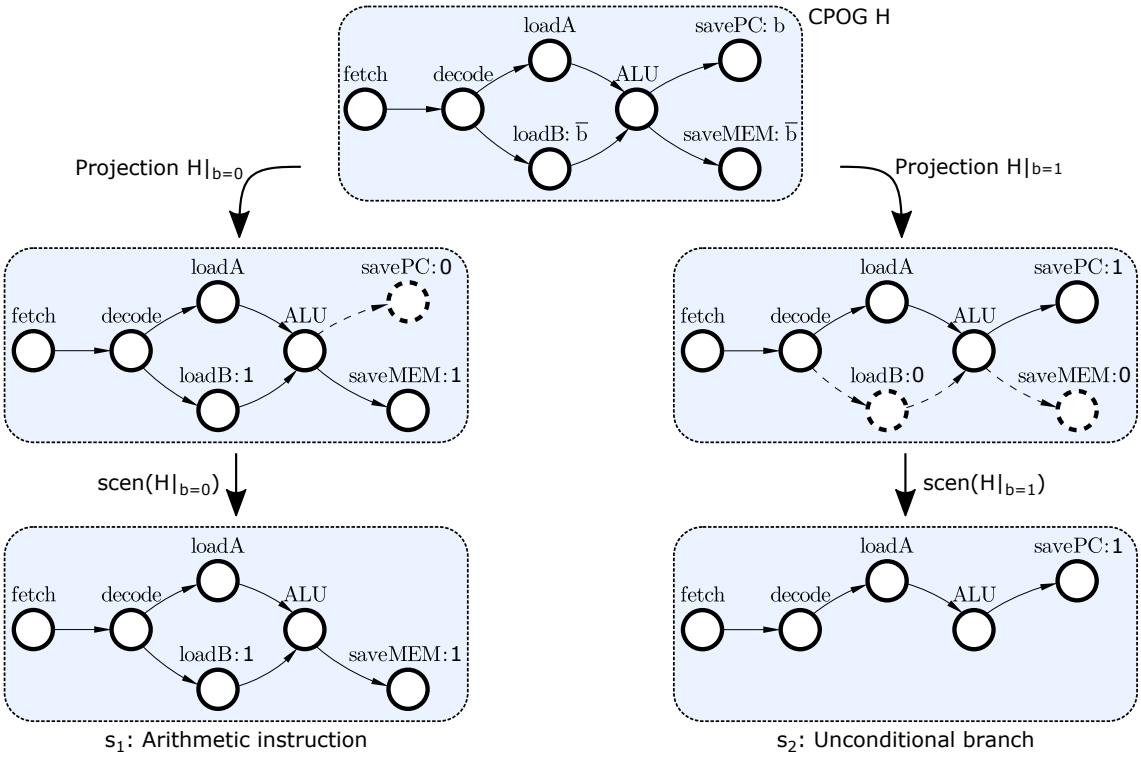


Figure 4.3: From the CPOG to its constituent POs.

4.1.1.3 Composition

Let $e = \{(s_1, c_1), .., (s_n, c_n)\}$ be a scenario encoding for a CPOG $H = (V, E, B, \phi)$. The following definitions will be used to formally state the *CPOG synthesis problem*, see Figure 4.3 for an easier understanding.

- A *projection* $H|_{c_i}$ applies the code c_i to all Boolean conditions of H. The result is a graph H_i, whose vertex/arc conditions are now fully evaluated to 1 or 0.
- The operation `scen`(H_i) removes vertices and arcs with 0 condition, and applies the transitive closure to the resulting graph, obtaining the scenario s_i.

The purpose of the above definitions is to let a code c_i select a scenario s_i from the CPOG:

$$\forall\, 1 \leq i \leq n, \texttt{scen}(H|_{c_i}) = s_i$$

The CPOG synthesis process uses the encoding e to synthesise the CPOG H. It produces the encoding functions $F(B) = \{f_1, f_2, ..., f_n\}$, so that the code $c_i \in e$ selects

the scenario $s_i \in e$. Following [53], we represent the CPOG H as the following linear combination of projections:

$$H = f_1 H|_{c_1} + ... + f_n H|_{c_n} = \sum_{1 \leq i \leq n} f_i H_i = \sum_{1 \leq i \leq n} f_i \mathsf{scen}^{-1}(s_i)$$

The CPOG synthesis requirement is satisfied if the encoding functions are orthogonal ($f_i f_j = 0, 1 \leq i < j \leq n$), and are not contradictions, i.e. $f_i \neq 0$ for all $1 \leq i \leq n$.

As an example, consider the encoding $\{(s_1, 0), (s_2, 1)\}$ of the scenarios in Figure 4.2. The resulting CPOG should be in the form of $H = f_1 H|_{c_1} + f_2 H|_{c_2}$ such that $\texttt{scen}(H|_{c_1}) = s_1$ and $\texttt{scen}(H|_{c_2}) = s_2$. The CPOG is represented by the linear combination $H = \overline{b} H|_0 + b H|_1$, and the encoding functions $f_1 = \overline{b}$ and $f_2 = b$ satisfy the synthesis requirement. Figure 4.3 shows the resulting CPOG H on top, the projections $H|_{b=0}$ and $H|_{b=1}$ in the centre, and the initial scenarios $\texttt{scen}(H|_{b=0})$ and $\texttt{scen}(H|_{b=1})$ at the bottom.

4.1.1.4 Hardware synthesis

The hardware synthesis step of the design flow extracts a set of Boolean equations from the synthesised CPOG, and obtains an implementation of the *hardware controller*. Its area, latency and power strongly correlate with the CPOG complexity [23], which can be seen as the number of Boolean literals of conditions ϕ. An operation $v \in V$ can be executed if:

1. it belongs to the current projection, i.e. $\phi(v) = 1$;
2. all preceding vertices have already been executed: $\forall u \in V, (u \prec v) \Rightarrow \texttt{ack}(u)$.

This is captured in terms of Boolean equations as follows:

$$\texttt{req}(v) = \phi(v) \wedge \prod_{\forall u \in V} [\phi(u) \wedge \phi((u, v)) \Rightarrow \texttt{ack}(u)],$$

where (u, v) is the arc from u to v, $\texttt{req}(v)$ is the request signal which activates the v operation, while $\texttt{ack}(u)$ is the acknowledgement signal which comes from the u operation, and indicates its completion. As an example, the hardware implementation (in the form of Boolean equations) of the CPOG on top of Figure 4.3 is shown below:

$$
\begin{cases}
\texttt{req}(fetch) = \texttt{go} \\
\texttt{req}(decode) = \texttt{ack}(fetch) \\
\texttt{req}(loadA) = \texttt{ack}(decode) \\
\texttt{req}(loadB) = \overline{b} \wedge \texttt{ack}(decode) \\
\texttt{req}(ALU) = \texttt{ack}(loadA) \wedge (\overline{b} \Rightarrow \texttt{ack}(loadB)) \\
\texttt{req}(savePC) = b \wedge \texttt{ack}(ALU) \\
\texttt{req}(saveMEM) = \overline{b} \wedge \texttt{ack}(ALU) \\
\texttt{done} = (b \Rightarrow \texttt{ack}(savePC)) \wedge (\overline{b} \Rightarrow \texttt{ack}(saveMEM))
\end{cases}
$$

The signals `go` and `done` are automatically added into set of operations to delimit the start and the end of a scenario execution. The Boolean equations above can be used to produce a gate-level description of the hardware controller (in the form of a Verilog file), whose corresponding digital circuit is shown in Figure 4.4. The controller is a combinational component, and is generated automatically by mapping the derived Boolean equations over a technology gate library introduced into the developed flow, see Section 4.1.4.

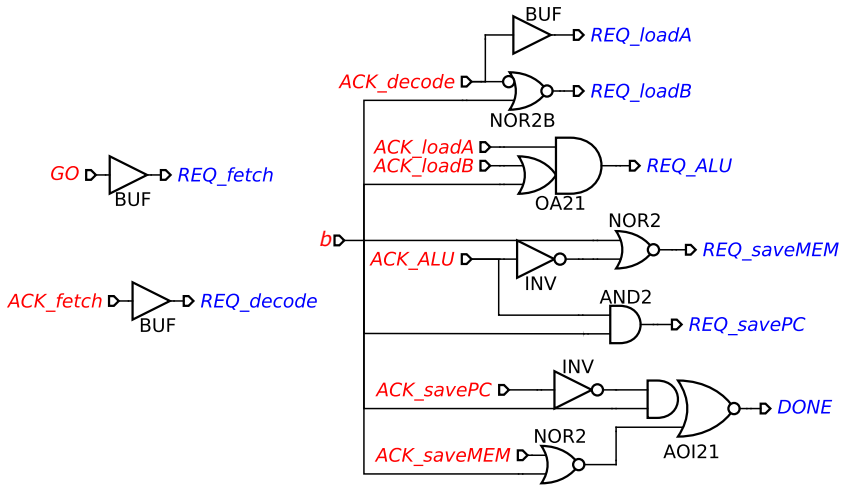


Figure 4.4: Hardware controller derived by the CPOG on top of Figure 4.3. Red signals are the inputs, while blue signals are the outputs.

The controller is in compliance with the scenario-based specification in Figure 4.2. The operations are executed according to the specified dependencies in response to input signals coming from the datapath units. As an example, the ALU is activated by raising the output signal `REQ_ALU` if:

- the arithmetic instruction is decoded (input $b = 0$), and the operations *loadA* and *loadB* have both been executed (inputs `ACK_loadA` and `ACK_loadB`);
- the unconditional branch is decoded (input $b = 1$), and the `loadA` has been executed (input `ACK_loadA`).

Afterwards, the controller can be connected to a set of synchronous or asynchronous datapath modules automatically. This part of the flow is named *interface synthesis*, and will be described in Section 4.1.4.2. The final implementation, including the generated controller, can be processed by conventional EDA tools.

4.1.2 Related work

The characteristics of the synthesised hardware controller correlate with the encoding selected [40]. In this work, we present a metric for extracting such a correlation, and an algorithm for approaching the efficient-behavioural composition heuristically. In this section, we report other encoding techniques available for the efficient composition of scenarios into a CPOG.

The *Single-literal encoding* [23] is based on the graph colouring algorithm [52]. It finds and encoding under the constraint that each Boolean equation $\phi(z)$ of the synthesised CPOG can have at most 1 literal. The number of Boolean variables $|B|$ determines the colours available for solving the graph colouring problem, and can be increased above $\lceil log_2|S| \rceil$ automatically.

The *SAT-based encoding* [40] uses SAT solvers (CLASP [65] or MINISAT [66]) for minimising the synthesised CPOG Boolean equations. The number of Boolean variables $|B|$ for encoding is set by the user. In this paper, we set $|B| = \lceil log_2|S| \rceil$.

In Chapter 5, we show that the above approaches do not scale well to high number of scenarios ($|S| > 15$).

4.1.3 The new scenario composition algorithm

The optimal scenario encoding problem is NP-complete [23]. Finding the encoding that optimises a target hardware characteristic can be only achieved by synthesising and comparing all available encodings. In practice, this *exhaustive search* is infeasible due

Table 4.1: Symmetric encodings derivable from e_1, e.g. e_2 is symmetric to e_1, as it can be obtained by negating the Boolean variable b_1 in all the codes in e_1.

Scenarios	$e_1(b_1b_2)$	$e_2(\overline{b_1}b_2)$	$e_3(b_1\overline{b_2})$	$e_4(\overline{b_1b_2})$
s_1	00	10	01	11
s_2	01	11	00	10
s_3	10	00	11	01
s_4	11	01	10	00

to the exponential growth of number of available encodings $|\mathcal{E}|$ when either the number of $|S|$ scenarios or $|C|$ codes increases, $|\mathcal{E}|$ defined in Section 4.1.3.1. This motivates the proposed *Heuristic encoding*, described in this section.

4.1.3.1 Symmetric encodings

It is inefficient to inspect encodings that result in similar hardware implementations. This is the case for *symmetric encodings*, which are best explained by an example. The encoding $e_1 = \{(s_1, 00), (s_2, 01), (s_3, 10), (s_4, 11)\}$ has three symmetric encodings: e_2, e_3 and e_4, see examples in Table 4.1.

A symmetric encoding can be obtained by negating one or more Boolean variables in all the codes of an encoding. We do not consider symmetric encodings, as the corresponding implementations differ only in terms of input inverters, which is insignificant. To rule out symmetric encodings we always encode the first scenario with the first available code. For being consistent across multiple scenario encoding runs, we encode the first scenario by the *zero code* 00..0 (i.e. $e = \{(s_1, 00..0), \cdots, (s_{|S|}, c_{|S|})\}$ for all $e \in \mathcal{E}$) if not constrained to encode another scenario (as we will explain in the next section).

The symmetry-breaking allows to restrict the universe of allowed encodings $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ to the set that satisfies the two properties below:

1. All encodings are different: $e_i \neq e_j$ for $1 \leq i < j \leq |\mathcal{E}|$.

2. No two encodings e_i and e_j are symmetric.

Given $|S|$ scenarios and $|C|$ codes, the size of the universe of encodings is:

$$|\mathcal{E}| = \frac{(|C|-1)!}{(|C|-|S|)!}$$

Note that at least $\lceil log_2 n \rceil$ Boolean variables are needed to encode $|S|$ scenarios ($|B| \geq \lceil log_2 |S| \rceil$). In this paper, we fix the number of such variables to the minimum, and restrict $|\mathcal{E}|$ using $|C| = 2^{\lceil log_2 |S| \rceil}$ codes, see Section 4.1.1.2.

4.1.3.2 Composition constraints

In real-life systems, there are *composition constraints* that restrict the space of allowed encodings, for example due to backward compatibility requirements. Consider the two scenarios in Figure 4.2, and assume that the following constraints must be met:

- The code of the arithmetic instruction (s_1) consists of an arbitrary 2-bit opcode, and two 3-bit operands A and B.
- The unconditional branch (s_2) consists of the fixed 00111 opcode, and a 3-bit branch offset.

The above requirements can be expressed with the *composition constraints* $G = \{(s_1, ??XXXXXX), (s_2, 00111XXX)\}$, which uses 8 Boolean variables $B = \{b_1, b_2, ..., b_8\}$.

- The arithmetic instruction 2-bit opcode ($b_1 b_2$) is denoted by ??, where each ? is a *don't care* bit that becomes either 0 or 1 in the encoding. Each X is a *don't use* bit, which is not used for selecting a PO from those contained in the CPOG. In fact, 6 bits are left unused for the two operands A ($b_3 b_4 b_5$) and B ($b_6 b_7 b_8$).
- The unconditional branch opcode ($b_1 b_2 b_3 b_4 b_5$) is fixed to 00111, the remaining 3 bits are left unused for the branch offset operand ($b_6 b_7 b_8$).

As shown in the above example, a *constraint* g is an assignment $g : B \rightarrow \{0, 1, ?, X\}$ of the set of Boolean variables B. Sets of constraints are used to express composition constraints $G = \{(s_1, g_1), (s_2, g_2), ..., (s_{|S|}, g_{|S|})\}$.

The presented algorithm handles composition constraints. As an example, the constraints set above can be satisfied by $\{(s_1, 10XXXXXX), (s_2, 00111XXX)\}$. On the other hand, $\{(s_1, 00XXXXXX), (s_2, 00111XXX)\}$ is an incorrect encoding as the code 00111000 selects both the two instructions. We say that the codes $00XXXXXX$ and $00111XXX$ are *overlapping* as they are both identified by the same code (e.g. 00111000).

To detect overlapping codes, let us define the overlapping function $\texttt{ovp}(c_1, c_2)$, which (1) takes as input two codes c_1 and c_2 that can contain *don't use* bits[1], (2) calculates the two sets of codes (with no *don't use* bits) that identify c_1 and c_2, and finally (3) computes their intersection. We say that c_1 and c_2 are overlapping if $\texttt{ovp}(c_1, c_2) \neq \emptyset$. For example, the codes 011 and $00X$ are not overlapping, as the result of the function ovp is the empty set $\emptyset$ (i.e. the codes cannot be identified by the same code), see below.

$$\texttt{ovp}(011, 00X) = \{011\} \cap \{000, 001\} = \emptyset$$

On the other hand, the two codes $11X$ and $1X0$ are said to be overlapping, as the result of the function ovp is 110, which indeed identifies both the two codes, see below.

$$\texttt{ovp}(11X, 1X0) = \{110, 111\} \cap \{100, 110\} = \{110\}$$

To enable the usage of *don't use* bits, we use the function ovp to reformulate the definition of encoding (in Section 4.1.1.2) as follows:

Definition 4.15. ***(Encoding) -*** *Encoding is a set of $|S|$ pairs $\{(s_1, c_1), .., (s_{|S|}, c_{|S|})\}$, where each scenarios s_i is encoded by the code c_i such that: $s_i \neq s_j \wedge \texttt{ovp}(c_i, c_j) = \emptyset$ for all $1 \leq i < j \leq |S|$.*

In addition, an encoding must also satisfy the composition constraints set by the user in order to be applicable to real-life systems. This formally means:

Definition 4.16. *An encoding $e = \{(s_1, c_1), .., (s_{|S|}, c_{|S|})\}$, as defined in 4.15, satisfies a set of composition constraints $G = \{(s_1, g_1), .., (s_{|S|}, g_{|S|})\}$ if it is possible to derive c_i from g_i for all $1 \leq i \leq |S|$ by substituting every don't care bit ? in g_i to either 0 or 1.*

For example, the encoding $\{(s_1, 100), (s_2, 110)\}$ satisfies the composition constraints $\{(s_1, 1??), (s_2, 1??)\}$ as the two constraints can be used to derive the two codes:

$$g_1 \xrightarrow{\checkmark} c_1 : \ g1 = \{b_1 = 1, b_2 = ?, b_3 = ?\} \xrightarrow{b_2 b_3 = 00} c_1 = \{b_1 = 1, b_2 = 0, b_3 = 0\}$$
$$g_2 \xrightarrow{\checkmark} c_2 : \ g2 = \{b_1 = 1, b_2 = ?, b_3 = ?\} \xrightarrow{b_2 b_3 = 10} c_2 = \{b_1 = 1, b_2 = 1, b_3 = 0\}$$

On the other hand, the encoding $\{(s_1, 11X), (s_2, 010)\}$ does not satisfy the composition constraints $\{(s_1, 1?X), (s_2, 1??)\}$ as g_2 cannot be used to derive c_2 (see b_1):

[1] To enable the usage of *don't use* bits, a code c becomes an assignment $c : B \rightarrow \{0, 1, X\}$ of the set of Boolean variables B.

$$g_1 \xrightarrow{\checkmark} c_1 : \ g1 = \{b_1 = 1, b_2 =?, b_3 = X\} \xrightarrow{b_2=1} c_1 = \{b_1 = 1, b_2 = 1, b_3 = X\}$$
$$g_2 \xrightarrow{\times} c_2 : \ g2 = \{\underline{b_1 = 0}, b_2 =?, b_3 =?\} \xrightarrow{b_2 b_3=10} c_2 = \{\underline{b_1 = 1}, b_2 = 1, b_3 = 0\}$$

The above definitions will be used to sketch a proof of correctness of the presented algorithm in Section 4.1.3.5. The implementation details for satisfying constraints and finding the initial encoding, prior to the heuristic optimisation, are described in Algorithm 1.

The function findInitialEncoding takes as **input** the Boolean variables B for encoding and the set of composition constraints G. The latter is an array of size $|S|$ whose indexes represent the scenarios and whose elements represent the constraints. As running example, we consider the constraints on 5 scenarios $\{(s_0, ???), (s_1, ??X), (s_2, ???), (s_3, 110), (s_4, ???), (s_5, ???)\}$ on the 3 variables $B = \{b_1, b_2, b_3\}$, which is represented by the array $G = (???, ??X, ???, 110, ???, ???)$.

Initially, the universe of codes C is initialised with $2^{|B|}$ codes **(line 2)**, and the encoding *enc* with $|S|$ no-code symbols $(-)$ as the encoding is initially empty **(line 3)**. The array *enc* represents the initial encoding, its indexes represent the scenarios and its elements represent the codes. In the example, C and *enc* are:

$$C = (000, 001, 010, 011, 100, 101, 110, 111)$$
$$enc = (-, -, -, -, -, -)$$

In **lines 4-7**, the codes paired with fully constrained scenarios (i.e. $? \notin G[i]$) are checked to be contained in the universe of code C **(line 5)** – if two or more scenarios are encoded by the same code, an *error* is returned. Subsequently, these scenarios are encoded by the provided codes **(line 6)**. The latter are removed from C **(line 7)**, as they cannot be used for encoding other scenarios. C and *enc* become:

$$C = (000, 001, 010, 011, 100, 101, 111)$$
$$enc = (-, -, -, \underline{110}, -, -)$$

In **lines 8-15**, partially constrained codes (i.e. $? \in G[i] \wedge G[i] \neq ?^{|B|}$) are turned into codes, i.e. the function randomAssignment$(G[i])$ turns their don't care bits ($? \in G[i]$) into binary values $\{0, 1\}$ randomly **(line 11)**. Such resulting codes (*code*) encode scenarios s_i

Algorithm 1: Algorithm for satisfying the given composition constraints and finding the initial encoding.

Function findInitialEncoding(B, G);
Input : Boolean variables B, a set of $|S|$ constraints G.
Output : Encoding *enc*, *error*, or *askUser*.
Parameter: $MAX(= 10)$ number of possibile iterations.
// Find the universe of codes, it contains the available codes for encoding
$C \leftarrow (0^{|B|}, \cdots, 1^{|B|})$;
// Initially, the encoding is empty, i.e. it contains a set of no-codes '$-$'
$enc \leftarrow (-_1, \cdots, -_{|S|})$;
// Encode the scenarios that have been fully constrained by the user
foreach i ***such that*** $? \notin G[i]$ **do**
    // Error if the user encodes two or more scenarios with the same code
    if $G[i] \notin C$ **then** return *error*;
    // Encode the i_{th} scenario $enc[i]$ by the code $G[i]$
    $enc[i] \leftarrow G[i]$;
    // Remove the used code $G[i]$ by the universe C, as $G[i]$ cannot be reused
    $C \leftarrow C \setminus G[i]$;
// Encode the scenarios that have been partially constrained by the user
foreach i ***such that*** $(? \in G[i]) \wedge (G[i] \neq ?^{|B|})$ **do**
    // Derive the *code* from the constraint $G[i]$ by substituting every ? with either 0 or 1 randomly ($? \rightarrow \{0,1\}$), see Definition 4.16
    $iteration \leftarrow 0$;
    do
        $code \leftarrow$ randomAssignment$(G[i])$;
    while $(code \notin C \ \wedge \ iteration{+}{+} < MAX)$;
    // If an available codes is not found, ask for user's intervention
    if $code \notin C$ **then** return *askUser*;
    $enc[i] \leftarrow code$;
    $C \leftarrow C \setminus code$;
// An error is returned if there are not enough codes to encode all remaining scenarios with the given constraints
if $|C| < |- \in enc|$ **then** return *error*;
// Encode the first scenario by the zero code for avoiding symmetric encodings across multiple scenario encoding runs
if $(G[0] = ?^{|B|}) \wedge (0^{|B|} \in C)$ **then**
    $enc[0] \leftarrow 0^{|B|}$;
    $C \leftarrow C \setminus enc[0]$;
// Encode remaining scenarios with available codes in C randomly
foreach i ***such that*** $G[i] = ?^{|B|}$ **do**
    $enc[i] \leftarrow$ pickRandom(C);
    $C \leftarrow C \setminus enc[i]$;
return *enc*;

(**line 14**), and are subsequently removed from C (**line 15**). If a valid code cannot be found, the parameter *askUser* is returned. In the latter case, the user is prompt to select one of the following choice: (1) increasing the algorithm effort, that is increasing the value of the MAX parameter; (2) relaxing or modifying the composition constraints set; (3) run the Exhaustive search - which is slow but always guarantees a solution. In the example, the constraint $\{??X\}$ is turned to $\{01X\}$, and is used to remove $\{010, 011\}$ from C. C and *enc* become:

$$C = (000, 001, 100, 101, 111)$$
$$enc = (-, \underline{01X}, -, 110, -, -)$$

The function randomAssignment can introduce overlapping codes. In the example, the constraint $G[1] = ??X$ cannot be turned to $11X$, as the latter is already used for encoding s_3 ($11X \notin C$). **Lines 10-12** can be repeated up to a MAX of 10 times to increase the probability of satisfying all constraints. If the constraint is still not satisfied, an *error* is returned (**line 14**). Notice that finding an initial encoding that satisfies the given constraints is a NP-hard problem that can be solved only by trying exhaustively all possibilities, for instance with a SAT solver [65, 66]. In practice, the presented heuristic works well and can be tuned by modifying the values of MAX in order to enable the algorithm to try more possibilities.

In **line 16**, an *error* is returned if the number of codes left for encoding ($|C|$) is less than the scenarios that need to be encoded ($|- \in enc|$). In this case, more codes and bits B are required for encoding the given S under the constraints G.

In **lines 17-19**, the first scenario s_0 is encoded by the zero code if it is unconstrained ($G[0] = ?^{|B|}$) and if the zero code has not been used ($0^{|B|} \in C$). This is necessary for avoiding symmetric encodings across multiple scenario encoding runs. In the example, C and *enc* become:

$$C = (001, 100, 101, 111)$$
$$enc = (\underline{000}, 01X, -, 110, -, -)$$

In **lines 20-22**, unconstrained scenarios ($G[i] = ?^{|B|}$) are encoded randomly. Codes left are extracted by C, used to encode scenarios s_i **(line 21)**, and subsequently removed from C **(line 22)**. In the example, C and *enc* become:

$$C = (101)$$

$$enc = (000, 01X, \underline{100}, 110, \underline{001}, \underline{111})$$

The **output** of Algorithm 1 is the encoding *enc*, which satisfies G and can be optimised via the heuristics that we will describe shortly.

4.1.3.3 Heuristic cost function

The main idea of the heuristic encoding algorithm is to encode similar scenarios by similar codes. Similarities between codes are determined using the classic *Hamming distance* metric [67]. Similarities between scenarios, on the other hand, are determined by referring to their partial order representation. Consider two scenarios in the form of Partial orders (reviewed in Section 3.1) $s_1 = (\mathcal{O}_1, \prec_1)$ and $s_2 = (\mathcal{O}_2, \prec_2)$. The distance between s_1 and s_2 is computed following the two rules below:

1. An operation $o \in \mathcal{O}_1$ counts as a difference if $o \notin \mathcal{O}_2$.
2. A dependency $(o_s \prec o_t) \in \prec_1$ counts as a difference if $(o_s \prec o_t) \notin \prec_2$, and if it connects two operations which are both present in the operation sets of the two scenarios: $(o_s \in \mathcal{O}_1 \wedge o_s \in \mathcal{O}_2) \wedge (o_t \in \mathcal{O}_1 \wedge o_t \in \mathcal{O}_2)$.

Distances between pairs of scenarios are elements of the *Scenario Distance Matrix* $\mathcal{SD}$. Distances between pairs of codes are elements of the *Code Distance matrix* $\mathcal{CD}$. Both $\mathcal{SD}$ and $\mathcal{CD}$ have size $|S| \times |S|$, where $|S|$ is the size of the scenario specification. Elements $\mathcal{SD}_{ij}$ and $\mathcal{CD}_{ij}$ represent the number of differences between the i_{th} and j_{th} scenarios and codes, respectively, in an encoding e in the form of $\{(s_i, c_i), (s_j, c_j), \cdots, (s_{|S|}, c_{|S|})\}$). These matrices are used to evaluate encodings heuristically via the below *cost function*:

$$\mathcal{F}(S, e) = \sum_{0 \leq i < j \leq |S|} (\mathcal{SD}_{ij} - \mathcal{CD}_{ij})^2 \tag{4.1}$$

Intuitively, minimising $\mathcal{F}$ means encoding similar scenarios with similar codes. We evaluated the cost function $\mathcal{F}$ empirically, by analysing several scenario specifications.

As an example, Figure 4.5a shows the analysis of a subset of 8 scenarios of the Intel 8051 [24] scenario specification, where the universe of encoding $\mathcal{E}$ is fully inspected,

and 5040 controllers are synthesised with a 90 *nm* technology library [68]. The size of the controllers is plotted against the heuristic value $\mathcal{F}$ of the corresponding encodings. Figure 4.5b, in turn, shows the analysis of the scenario specification of the Arm Cortex M0+ [27], composed of 11 scenarios. In the figure, 10^2 controllers produced by the proposed algorithm, described in Section 4.1.3.4, are compared to 10^5 controllers produced by encoding scenarios randomly.

The two figures highlight the existence of a correlation between the controller area and $\mathcal{F}$, and suggest the following two claims:

- the likelihood of synthesising efficient implementations is higher where $\mathcal{F}$ is lower, see *Promising candidates* in Fig. 4.5a;
- the likelihood of synthesising efficient implementations is proportional to the number of encodings inspected, due to the inaccuracy of the heuristics, see the *Variability* span in Fig. 4.5b.

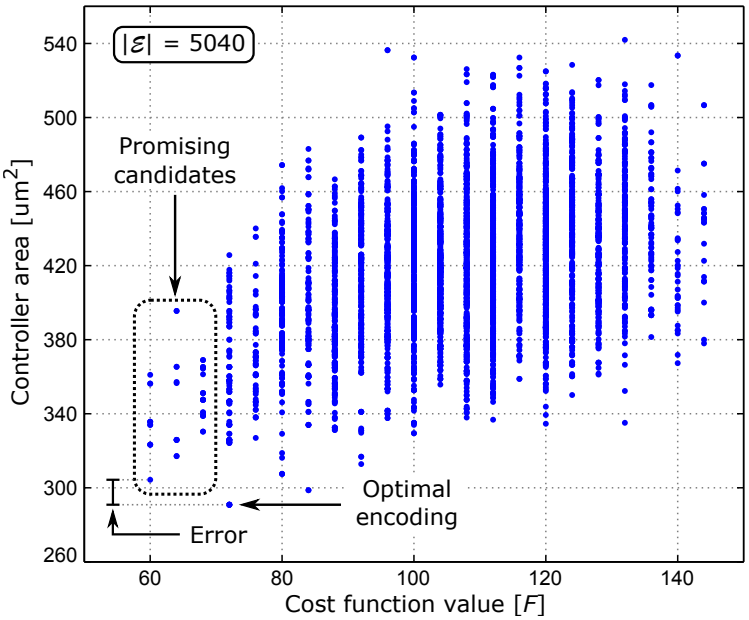
4.1.3.4 The heuristic encoding algorithm

The presented heuristic algorithm is based on the cost function $\mathcal{F}$, and on a proposed implementation of the simulated annealing (SA) [69]. The latter is a heuristic method for solving optimisation problems where a function must be minimised in a large search space. The algorithm pseudo-code is shown in the Algorithm 2.

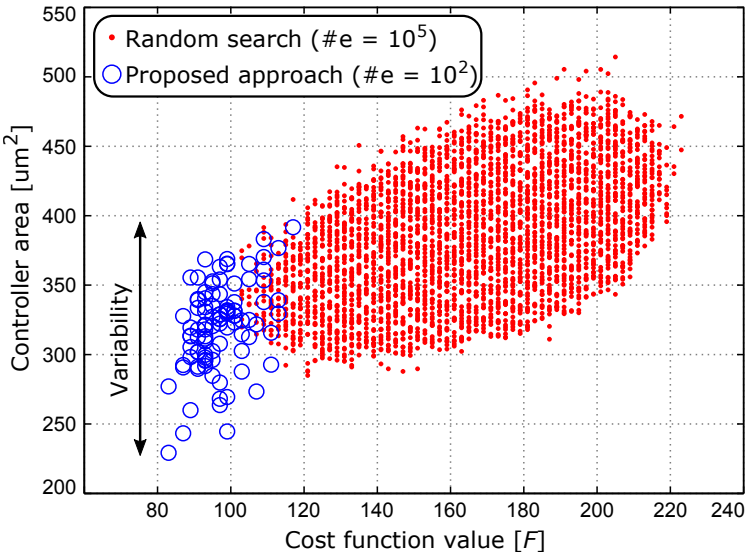
The **inputs** of the function heuristicEncoding are the Boolean variables B, the scenarios S and constraints G. We continue the running example used for the Algorithm 1, where $G = \{???, ??X, ???, 110, ???, ???\}$ were turned to $enc = \{000, 01X, 100, 110, 001, 111\}$ by the findInitialEncoding function **(line 2)**. This encoding is initially copied into enc_{best} **(line 3)**, which represents the best encoding found during the SA search.

Simulated annealing parameters were calibrated experimentally. The initial temperature is $t_0 = 10$. The cooldown factor alpha is $a = 0.996$, and the ending temperature is $t_e = 0.1$. These parameters can be modified for increasing or decreasing the number of iterations for the SA optimisation.

Line 4 initialises the universe of codes C. The code of the first scenario $enc[0]$ is removed for avoiding symmetric encodings.



(a) All controllers ($\mathcal{E}$) plotted with respect to the cost function $\mathcal{F}$ (Intel 8051 [24], subset of 8 scenarios). The *promising candidates* are the encodings that we aim to identify with the presented cost function F and synthesise, as they are more likely to result in more efficient controllers. Notice that the *optimal encoding* is not included in the promising candidates, but that the area overhead of the heuristic controller in comparison to the optimal one (see the *error* span) is small and arguably acceptable.



(b) 10^2 heuristic and 10^5 random controllers are compared, and plotted with respect to $\mathcal{F}$ (Arm Cortex M0+ [27], 11 scenarios). Notice the high area *variability* of the heuristic controllers, i.e. the difference between the biggest and smallest heuristic controllers, which suggests that it is important to inspect as many heuristic encodings as possible to increase the likelihood of finding efficient controllers.

Figure 4.5: The presented cost function is studied over two benchmarks.

Algorithm 2: The presented Heuristic encoding algorithm.

Function heuristicEncoding(B, S, G);
Input : Boolean variables B, scenarios S and their constraints G.
Output : Heuristic encoding enc.
Parameters: $t_0 = 10, a = 0.996, t_e = 0.1$

$enc \leftarrow$ findInitialEncoding(B, G);

// The initial encoding is also the best encoding

$enc_{best} \leftarrow enc$;

// C contains all available codes for encoding. The code of the first scenario $enc[0]$ is removed from C for avoiding symmetric encodings

$C \leftarrow (0^{|B|}, \cdots, 1^{|B|}) \setminus enc[0]$;

// Our Simulated Annealing implementation: the initial temperature t_0 slowly decreases by the factor a until the temperature t_e is reached

while $(t_0 > t_e)$ **do**

  // Save current encoding in enc_{next}

  $enc_{next} \leftarrow enc$;

  // Pick a random scenario and code, notice that the first scenario s_0 cannot be selected for avoiding symmetric encodings

  $i \leftarrow$ pickRandomScenario$(1 \leq i < |S|)$;

  $code \leftarrow$ pickRandomCode(C);

  // If $code$ is used in enc_{next}, then swap the codes within the encoding. We use the symbol $\in$ because the codes in enc_{next} may contain *don't use* bits X, e.g. $001 \in 00X = \{000, 001\}$, while $001 \notin X00 = \{000, 100\}$

  if $\exists j$ ***such that*** $code \in enc_{next}[j]$ **then**

    $enc_{next}[i] \leftrightarrow enc_{next}[j]$;

  else

    $enc_{next}[i] \leftarrow code$; // Else, swap the code of s_i with the extracted $code$

  // If the new encoding enc_{next} satisfies the given constraints G

  if $\forall i$ *satisfy*$(enc_{next}[i], G[i])$ **then**

    // Replace enc_{best} if its cost is higher than the cost of enc_{next}

    if $\mathcal{F}(S, enc_{next}) < \mathcal{F}(S, enc_{best})$ **then**

      $enc_{best} \leftarrow enc_{next}$;

    // Replace enc either if its cost is higher than the cost of enc_{next}, or if a randomly extracted value v is lower than a certain threshold $e^{-\frac{d}{t_0}}$ (an encoding with a higher cost is selected)

    $d \leftarrow \mathcal{F}(S, enc_{next}) - \mathcal{F}(S, enc)$;

    $v \leftarrow$ pickRandomValue$(0 \leq v < 1)$;

    if $v < e^{-\frac{d}{t_0}}$ **then**

      $enc \leftarrow enc_{next}$;

  // cooling the temperature t_0 by the factor a

  $t_0 \leftarrow t_0 \times a$;

return enc_{best};

Lines 5-21 minimise the initial encoding heuristic value $\mathcal{F}(S, enc)$ by repeatedly swapping pairs of codes in the encoding, until the initial temperature t_0 reaches the ending temperature t_e **(line 5)**. **Line 6** stores the current encoding *enc* into the the next encoding enc_{next}. **Lines 7-8** select a random scenarios s_i in *enc* ($1 \leq i < |S|$), and a random code c_j in C, respectively. Such indexes are used for swapping codes in enc_{next}. In **line 9**, we use the symbol $\in$ to check if a *code* extracted randomly is used for encoding a scenario in the encoding (see $code \in enc_{next}[j]$). This symbol is used because scenarios might be encoded by codes having *don't use* bits. In our example, s_1 is encoded by $01X$ and is thus identified by the set of codes $\{010, 011\}$ (i.e. both these two codes $\in 01X$). Coming back to our running example, if $i = 4$ and $j = 7$, the fourth scenario (encoded by 001) is swapped with the code 111 (which identifies s_5 in *enc*). *enc* and enc_{next} become:

$$enc = (000, 01X, 100, 110, \underline{\underline{001}}, \underline{111})$$
$$enc_{next} = (000, 01X, 100, 110, \underline{111}, \underline{\underline{001}})$$

The next encoding enc_{next} is considered if it satisfies the composition constraints G **(line 13)**. The function satisfy (see Algorithm 3) checks that the bit size of the code matches the bit size of the constraint (**line 2**), and that the bits constrained by $\{0, 1, X\}$ hold these values in the final code (**lines 4-6**). Notice that bits constrained by ? do not need to be checked, as both logic values $\{0, 1\}$ satisfy such constraints.

Algorithm 3: Implementation of the function for checking if a code satisfies a constraint.

```
1 Function satisfy(c, g);
  Input  : A code c, and a constraint g.
  Output : Boolean values True or False.
2 if |c| ≠ |g| then return False; // The code and the constraints are of the same size
3 foreach 1 ≤ i ≤ |g| do
4     if (c[i] = 1) ∧ (g[i] = 0) then return False;   // Bits constrained by {0,1,X} have
5     if (c[i] = 0) ∧ (g[i] = 1) then return False;   // to keep these values in the code
6     if (c[i] ≠ X) ∧ (g[i] = X) then return False;
7 return True;
```

In **lines 14-15**, enc_{next} replaces the best encoding enc_{best} found during the SA optimisation if the former has a lower heuristic value than the latter, i.e. $\mathcal{F}(S, enc_{next}) < \mathcal{F}(S, enc_{best})$. In **lines 16-19**, enc_{next} also replaces *enc* either if the former has a lower heuristic value than the latter (i.e. $v < e^{-\frac{d}{t_0}}$ for all $0 \leq v < 1 \wedge d \leq 0$), or if the extracted random value

v is lower than $e^{-\frac{d}{t_0}}$, with $d > 0$. In the second case, a worse encoding (with a higher $\mathcal{F}$) replaces *enc*.

The randomness allows the heuristicEncoding to return a different enc_{best} (**output**) at every execution. The solution space is connected, as all $e \in \mathcal{E}$ are reachable by a set of swap moves.

The current implementation of the algorithm is run in a single thread of execution. However, multiple instances of the heuristicEncoding function can be run on multiple threads, resulting in several encodings to be produced concurrently. The parallelisation of the presented algorithm is left as future research.

4.1.3.5 Correctness

An encoding *enc*, constrained by composition constraints G, is said to be *correct* if it meets the definition 4.15, and if it is satisfies G (according to the definition 4.16).

Whenever Algorithm 1 terminates, a *correct* encoding *enc* is returned by construction. I.e. the result *enc* is constructed by selecting the codes for encoding from the universe C, which only contains valid codes being derived by the number of bits $|B|$ selected for encoding (**line 2**). Fully constrained scenarios are encoded by their given constraint (see **line 6**), partially constrained scenarios are encoded by codes derived by replacing every ? by either 0 or 1 as described by the definition 4.16 (see **lines 9-14**), and non-constrained scenarios are encoded by codes that have not been used previously, see **line 21**. Thus, the resulting *enc* always satisfies G. Also, the resulting *enc* meets the definition 4.15, as overlapping codes cannot be introduced in the final result: a code is removed from C every time that it is used for encoding a scenario, see **lines 7**, **15**, **19** and **22**.

On the other hand, Algorithm 1 generates an *error* if the constraints G cannot be met for any of the following two reasons:

- The user introduces overlapping constraints, see **line 5**.
- The number of codes $|C|$ is not enough for encoding a set of scenarios with size $|G|$ with the given constraints G, see **line 16**.

In these cases, the constraints cannot be satisfied, and the user is requested to fix the composition constraints set.

The algorithm may also ask for the user intervention (*askUser*) in the case that a partially constrained code is not turned into a code c left for encoding ($c \notin C$) in any of the MAX iterations, see **lines 9-13**. In this case the user can either choose to increase the effort of the algorithm, to relax the encoding constraints, or to run an exhaustive search.

In regards to Algorithm 2, the function heuristicEncoding handles the output of the previous function *enc*, and advances to enc_{best} through a sequence of swap moves that inspects many intermediate encodings enc_{next}. The initial encoding *enc* returned by the Algorithm 2 in **line 2** is *correct* by construction, as we showed in the previous paragraph. Each intermediate encoding derived by a swap move (see **lines 7-12**) always meet the definition 4.15, as no sequences of swaps can introduce overlapping encodings: the code of a scenario can either be swapped with a code of another scenario, or with a code which was is not used (in C) for encoding other scenarios. Intermediate encodings replace enc_{best} if they satisfy the constraints G (this is checked by the Algorithm 3), thus they also meet the definition 4.16. Consequently, enc_{best} is also *correct* by construction. The two algorithms always terminate, as there are not infinite loops.

4.1.3.6 Time complexity analysis

The function findInitialEncoding is constituted by a sequence of three loops. The first one (**lines 4-7**) encodes fully constrained scenarios by moving their codes into the encoding *enc*. Its complexity only depends on the number of fully constrained scenarios introduced: $\mathcal{O}(|S|)$. The second loop (**lines 8-15**) encodes partially constrained scenarios by looping over the bits $|B|$ of each constraint, in order to flip every ? to $\{0,1\}$. Thus, its complexity is: $\mathcal{O}(|S| \cdot |B|)$. The third loop (**lines 20-22**) makes use of the function pickRandom ($\mathcal{O}(1)$) to extract codes left in the code universe C and encode the unconstrained scenarios. Its complexity depends on the number of scenarios: $\mathcal{O}(|S|)$. Consequently, the complexity of the Algorithm 1 ($\mathcal{A}_1$) comes from the second loop. In this paper, we assume that $|B| = \lceil log_2 |S| \rceil$, hence the below equation:

$$\mathcal{O}(\mathcal{A}_1) = \mathcal{O}(|S|) + \mathcal{O}(|S| \cdot |B|) + \mathcal{O}(|S|) = \mathcal{O}(|S| \cdot \log |S|)$$

On the other hand, the function heuristicEncoding, excluding the internal findInitia-

lEncoding function in line 2, is constituted by a loop that implements an *exponential multiplicative cooling* strategy of the simulated annealing algorithm (SA) [69], i.e. an initial temperature t_0 is multiplied by a constant factor a at each iteration, until an ending temperature t_e is reached. This causes a fixed number of iterations n that can be tweaked by modifying these parameters. At each iteration of the SA, the most computationally expensive statements are in **lines 13** and **22-28**: where the enc_{next} is checked against the constraints G, and in **lines 14** and **16**: where the function $\mathcal{F}$ has to be computed. The former has a complexity of $\mathcal{O}(|S| \cdot \lceil \log_2 |S| \rceil)$, as the encoding has to be checked for every bit of each constraint. The latter has a complexity of $\mathcal{O}(|S|^2)$, see the cost function $\mathcal{F}$ in Formula 4.1. Consequently, Algorithm 2 ($\mathcal{A}_2$) has the following complexity:

$$\mathcal{O}(\mathcal{A}_2) = n \cdot \left[\mathcal{O}\left(|S| \cdot \lceil \log_2 |S| \rceil\right) + \mathcal{O}\left(|S|^2\right)\right] = \mathcal{O}\left(n \cdot |S|^2\right)$$

The described time complexity analysis disregards the implementation details of the further set of functions (e.g. `pickRandomScenario`) that the presented algorithms rely on. However, if implemented reasonably, this further set of functions does not increase the time complexity illustrated above.

4.1.4 Design automation

The design methodology described in Section 4.1.1 is implemented in the tool SCENCO [68] (i.e. *SCENario ENCOder*), and integrated in the WORKCRAFT design environment [34]. The developed tool features the following encoding algorithms.

1. **Exhaustive search** fully explores the universe of encodings $\mathcal{E}$.
2. **SAT-based encoding** and **Single-literal encoding** are described in Section 4.1.2.
3. **Heuristic encoding** is the proposed algorithm (Section 4.1.3).
4. **Random search** encodes scenarios randomly.
5. **Sequential encoding** assigns codes sequentially,
   i.e. $\{(s_1, 000), (s_2, 001), (s_3, 010), ...\}$.

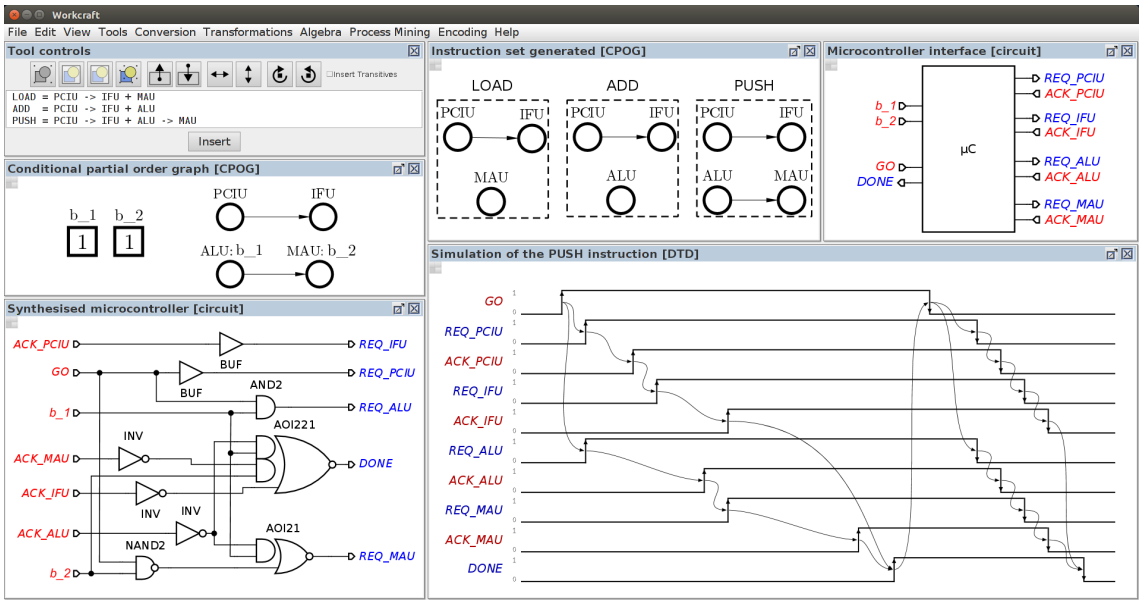


Figure 4.6: Methodology based on the CPOG shown within WORKCRAFT.

SCENCO relies on *Espresso* [70] for Boolean minimisation, and *Abc* [71] for technology mapping, whose library is specified in the GenLib format [72]. *Abc* is also used for producing Verilog files. SCENCO also uses *Clasp* [65] and *MiniSAT* [66] SAT solvers for supporting the SAT algorithms.

This section is divided as follows. Section 4.1.4.1 describes an example of the applied CPOG design methodology in WORKCRAFT. Section 4.1.4.2 describes how to synthesise the interface between a controller (derived by a CPOG) and the controlled datapath modules.

4.1.4.1 Design methodology in Workcraft

SCENCO and the CPOG design methodology are integrated in WORKCRAFT [33] [34]. The latter is a design environment that supports many formal models that have graphs as their underlying structure. WORKCRAFT supports features such as visual editing, simulation, synthesis and analysis.

As an example, Figure 4.6 guides the reader through the specification, synthesis and simulation of a microcontroller in WORKCRAFT, starting from the three simple scenarios (LOAD, ADD and PUSH).

1. **Scenario-based specification**: the three scenarios are introduced textually (*Tools controls* window), and are parsed and converted into partial orders (*Instruction set generated [CPOG]* window). Scenarios can be also entered or edited graphically.

2. **Scenario encoding**: scenarios are encoded using the chosen algorithm (*Encoding* menu). The encoding $\{(\mathsf{LOAD}, 01), (\mathsf{ADD}, 10), (\mathsf{PUSH}, 11)\}$ is found in the example, with two Boolean variables: $B = \{b_1, b_2\}$.

3. **Composition**: CPOG is synthesised automatically (*Conditional partial order graph [CPOG]* window);

4. **Hardware synthesis**: the hardware microcontroller is synthesised from the CPOG (*Synthesised microcontroller [circuit]* window). Available gates are specified in the form of a GenLib [72] technology library. The microcontroller interface is highlighted in the window *Microcontroller interface [circuit]*.

The hardware controller can be simulated and formally verified using other tools available in the WORKCRAFT framework. For example, the window *Simulation of the PUSH instruction [DTD]* shows a simulation of the PUSH scenario of the controller. The sequence of executed operations and their dependencies are shown.

4.1.4.2 Interface synthesis

The synthesis of the interface between the controller and the datapath has been automated in the developed EDA tool [68], relying on the ideas elaborated in [73] and summarised below.

The controller can be interfaced either with asynchronous datapath modules, relying on the reqest/acknowledge handshake, and to synchronous modules using *matched delays* [25], which produce acknowledgement signals after a chosen delay. In turn, since the controller resets request signals only at the end of each scenario execution, *decouple* and *merge* [73] are needed to release datapath modules immediately after they acknowledge their completion. Also, *merge* is used when a module is executed multiple times within a scenario, see a schematic of the interface in Figure 4.7.

The developed tool [68] takes as input the datapath modules in the form of Verilog,

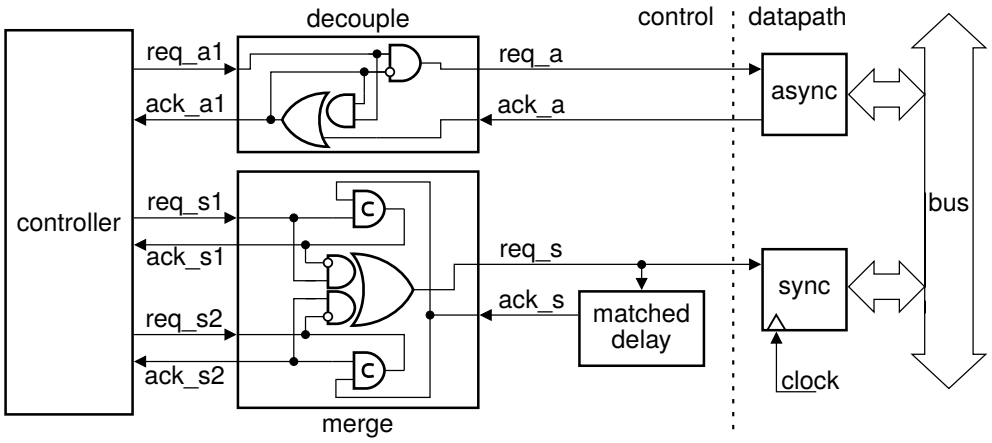


Figure 4.7: The interface between controller and datapath. The controller interfaces to asynchronous components via req/ack interface, and to synchronous ones via matched delays. The decouple and merge modules release a datapath component after the end of its execution. Merge is used when a component is accessed multiple times within within a scenario.

and interfaces them to the synthesised controller automatically. The produced Verilog file contains the final system implementation, see Figure 4.1.

4.1.5 Summary

We presented a new scenario composition algorithm, and applied to the methodology based on Conditional Partial Order Graphs. In Chapter 5, we evaluate the proposed algorithm on an extensive set of case studies for highlighting its characteristics.

4.2 Composition of dataflow structures

In the previous section, we described a novel approach to compose behavioural scenarios efficiently. In this section, on the other hand, we consider the following problem:

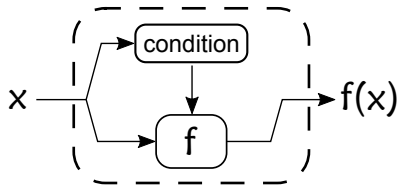
Problem: *Static Dataflow Structures are a known formalism that abstracts the Petri net token-game providing a simple mechanism to represent static event-based systems at the high level. The available formalisation does not allow one to represent dynamic systems, where the flow of events depends on run time conditions. Is it possible to extend the model for breaking this limitation?*

To overcome this limitation, we describe the novel *Dataflow Structures* formalism, which allows to compose Static Dataflow Structures and capture the behaviour of

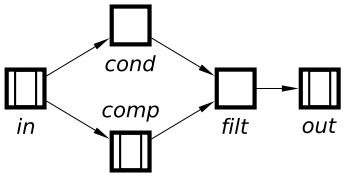
dynamic systems, e.g. dynamically reconfigurable circuits. This section is divided as follows. Section 4.2.1 motivates the new formalism by showing that SDFSs cannot capture asynchronous dynamic reconfigurability. Section 4.2.2 presents the new Dataflow Structures formal model. Section 4.2.3 shows an *inefficient* methodology for composing asynchronous circuit scenarios in the form of SDFS. Sections 4.2.4 and 4.2.5 describe the design automation developed for the presented model. The related work in the field and a final summary of the section are in Section 4.2.6. Part of the content of this section has been/will be published in [29,30].

4.2.1 Motivation

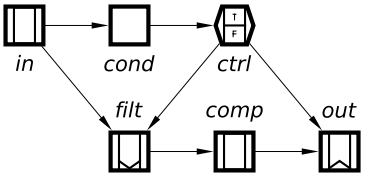
In Section 3.4, we introduced the Static Dataflow Structures formalism, and we said that it is used to model the behaviour of asynchronous circuits by abstracting away low-level implementation details. However, SDFSs cannot handle *dynamic reconfigurability* according to its formulation in [26]. In this section, we describe two examples for highlighting this issue and motiving our research.



(a) High-level view.



(b) SDFS model.



(c) DFS model.

Figure 4.8: Conditional application of a function.

As the first motivating example for the DFS model, consider an asynchronous circuit that applies a computationally expensive pipelined function f (node `comp`) only to those data items that satisfy an easily-checked *condition* (`cond`), e.g. computing a square root only for non-negative numbers. This example can be specified as $\texttt{in} \rightarrow [cond]\texttt{comp} \rightarrow \texttt{out}$ using the algebra of *Parameterised Graphs* [61], which can be used

as high-level specification language for the DFS model. See the high-level view of this example in Figure 4.8a.

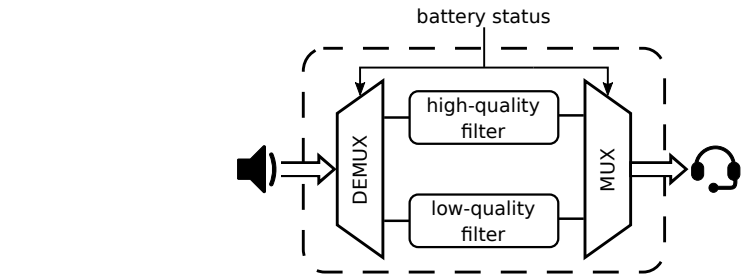
Figure 4.8b shows a possible SDFS model of this described system. An incoming data item (or simply *token*) is duplicated by the `in` register (1) to the `cond` node, whose purpose is to check if the token satisfies a condition; and (2) to the `comp` node, which applies the function f to the token and computes the result. The latter is finally filtered by the `filt` node, which addresses the result to the output register `out` only if the condition is satisfied. This is an inefficient model of this example, as both `cond` and `comp` must be executed before filtering unneeded results. This limits the performance and degrades the power consumption of the pipeline to the worst-case scenario.

Figure 4.9: Static and dynamic nodes included in the Dataflow Structures model.

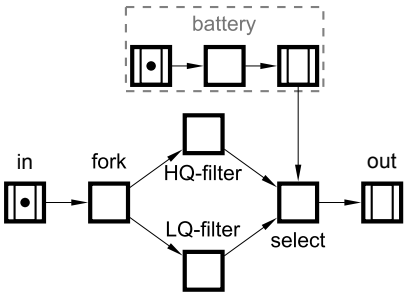
To overcome this issue and adequately model such a dynamic behaviour, we introduce the DFS formalism that extends the SDFS with three new types of registers: *control*, *push* and *pop*. These three nodes are shown in Figure 4.9, and are used to model circuit reconfigurability.

Figure 4.8c shows a DFS model of the above example, which applies the `cond` predicate to incoming tokens, and produces a `True` or `False` token in `ctrl` that either applies or bypasses the function *comp*. In the case of a `True` token, the function *comp* is applied: the push and pop registers `filt` and `out` act as static registers, and the data is propagated from `in` to `out` passing by `comp`. On the other hand, in the case of a `False` token, the function `comp` is bypassed: the incoming token is stored and *destroyed* by `filt` (i.e. the token is not propagated to `comp`), and an 'empty' token (i.e. a token that is not paired with any valid data) is *produced* by `out`.

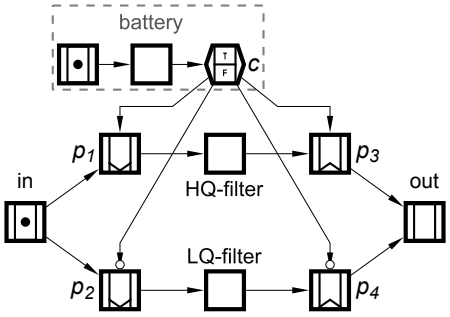
In the first example, we showed the conditional application of a function. As a second motivating example, instead, we show the conditional selection of a function. Consider a hardware module in a mobile phone that is in charge of processing the audio related to a phone call for removing the background noise. The incoming audio data has to



(a) High-level view.



(b) SDFS model.



(c) DFS model.

Figure 4.10: Selection of a noise filter for audio processing.

be processed by a high-quality (HQ) filter when the device has high battery level, and by a low-quality (LQ) filter when the phone is in power saving mode, see Figure 4.10a. This can be expressed as `in` $\rightarrow$ ($[battery]$`HQ-filter` + $[\overline{battery}]$`LQ-filter`) $\rightarrow$ `out` in the algebraic form.

Figure 4.10b shows a possible SDFS model of the above idea. The `fork` node cannot propagate an incoming audio data only to the path in charge of processing the data, and therefore duplicates the token to both the two paths, triggering the execution of the two filters. The `select` node takes as input the two results produced by the filters, and selects the one requested according to the battery level. Its output is propagated to the `out` node. This is an inefficient approach to the representation of this system, as both the HQ and LQ filters are always executed, worsening both the performance and energy consumption of the device. The performance is decreased because the latency of the pipeline is always given by the filter with the maximum latency, i.e. the `select` node has to wait for the two results before propagating the processed input to the output. The energy consumption is also increased because one of the two filters is always executed unnecessarily.

Figure 4.10c shows a possible DFS model of this reconfigurable filter. The *battery* nodes check the left amount of charge and generate a `True` or `False` token in the control register `c`, which selects either the high- or the low-quality filter. In the case of a `True` token, for example, the push register `p1` and the pop register `p3` behave as static registers and propagate the incoming token to the `HQ-filter`, and subsequently to the output `result`. At the same time, the `True` token in `c` is inverted and propagated to the registers `p2` and `p4` as a `False` token via the *inverting arcs* (denoted graphically as $\multimap$). These registers are used to bypass the `LQ-filter` by consuming the incoming token at `p3` and producing a `False` (i.e. empty) token at `p4`. In Section 4.2.3, we will illustrate a simulation of this example relying on the behavioural semantics of the DFS model, which is described in the next section.

These two examples show that SDFS cannot model *dynamic behaviours*, i.e. scenarios executed conditionally. This motivates the proposed Dataflow Structures.

4.2.2 The Dataflow Structures model

In this section, we introduce the DFS model, which extends the SDFS with three additional types of registers and allows to model dynamic behaviours. These are needed for describing circuit reconfigurability.

Figure 4.9 shows the full set of nodes of the DFS. It includes the *static* SDFS set comprising logic nodes L and static registers R (renamed R_{stc} in the DFS); and the three new `control` (R_{ctrl}), `push` (R_{push}) and `pop` (R_{pop}) types of *dynamic* registers, $R_{stc} \cup R_{ctrl} \cup R_{push} \cup R_{pop} = R$. The next paragraphs describe these new nodes.

Control registers R_{ctrl}: These registers can store `True` or `False` tokens. They can propagate the regular or inverting polarity of their hold tokens by means of *regular* ($\rightarrow$) or *inverting* ($\multimap$) arcs. Control registers in the inverting arc preset of a node d (i.e. connected by inverting arcs) are denoted as $\bullet^0 d$, while those in regular arc preset as $\bullet^1 d$. A dynamic node d is said to be *false-controlled* if the condition M_f^c is satisfied, and is said to be *true-controlled* if the condition M_t^c is satisfied, see Figure 4.11. Control registers can implement the AND, OR and PLAIN[2] Boolean functions, while push and pop registers can only implement the PLAIN Boolean function, denoted as $M_{t/f}^{c.PLAIN}$.

[2]The PLAIN Boolean logic function is the one implemented by an asynchronous C-element [25].

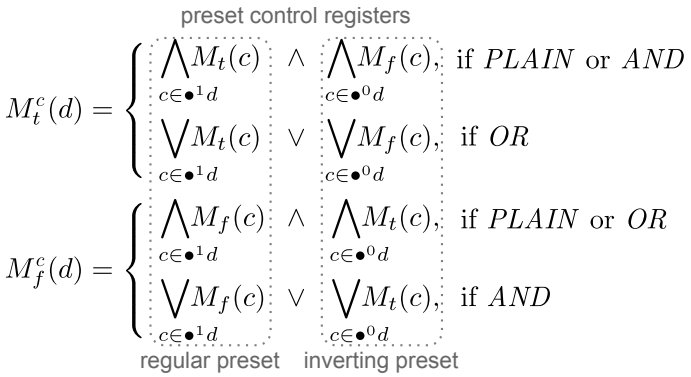


Figure 4.11: Conditions for determining if a dynamic node is false- or true-controlled.

As an example, a control register that implements the OR function is true-controlled if at least one control register in its regular arc preset store a `True` token (M_t), OR if at least one of those in inverting arc preset store a `False` token (M_f). On the other hand, it is said to be false-controlled if all control registers in the regular arc preset store a `False` token, AND all of those in inverting arc preset store a `True` token.

Push registers R_{push}: when true-controlled, they behave as static registers. When false-controlled, on the other hand, they store and *destroy* an incoming token, i.e. the data is not propagated to the register postset. For simplifying their description, push registers are said to handle `True` tokens when true-controlled (i.e. which represent real data items, and are denoted graphically as •), and `False` tokens when false-controlled (i.e. empty data items, and are denoted as ○).

Pop registers R_{pop}: when true-controlled, they behave as static registers and handle `True` tokens. When false-controlled, they *produce* a `False` (or empty) token that is propagated to the postset nodes.

In the light of the above considerations, the new dynamic set of nodes refines the SDFS behavioural semantics (reviewed in Section 3.4) as follows. Figure 4.12 describes the behaviour of logic nodes $l \in L$. Their *evaluation state* C is defined using the *evaluate* $C_\uparrow$ and *reset* $C_\downarrow$ functions, similar to what happens for Set-Reset latches $Q = S \vee \overline{R} \wedge Q$. A logic node can be evaluated when its preset logic is evaluated, preset registers are *marked* (i.e. contain valid data, see function M), and preset push are holding True tokens (M_t). Symmetrically, a logic node can be reset when its preset logic is reset, preset registers (except for push) are *unmarked* (i.e. contain no data, see function $\overline{M}$), and preset

push do not store a True token ($\overline{M_t}$). Intuitively, logic nodes are passive to the handshake mechanism, and are evaluated when all their inputs are available; they are reset when their inputs do not contain valid data.

$$C(l) = C_{\uparrow}(l) \vee \overline{C_{\downarrow}(l)} \wedge C(l), \text{ with } l \in L$$

$$C_{\uparrow}(l) = \bigwedge_{k \in {}^{\bullet}l \cap L} C(k) \wedge \bigwedge_{r \in {}^{\bullet}l \cap (R \setminus R_{push})} M(r) \wedge \bigwedge_{p \in {}^{\bullet}l \cap R_{push}} M_t(p)$$

$$C_{\downarrow}(l) = \bigwedge_{k \in {}^{\bullet}l \cap L} \overline{C(k)} \wedge \bigwedge_{r \in {}^{\bullet}l \cap (R \setminus R_{push})} \overline{M(r)} \wedge \bigwedge_{p \in {}^{\bullet}l \cap R_{push}} \overline{M_t(p)}$$

preset logic | preset registers but push | preset push registers

Figure 4.12: Logic nodes.

The *marking state* M of static register nodes $r \in R_{stc}$, see Figure 4.13, is determined using the *marking* $M_{\uparrow}$ and *reset* $M_{\downarrow}$ functions. A static register can be marked if its preset logic is evaluated, R-preset registers are marked (with push holding True token), R-postset registers are unmarked (pop are allowed to hold False token though). Symmetrically, it can be unmarked ($M_{\downarrow}$) if its preset logic is reset, R-preset registers are unmarked (push are allowed to hold False token though), and R-postset registers are marked (with pop holding True token). In other words, a register can store a data value when all its inputs are available and its R-postset registers contain *spacers* (i.e. no tokens and False tokens in pop); and can be reset when the data stored has been propagated to its R-postset. This semantics models the *4-phase handshake protocol* as described in [25].

$$M(r) = M_{\uparrow}(r) \vee \overline{M_{\downarrow}(r)} \wedge M(r), \text{ with } r \in R_{stc}$$

$$M_{\uparrow}(r) = \bigwedge_{l \in {}^{\bullet}r \cap L} C(l) \wedge \bigwedge_{q \in {}^{\star}r \cap (R \setminus R_{push})} M(q) \wedge \bigwedge_{p \in {}^{\star}r \cap R_{push}} M_t(p) \wedge \bigwedge_{q \in r^{\star} \cap (R \setminus R_{pop})} \overline{M(q)} \wedge \bigwedge_{p \in r^{\star} \cap R_{pop}} \overline{M_t(p)}$$

$$M_{\downarrow}(r) = \bigwedge_{l \in {}^{\bullet}r \cap L} \overline{C(l)} \wedge \bigwedge_{q \in {}^{\star}r \cap (R \setminus R_{push})} \overline{M(q)} \wedge \bigwedge_{p \in {}^{\star}r \cap R_{push}} \overline{M_t(p)} \wedge \bigwedge_{q \in r^{\star} \cap (R \setminus R_{pop})} M(q) \wedge \bigwedge_{p \in r^{\star} \cap R_{pop}} M_t(p)$$

preset logic | R-preset registers but push | R-preset push registers | R-postset registers but pop | R-postset pop registers

Figure 4.13: Static register nodes.

In the equations in Figure 4.13, the function M_t determines if a push or pop register $p \in R_{push} \cup R_{pop}$ is marked with a `True` token (i.e. behaving as a static register), and the function M_f if it is marked with a `False` token. We also use these functions to determine

the marking state $M(p)$ of push and pop registers, see Figure 4.14. The marking of push and pop registers is defined as a non-deterministic choice between being marked with a `True` or `False` token. A register p can be marked with a `True` (`False`) token when it satisfies the marking function $M_{\uparrow}$ (see Figure 4.13), and when it is true- (false-) controlled according to the *PLAIN* Boolean function $M_t^{c.PLAIN}$ ($M_f^{c.PLAIN}$), see Figure 4.11. On the other hand, p can be reset if it is marked (M_t or M_f) and satisfies the reset function $M_{\downarrow}$.

$$
\begin{aligned}
M(p) &= M_t(p) \vee M_f(p), \quad \text{with } p \in R_{push} \cup R_{pop} \\
M_t(p) &= M_{\uparrow}(p) \ \wedge \ M_t^{c.PLAIN}(p) \ \vee \ \overline{M_{\downarrow}(p)} \ \wedge \ M_t(p) \\
M_f(p) &= M_{\uparrow}(p) \ \wedge \ M_f^{c.PLAIN}(p) \ \vee \ \overline{M_{\downarrow}(p)} \ \wedge \ M_f(p)
\end{aligned}
$$

Figure 4.14: Push and pop registers.

The behavioural semantics of control registers $c \in R_{ctrl}$ (see Figure 4.15) is analogous to the one of push and pop registers p , with the only difference that control registers can implement also the AND or OR Boolean functions, see Figure 4.11.

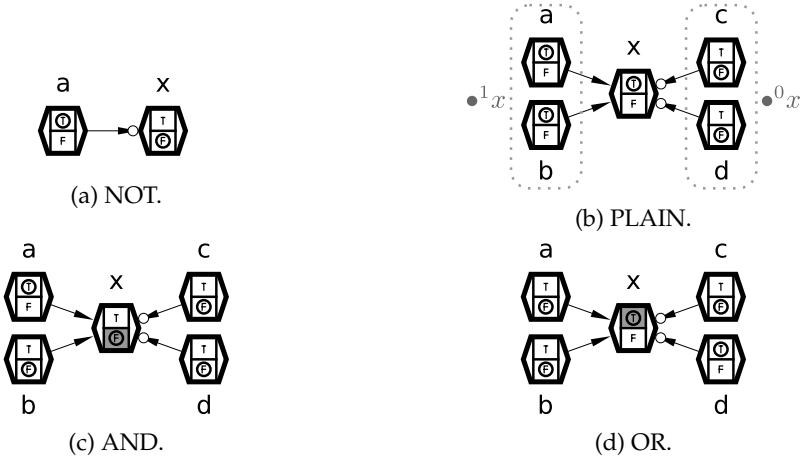
$$
\begin{aligned}
M(c) &= M_t(c) \vee M_f(c), \quad \text{with } c \in R_{ctrl} \\
M_t(c) &= M_{\uparrow}(c) \ \wedge \ M_t^{c}(c) \ \vee \ \overline{M_{\downarrow}(c)} \ \wedge M_t(c) \\
M_f(c) &= M_{\uparrow}(c) \ \wedge \ M_f^{c}(c) \ \vee \ \overline{M_{\downarrow}(c)} \ \wedge M_f(c)
\end{aligned}
$$

Figure 4.15: Control registers.

The DFS can implement a Boolean algebra using `True/False` tokens and NOT/-PLAIN/AND/OR Boolean functions. We conclude the formal description of the DFS formalism by showing the graphical representation of the implemented Boolean functions on control registers, see Figure 4.16.

The *NOT* Boolean function is implemented by means of an inverting arc (—●○), which propagates the inverted polarity of a token. As an example, Figure 4.16a shows two control registers connected by an inverting arc: the `True` token in a is propagated to x as a `False` token.

In regards to the *PLAIN* function, a node is said to be true-controlled if all its control registers in the regular arc preset hold a `True` token, while those in inverting arc preset hold a `False` token. Symmetrically, it is said to be false-controlled if all control register in the regular arc preset hold a `False` token, and those in inverting arc preset hold a `True`



(a) NOT. (b) PLAIN. (c) AND. (d) OR.

Figure 4.16: Examples of the four Boolean functions implemented in the DFS.

token. In case of a *mismatch* (i.e. a node sees both True and False tokens in its preset), the node is disabled, which may lead to a deadlock. The reachability of such problematic states needs to be formally verified (which has been automated in our design flow). In the example in Figure 4.16b, x implements the PLAIN function and it is true-controlled because the control registers in the regular arc preset ($\bullet^1 x = \{a, b\}$) store `True` tokens, and the control registers in the inverting arc preset ($\bullet^0 x = \{c, d\}$) store `False` tokens.

On the other hand, a control register that implements the AND function is denoted graphically with the `False` polarity shadowed (see Figure 4.16c), while one that implements the OR function is drawn with the `True` polarity shadowed (see Figure 4.16d). The control register x in Fig. 4.16c is false-controlled as the register b stores a `False` token. On the other hand, the control register x in Fig. 4.16d is true-controlled as the c register stores a `False` token.

4.2.2.1 Simulation of the motivating example

In this section, we show and describe a graphical simulation of the DFS model of the reconfigurable noise filter, introduced as second motivating example in Section 4.2.1. The purpose of this section is to apply the just described behavioural semantics of the formal model, and simplify the reader's understanding.

Figure 4.17 shows a possible simulation trace of the DFS model of the reconfigurable noise filter. The marking and evaluating functions are highlighted in green, and the reset

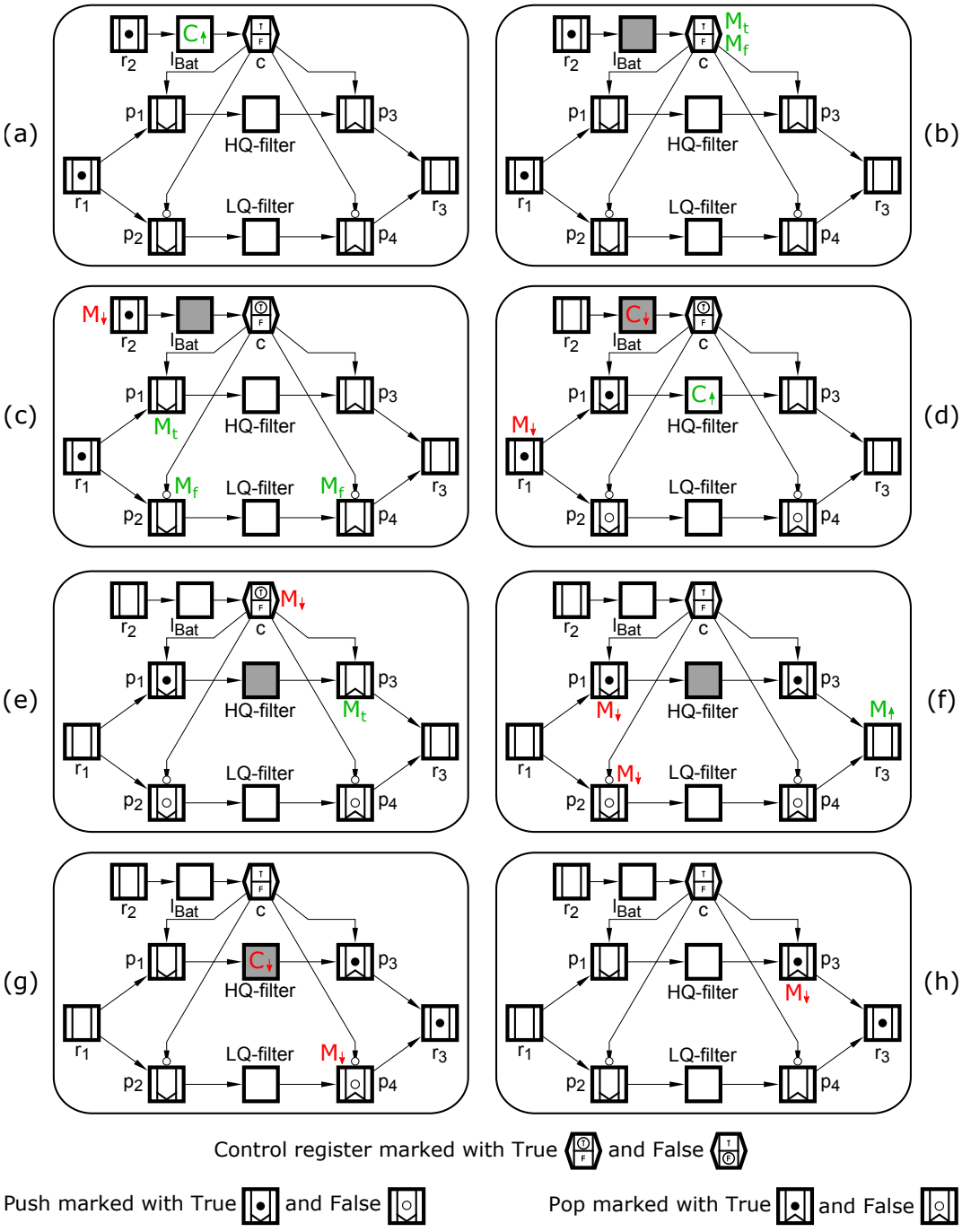


Figure 4.17: A possible simulation trace of the DFS model of the reconfigurable noise filter.

functions are in red for an easier visualisation. The most relevant steps of the simulation are described in the next paragraphs.

(a-b) The data audio stored in the register r_1 is on hold to be processed by one of the two filters. The control register c is waiting to be marked either with a `True` or `False` token, depending on the battery level provided by the combination logic l_{Bat}. In the simulation, we assume that the battery level is high enough (c marked with a `True`) to process the incoming data with the high-quality filter.

(c-e) The control register c is marked with a `True` token, which selects the high-quality filter and bypasses the low-quality filter. I.e. the push and pop registers p_1 and p_3 are true-controlled as they are connected to their control register preset c with regular arcs. Consequently, these nodes behave as static registers and propagate the incoming token from r_1 to the HQ filter, and subsequently to the output r_3. Concurrently, the push and pop registers p_2 and p_4 are false-controlled due to their incoming inverting arcs from c. Thus, these registers are forced to bypass the `LQ-filter` by destroying and producing a `False` token.

(f-h) The static register r_3 can store the processed audio token, as its preset pop registers are both marked. The high-quality filter can be reset in order to be ready to process a new incoming token. Note that the low-quality filter has never been evaluated, and thus does not need to be reset.

The simulation shows that the DFS model can capture the behaviour of the reconfigurable noise audio filter, where the two internal scenarios (high- and low-energy filters) are selectively executed.

4.2.3 Composition of scenarios

The presented DFS formalism enables engineers to compose scenarios in the form of SDFSs. Here, we show that this process can be easily generalised relying on the notion of inefficient scenario composition described in the introductory chapter.

As an example, Figure 4.18 shows the DFS structure for composing four different behavioural scenarios. The *Selection interface* nodes select which of the four scenarios

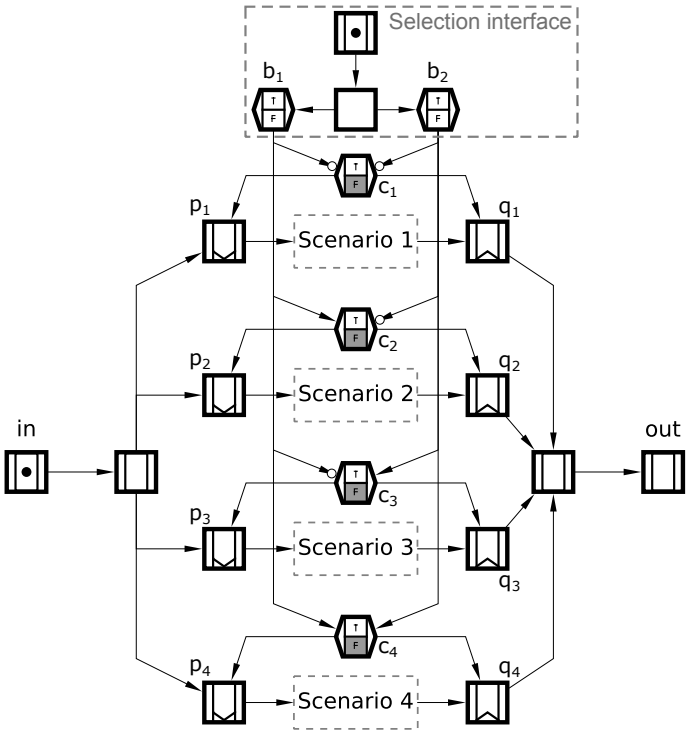


Figure 4.18: DFS generic structure for the execution of 4 scenarios.

have to be executed, relying on the encoding $\{(s_1, 00), (s_2, 10), (s_3, 01), (s_4, 11)\}$ on two bits modelled by the control registers $\{b_1, b_2\}$. At the input and output of every i_{th} scenario there are a push (p_i) and a pop (q_i) register, which are controlled by a control(c_i) register that implements the AND Boolean function. The regular and inverting arcs going from the registers $\{b_1, b_2\}$ to $c_{\{1,\cdots,4\}}$ implement the encoding.

With the above structure, an incoming token can go from the input register `in` to the output `out` passing by the scenario selected by the selection interface. As an example, in the case of a `False` and `True` tokens in b_1 and b_2 (code 01), *Scenario 3* is executed and all the others are bypassed.

This structure can be employed in several applications. For example, rather than having only two types of filter (as in the example in Figure 4.10), one might want to have a finer granularity of Energy-Quality trade-offs and implement more filters. Also, this structure reflects the behaviour of dataflow processors, which are requested to process incoming data items via different instructions (scenarios).

As we discussed in Section 1.2, however, the above represents an *inefficient* approach

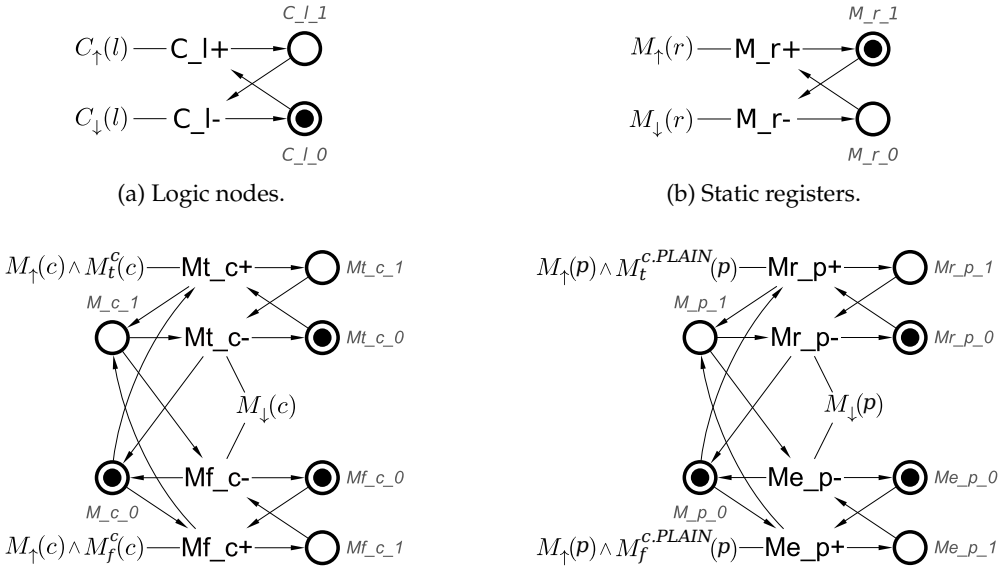
to scenario composition. Reusing common functionalities among scenarios is desirable for optimising many design criteria (e.g. area, power). An *efficient* approach is possible with the DFS methodology, but is currently not automated. In Chapter 5, we present a methodology for designing and implementing asynchronous reconfigurable pipelines that is based on the idea of efficient scenario composition.

4.2.4 Execution semantics expressed with Petri nets

The execution semantics of the novel Dataflow Structures, described in Section 4.2.2, can be expressed via different general purpose formalisms, e.g. finite state machines, Event-B [74], process algebra [75], Petri nets (reviewed in Section 3.3). In this section, we show how to express the described DFS behavioural semantics using Petri nets, for enabling designers to reuse the wide and existing tool-sets for PN that are available in the WORKCRAFT [34] design environment, where DFS has been also integrated. All the features provided in WORKCRAFT are summarised in Section 4.2.5. In this section, we describe the methodology for translating the DFS models into Petri nets.

The state of static nodes (see Figure 4.9) can be expressed by one Boolean variable. The evaluation state of a logic node $l \in L$ can be represented by `C_l`, i.e. a node is evaluated if `C_l`$= 1$, a node is reset if `C_l`$= 0$. Similarly, the marking state of a static register $r \in R_{stc}$ can be also expressed by `M_r`, i.e. a node is marked if `M_r`$= 1$, and reset if `M_r`$= 0$. These concepts can be represented by the Petri nets in Figures 4.19a and 4.19b, where the place `C_l_1` (`M_r_1`) represents the state in which a logic (stati register) node is evaluated (marked), while the the place `C_l_0` (`M_r_0`) represents the state in which a logic (static register) node is reset (reset). The transitions `C_l+` and `C_l-` allow a logic node to change its state, they can be fired depending on the conditions $C_{\uparrow}$ and $C_{\downarrow}$ (interconnected with read arcs), respectively, as described in Equations 4.12. The transitions `M_r+` and `M_r-` allow a static register node to change its state, they can be fired depending on the conditions $M_{\uparrow}$ and $M_{\downarrow}$, respectively, as described in Equations 4.13.

On the other hand, dynamic nodes (see Figures 4.19c and 4.19d) can be marked either with `True` or `False` tokens. For such nodes, at least two Boolean variables are needed to identify the three states of a dynamic register node: (1) node unmarked, (2) node marked with a `True` token, (3) node marked with a `False` token. For a control register c,



(a) Logic nodes.

(b) Static registers.

(c) Control registers.

(d) Push and pop registers.

Figure 4.19: Petri Net models of the Dataflow Structures nodes.

for example:

(1) *c* unmarked - The places `M_c_0`, `Mt_c_0`, and `Mf_c_0` contain a token. In this state, the transitions `Mt_c+` and `Mf_c+` are enabled by the conditions $M_\uparrow \wedge M^c_{t/f}$ (via read arcs). Only one of these transitions can fire, i.e. *c* can be marked either with a `True` or a `False` token.

(2) *c* marked with a `True` token - The places `M_c_1`, `Mt_c_1` and `Mf_c_0` contain a token. In this state, the transition `Mt_c-` is enabled, and can fire if the condition $M_\downarrow$ is satisfied, leading *c* to be unmarked.

(3) *c* marked with a `False` token - The places `M_c_1`, `Mf_c_1` and `Mt_c_0` contain a token. In this state, the transition `Mf_c-` is enabled, and can fire if the condition $M_\downarrow$ is satisfied, leading *c* to be unmarked.

Notice that the transitions `Mt_c+` and `Mf_c+` are both enabled if $\bullet c \cap R_{ctrl} = \emptyset$, i.e. a data token that reaches a dynamic node, which is neither false- nor true-controlled, is converted to a control token (`True` or `False`) via a non-deterministic choice.

The Petri net description of push an pop registers *p* is analogous to the PN description of control registers, with the difference that the former can only implement the PLAIN

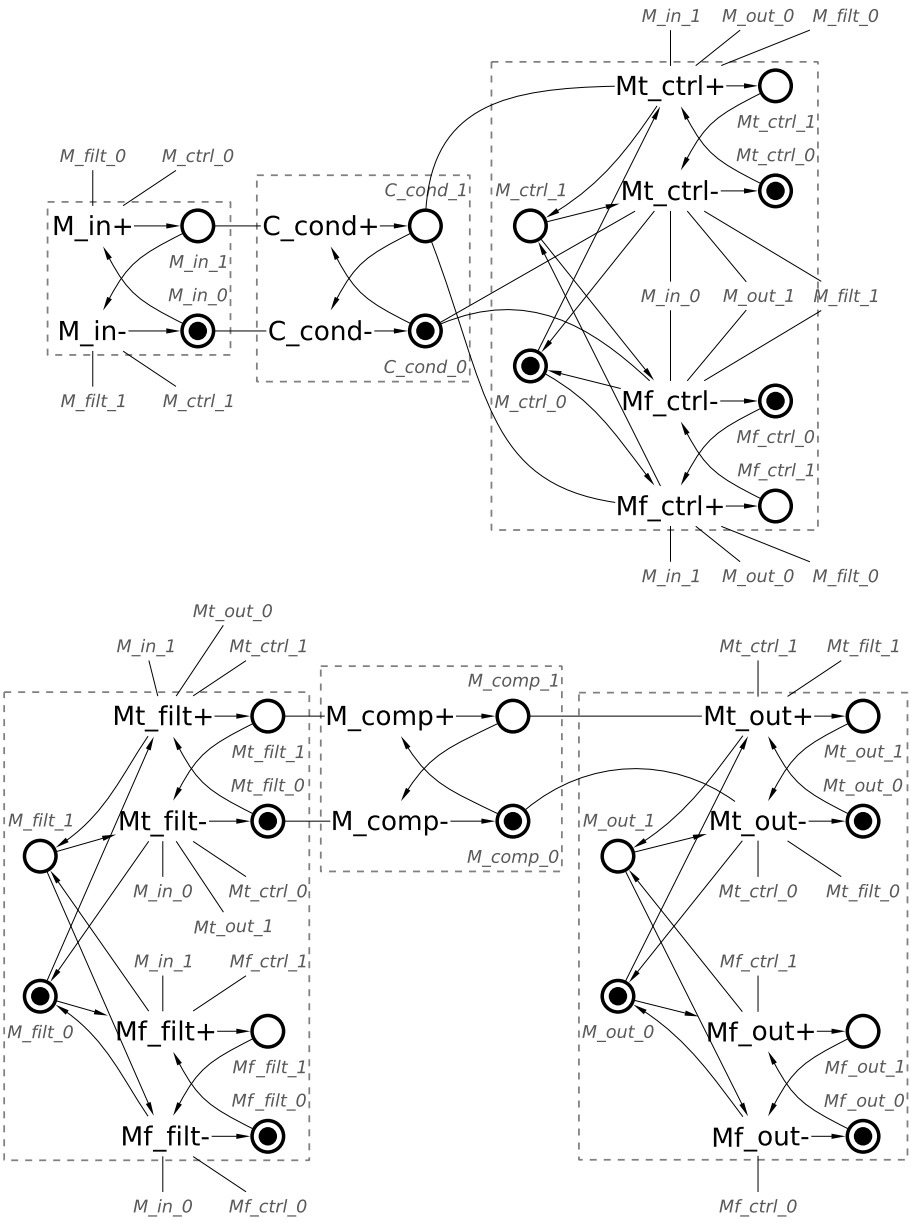


Figure 4.20: Petri net description of the DFS Filter in Figure 4.8c.

Boolean function. I.e. A register p can be marked with a `True` or `False` token if one of the following conditions is satisfied, respectively: $M_{\uparrow} \wedge M_t^{c.PLAIN}$, and $M_{\uparrow} \wedge M_f^{c.PLAIN}$

As an example of conversion from DFS to PN, Figure 4.20 shows the Petri net corresponding to the DFS model in Figure 4.8c. We refer the reader to [29] for a detailed description of this example.

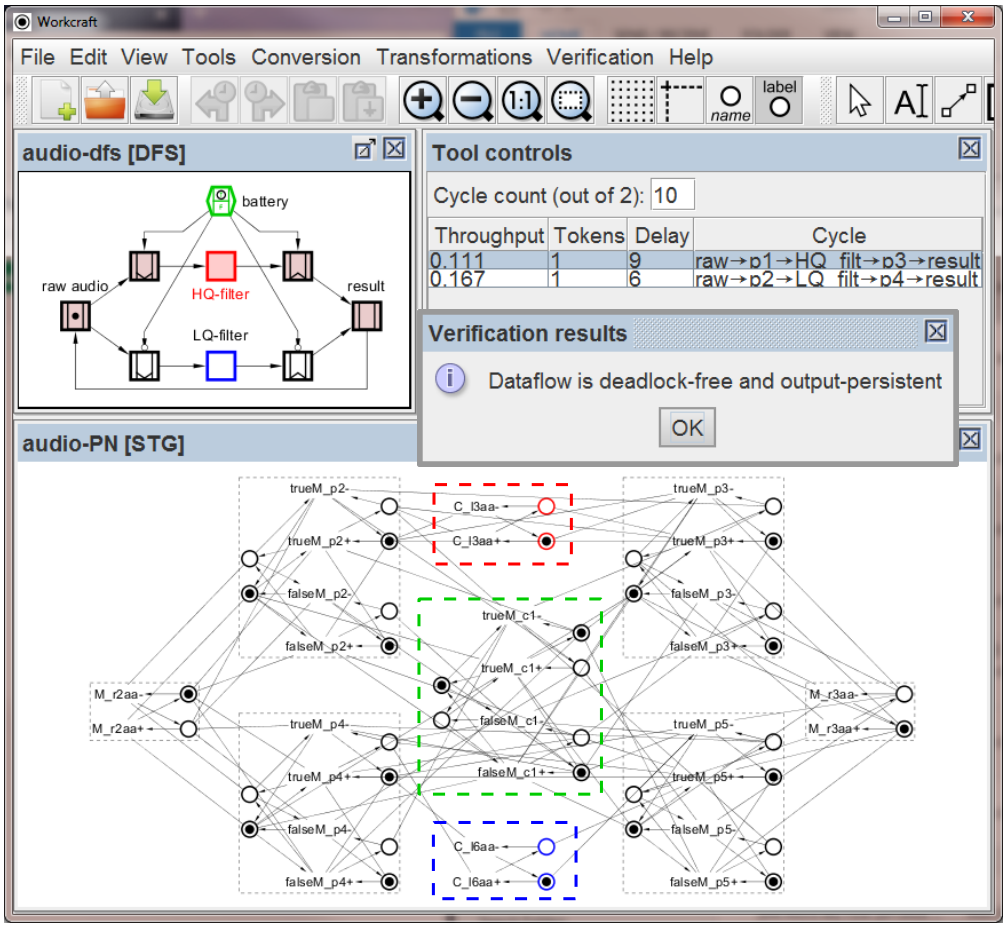


Figure 4.21: Screenshot of WORKCRAFT while handling the DFS digital camera model.

4.2.5 Design automation

The Dataflow Structures formalism has been implemented in the WORKCRAFT design environment [33,34]. The latter provides the following functions (see Figure 4.21):

Editing - Convenient editing of dataflow structures by means of the cross-platform graphical user interface.

Performance analysis - Functional delay values can be assigned to the Dataflow structure nodes. Such values are used to report the slowest paths of the circuits and the bottleneck nodes. This information can be used to improve the circuit performance. Figure 4.21 shows the performance analysis feature applied to the DFS model of the reconfigurable noise filter, where the slowest delay has been

assigned to the `HQ-filter` node, whose name is highlighted in red. In the sub-window "audio-dfs [DFS]", the nodes that compose the slowest path of the model are coloured in red: from the `raw audio` register, to the `result` via the *HQ-filter*. In the "Tool control" windows, all the possible paths of the models are reported and ranked by their throughput.

Encoding - Different scenarios, expressed with the presented Dataflow Structures, can be encoded relying on the Boolean functions implemented in the control registers, as we illustrated in Figure 4.18.

Hardware synthesis - The DFS model can be exported automatically into a Verilog description of the corresponding asynchronous circuit, where the DFS nodes are mapped to a library of pre-built components and are interconnected according to the arcs described in the model. The implementation of such components determines the characteristics of the final circuit. In Chapter 5.2, we show the implementation style chosen for the designed asynchronous accelerator. Users can also choose to convert DFSs into PNs, and synthesise a custom circuit using the existing tools for PN logic synthesis, e.g. Petrify [54].

Formal verification - The DFS models are mechanically converted to Petri nets to use the formal verification tools developed by the Petri net community. In the sub-window "audio-PN [STG]", the PN model of the reconfigurable audio filter is shown, the colours of the DFS nodes match the colours of the boxes surrounding the corresponding PN nodes. In WORKCRAFT, Petri nets can be processed by MPSAT [76] that detects deadlocks or functional hazards. Users can express the queries to run via MPSAT in Reach language [77]. In Figure 4.21, the "Verification results" pop-up window informs that the DFS is deadlock-free and output-persistent.

4.2.6 Related work

The Dataflow Structures is a *Model of Computation* (MoC), i.e. a set of laws that describes the interaction of the components of a system. Interested readers can find the description

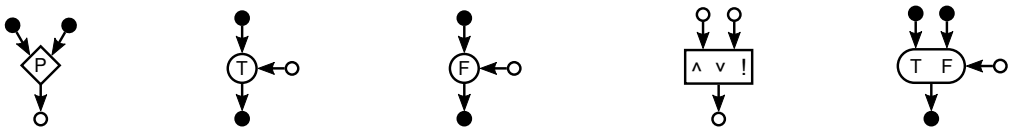
(a) Demultiplexer. (b) Multiplexer.

Figure 4.22: DFS models of a demultiplexer and multiplexer.

of a number of dataflow-based MoCs in [78]. In the next paragraphs, we relate the presented Dataflow Structures formalism to similar existing MoCs available in literature.

In [25], Sparsø and Furber introduced the SDFS formal model, and showed how to apply it to the description of asynchronous circuits at a high-level of abstraction. The model is characterised by logic and register nodes, described in Section 3.4, and by two further primitives meant to be passive to the handshake communication mechanism and necessary to describe hardware reconfigurability: the *demultiplexer* and *multiplexer*. These two nodes were not taken into consideration when the SDFS was formalised [26] for not overcomplicating its behavioural semantics. With the dynamic extension that we propose, demultiplexers and multiplexers can be described with the usage of registers, see Figure 4.22 (these primitives are used in our second motivating example in Section 4.2.1). This enables the semantics of the model to stay relatively simple as the new nodes, being an extended types of registers, have similar behavioural equations.

The idea of using Boolean-controlled nodes for describing conditional behaviours in data-flow graphs goes back to the 70s, when Dennis et al. described a dataflow MoC that enables the representation of static and conditional behaviours [79]. Similarly to what we elaborated, this model could handle both *data* and *control* types of tokens – data tokens are used to abstract data values, while control tokens (based on Boolean conditions) manage the topology, thereby the behaviour, of the model itself at runtime. Figure 4.23 shows the five nodes of this model as described in the original paper, • represents data links for the propagation of data tokens, and ◦ represents control links for the propagation of control tokens. The *decider* node takes two data tokens as input, tests an internal predicate P and produces a control token (either True or False). The *T-gate* and *F-gate* nodes behave as the shown push registers: they propagate the input data



(a) Decider. (b) T-gate. (c) F-gate. (d) Boolean operator. (e) Merge.

Figure 4.23: Nodes of the MoC conceived by Dennis.

token to the output only if the input control token is satisfied (True for the T-gate, and False for the F-gate), otherwise the input data token is consumed from the input link and destroyed. The *Boolean operator* node computes Boolean operations over multiple control tokens. Finally, the *merge* node behaves as a multiplexer, i.e. the input control token selects which of the input data tokens have to be propagated to the output.

(a) Switch. (b) Select.

Figure 4.24: Control-flow nodes the BDF MoC.

Another formalism based on this idea is the *Boolean-controlled Dataflow* (BDF) [80]. This model also handles both data and control types of tokens for managing the model topology and describing conditional behaviours. The BDF extends the Synchronous Dataflow (SDF) [81] formalism by introducing two primitives: *switch* and *select*, shown in Figure 4.24, which practically behave as demultiplexers and multiplexers enabling the description of conditional behaviours.

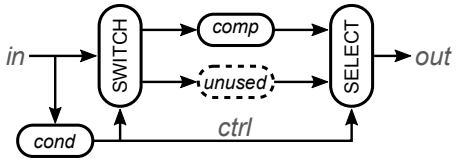


Figure 4.25: BDF description of the DFS model in Figure 4.10.

Unlike these MoCs, we argue that the nodes constituting the presented DFS formalism are *atomic*, i.e. decomposing any of them would not give any advantage in terms of representative capabilities. We already showed that the primitives representing demultiplexers, such as the above merge, switch and select, can be broken down into

DFS models made of registers, see Figure 4.22. This provide advantages in a number of reconfiguration scenarios where, as an example, only one part of a system as to be controlled (see our first motivating example in Figure 4.10). This latter case cannot be modelled with the solely usage of a demultiplexer and multiplexer without incurring in an unused part, see Figure 4.25.

It is also worth mentioning the *Boolean Parametric Data Flow* (BPDF) [82] formalism. This is different with respect to the previously described models, as the control flow is separated by the dataflow. Boolean conditions are paired to the model arcs *statically* (i.e. during the model description), and disable them if such conditions are not satisfied – similarly to CPOGs. BPDF is supported by automated verification features (e.g. liveness, boundedness), and efficient scheduling techniques to assign internal operations to sequential or multi-core systems. BPDF, unlike the described DFS, is also applied to scheduling of concurrent processes rather than to hardware design.

Push and pop registers, as shown in our motivating examples, provide data tokens to their intermediate nodes (e.g. see *comp* in Figure 4.8c) only when their execution is required, disconnecting them from the rest of the system otherwise. This idea takes the name of *operand isolation* and was proposed in [83]. In this paper, the authors propose two primitives – namely *receive* and *send*, formally described by the Three-Valued Logic [84] – that can act as the presented push and pop registers and isolate components of asynchronous circuits when not required for saving dynamic power. Our work, in comparison, employs this concept at a higher level of abstraction reusing formalisms that are well known to the asynchronous community. Also, the receive and send primitives are only described behaviourally in [83], whereas, in Section 5.2, we propose a standard-cell based implementation for the push and pop registers that can be employed out-of-the-box in asynchronous circuits functioning with the 4-phase handshake protocol.

4.2.7 Summary

The presented DFS formalism abstracts away the implementation details of digital circuits, providing an environment where engineers can focus on circuit functionality. However, the described behavioural semantics can only be applied to circuits function-

ing with the *4-phase handshake protocol* [25], i.e. a spacer interleaves between two valid data values. The description of an additional semantics for circuits that make use of the *2-phase handshake protocol* [25] is left for future research, see Section 6.2.

In this section, we described the new Dataflow Structures model, which extends the Static Dataflow Structures with an additional set of nodes that we use to capture dynamic behaviours of asynchronous circuits. Afterwards, we showed a possible simulation trace of one of the used motivating examples. Finally, we presented the developed design automation for converting DFS models into Petri nets and enjoying the existing back-end tools for automated verification and hardware synthesis. In the *Case studies* Chapter, we will show how to derive functional digital circuits from DFS models by mapping DFS nodes to a gate-level library of combinational and sequential digital components. Our final goal will be to design an asynchronous reconfigurable accelerator for the ordinal pattern encoding.

4.3 Decomposition of system specifications

So far, we presented two novel approaches to behavioural composition of scenarios. In the case of the CPOG, we described an algorithm for the efficient composition of partial orders that supports real-life constraints. In the case of the DFS formalism, we extended SDFSs with a set of nodes that enables one to compose static asynchronous scenarios into dynamically reconfigurable asynchronous systems. In this section, we consider the following problem:

Problem: *Understanding systems that are made of concurrent processes is hard. In many applications, it is convenient to untangle such systems into separate scenarios. Is it possible to derive an automated procedure for decomposing complex concurrent systems into good-looking and clear scenarios?*

To answer this question, we present the *Process Windows* formalism, which enables to decompose complex systems automatically via a custom set of rules. This section is divided as follows. In Section 4.3.1, we describe the main idea behind this work by expanding the description of our motivating example (see Section 2.3). In Section 4.3.2, we formalise the definition of Process Windows. In Sections 4.3.3 and 4.3.4, we describe

the algorithms for decomposing a system into collections of windows, and for deriving the conditions needed to orchestrate these windows, respectively. Applications and examples of Process Windows are discussed in Section 4.3.5. Finally, we discuss the related work in the field in Section 4.3.6.

This section reuses definitions, figures and tables of [31]. My main contribution in this work is in Section 4.3.4: the automated generation of the Boolean equations for controlling the windows, and the development of the *Shutters* tool [32].

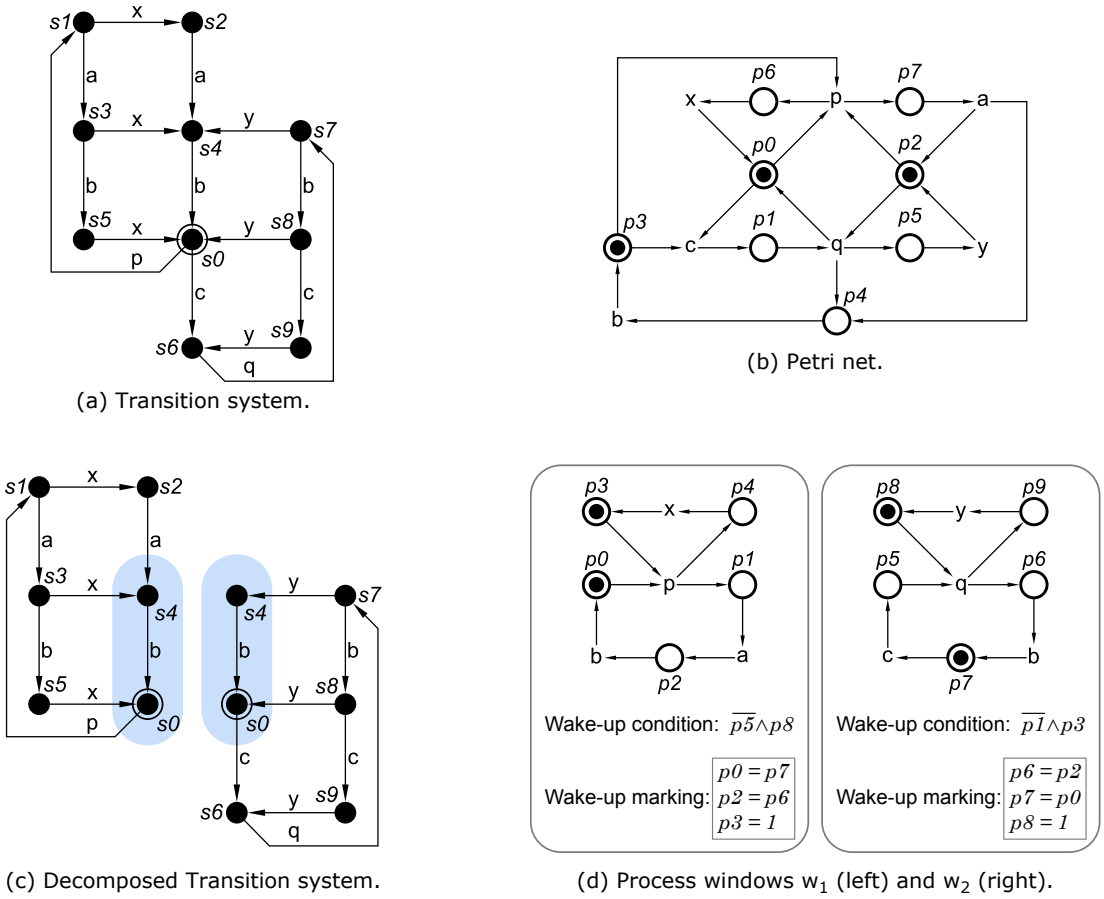
4.3.1 The idea with an example

As discussed in the *Motivation* chapter (see Section 2.3), it is difficult to understand complex system specifications constituted of many concurrent behaviours. Decomposing such specifications into simpler scenarios would facilitate their understanding.

Our motivating example is reported in Figure 4.26 and summarised below. A transition system (in Fig. 4.26a) and its corresponding Petri net (in Fig. 4.26b) are decomposed into two simpler transition systems (in Fig. 4.26c) and their corresponding Process windows w_1 and w_2 (in Fig. 4.26d). The latter representation includes two Petri nets, whose behaviour is coordinated by the shown *wake-up conditions* and *wake-up marking* equations.

The idea behind the decomposition process is that every window has to model a part of the Transition System (TS), with the condition that the union of all the extracted windows would recover the complete system functionality. In the motivating example, w_1 models the subset of states $\{s0, s1, s2, s3, s4, s5\}$ (left-hand TS in Fig. 4.26c), while w_2 models the subset of states $\{s0, s4, s6, s7, s8, s9\}$ (right-hand TS in Fig. 4.26c). The union of the two windows (i.e $w_1 \cup w_2$) recovers the initial TS in Fig. 4.26a $\{s0, \cdots, s9\}$, while their intersection (i.e. $w_1 \cap w_2$) returns the *overlapping states* $\{s0, s4\}$, which are highlighted in blue in Fig. 4.26c. Such states are important as they 'bridge' the two windows, i.e. they enable the system to move from one window to another.

To clarify the above concepts, Figure 4.27 shows a graphical simulation of the motivating example. Each window is *transparent* when it models the current state of the TS, and it is *opaque* (grey) otherwise. An opaque window is not needed for modelling the TS current state and can therefore forget its current marking. As an example, the



(a) Transition system.

(b) Petri net.

(c) Decomposed Transition system.

(d) Process windows w_1 (left) and w_2 (right).

Figure 4.26: The motivating example.

two windows are both transparent when the TS is in the initial state $s0$ (i.e. both the two windows model this state). From $s0$, the system can either move to $s1$ with the transition p ($s0 \xrightarrow{p} s1$) or to $s6$ with the transition c ($s0 \xrightarrow{c} s6$). In the former case, w_1 models the state $s1$ and w_2 becomes opaque. In the latter case, w_2 covers the state $s6$ and w_1 becomes opaque. Every state of the TS is modelled by at least a window. The TS can move within a window indefinitely (e.g. $s0 \xrightarrow{p} s1 \xrightarrow{x} s2 \xrightarrow{a} s4 \xrightarrow{b} s0$), but can only move to a different window through an overlapping state such as $s0$ (e.g. $s5 \xrightarrow{x} \underline{s0} \xrightarrow{c} s6$).

4.3.1.1 Wake-up markings

Table 4.2 shows the markings of the Petri nets included in Process Windows in Fig. 4.26d for representing every state of the TS in Fig. 4.26a. Each state of the system is covered

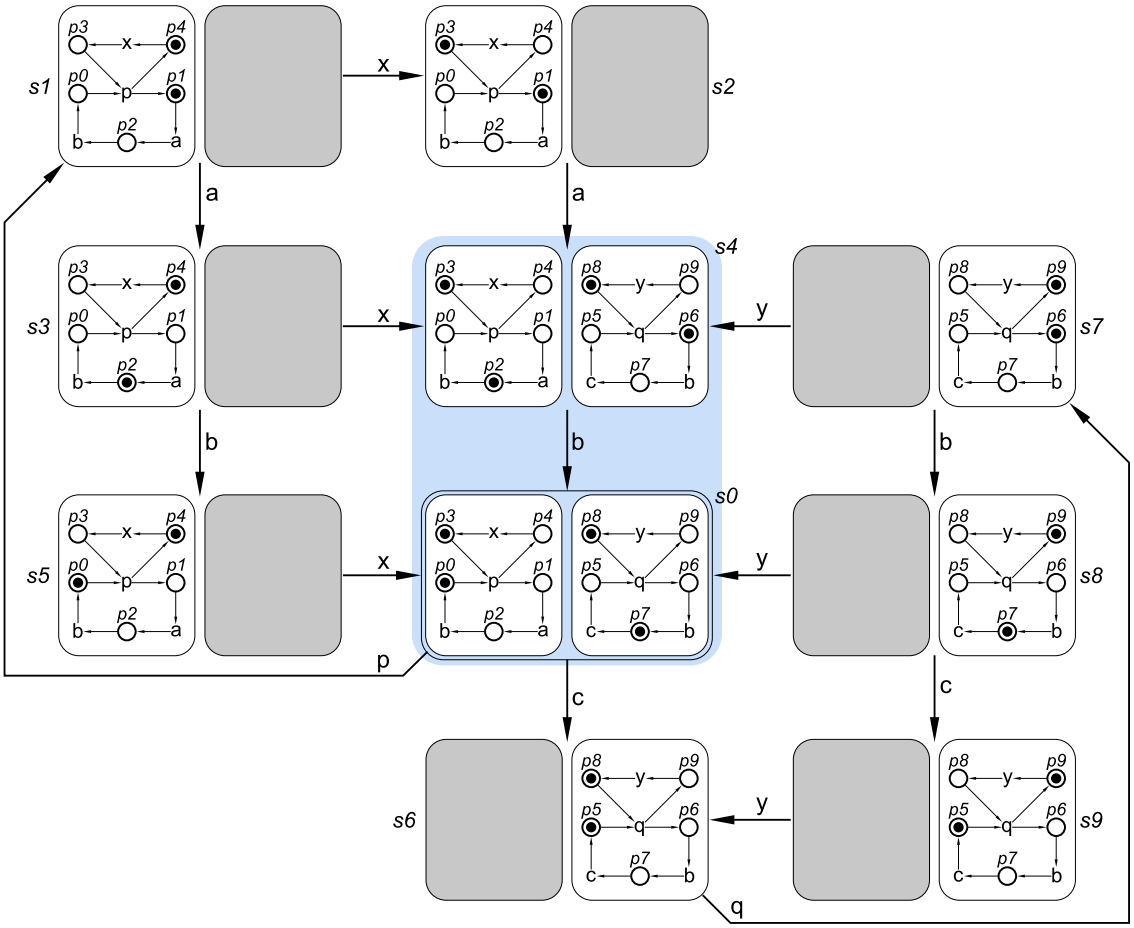


Figure 4.27: Simulation of the motivating example.

by at least one marking. When a window becomes opaque, the marking is forgotten ($-$) as the window does not need to store the state of the TS. When a window *wakes up* (i.e it becomes transparent), it recovers the TS current state from a previously active window.

Window w_1, for instance, may become transparent either in the state $s0$ or $s4$, see Fig. 4.27. When this happens, w_1 observes the state of the remaining active windows (w_2 in this case) to figure out its initial marking.

- w_1 is initialised with $\{p0, p3\}$ when it wakes up in $s0$ (the marking of w_2 is $\{p7, p8\}$).
- w_1 is initialised with $\{p2, p3\}$ when it wakes up in $s4$ (the marking of w_2 is $\{p6, p8\}$).

Via Boolean minimisation, we can derive that: (1) the place $p3$ needs to be always marked when w_1 wakes up **(i.e. $p3 = 1$)**, (2) the place $p0$ needs to be marked only when w_1 wakes

Table 4.2: States and net markings in Fig. 4.27.

State	Marking of w_1	Marking of w_2
$s0$	$\{p0, p3\}$	$\{p7, p8\}$
$s1$	$\{p1, p4\}$	-
$s2$	$\{p1, p3\}$	-
$s3$	$\{p2, p4\}$	-
$s4$	$\{p2, p3\}$	$\{p6, p8\}$
$s5$	$\{p0, p4\}$	-
$s6$	-	$\{p5, p8\}$
$s7$	-	$\{p6, p9\}$
$s8$	-	$\{p7, p9\}$
$s9$	-	$\{p5, p9\}$

up in $s0$ **(i.e. $p0 = p7$)**, and (3) the place $p2$ needs to be marked only when w_1 wakes up in $s4$ **(i.e. $p2 = p6$)**.

Symmetrically, window w_2 looks at the marking of w_1 when it wakes up:

- w_2 is initialised with $\{p7, p8\}$ when it wakes up in $s0$ (the marking of w_1 is $\{p0, p3\}$).
- w_2 is initialised with $\{p6, p8\}$ when it wakes up in $s4$ (the marking of w_1 is $\{p2, p3\}$).

Via Boolean minimisation, we can derive that: (1) the place $p8$ needs to be always marked when w_2 wakes up **(i.e. $p8 = 1$)**, (2) the place $p7$ needs to be marked only when w_2 wakes up in $s0$ **(i.e. $p7 = p0$)**, and (3) the place $p6$ needs to be marked only when w_2 wakes up in $s4$ **(i.e. $p6 = p2$)**. See these wake-up markings in Fig. 4.26d at the bottom of w_2.

The above conditions (highlighted in bold in the above) are named *wake-up markings*, and are shown at the bottom of the two windows in Fig. 4.26d.

4.3.1.2 Wake-up conditions

While wake-up markings initialise a window to model the current state of a TS, the *wake-up conditions* evaluate to `True` in all the states where a window have to wake up.

Window w_1 observes the marking of the other transparent window w_2 to figure out when to wake up. w_1 wakes up either when w_2 moves to $s0$ (marking $\{p7, p8\}$), or to $s4$ (marking $\{p6, p8\}$), i.e.$(p6 \wedge p8) \vee (p6 \wedge p8)$. This can be simplified as $\overline{\boldsymbol{p5}} \wedge \boldsymbol{p8}$ via Boolean minimisation.

On the other hand, window w_2 observes the marking of the other transparent window w_1 to figure out when to wake up. w_2 wakes up either when w_1 moves to

$s0$ (marking $\{p0, p3\}$), or to $s4$ (marking $\{p2, p3\}$), i.e.$(p0 \wedge p3) \vee (p2 \wedge p3)$. This can be simplified as $\overline{\boldsymbol{p1}} \wedge \boldsymbol{p3}$ via Boolean minimisation.

These conditions are also shown at the bottom of each window in Figure 4.26d in our motivating example. In the next sections, we describe how to derive automatically Process Windows, wake-up markings and wake-up conditions.

4.3.2 The Process Windows model

In this section, we formalise the description of Process Windows formalism, providing its definition (see Definition 4.17), the rules for decomposing an LTS (see Definition 4.18), the notation that we use for representing transitions between LTS states, and finally the properties that one might want to enforce for having a set of "easy to understand" windows. The provided definitions are based on the description of an LTS as reviewed in the *Background* chapter of this dissertation, see Section 3.5.

Definition 4.17. ***(Process window) [31] -*** *Given an LTS $A = (S, T, L, s_0)$, a process window (or simply window) of A is another LTS $w = (S_w\{\perp_w\}, T_w, L_w, s_{0_w})$ such that:*

- *$S_w \subseteq S$, $L_w \subseteq L$ and $T_w \subseteq T$. L_w (S_w) strictly contains the labels (states) paired to T_w.*
- *$\perp_w \notin S$ represents the inactive state.*
- *If $s_0 \in S_w$, then $s_{0_w} = s_0$, otherwise $s_{0_w} = \perp_w$.*

Intuitively, a window is a subset of states and transitions of an LTS. The rules for decomposing an LTS into a set of windows are reported below:

Definition 4.18. ***(Window Decomposition) [31] -*** *Given an LTS $A = (S, T, L, s_0)$, a Window Decomposition (WD) of A is a set of LTS windows $\{w_1, \cdots, w_n\}$, with $w_i = (S_i \cup \{\perp_i\}, T_i, L_i, s_{0_i})$, such that:*

$$S = \bigcup_i S_i, \qquad T = \bigcup_i T_i, \qquad L = \bigcup_i L_i$$

Additionally, the following conditions hold for every w_i:

- *All $\perp_i$ are different.*
- *The underlying graph induced by T_i is connected.*
- *$T_i \not\subseteq T_j$ for any $i \neq j$.*

Intuitively, every state and transition of an LTS must be covered in a WD, and there should not be any *redundant windows*, i.e. a window which is equal or contained by another window.

A window decomposition $W = \{w_1, \cdots, w_n\}$ of an LTS $A = (S, T, L, s_0)$ is a set of windows whose state (i.e. window marking) evolves depending on the transitions that are triggered in the associated LTS. The state of a window decomposition W is represented by a set of states $\vec{s} = (s_1, \cdots, s_n)$, where s_i is the state of the window w_i. "*There is a one-to-one correspondence between the states of W and the states of the associated LTS. For every state $s_x \in S$ of an LTS, the associated state in W will be $\vec{s}_x = (s_1, \cdots, s_n)$ such that for every w_i: $s_i = s_x$ if $s_x \in S_i$, and $s_i = \perp_i$ if $s_x \notin S_i$*" [31].

As an example, the WD in Figure 4.26d, representing the transition system in Figure 4.26a, may evolve as follows starting from the initial state $\vec{s}_0 = (s_0, s_0)$ (see also simulation in Figure 4.27):

$$(s_0, s_0) \xrightarrow{p} (s_1, \perp_2) \xrightarrow{a} (s_3, \perp_2) \xrightarrow{x} (s_4, s_4) \xrightarrow{b} (s_0, s_0) \xrightarrow{c} (\perp_1, s_6) \cdots$$

Whenever a transition of the LTS is triggered, the WD moves to a different state where each window is either modelling the LTS state (s_i) or sleeping ($\perp_i$). When the transition p is triggered, the WD moves to $(s_1, \perp_2)$: w_1 is in the state s_1 and w_2 is sleeping. The window w_2 wakes up when the transition x is triggered ($(s_3, \perp_2) \xrightarrow{x} (s_4, s_4)$), and its state is initialised to s_4. A WD can model its associated LTS indefinitely.

4.3.2.1 Structural properties

When a complex LTS has to be decomposed into a set of simpler windows, one might want to derive specific classes of Petri nets that guarantee a graphically simpler scenario specification [85], such as Marked Graphs [86] choice-free PNs [87], or free choice PNs [88]. Following, we report three properties elaborated in [87] and [89], which we use in our automated approach for extracting "easy-to-understand" windows.

Definition 4.19. ***(Forward and backward persistence) -*** *Let t_1 and t_2 be two transitions $\in T$ of a given LTS. They are said to be forward persistent if the following condition holds:*

$$\forall s_0 \in ES(t_1) \cap ES(t_2), \quad s.t. \quad s_0 \xrightarrow{t_1} s_1 \wedge s_0 \xrightarrow{t_2} s_2 : \quad s_1 \in ES(t_2) \wedge s_2 \in ES(t_1)$$

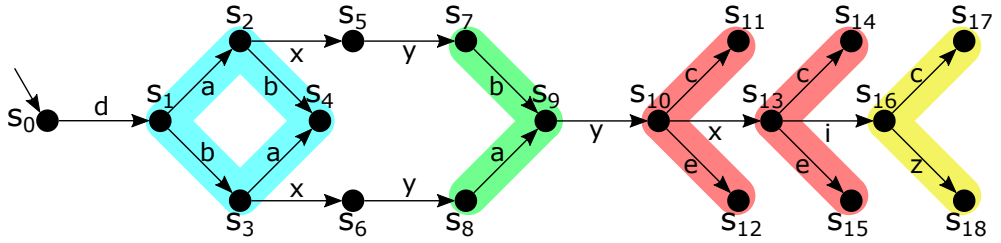


Figure 4.28: To clarify Definitions 4.19 and 4.20, we show an LTS and describe its properties. Transitions a and b are forward persistent (see *blue shadow*), but are not backward persistent (see *green shadow*). The transitions c and e are not forward persistent, but are in forward free choice (see *red shadow*). Lastly, the transitions c and z are neither forward persistent, nor in forward free choice (see *yellow shadow*).

Symmetrically, the transitions are said to be backward persistent if the following condition holds:

$$\forall s_0 \in BES(t_1) \cap BES(t_2), \quad s.t. \quad s_1 \xrightarrow{t_1} s_0 \wedge s_2 \xrightarrow{t_2} s_0 : \quad s_1 \in BES(t_2) \wedge s_2 \in BES(t_1)$$

If two transitions are forward persistent, the execution of any of the two does not disable the execution of the other one, which will be executed in the next iterations. An LTS is said to be forward (backward) persistent, if every pair of transitions is forward (backward) persistent.

Definition 4.20. ***(Free choiceness)*** **-** *If two transitions t_1 and t_2 of an LTS are not forward persistent, they are said to be in forward free choice if they are enabled in the same states, i.e. $ES(t_1) = ES(t_2)$. Symmetrically, if the two transitions are not backward persistent, they are said to be in backward free choice if their execution lead to the same states, i.e. $BES(t_1) = BES(t_2)$.*

As an example, Figure 4.28 shows an LTS where the properties of the above definitions are highlighted and described in the caption.

4.3.3 Extracting windows from system specifications

This section describes the algorithm for producing a window decomposition from a given LTS. It was inspired by a technique used to extract LTS slices in the area of process mining [85]. It is based on a SAT-formulation of the problem, which is solved by Pseudo-Boolean Optimisation approach [90].

The SAT-formulation that we use is based on finding $|T|$ Boolean variables, one for each transition of a given LTS. Each window is identified by a subset of transitions $t \in T$.

The term $W(T)$ identifies all the SAT properties $P_i(T)$ that constitute the SAT formula, which is used for deriving the window decomposition, see Equation 4.2.

$$W(T) = P_1(T) \wedge P_2(T) \wedge \cdots \wedge P_n(T) \tag{4.2}$$

The next sections describe the properties that can be enforced by inserting Boolean constraints in the SAT formula.

4.3.3.1 Property 1: Forward and Backward Persistence

This property relies on Definition 4.19. It is important for deriving simple windows, where there are no transitions that disable other transitions. Below, we report the Boolean formulation of the property from [31].

> *"Let s, s_1, $s_2 \in S$ and a, $b \in L$, with $a \neq b$, such that $t_1 = (s, a, s_1)$ and $t_2 = (s, b, s_2)$. Let $T_{s_2,a} = \{t_i = (s_2, a, s_i) | s_i \in S\}$ be the set of transitions enabled in s_2 with transition a. Then the following constraints is added to the formula to guarantee the persistence of a:*
>
> $$(t_1 \wedge t_2) \Longrightarrow (t_{i_1} \vee \cdots \vee t_{i_k})$$
>
> *where $t_{i_1}, \cdots, t_{i_k}$ are the elements of $T_{s_2,a}$. Notice that, by symmetry, this constraint will also be applied for b's persistence. It also works for non-deterministic LTSs in which $|T_{s_2,a}| > 1$. In case $T_{s_2,a} = \emptyset$, the constraint is reduced to: $\overline{t_1} \vee \overline{t_2}$."*

Backward persistency-related constraint has a dual formulation that corresponds to the forward persistency constraint when the direction of all transitions is reversed.

A forward and backward persistency constraint has to be inserted in $W(T)$ for every pair of transitions that are enabled in any state of the LTS.

4.3.3.2 Property 2: Determinism

To guarantee windows to be deterministic, one needs to enforce the transitions of all non-deterministic states to move the state of an LTS to a state contained by a different window. This can be achieved by imposing every window to contain at most one transition of non-deterministic states.

4.3.3.3 Property 3: Connectedness

To guarantee the connectedness of every window, one needs to enforce every state to have at least one incoming transition (t_{in_i}), and at least one outgoing transition (t_{out_i}). This ensures that neither *source* nor *deadlock* states are created in a window. We report below the formal constraint from [31]:

> *"Formally, for any state $s \in S$, we define $T_{in}(s) = \{t_{in_1}, \cdots, t_{in_m}\}$ and $T_{out}(s) = \{t_{out_1}, \cdots, t_{out_m}\}$ as the set of incoming and outgoing transitions of s, respectively. For any state in which $T_{in}(s) \neq \emptyset$ and $T_{out}(s) \neq \emptyset$, the following constraint is added:*
>
> $$(t_{in_1} \vee \cdots \vee t_{in_m}) \iff (t_{out_1} \vee \cdots \vee t_{out_m})$$

4.3.3.4 Algorithm: Window decomposition

Algorithm 4 describes the WindowDecomposition function, which takes an LTS as input and returns a window decomposition.

Algorithm 4: Generation of a Window Decomposition.

```
1  Function WindowDecomposition (S, L, T, s_0)
2      F ← SAT formula (4.1) ;                          // for property encoding
3      T' = T ;                                         // T' contains the uncovered transitions
4      i ← 1 ;                                          // Window index
5      while T' ≠ ∅ do
6          Cost ← Σ_{t∈T'} t ;                          // max uncovered transitions
7          T_i ← PseudoBooleanOptimization(F, Cost);
8          T_i ← LargestConnectedComponent(T_i);
9          L_i ← The labels associated to T_i;
10         S_i ← States from S adjacent to T_i;
11         s_ini ← s_0 ∈ S_i ? s_0 : ⊥_i;
12         w_i = LTS(S_i, L_i, T_i, s_ini);
13         T' ← T'\T_i;
14         i ← i + 1;
15     return {w_1, ··· , w_n};                         // The window decomposition
```

At the beginning of the function, the SAT formula is initialised with the properties described in the above sections (see **line 2**). The set of uncovered transitions T' contains the transitions that are not yet covered by any of the previously extracted windows of the derived WD. In **line 3**, T' is initialised with all LTS transitions T.

In **lines 5-14**, a window is extracted by the set of uncovered transitions, until the whole LTS is covered (termination is when $T' = \emptyset$). The window extraction is guided by the PseudoBooleanMinimisation function (see **line 7**), which takes as input the SAT formula F and the variable Cost computed by the sum of all yet uncovered transitions, and returns the transitions T_i that are selected to model the window w_i. It is possible that more than a single connected set of transitions is present in T_i. In this case, the biggest connected component is returned by the LargestConnectedComponent function (see **line 8**).

In **lines 9-12**, the window w_i is built by considering the states and labels of the selected transitions T_i. Also, the initial state s_{ini} is initialised to be either the initial of the LTS, or to be in set of inactive states. If the latter is the case, the window is not needed to model the initial state of the LTS and starts by being inactive.

Before proceeding to the next window $i + 1$ (see **line 14**), the extracted transitions are removed from the set of uncovered transitions (see **line 13**). The algorithm returns the final window decomposition (see **line 15**).

The decomposition of LTSs into window decompositions has been automated in the *Cats* tool [91]. We used Minisat [66] as SAT solver, and the PBLib library [92] for Pseudo-Boolean optimisation. We refer the reader to [31] for the proof of correctness of the window decomposition algorithm.

4.3.4 Deriving window conditions

This section describes the developed technique for deriving the *wake-up* and *wake-up marking* conditions automatically. These are needed to orchestrate the behaviour of extracted windows of a system.

The derivation of these Boolean conditions has been automated in the developed *Shutters* tool [32]. The conditions are derived by extracting truth tables that describe the behaviour of the windows, and synthesising Boolean equations from such tables using the tool *Espresso* [70]. The equations can be optionally refactored by relying on the tool *Abc* [71].

4.3.4.1 Deriving wake-up conditions

Let S_w be the set of states modelled by a window w, with S_w being a subset of the full set of states S of the LTS, i.e. $S_w \subset S$. The truth table entries of the wake-up condition $c(w, S_w)$ of the window w can be specified as:

- $\boldsymbol{c(w,s) = 0}$, in all the states of the labelled transition system that are not included in S_w, i.e. $s \notin S_w$. In these states, window w is off and does not have to wake-up.
- $\boldsymbol{c(w,s) = 1}$, in all the states of the labelled transition system that *enter* the window w, i.e. $\forall s \in S_w : (\exists s' \notin S_w : s' \xrightarrow{e} s)$. In these states, window w has to wake up and start modelling the LTS.
- $\boldsymbol{c(w,s) = X}$, in all the other states that are contained in the window, but that cannot be entered from the outside. In fact, we *don't care* if the wake-up condition is either True or False when a window is already on. Such values help minimise the final wake-up conditions.

As an example, the truth table of the wake-up condition related to the windows in Figure 4.26d are:

$$c(w_1, S_{w_1}) = \begin{cases} c(w_1, s_6) = 0,\ c(w_1, s_7) = 0,\ c(w_1, s_8) = 0,\ c(w_1, s_9) = 0 \\ c(w_1, s_0) = 1,\ c(w_1, s_4) = 1 \\ c(w_1, s_1) = X,\ c(w_1, s_2) = X,\ c(w_1, s_3) = X,\ c(w_1, s_5) = X \end{cases}$$

$$c(w_2, S_{w_2}) = \begin{cases} c(w_1, s_1) = 0,\ c(w_1, s_2) = 0,\ c(w_1, s_3) = 0,\ c(w_1, s_5) = 0 \\ c(w_1, s_0) = 1,\ c(w_1, s_4) = 1 \\ c(w_1, s_6) = X,\ c(w_1, s_7) = X,\ c(w_1, s_8) = X,\ c(w_1, s_9) = X \end{cases}$$

Since we use Petri nets to model transition systems, it is convenient to represent LTS states with PN markings. The above truth tables can be thus represented as follows.

The above truth tables are synthesised to the conditions $c(w_1, S_{w_1}) = \overline{p5} \wedge p8$ and $c(w_1, S_{w_1}) = \overline{p1} \wedge p3$ by using *Espresso* [70]. In the developed tool, we implemented the *positive mode*, which generates conditions with only positive literals. It is, in fact, always

Marking (s)	**Boolean vector**	$c(w_1, S_{w_1})$	**Marking (s)**	**Boolean vector**	$c(w_2, S_{w_2})$
$\{p5, p8\}$ ($s6$)	(1,0,0,1,0)	0	$\{p1, p4\}$ ($s1$)	(0,1,0,0,1)	0
$\{p6, p9\}$ ($s7$)	(0,1,0,0,1)	0	$\{p1, p3\}$ ($s2$)	(0,1,0,1,0)	0
$\{p7, p9\}$ ($s8$)	(0,0,1,0,1)	0	$\{p2, p4\}$ ($s3$)	(0,0,1,0,1)	0
$\{p5, p9\}$ ($s9$)	(1,0,0,0,1)	0	$\{p0, p4\}$ ($s5$)	(1,0,0,0,1)	0
$\{p7, p8\}$ ($s0$)	(0,0,1,1,0)	1	$\{p0, p3\}$ ($s0$)	(1,0,0,1,0)	1
$\{p6, p8\}$ ($s4$)	(0,1,0,1,0)	1	$\{p2, p3\}$ ($s4$)	(0,0,1,1,0)	1
otherwise	otherwise	X	otherwise	otherwise	X

possible to synthesise conditions where no literals are negated, since every state of the LTS is identified by a net marking, i.e. the final wake-up condition can be the result of the OR-wise Boolean operation of all the states where a window has to wake-up, see below.

$$c(w_1, S_{w_1})_{pos} = s0 \vee s4 = (p7 \wedge p8) \vee (p6 \wedge p8)$$
$$c(w_2, S_{w_2})_{pos} = s0 \vee s4 = (p0 \wedge p3) \vee (p2 \wedge p3)$$

The above equations can be refactored by using *Abc* [71], and the following simpler conditions can be derived: $p8 \wedge (p7 \vee p6)$, and $p3 \wedge (p0 \vee p2)$, respectively.

4.3.4.2 Deriving wake-up marking conditions

We use a similar approach to derive the wake-up marking conditions. Let $m(w, p, s)$ be the wake-up marking of a place p in a window w in a state s. Its truth table can be formulated as follows:

- $\boldsymbol{m(w, p, s) = 0}$, if we can *enter* the window w in the state s, and the place p is unmarked. E.g. in Fig. 4.27, it is possible to enter w_1 in the state $s0$, and the places $\{p1, p2, p4\}$ are unmarked.
- $\boldsymbol{m(w, p, s) = 1}$, if we can *enter* the window w in the state s, and the place p is marked. E.g. in Fig. 4.27, it is possible to enter w_1 in the state $s0$, and the places $\{p0, p3\}$ are marked.
- $\boldsymbol{m(w, p, s) = X}$ otherwise. We are only interested in detecting whether a place p should be marked when a window w is entered.

As an example, the truth tables related to the wake-up marking conditions of the places $\{p0, p2, p3\}$ are shown in the above tables. These can be used to synthesise the

Marking (s)	Boolean v.	$m(w_1, p_0, s)$
$\{p7, p8\}$ ($s0$)	(0,0,1,1,0)	1
$\{p6, p8\}$ ($s4$)	(0,1,0,1,0)	0
otherwise	otherwise	X

Marking (s)	Boolean v.	$m(w_1, p_2, s)$
$\{p7, p8\}$ ($s0$)	(0,0,1,1,0)	0
$\{p6, p8\}$ ($s4$)	(0,1,0,1,0)	1
otherwise	otherwise	X

Marking (s)	Boolean v.	$m(w_1, p_3, s)$
$\{p7, p8\}$ ($s0$)	(0,0,1,1,0)	1
$\{p6, p8\}$ ($s4$)	(0,1,0,1,0)	1
otherwise	otherwise	X

following wake-up marking conditions for window w_1 in Figure 4.26d: $p0 = p7$, $p2 = p6$ and $p3 = 1$, respectively.

4.3.5 Applications of the model

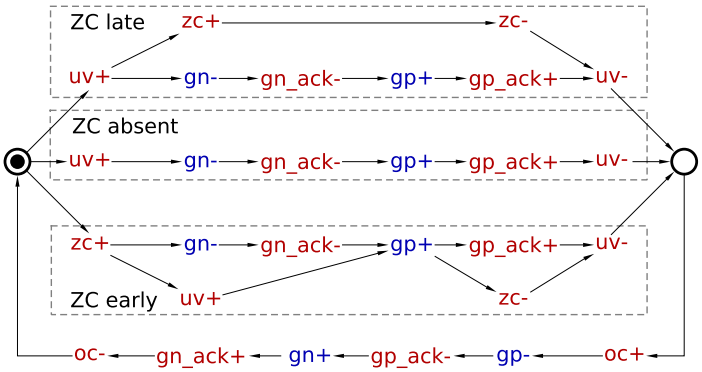
Process Windows are important for simplifying the understanding of concurrent system specifications, as discussed in the *Motivation* Chapter of the thesis. In this section, we discuss a couple of applications that benefit from this presented formalism.

Asynchronous hardware design

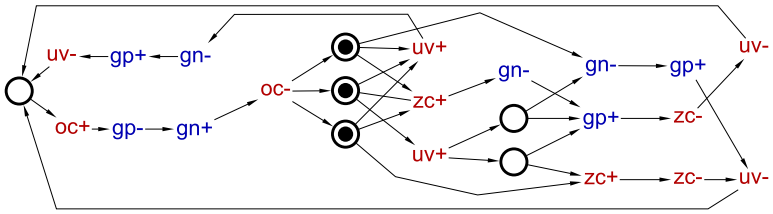
Asynchronous circuits are a particular class of hardware systems that do not rely on timing assumptions. Their sequential elements, rather than synchronising on a periodic signal (commonly named CLOCK), communicate via *handshake mechanisms* based on causality dependencies. Every internal component functions whenever the data is available at the input, and produces results at the output, making the whole system truly concurrent. This type of systems are typically represented by models that can express concurrency, e.g. Partial Orders, Signal Transition Graphs, Petri Nets, Dataflow graphs (all models that we introduced and dealt with previously).

However, even by using one of these models, system specifications can become difficult to understand. Consider, for example, the STG-based system specification of a controller for a power buck converter [14] in Figure 4.29a. This specification has been obtained by careful analysis of the controller by the designer, who analysed its behaviour and drew the STG specification by separating three different scenarios.

Designing the STG in Figure 4.29a is a difficult task. To prove this, we show the STG specification derived automatically from the LTS of the controller by Petrify [54],



(a) Specification derived manually.



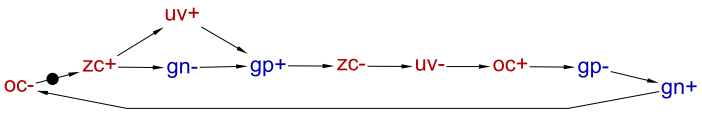
(b) Specification derived automatically by Petrify.

Figure 4.29: STG specifications of a buck controller.

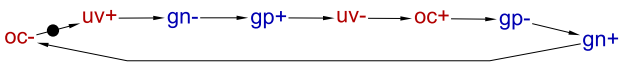
see Figure 4.29b. From this specification, it is arguably more difficult to identify internal system scenarios and understand the behaviour of the controller.

On the other hand, Figure 4.30 shows the scenario specification extracted automatically from the controller transition system in Figure 4.29b by means of the methodology based on Process Windows. The three scenarios identified by the presented algorithm for window decomposition (in Section 4.3.3) match the ones identified by the designer in Figure 4.29a. The derived wake-up marking conditions (derived by the approach in Section 4.3.4) are the same for the three scenarios: a token is placed in the arc $\texttt{oc-}\rightarrow\texttt{zc+}$, which corresponds to the left-most place in Figure 4.29a where the scenarios start. The wake-up conditions, in turn, monitor this overlapping state where the execution of these three scenarios is triggered. In this case, the PW methodology is successful by substantially reducing the effort required by the designer to understand a complex specification.

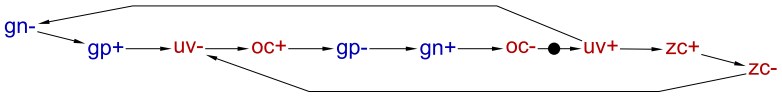
In the context of hardware design, this methodology can be also useful for reducing the effort required to synthesise circuits of complex specifications. The established



(a) 'ZC late' scenario.



(b) 'ZC absent' scenario.



(c) 'ZC early' scenario.

Figure 4.30: STG specification of a buck controller derived by the PW methodology.

technique for automated synthesis of asynchronous controllers based on STGs [55], in fact, cannot handle large specifications, as we will show in the *Case studies* Chapter. Whenever Petrify [54] fails, one option would be to extract internal scenarios and synthesise them in isolation. Finally, the extracted wake-up and wake-up marking equations can be used for switching between scenarios in the final implementation.

Process mining

We believe that the presented Process Windows can also provide significant advantages in the area of *Process Mining*, e.g the discovery of process models from (software) execution traces (or *logs*). In many cases, models mined by logs of complex systems are, in turn, difficult to read and understand as they represent the systems as a whole. We argue that these models would benefit by the presented flow for splitting event-based systems into scenarios. Following, we describe an example for highlighting this idea.

Consider the log of a computer program in Figure 4.31(left), and the corresponding Labelled Transition System on the right-hand side of the Figure. The LTS describes the behaviour of the program by its output.

Figure 4.32 shows the Petri net synthesised automatically by Petrify [54] using the theory of regions and label splitting. The PN describes all relations between the behaviours of the system, but it is difficult to understand since such behaviours are

Traces:

```
adeac
acdbeacb
abceac
acbdecab
adeca
acbeca
```

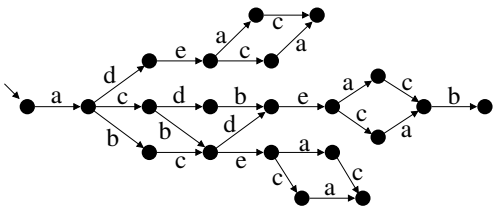


Figure 4.31: Log of traces and Labelled Transition System.

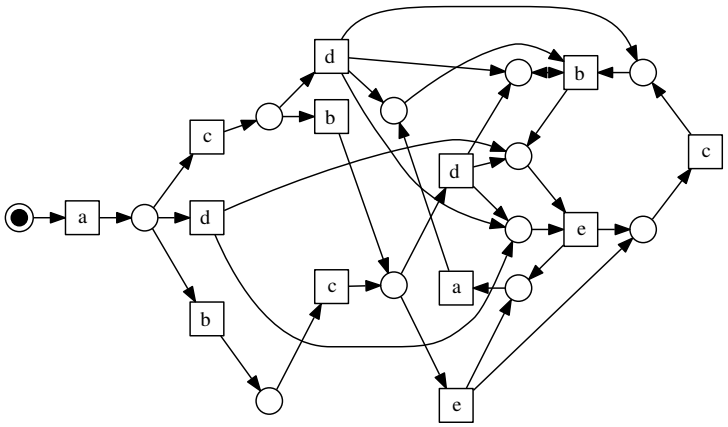


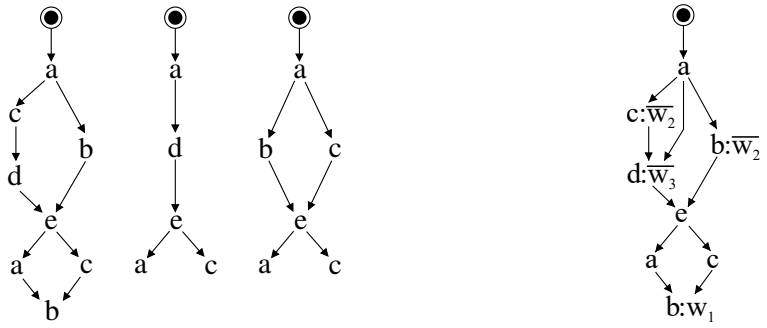
Figure 4.32: Petri net of the program log synthesised automatically by Petrify.

represented in the same structure.

Figure 4.33a shows the windows extracted via the algorithm presented in Section 4.3.3, w_1 to w_3 from the left to the right. The execution traces of the program are well defined, and easier to understand than when the log was represented via a LTS or a Petri net. In addition, one could compose the three scenarios of the system back again into a CPOG in order to emphasise the common events between the three scenarios. Figure 4.33b shows the CPOG obtained by the composition of the three scenarios by using the one hot encoding $\{(w_1, 100), (w_2, 010), (w_3, 001)\}$. For example, by activating the second window (code 010), the events b and c are disabled and the CPOG behaves as the w_2. The fundamentals of the CPOG methodology are in Chapter 3.

4.3.6 Related work and summary

We presented the new Process Windows formalism, and showed a couple of applications that can benefits from the capabilities of this new formalism. To conclude, we briefly



(a) Extracted windows. (b) Window overlay with CPOG.

Figure 4.33: The program log represented via the methodology based on Process Windows.

describe the related work in the field.

Process Windows have been inspired by [85], where a technique for extracting scenarios from software logs to derive process models is presented. However, while in [85] each log trace is fully covered by at least one scenario, this is not the case for our approach where a log trace can travel across multiple scenarios to be covered.

The idea of using scenarios for simplifying system understanding originates by an investigation of how scenarios are used in different fields [47]. In the context of hardware design, for example, there are other methodologies that are based on scenarios, such as the ones introduced in this dissertation based on CPOGs and DFSs, or the one based on *Parameterised graphs* [61]. In the area of system protocols, the *Message-Sequence Charts* [93] and the *Live Sequence Charts* [94] are widely used scenario-based approaches. In these, scenarios are specified via sequence of messages exchanged between functional units of a system. These methodologies are supported by automated software synthesis features, and can be specified by well-known programming languages such as C++. Finally, it's also worth mentioning *Oclets* [95] that use Petri nets for describing scenarios and anti-scenario (i.e. behaviours that do not have to occur) of a system; *Untanglings* [96] that represent the behaviour of a system by its acyclic partial runs. Untanglings are used for model checking rather than for system specification. And *Structured Occurrence Nets* [97], which can specify relations betweens scenarios. The scenarios of all these approaches, however, need to be specified by hand. They are not extracted automatically as in PW.

Chapter 5

Case studies

In the previous Chapter, we presented our contributions in the area of scenario-based design. We described a new approach for composing scenarios efficiently; the Dataflow Structures formalism that enables one to compose static dataflow scenarios of asynchronous circuits; and the Process Windows formalism that enables the automated decomposition of complex system specification into simpler scenarios.

In this chapter, we show the importance of scenarios on three real-life case studies that also validate some of our previous contributions. In Section 5.1, we evaluate the new scenario composition approach by synthesising various types of control architectures. In Section 5.2, we present a DFS-based methodology to design reconfigurable asynchronous pipelines, and use it to design a hardware accelerator (also relying on CPOGs). In Section 5.3, we present an FPGA accelerator for computational drug discovery, where the scenarios of systems are too many to be specified, but yet they all need to be processed. We refer the reader to Figure 1.3 for a diagram of the dependencies of the next sections.

5.1 Control synthesis

In Section 4.1, we described how to employ the presented scenario composition technique for composing scenarios (expressed as partial orders) into a CPOG. The latter benefits from automated *behavioural synthesis* features to derive digital hardware controllers.

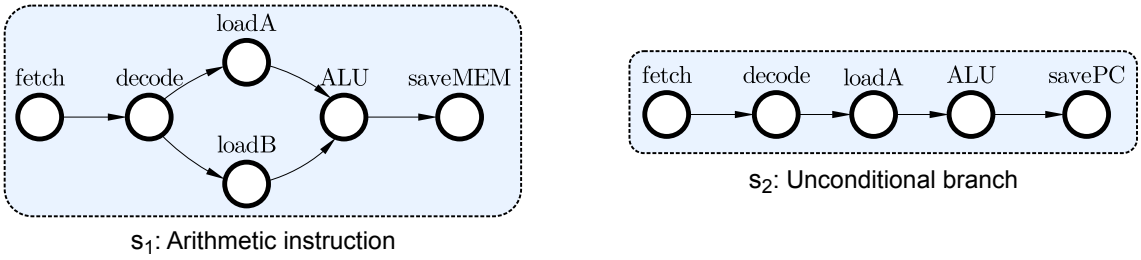


Figure 5.1: A CPOG-based processor specification comprising two instructions.

In this section, we compare this *proposed* composition technique to other existing approaches to scenario composition (reviewed in Section 4.1.2), and to existing high-level approaches that make use of automated behavioural synthesis techniques to produce hardware controllers (which will be reviewed shortly in Section 5.1.1).

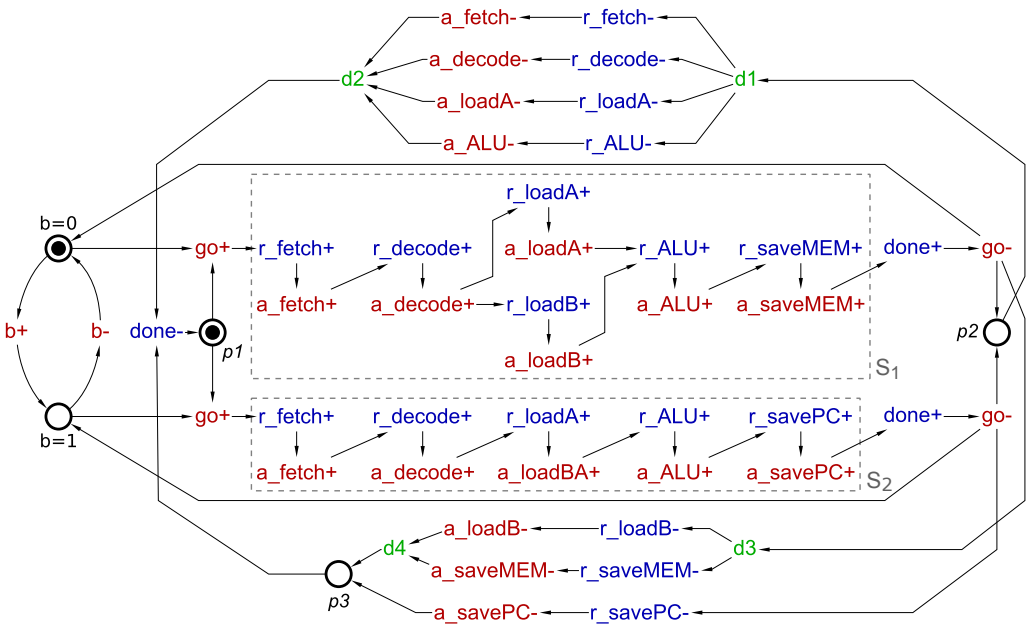
For this comparison, we consider a wide set of benchmarks coming from three domains: ad-hoc controllers, processor instruction sets and process mining in Sections 5.1.3, 5.1.4 and 5.1.5, respectively. All these benchmarks are included in the higher level domain of *control architectures*, i.e. components that have to control parts of a system. The experimental setup and notation used for benchmarking are discussed in Section 5.1.2. The content of this section has been published in [27,28].

5.1.1 Related work

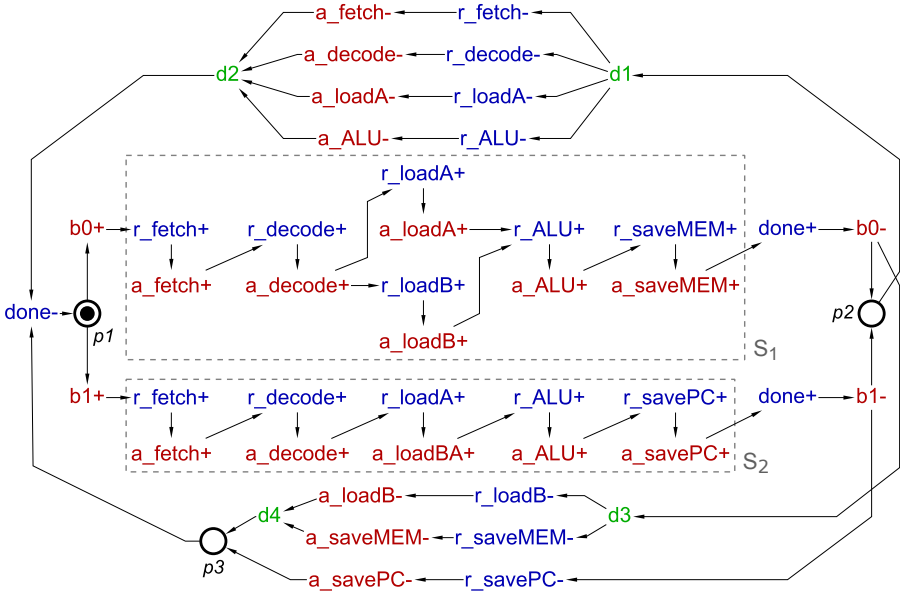
Behavioural synthesis is not new and several other approaches exist that allow the designer to formally describe the behaviour of a control hardware architecture and synthesise the corresponding hardware implementation. The two most relevant approaches are: the work by Cortadella et al. [55] that is based on Signal Transition Graphs (STG) as the formal specification model for synthesising asynchronous controllers; and the work by De Micheli [98] that uses synchronous Finite State Machines (FSM) to derive controllers implemented as microcode memories or hard-wired control units.

In [53], CPOGs were compared to STGs and FSMs in terms of their compactness and ease of use when specifying asynchronous circuits. Below we highlight the main reasons for using CPOGs in the broader context of scenario-based synthesis.

- The separation of datapath (scenarios) and control (encoding) abstraction layers



(a) Binary encoding on one bit b. The arithmetic instruction scenario s_1 can be executed when there is a token in the place $b = 0$, while the unconditional branch scenario s_2 when there is a token in the place $b = 1$. The $go+$ signal starts the scenarios.



(b) One-hot encoding on two bits $\{b0, b1\}$. The arithmetic instruction scenario s_1 is executed when the signal $b0+$ fires, while the unconditional branch scenario s_2 when the signal $b1+$ fires.

Figure 5.2: STGs of the processor specification in Figure 5.1. Two types of encoding are used as interface to run the internal scenarios.

enables scenarios to remain unchanged when the encoding changes.

- Underlying partial orders can efficiently represent highly concurrent systems without incurring exponential state explosion.
- Scenario composition allows CPOGs to remain compact even when the size of the specification grows.
- Opportunity to minimise various design criteria (e.g. area, power, latency) by scenario encoding, which is our main goal.

In this section, we compare CPOGs, STGs and FSMs practically by synthesising real scenario-based specifications. Our benchmarks highlight that: (1) the STG methodology does not scale to specifications that include many scenarios, (2) the presented approach shows better results than the FSM methodology.

As an example of specifications, Figures 5.2 and 5.3 show STGs and a FSM models of the processor scenarios in Figure 4.2b. In these figures: red transitions are the inputs of the designed controller, blue ones are the outputs and green ones (present only in STGs) are dummy transitions (used only to simplify models). The prefixes '`r_`' and '`a_`' stand for 'request' and 'acknowledge', respectively.

In the STGs in Figure 5.2, the two scenarios are mutually excluded via the choice place `p1`, and the causality dependencies of their operations are modelled via sequences of request/acknowledge transitions. Two types of specifications are shown: in Figure 5.2a, scenarios are encoded by the binary encoding (the same one that we used in the CPOG in Figure 4.3); in Figure 5.2b, scenarios are encoded by the one-hot encoding (the `go` signal is removed, as it is redundant for starting the scenarios). The STG with the binary encoding is more complex due to the higher number of signals and transitions, and cannot be handled by the corresponding tool-chain for hardware synthesis (this will be shown shortly). Thus, we compared the experimental results produced by the proposed algorithm to one-hot encoding STGs. STG specifications are handled by the EDA tools Petrify [54] and MPSat [76], which synthesise asynchronous implementations using different algorithms. Petrify uses binary decision diagrams [55], while MPSat uses Petri net unfoldings [99].

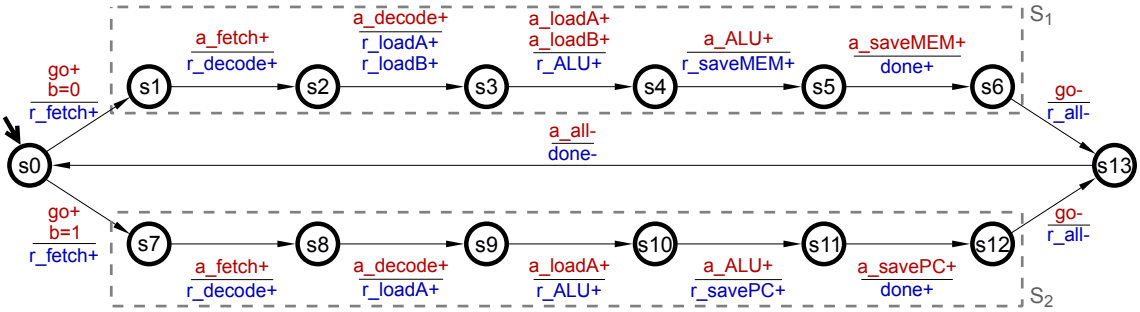


Figure 5.3: FSM (with binary encoding) of the processor specification in Figure 4.2b.

In the FSM specification in Figure 5.3, the two scenarios are selected via one bit b observed at the rising edge of the go signal, which starts the computation. Upon the completion of each scenario, all output requests r_all are reset, and the FSM returns to the initial state s0 when all input acknowledgements a_all are also reset. In our experiments, FSM specifications are handled by Design Compiler [100] that derives synchronous controllers. We applied concurrency reduction [53] to some of the considered FSM specifications not to incur state explosion. All used benchmarks are available online [68].

5.1.2 Configuration and notation for benchmarking

The experimental results that we present (see Tables 5.1, 5.2 and 5.3) have been obtained on an Intel-i7-3610QM 2.30GHz CPU, equipped with 8 GB DDR3 1600 MHz RAM memory. Benchmarks are:

1. Specified in the form of partial orders in WORKCRAFT [34], and synthesised by the developed tool SCENCO [68] using the CPOG composition and synthesis mechanism.

2. Specified as STGs in WORKCRAFT, and synthesised by Petrify [54] and MPSat [76].

3. Specified as synchronous FSMs in VHDL, and synthesised by Synopsys Design Compiler [100].

The same 90nm gate library is used for technology mapping.

For presenting the experimental results, we use the following notation. **#e** denotes the number of encodings generated and synthesised by the proposed algorithm. The

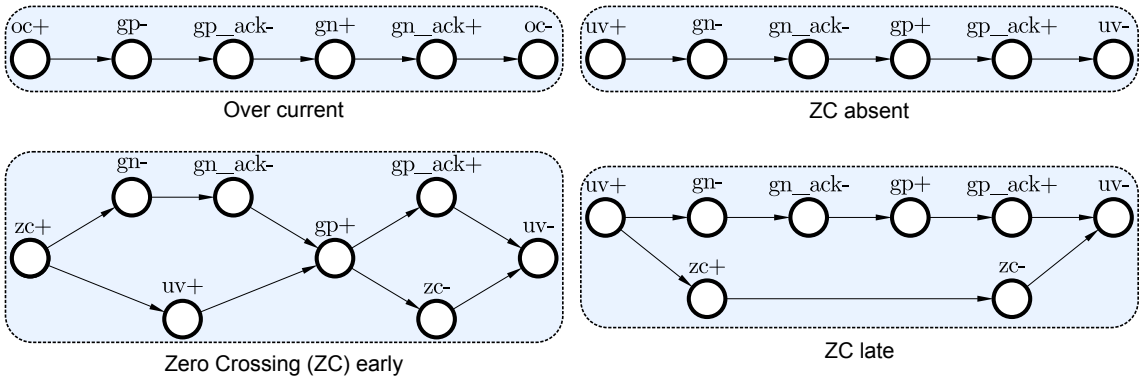


Figure 5.4: Scenario-based specification of a power management controller of a buck converted.

smallest controller out of these is shown as result. **Area ($|B|$)** denotes the area [μm^2] of the resulting controller, with the number of bits used for encoding in brackets. **RT** denotes the tool runtime [s], which is the time that goes from parsing the specification to obtaining the final implementation. We only consider results produced within a runtime of 1 hour, denoted in turn as timeout **TO**. Finally, we use the **dash** character '—' when a behavioural synthesis approach cannot be applied to a benchmark due to a technical issue, in this case we explain the reasons of the failure in the **evaluation** paragraphs.

Controllers derived by FSM and STG include sequential components (registers and C-elements, respectively) for holding system states. In the results, we only consider the combinational part of the controllers for not penalising them in the final evaluation.

5.1.3 Ad-hoc controllers

The first set of benchmarks includes an on-chip power management controller of a buck converter [14], and an asynchronous controller for the reconfigurable pipeline of our produced hardware accelerator, described in Section 5.2.

The power management controller is required to regulate the activation of the PMOS (gp) and NMOS (gn) transistors in response to three signals coming from sensors within the power regulator: over-current (oc), under-voltage (uv) and zero-crossing (zc). The two transistors must never be on at the same time to avoid a short circuit. In Section 3.3.1, we described part of its specification in the form of STGs. In Figure 5.4, on the other hand, we show the four *specification* scenarios in the form of partial orders

Table 5.1: Comparison of CPOG scenario encoding algorithms over the ad-hoc controller benchmarks. *Units of measure:* Area ($|B|$) = [μm^2] (number of bits).

Model	$\|S\|$	**Exhaustive**	**Single-literal**	**SAT-based**	**Proposed #e = 10**
Buck controller	4	266 (2)	**261 (3)**	266 (2)	266 (2)
Processor controller	13	TO	**357 (12)**	TO	481 (4)

that compose the controller, and describe two of these scenarios below.

Zero Crossing (ZC) absent scenario: when the uv condition is detected (event uv+), the NMOS transistor must be switched off (event gn-) and the PMOS has to be subsequently switched on (event gp+).

ZC late scenario: The same two operations (switching off and on the transistors NMOS and PMOS, respectively) have to be performed if the condition zc is detected right after the condition uv.

The controller of the fabricated dataflow processor has to handle a 16-stage reconfigurable pipeline for computing the *ordinal pattern encoding* of long data streams (we will describe this calculation in Section 5.2.1). The controller manages the energy-quality of the result by controlling the number of active pipeline stages. It is an important case study, as it was fabricated in an ASIC and tested. Three of its 13 scenarios that compose the reconfigurable pipeline are specified below in the text-form, i.e. the scenario s_1 activates 4 pipeline stages, the s_2 activates 5 stages, up to the scenario s_{13} that activates all 16 stages of the pipeline, see the corresponding partial orders in Figure 2.3.

$$s_1 = \mathsf{stage1} \rightarrow \mathsf{stage2} \rightarrow \mathsf{stage3} \rightarrow \mathsf{stage4}$$

$$s_2 = \mathsf{stage1} \rightarrow \mathsf{stage2} \rightarrow \mathsf{stage3} \rightarrow \mathsf{stage4} \rightarrow \mathsf{stage5}$$

$$\vdots$$

$$s_{13} = \mathsf{stage1} \rightarrow \mathsf{stage2} \rightarrow \cdots \rightarrow \mathsf{stage16}$$

Evaluation: Table 5.1 shows the results upon application of the state-of-the-art encoding algorithms. The Single-literal encoding produces a 1.9% smaller *buck controller* in comparison to other approaches, and uses one more variable than needed ($|B| = 3$). On

2 variables, the optimal controller is generated by the Exhaustive search by definition. Such a controller is also achieved by the SAT-based and by the proposed Heuristic algorithm.

The *reconfigurable pipeline controller* is not produced within the considered timeout by the Exhaustive and the SAT-based algorithms, due to the complexity of the corresponding scenario specification. The Single-literal controller is $\simeq$ 25.8% smaller than the controller produced by the proposed encoding technique, and uses 3$\times$ more variables. The final design implements the controller produced by the proposed encoding technique, as the final design was constrained by the pins of the external package.

The runtime of the tool for processing the above benchmarks is always less than 1 s.

5.1.4 Processor instruction sets

The second set of benchmarks includes different subsets of instructions of the ARM Cortex M0+ [27], Texas Instruments MSP430 [40] and Intel 8051 [12, 24]. These processor specifications were derived by analysing their corresponding ISA reference manuals, and identifying classes of instructions (specification[1] scenarios) that share similar functionalities and addressing modes.

In regards to the design of real processors, the above manual scenario extraction approach is not ideal to obtain accurate specifications. However, recent research on specification languages for processor architectures (see Section 4.1.1.1) enables to fully specify the behaviour of modern systems comprising hundreds of instructions, and to derive accurate specifications for synthesising real processors. In this context, the presented algorithm is important as it scales well to hundreds of scenarios (as we show in Section 5.1.5) making the CPOG-methodology suitable to the design of such modern systems.

We fully describe the ARM Cortex M0+ scenario specification in [27] (also see Appendix A). This processor has an ISA constituted of 68 instructions. The specification composed of 11 scenarios and 6 datapath modules (scenario operations) models 61 of these instructions. As an example, two scenarios of the specification are shown in

[1]See the difference between specification, implementation, and activation scenarios in Chapter 2.

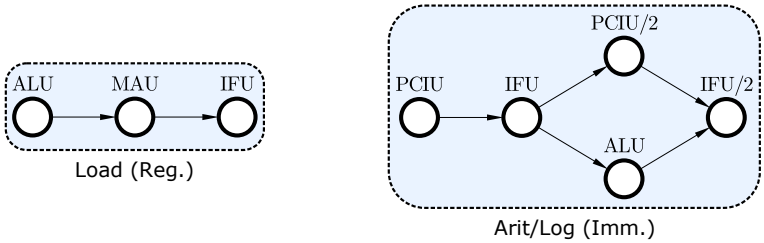


Figure 5.5: Two scenarios of the ARM Cortex M0+ specification.

Figure 5.5 and described below.

Load (reg.) covers the *LDR (reg.)* instruction. The ALU operation computes the memory address, the MAU loads a value from the memory and stores it into a specified register. The IFU fetches a new processor instruction.

Arit/Log (Imm.) covers arithmetical, logical and data transfer instructions with immediate addressing, e.g. *ADD (imm.)*, *LSR (imm.)*. An immediate value is fetched from the instruction register (PCIU $\rightarrow$ IFU), and used as operand for the selected operation (ALU). The result is stored into a specified register. The ALU operation is executed concurrently with the program counter incrementation (PCIU/2). The resulting *PC* is used for fetching a new instruction (IFU/2).

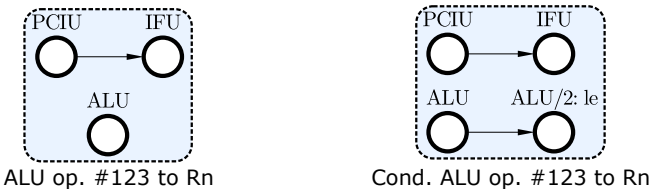


Figure 5.6: Two scenarios of the Texas Instruments MSP430 specification.

The Texas Instruments (TI) MSP430 scenario specification has been introduced in [40]. The specification composed of 8 scenarios and 7 datapath modules models the full instruction set composed of 51 instructions. This benchmark is important as some of its scenarios have conditional elements. As an example, two of its scenarios are shown in Figure 5.6 and described below:

ALU op. #123 to Rn executes an arithmetic operation between two general purpose registers $\{A, B\}$, and writes the result back into one of them (ALU). Operations PCIU $\rightarrow$ IFU fetch a new instruction concurrently.

Cond. ALU op. #123 to Rn executes the above arithmetic operation between two general purpose registers $\{A, B\}$ conditionally (ALU/2), on the condition $le = A < B$ set by the ALU. The flag le is input to the synthesised controller, and is used for managing the activation of ALU/2.

The second scenario is said to be *conditional*, and can be described in the form of a CPOG. Conditional scenarios can be composed regularly with other scenarios, see [23] for further details.

The Intel 8051 specification supported the design of an asynchronous version of this processor [12]. It comprises 37 scenarios and 17 datapath modules that model 255 processor instructions. It is important to the CPOG validation, as it contains $3\times$ more scenarios and $2\times$ more operations than the other processor benchmarks.

Evaluation: Table 5.2 shows the results of the applied state-of-the-art CPOG encoding algorithms to the described set of processor benchmarks. This set is also used to compare the proposed algorithm based on CPOGs to the methodologies based on FSMs and STGs, as it is the most diverse set being characterised by (1) specifications of different sizes (from 4 to 37 scenarios), (2) specifications with conditional scenarios (see TI MSP430), (3) specifications comprising a different number of datapath modules (from the ARM processor with 6, to the Intel with 17). For these reasons, it is able to highlight the characteristics of all used approaches to behavioural synthesis.

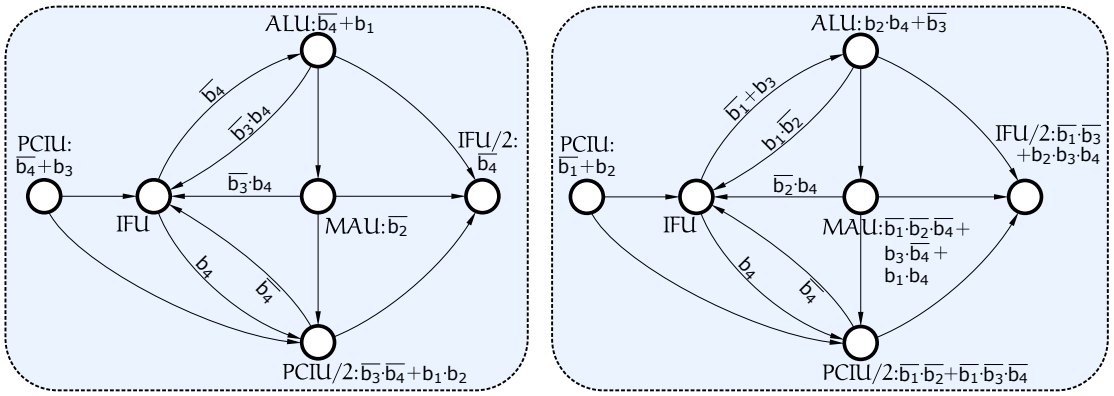
The *Exhaustive search* produces the smallest instruction decoders using $\lceil log_2|S| \rceil$ variables. In practice, it is applicable to specifications that contains up to 8 scenarios, as its runtime increases exponentially with the specification size.

The *Single-literal* encoding produces the smallest instruction decoders in most of the cases when it does not exceed the time limit. However, synthesised decoders might not be applicable to real processors, as the code size $|B|$ is fixed by the algorithm rather than by the processor (op)code specifications.

The *SAT-based encoding* produces decoders with an average overhead of $\simeq$ 7.4% in comparison to Exhaustive decoders. The current implementation does not support scenarios in the form of CPOGs (see missing results $-$ in the TI MSP430 rows). The runtime of the SAT-based and Single-literal approaches increase exponentially (exceeding the timeout) when $|S|$ grows.

Model	$\|S\|$	Exhaustive		Single-literal		SAT-based		Proposed #e = 10		Proposed constr., #e = 10		FSM (seq. encod.) dc_shell		STG (one-hot encod.) Petrify / MPSat	
		Area ($\|B\|$)	RT	Area ($\|B\|$)	RT	Area ($\|B\|$)	RT	Area ($\|B\|$)	RT	Area ($\|B\|$)	RT	Area ($\|B\|$)	RT	Area ($\|B\|$)	RT
ARM Cortex M0+	4	**162 (2)**	1	177 (4)	1	**162 (2)**	1	**162 (2)**	2	167 (2)	1	193 (2)	5	265 / 200 (4)	51 / 93
	5	**179 (3)**	162	202 (5)	1	201 (3)	1	182 (3)	2	192 (3)	2	227 (3)	7	243 / 242 (5)	47 / 295
	6	201 (3)	524	**198 (5)**	1	225 (3)	3	201 (3)	2	241 (3)	2	303 (3)	7	– / 263 (6)	– / 270
	7	209 (3)	1051	**198 (5)**	1	225 (3)	3	226 (3)	2	249 (3)	2	316 (3)	7	– / 319 (7)	– / 409
	8	180 (3)	1005	**165 (5)**	2	224 (3)	2	224 (3)	2	230 (3)	2	345 (3)	7	– / 343 (8)	– / 981
	9	TO	TO	**220 (5)**	1	269 (4)	1	224 (4)	2	235 (4)	2	457 (4)	6	– / 456 (9)	– / 2232
	10	TO	TO	**218 (5)**	1	232 (4)	1	241 (4)	2	252 (4)	2	467 (4)	10	– / TO	– / TO
	11	TO	TO	**212 (5)**	1	246 (4)	2	249 (4)	2	279 (4)	2	498 (4)	7	– / TO	– / TO
Texas Instruments MSP430	4	**154 (2)**	1	177 (4)	1	–	–	**154 (2)**	2	171 (2)	1	288 (2)	5	292 / TO (4)	119 / TO
	5	**174 (3)**	162	181 (6)	1	–	–	180 (3)	2	193 (3)	1	294 (3)	7	– / TO	– / TO
	6	**189 (3)**	489	191 (7)	2	–	–	205 (3)	2	235 (3)	1	384 (3)	7	– / TO	– / TO
	7	252 (4)	1059	**223 (8)**	1	–	–	276 (3)	2	293 (3)	2	376 (3)	7	– / TO	– / TO
	8	**299 (4)**	1145	304 (8)	1	–	–	321 (3)	2	345 (3)	2	390 (3)	6	– / TO	– / TO
Intel 8051	4	175 (2)	1	**166 (3)**	1	175 (2)	1	175 (2)	2	175 (2)	1	240 (2)	7	– / TO	– / TO
	5	175 (3)	165	**170 (4)**	1	189 (3)	1	178 (3)	2	193 (3)	2	335 (3)	7	– / TO	– / TO
	6	214 (3)	521	**196 (5)**	1	235 (3)	1	226 (3)	2	224 (3)	1	377 (3)	7	– / TO	– / TO
	7	**234 (3)**	1111	242 (6)	1	239 (3)	1	240 (3)	2	260 (3)	1	422 (3)	7	– / TO	– / TO
	8	295 (3)	1145	**267 (7)**	1	TO	TO	302 (3)	2	312 (3)	2	498 (3)	7	– / TO	– / TO
	9	TO	TO	**286 (8)**	1	303 (4)	13	322 (4)	2	347 (4)	2	519 (4)	10	– / TO	– / TO
	10	TO	TO	464 (9)	2	TO	TO	**335 (4)**	2	368 (4)	2	565 (4)	10	– / TO	– / TO
	15	TO	TO	TO	TO	TO	TO	**679 (4)**	2	736 (4)	2	943 (4)	12	– / TO	– / TO
	20	TO	TO	TO	TO	TO	TO	**822 (5)**	2	842 (5)	2	1187 (5)	15	– / TO	– / TO
	25	TO	TO	TO	TO	TO	TO	**1147 (5)**	3	1202 (5)	3	1519 (5)	17	– / TO	– / TO
	30	TO	TO	TO	TO	TO	TO	**1376 (5)**	4	1450 (5)	4	1879 (5)	20	– / TO	– / TO
	35	TO	TO	TO	TO	TO	TO	**1690 (6)**	4	1741 (6)	4	2159 (6)	21	– / TO	– / TO
	37	TO	TO	TO	TO	TO	TO	**1879 (6)**	4	2037 (6)	4	2336 (6)	20	– / TO	– / TO

Table 5.2: The proposed algorithm is compared with existing CPOG composition techniques, and with the FSM and STG synthesis approaches over 26 processor instruction set benchmarks. Bold results are the smallest controllers for each model. *Units of measure:* Area ($|B|$) = [μm^2] (number of bits), Runtime (RT) = [s].



(a) Proposed encoding. (b) Random encoding.

Figure 5.7: ARM Cortex M0+ system specifications in the form of CPOG.

On average, the *Proposed encoding* produces implementations with an area overhead of $\simeq$ 4.5% in comparison to optimal solutions. It scales to higher number of scenarios (see Intel 8051 results), and supports scenarios in the form of CPOGs (see TI MSP430 results). The runtime is always within the timeout. As an example, Figure 5.7a shows the ARM system specifications obtained by composing its 11 constituent scenarios via the proposed encoding, and Figure 5.7b shows the one derived via the random search algorithm. The 'proposed' CPOG contains shorter conditions ϕ, which is why the corresponding controller has also a smaller size.

We also run the *Proposed encoding* by constraining $\left\lceil \frac{|S|}{2} \right\rceil$ scenarios of every processor specification randomly, using $\{0, 1, ?, X\}$. The resulting decoders always satisfy the composition constraints given, and have an overhead of $\simeq$ 12.4%, on average, in comparison to optimal implementations.

Finally, we used the behavioural synthesis approaches based on *Finite State Machines* (FSM) and *Signal Transition Graphs* (STG) to show that the proposed methodology shows better results in comparison to these established techniques in the field. The approach based on synchronous FSM and Design Compiler (known as *dc_shell* in the Synopsys tool-chain) is always able to synthesise controllers from the given specifications with the usage of the *sequential encoding*. Synthesised implementations show an average area overhead of $\simeq$ 56% in comparison to the proposed unconstrained approach. The processing runtime is comparable.

On the other hand, the methodology based on STG is never able to synthesise implementations from the given specifications with the *sequential encoding*. The results shown on the table are derived with the *one-hot encoding*, which simplifies the specifications by replacing the go transitions and their dependencies with the codes, see all benchmarks in [68]. However, even after this simplification, the methodology is not often successful. In most cases, Petrify returns the error "*support too big for minimisation*" (see missing results − on the left-hand side of the STG column), and MPSat does not find a solution within the given time limit (see TO entries on the right-hand side). MPSat is partially successful with the ARM Cortex M0+, whose scenarios include fewer datapath modules (6 compared to the 17 modules of the Intel 8051) and which does not include conditional scenarios (as the TI MSP430). On average, the methodology based on STG has an area overhead of $\simeq$ 43% in comparison to the proposed unconstrained approach, and a much higher synthesis runtime.

5.1.5 Software output logs

The third set of benchmarks includes scenario specifications that describe a set of different software output logs [101]. They come from the process mining community: artificial logs derived from the simulation of a process model (`BigLog1`, `Log2`, `Caise2014`), and real-life traces in different other contexts (`purchasetopay`, `incidenttelco`, `svn_log`, `telecom`, `documentflow`).

Due to the size of these benchmarks (from 16 to 651 scenarios), we compare the *Proposed encoding* to the *Sequential encoding* and *Random search*, as the other CPOG algorithms always exceed the time limit. The proposed encoding is applied in three configurations: **(a)** #e set to 1 , **(b)** #e set to 10 , **(c)** with #e $=$ 1 and the Simulated Annealing parameters modified in such a way to allow $\times$10 more iterations for the optimisation (SA $\times$10).

Evaluation: On average, the area of the controllers found by the proposed encoding in configurations 1 and (2) are 4.7% (9.8%) more efficient, in terms of area, than sequential controllers, and 12.9% (18%) more efficient than random controllers. In turn, the results produced by the proposed encoding in configuration 3 are 13.2% and 21.7% more efficient, on average, than the sequential and random implementations, respectively.

Model	$\|S\|$	**Sequential**		**Random search**		**(a) Proposed #e = 1**		**(b) Proposed #e = 10**		**(c) Proposed #e = 1, SA ×10**	
		Area ($\|B\|$)	Runtime	Area ($\|B\|$)	Runtime	Area ($\|B\|$)	Runtime	Area ($\|B\|$)	Runtime	Area ($\|B\|$)	Runtime
BigLog1	16	503 (4)	1	530 (4)	1	495 (4)	1	**389 (4)**	2	447 (4)	1
(S) Purchasetopay	20	581 (5)	1	713 (5)	1	502 (5)	1	**434 (5)**	3	546 (5)	1
BigLog2	26	953 (5)	1	906 (5)	1	699 (5)	1	**565 (5)**	3	568 (5)	1
Log2	32	1163 (5)	1	1271 (5)	1	969 (5)	1	825 (5)	3	**815 (5)**	1
Incidenttelco	77	3531 (7)	1	3490 (7)	1	2953 (7)	1	2927 (7)	6	**2613 (7)**	1
(M) Svn_log	92	3215 (7)	1	3274 (7)	1	2928 (7)	1	2769 (7)	5	**2363 (7)**	2
Telecom	122	4417 (7)	1	4644 (7)	1	4123 (7)	1	4237 (7)	7	**3938 (7)**	2
Colibrilog	167	6870 (8)	2	7777 (8)	2	7212 (8)	3	**6789 (8)**	11	6852 (8)	6
Caise2014	401	37314 (9)	11	38339 (9)	11	37234 (9)	14	37180 (9)	115	**35616 (9)**	32
(L) Log1-filtered	402	**13910 (9)**	3	20215 (9)	4	18212 (9)	6	18004 (9)	46	14343 (9)	31
Documentflow	651	**21131 (10)**	8	25222 (10)	8	24700 (10)	12	24623 (10)	112	22646 (10)	57

Table 5.3: Three configurations of the proposed algorithm are compared with trivial CPOG composition techniques on 11 software output logs divided in (S)mall, (M)edium and (L)arge sizes. Bold results are the smallest controllers for each model. *Units of measure:* Area ($|B|$) = [μm^2] (number of bits), Runtime (RT) = [s].

On average, the Sequential encoding produces $\simeq 8.56\%$ smaller controllers in comparison to the Random search algorithm. Such a good result is due to a certain degree of similarity between pairs of subsequent scenarios, which are encoded naturally by pairs of subsequent and similar codes by the Sequential algorithm.

In Table 5.3, the benchmarks are divided in three sets of different sizes, from the bottom to the top: **(S)**mall ($10 < |S| < 30$), **(M)**edium ($30 < |S| < 400$) and **(L)**arge ($400 < |S| < 652$). See below consideration:

Small set, configuration 2 of the proposed encoding finds the best results, as the higher number of encodings inspected (#e = 10) provides a higher chance to produce a good result. The increased number of SA iterations of configuration 3 is not justified in this set due to the small $|S|$.

Medium set, configuration 3 finds the best results, as the higher $|S|$ justifies a longer optimisation time provided to the Simulated Annealing optimisation.

Large set, configuration 3 finds smaller controllers in comparison to configurations 1 and 2. However, these benchmarks highlight the heuristic (inaccurate) component of the proposed approach, which may find worse controllers in comparison to trivial algorithms. For this set, a higher number of SA iterations would be justified for obtaining good results.

This set of benchmarks shows that the proposed approach can handle specifications of hundreds of scenarios. Also, it can be tuned as much as needed by modifying the time for the SA optimisation.

5.1.6 Conclusion

In this section, we evaluated the novel scenario composition approach (described in Section 4.1) on an extensive set of benchmarks, and also compared it to the state-of-the-art composition algorithms for CPOG, and to the behavioural synthesis techniques based on FSM and STG.

Table 5.4 summarises the comparison of all CPOG composition techniques, relying on the experimental results shown previously. The proposed algorithm, unlike the already

Table 5.4: Features of the CPOGs compositional algorithms. **Max $|S|$**: maximum number of scenarios supported. **CPOG-scenarios**: support of scenarios in the form of CPOG. **Constraints**: support of composition constraints.

	Exhaustive	**SAT-based**	**Single-literal**	**Proposed**
Max $\|S\|$	8	$\simeq 10/15$	$\simeq 10/15$	$\simeq 650$
CPOG-scenarios	$\surd$		$\surd$	$\surd$
Constraints				$\surd$

existing techniques, handles hundreds of scenarios with a good area/synthesis runtime trade-off, and supports composition constraints. It also supports conditional scenarios for modelling behaviours that contain dynamic branching. Also, the experimental results highlight that the CPOG methodology produces more efficient implementations (in terms of area) than the approaches based on FSMs and STGs. The latter can be applied only to relatively compact models.

5.2 Reconfigurable asynchronous pipelines

In Section 4.2, we presented the Dataflow Structures formalism, and showed that it can be used for modelling the behaviour of dynamically reconfigurable asynchronous circuits by describing their constituent static scenarios and then composing them.

In this section, on the other hand, we validate the Dataflow Structures by designing an asynchronous accelerator that is meant to compute the *Ordinal Pattern Encoding* (OPE), introduced in Section 5.2.1. The accelerator relies on a pipeline that can be dynamically reconfigured for performing a set of OPE computations of different sizes. Implementation scenarios have been fundamental for the design of this pipeline, where the number of active stages is decided at runtime by a control logic derived from CPOGs. Sections 5.2.2 and 5.2.3 describe a methodology based on the DFS to model and implement reconfigurable asynchronous pipelines, and applies it to the OPE case study. Our prototype implements both a static and a reconfigurable pipeline, and has been fabricated on an Application Specific Integrated Circuit (ASIC). The chip is evaluated in Section 5.2.4: the two pipelines are compared for highlighting advantages and drawbacks of asynchronous dynamic reconfigurability. The related work in the field, consisting of existing approaches to design dataflow pipelines, and conclusion are finally

discussed in Section 5.2.5. Part of the content of this section has been/will be published in [29,30].

We conclude our introduction with a couple of clarifications. Another solution for accelerating the ordinal pattern encoding would be to use Field Programmable Gate Arrays (FPGA) and *static reconfigurability*. The latter is different than dynamic reconfigurability, as hardware architecture changes are achieved via total or partial circuit re-synthesis rather than via extra on-board control logic. In our work, we focus on studying dynamic asynchronous reconfigurability, and therefore we do not compare our ASIC accelerator to FPGA implementations. However, we refer the reader to two papers where ASIC and FPGA characteristics are compared [102, 103] for highlighting the differences between these platforms.

Also, it is not in the scope of our work to compare our asynchronous accelerator to its synchronous counterpart, as our goal is to validate the proposed Dataflow Structures that is meant to improve asynchronous design. Synchronous and asynchronous circuits have been extensively compared in literature under many perspectives – e.g. in terms of power consumption [104], performance [105], resilience [106] – and their advantages and drawbacks are known [25]. Also, the style of implementation that we chose for our design (Null Convention Logic [107]) has been studied and compared to synchronous logic [108,109]. Thus, we believe that one more comparison would be redundant.

5.2.1 Introduction to ordinal pattern encoding

Let X be a series of numbers in the form of $\{x_1, x_2, \cdots, x_N\}$. The ordinal pattern encoding of X is the permutation $\pi = \{k_1, k_2, \cdots, k_N\}$ such that the series of numbers $X' = \{x_{k_1}, x_{k_2}, \cdots, x_{k_N}\}$ is in ascending order (i.e. $x_{k_1} \leq x_{k_2} \leq \cdots \leq x_{k_N}$), e.g. let X be $\{5, 2, 10\}$, then its OPE π is $\{2, 1, 3\}$ such that $X' = \{2, 5, 10\}$.

The OPE finds application in different areas where the analysis of numerical series is important: from stock market prediction, to medical data analysis. In these applications, its purpose is to detect repetitive patterns within long data streams in order to predict future data values. To be really effective and discover hidden patterns within a stream, however, the OPE needs to be applied to subsets of a series with different sizes.

Index	*Window*	*OPE* (π)
1	$(3,1,4,1,5)$	$(3,1,4,2,5)$
2	$(1,4,1,5,9)$	$(1,3,2,4,5)$
3	$(4,1,5,9,2)$	$(3,1,4,5,2)$
4	$(1,5,9,2,6)$	$(1,3,5,2,4)$

For example, the above table shows the application of the OPE to every subset (or window) of size 5 of the series $\{3,1,4,1,5,9,2,6\}$. At the start of the computation (*Index* 1), the first window of the series $\{3,1,4,1,5\}$ is processed and the result is $\{3,1,4,2,5\}$. Afterwards, the index of the window is increased (*Index* 2) and the second window $\{1,4,1,5,9\}$ is processed resulting in $\{1,3,2,4,5\}$. At the end of the stream (*Index* 4), the final window $\{1,5,9,2,6\}$ is processed and the result is $\{1,3,5,2,4\}$.

The results are different, indeed, if the same data stream is processed by considering a window of size 6, see below.

Index	*Window*	*OPE* (π)
1	$(3,1,4,1,5,9)$	$(3,1,4,2,5,6)$
2	$(1,4,1,5,9,2)$	$(1,4,2,5,6,3)$
3	$(4,1,5,9,2,6)$	$(3,1,4,6,2,5)$

In this section, we apply the DFS methodology to the design of an asynchronous accelerator for performing a range of OPE computations of different window sizes via dynamic hardware reconfiguration. Our design has been inspired by [35], where a *hardware-oriented* (i.e. sorting-free) algorithm capable of computing the ordinal pattern encoding is presented.

To avoid sorting, every incoming window of a series is converted into a *Lehmer code*. Let W be a window $\{w_1, w_2, \cdots, w_N\}$, the Lehmer code $\mathcal{L}$ of W is $\{l_1, l_2, \cdots, l_N\}$, where $l_i = \#\{w_j : i < j \ \wedge \ w_j < w_i\}$. In practice, every value of $\mathcal{L}$ represents the number of positions that the corresponding value in W must be delayed to make the window sorted. For example, the Lehmer code of the window $\{3,1,4,1,5\}$ is $\{2,0,1,0,0\}$. From the right to the left, w_3 has to be delayed of 1 position first (i.e. the window becomes $\{3,1,1,4,5\}$),

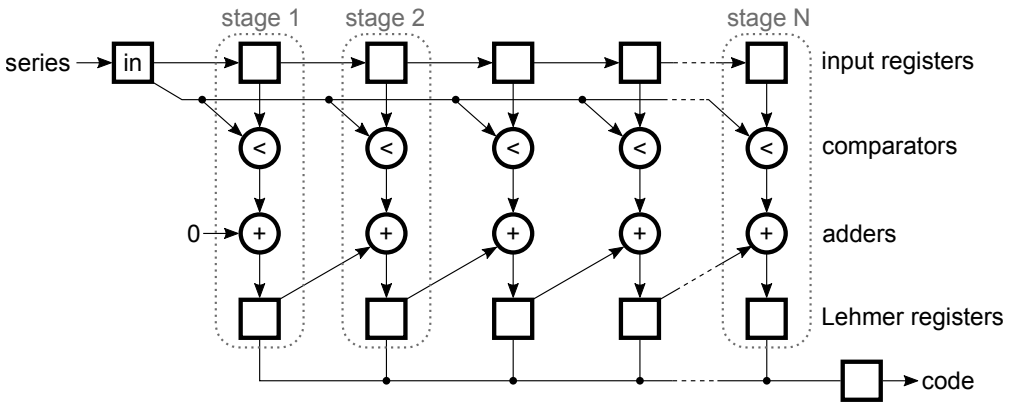


Figure 5.8: The N-stage pipeline for computing the OPE. The data stream is propagated through the *input registers*, whose values are compared by the *comparators* to the latest incoming value hold in the `in` register. The *adders* sum the comparator results with the results of the preceding pipeline stages stored in the *Lehmer registers*, whose values are compressed into the final *code*.

and subsequently w_1 has to be delayed of 2 positions: $\{1, 1, 3, 4, 5\}$.

The Lehmer code can be used to obtain the OPE [35], and can be computed in hardware efficiently by relying on the following two properties:

1. If $\mathcal{L}(W) = (l_1, l_2, \cdots, l_N)$, then the Lehmer code of $W' = (w_2, \cdots, w_N)$ (where the oldest value of the window is removed) is $\mathcal{L}(W') = (l_2, \cdots, l_N)$. In other words, the value of l_i is independent by older values of the stream w_k, where $k < i$.

2. If $\mathcal{L}(W') = (l_2, \cdots, l_N)$, then the Lehmer code of $W'' = (w_2, \cdots, w_N, w_{N+1})$ (where a new value enters the window) is $\mathcal{L}(W'') = (l_2^+, \cdots, l_N^+, 0)$, where:

$$l_i^+ = \begin{cases} l_i + 1 & \text{if } w_i > w_{N+1} \\ l_i & \text{otherwise} \end{cases}$$

   Intuitively, the Lehmer bit of the most recent value (l_{N+1}) is always 0, while the remaining bits have to be increased by 1 (i.e. such values need to be delayed by one more position to make this window sorted) if their corresponding window values (w_i) are bigger than the new one (w_{N+1}).

In [35], these above properties are used to elaborate a hardware approach to compute OPEs of incoming data streams very efficiently. The Lehmer code of each window is computed by relying on comparators, half-adders, and registers. The produced result is

reused to compute the Lehmer code of the next window. Figure 5.8 shows the schematic of this algorithm [35], where the squares (□) represent registers storing data values and circles (◯) represent operations between such values. Each value of the incoming data series propagates through the *input registers*. At every cycle, the new incoming value of a window (`in` register) is compared to all the remaining values of the window. Each comparator generates a logic '1' if `in` is smaller than the window value, otherwise it generates a '0'. The comparator results are added to the results of the previous stages, which are stored in the *Lehmer registers*. Such registers hold the Lehmer code of the current window. In our implementation, the values of such registers are compressed into the output *code*, which can be used to compute the OPE of the whole data stream. In the figure, we highlight the stages of the pipeline. The number of active stages determines the size of the OPE computation, e.g. 5-stage pipeline computes the OPE of size 5.

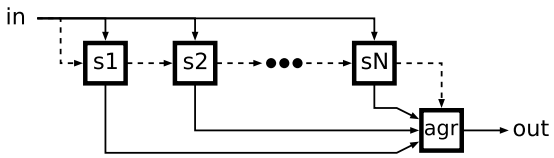
The described hardware architecture is an optimal testbench to validate the formal model that we proposed in Section 4.2, namely Dataflow Structures. DFS models are capable of modelling dynamic reconfigurability of asynchronous circuits, which is what we need to design an accelerator for processing OPE of different sizes via pipeline depth reconfiguration. In the next sections we present a methodology to design asynchronous reconfigurable pipelines, and apply it to the OPE case study.

5.2.2 Modelling reconfigurable asynchronous pipelines

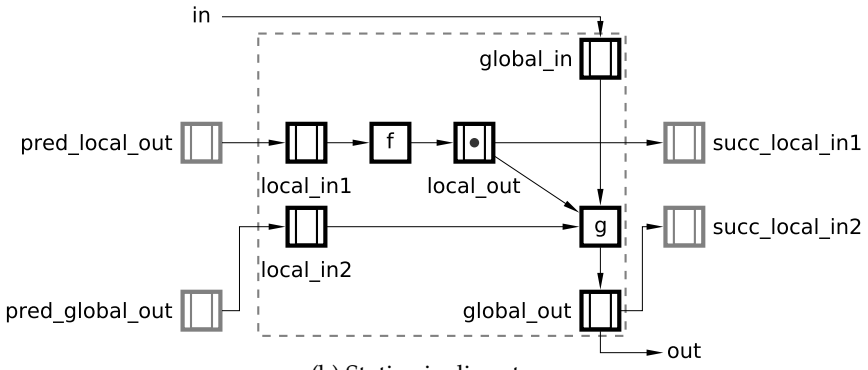
Figure 5.9a shows a generic pipeline structure comprising N stages, which interface to each other via *local channels* (dashed arcs), and are connected to the common input `in` and aggregated output `out` via *global channels* (solid arcs).

A DFS model for a pipeline stage is shown in Figure 5.9b. It applies a function `f` to the tokens in the `local_in1` register (input data from the previous stage local output) and produces a token in the `local_out` register (local output data for the next stage). The produced token, paired with the common input token in `global_in` and the global output of the previous register in `local_in2`, are passed to a function `g`, which produces a `global_out` token, used to aggregate the output of all stages.

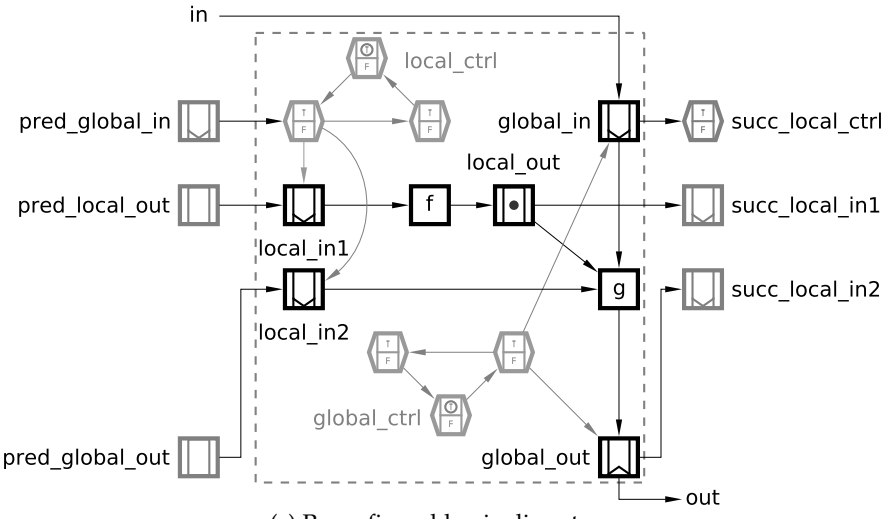
As for the OPE case study, one typical reconfiguration scenario for such a pipeline is to change its *depth* (i.e. the number of stages) depending on the application requirements.



(a) Generic N-stage pipeline.



(b) Static pipeline stage.



(c) Reconfigurable pipeline stage.

Figure 5.9: Pipeline with local and global stage interfaces.

We design a reconfigurable generic pipeline that is capable of using a given number of initial stages, bypassing the remaining ones. Figure 5.9c shows our DFS design of a reconfigurable pipeline stage. The stage local input is implemented as a pair of push registers `local_in1` and `local_in2` controlled by the `local_ctrl` structure. The `global_in` and `global_out` are push and pop registers, respectively, controlled by the `global_ctrl` structure. Both `local_ctrl` and `global_ctrl` are 3-register loops (the minimum number of registers required for a token oscillation). To include a stage into the reconfigurable pipeline, these control loops need to be initialised with the `True` tokens; to exclude it – with the `False` tokens. Note that a token starts oscillating in `local_ctrl` only if the previous stage is included in the pipeline and `global_in` operates as a static register – this is done to prevent the stage operation when the previous stage is inactive.

The model of the OPE reconfigurable pipeline

Using the DFS-based static and reconfigurable pipeline stages described previously, Figure 5.10 shows our DFS model of the reconfigurable OPE pipeline.

The pipeline works as in the schematic shown in Figure 5.8. The incoming data tokens of numerical series propagate through the pipeline via the `local_in1` and `local_out` registers of the stages, and are compared (by the logic node $<$) to the new incoming data item (in the `data` register) that propagates to the `global_in` register of each stage. The result of the comparison is added (by the logic node $+$) to the OPE delay result of the previous stage (coming from the `local_in2` register) and stored into the `global_out` register of each stage. The `global_out` registers store the OPE delay values, which are aggregated into the final `code` register.

In the example of this figure, the first stage `s1` is always included and is therefore implemented in the static style; the remaining stages are reconfigurable. Note that the stage `s2` is optimised by reusing `global_ctrl` to control both the local and global interfaces, which is possible because `s1` is always included in the pipeline and its `global_in` is a static register, not a push.

Using the developed WORKCRAFT plugin, the DFS model of the reconfigurable OPE pipeline can be visually simulated and formally verified at the abstract technology-independent level where data is represented by abstract tokens. Several cases of

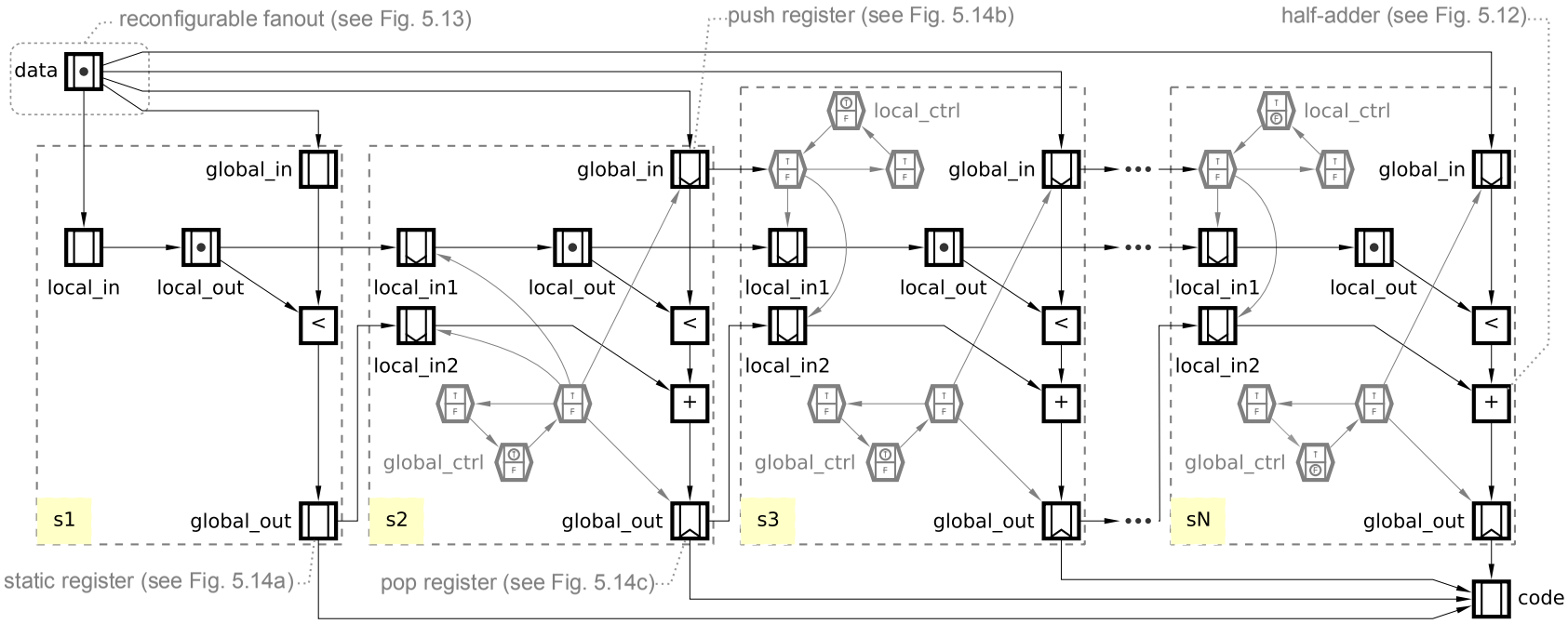
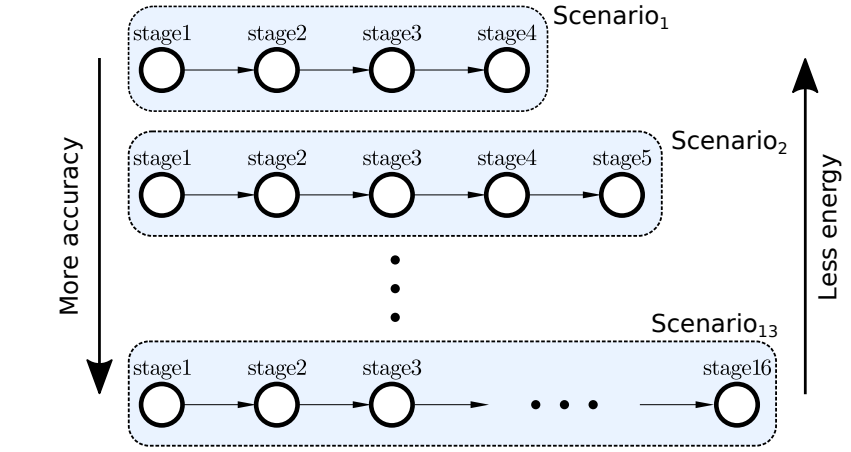
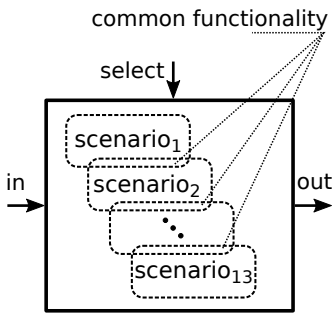


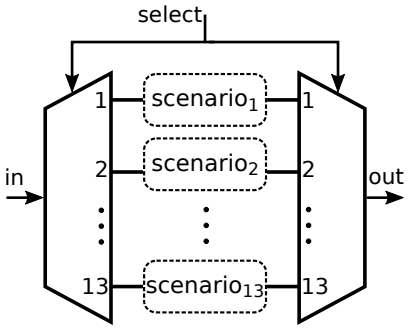
Figure 5.10: The DFS model of the reconfigurable OPE pipeline, from 1 to N stages that corresponds to the OPE window size. The model has been derived by the efficient composition of static and reconfigurable pipeline stages (described in Fig. 5.9). The first stage $s1$ is static as it is always present, while the remaining stages can be disabled by the corresponding control registers. Notice that the second stage $s2$ has been optimised, and the `local_ctrl` control registers have been removed as the preceding stage $s1$ is always active. The grey-shaded control registers are needed to coordinate the behaviour of the pipeline in the model, but are substituted by a combinational control unit (described in Section 5.1.3) in the final hardware implementation. In Section 5.2.3, we show that the DFS model can be mapped to a digital circuit using a library of asynchronous components, and describe a few examples on the key parts of the pipeline, e.g. reconfigurable fanout.



(a) Implementation scenarios of the OPE-pipeline.



(b) Efficient composition.

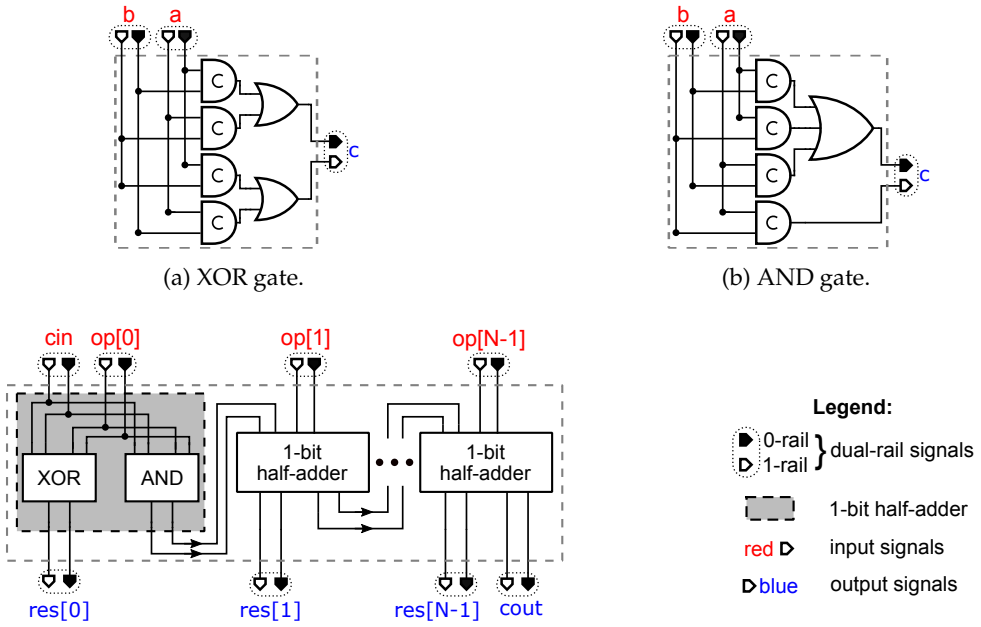


(c) Inefficient composition.

Figure 5.11: The scenarios of the OPE-pipeline, and two approaches to their composition.

deadlock and non-persistent behaviour (mostly due to incorrect initialisation of control registers) were identified, analysed and corrected during the design process.

The model of the pipeline has been obtained by composing the static and reconfigurable pipeline stage templates shown in Figure 5.9. The composition is said to be *efficient* (see Figure 5.11b) because the functionality of each stage is reused in all the implementation scenarios where a pipeline stage is included, e.g. stage 7 is synthesised once and used whenever one has to compute the OPE with size ≥ 7, i.e. for all s_i in $4 \leq i \leq 13$, see Figure 5.11a. As opposed to the *inefficient* scenario composition approach (see Figure 5.11c), where each scenario of the pipeline is synthesised in isolation, causing stages to be synthesised multiple times. The number of active pipeline stages determines the size of the OPE computation, and consequently also the *quality* (i.e. seen as the capability of analyse longer series) and energy consumption of the chip, see Figure 5.11a.



(a) XOR gate.

(b) AND gate.

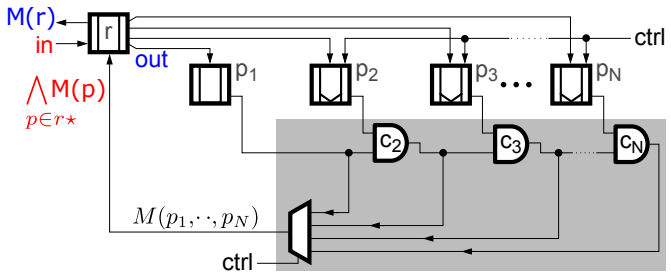
(c) N-bit half-adder.

Figure 5.12: Implementation of an asynchronous dual-rail N-bit half-adder, with NCL-D gates.

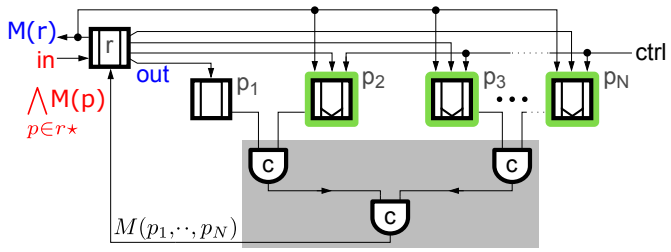
5.2.3 Implementing reconfigurable asynchronous pipelines

The DFS model was translated into a circuit implementation netlist using a library of pre-built NCL-D style asynchronous dual-rail components (comparator, half-adder, and a set of registers) that rely on the 4-phase communication protocol [107]. A conventional flow was subsequently employed for technology mapping, layout, and place-and-route tasks. Here, we describe the implementation of some of these components, which can be reused as IP blocks for other circuits that make use of the same style and protocol.

As an example of a combinatorial logic node, the circuit behind the *N-bit half-adder* embedded in the OPE stages (node + in Figure 5.10) is shown in Figure 5.12. Its purpose is to increment the result of the previous stage (`op` operand contained in the `local_in2` register) by the comparator result (`cin` operand in the node <), which can either be a logic 0 or 1. The N-bit half-adder is implemented as a chain of 1-bit half-adders (see Figure 5.12c) made of NCL-D XOR and AND gates (see Figures 5.12a-b). In the schematics shown in this section, pins highlighted in red are the input signals, and the ones in blue are the output signals. Dual-rail signals have their pins coloured as black



(a) Reconfigurable interconnections.



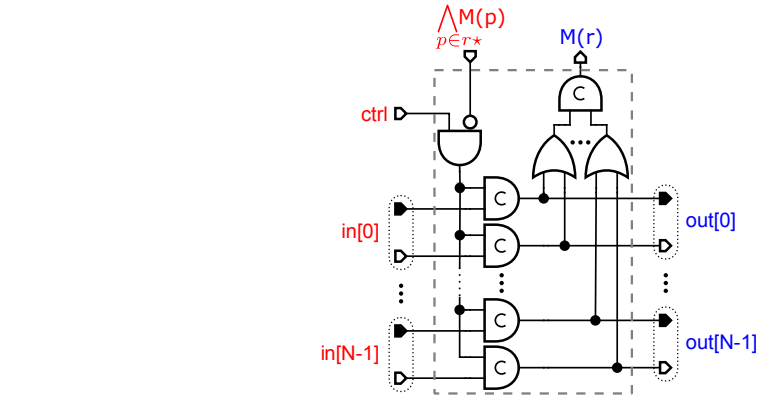
(b) Distributed reconfiguration.

Figure 5.13: Asynchronous reconfigurable fanout implementations.

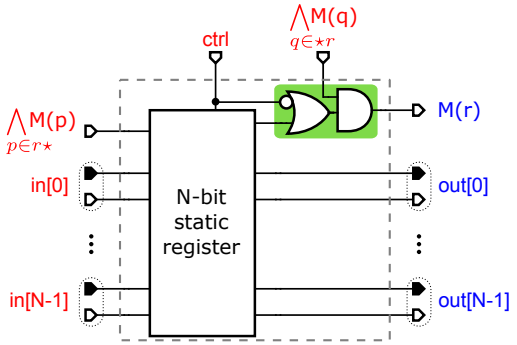
and white pairs, the former represents the 0-rail and the latter represents the 1-rail, see legend at the bottom of Figure 5.12.

While logic nodes are passive to the handshake mechanism, register nodes have an active role. Their implementation is at the core of asynchronous reconfiguration mechanisms. Figure 5.13 shows two approaches to implement the handshake between `r` (`data` node in Figure 5.10) and its N reconfigurable fanout registers `p` (`global_in` nodes) representing the pipeline stages. The pipeline can be shortened by disabling the latest push registers, e.g. if the pipeline depth is 2: $\{p_1, p_2\}$ are enabled, if it is 3: $\{p_1, p_2, p_3\}$ are enabled, and so on.

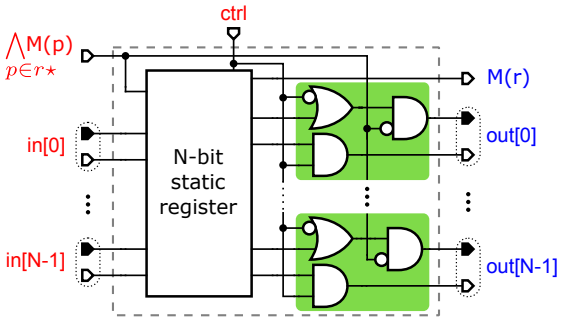
In the approach in Figure 5.13a, all registers (including push and pop) are mapped to the *N-bit static register* implementation in Figure 5.14a, which enables the usage of 4-phase dual-rail asynchronous communication under the delay-insensitive timing model, as described in [110]. The `ctrl` input enables a register, and it is omitted for the static registers in Figure 5.13a that are indeed always enabled. The `M(r)` output is the register marking state, while the `M(p)` input is the marking state coming from the R-postset of `r`. In this approach, the fanout reconfiguration is controlled by modifying the



(a) N-bit static register.



(b) N-bit push register.



(c) N-bit pop register.

Figure 5.14: Static and dynamic register implementations.

acknowledgement interconnection between registers, i.e. the registers p are connected to r through a daisy chain of C-elements and a multiplexor. The latter selects which marking state $\mathtt{M}(\mathtt{p}_1, \cdots, \mathtt{p}_N)$ to consider according to the number of push registers enabled by ctrl, e.g. if 3 stages are enabled, the multiplexor propagates the output of the C-element $\mathtt{c}_3$: $\mathtt{M}(\mathtt{p}_1, \mathtt{p}_2, \mathtt{p}_3)$, see grey part in Figure 5.13a.

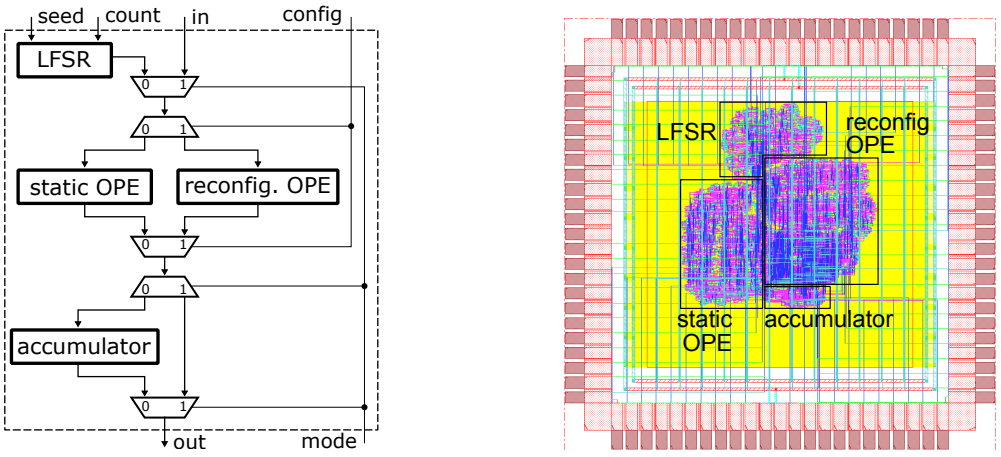
In the approach in Figure 5.13b, on the other hand, the reconfiguration mechanism is distributed across the implementation of push and pop registers. In this approach, static registers are mapped to the structure in Figure 5.14a, and push and pop registers are mapped to Figures 5.14b and 5.14c, respectively. At the core of both the *N-bit push* and *pop registers* there is a static register. Push registers are extended with extra-logic (highlighted in green) that allows them to behave as static registers when enabled ($\mathtt{ctrl} = 1$), and to propagate an 'empty' marking state M(r) driven by the R-preset

marking state M(q) when disabled (ctrl = 0), see Figure 5.14b. Pop registers, in turn, also extend the static register implementation with extra-logic (in green) that allows them to behave as static registers when enabled, and propagate an 'empty' data value out encoded by a logic 0, driven by their R-postset marking state M(p), when disabled, see Figure 5.14c. This approach allows the interconnection of r and its fanout registers p via a tree of C-elements for the propagation of the fanout marking state $\mathtt{M}(\mathtt{p}_1, \cdots, \mathtt{p}_N)$, see grey part in Figure 5.13b. Notice that the implementation of push and pop registers can be customised to accommodate application needs, e.g. pop registers propagating a logic 1 as 'empty' output data value.

The proposed implementation of push and pop registers allows them to behave as the corresponding nodes of the DFS formalism. Hence, one can place such registers at the input and output of a block/scenario, as in Figure 4.18, to isolate this from unnecessary switching activity when not in use. In literature, this approach is referred to as *operand-isolation*, and can save large amount of dynamic circuit power [83]. The reconfigurable interconnections approach does not make use of such elegant implementations of push and pop registers, but achieves the same goal by dynamically reconfiguring the interconnections between static registers.

Our prototyped OPE chip makes use of the *reconfigurable interconnections* approach to implement all internal reconfiguration scenarios, e.g. final code aggregation. The chip evaluation (see Section 5.2.4) highlighted a delay in the computation caused by the daisy chain of C-elements, which led us to design the improved *distributed reconfiguration* approach for avoiding this issue in future prototypes. This latter approach has two practical advantages: (a) it is more scalable due to the tree of C-elements that has a logarithmic latency with respect to the number of pipeline stages, and due to the lack of the multiplexor. (b) It is more flexible, as push and pop registers can acknowledge 'empty' marking states and data values when disabled. This enables such registers to be disabled in any order without affecting interconnections to their preset and/or postset. We tested the correctness of the distributed reconfiguration approach by functional simulation. However, the physical implementation and evaluation of this approach is left for future research, see Section 6.2.

We used a combinational control unit for managing the ctrl bus and controlling the



(a) High-level structure.

(b) Floorplan.

Figure 5.15: Ordinal pattern encoding chip.

pipeline depth (described in details in Section 5.1.3). The control registers shown in Figure 5.10 were not synthesised.

The presented library of components has been used to map our DFS model of the OPE pipeline to a digital circuit described in Verilog. Our implementation belongs to the class of *Quasi Delay Insensitive* asynchronous circuits [25]. However, engineers can design their own libraries of components, and implement other classes of circuits (e.g. delay insensitive), handshake protocols (e.g. bundled-data) and implementation styles (e.g. NCL-X) while being independent from circuit functionality. This is the main advantage of using a higher-level model such as the Dataflow Structures.

5.2.4 Evaluation of the fabricated prototype

Figure 5.15a shows the top-level schematic of the designed evaluation chip. It comprises two implementations of the OPE pipeline, *static* and *reconfigurable*, that are selected by the config input. The static implementation is a fixed 18-stage pipeline that can only compute the OPE with window size set to 18. The reconfigurable pipeline, on the other hand, supports 16 different depth settings, from 3 to 18 stages. Notice that the pipeline depth determines the OPE window size. The reconfigurable pipeline has an area overhead of $\simeq$26% in comparison to the static pipeline due to the extra control logic for dynamic reconfiguration.

The chip can be used in *normal* or *random* mode, as selected by the mode input. In

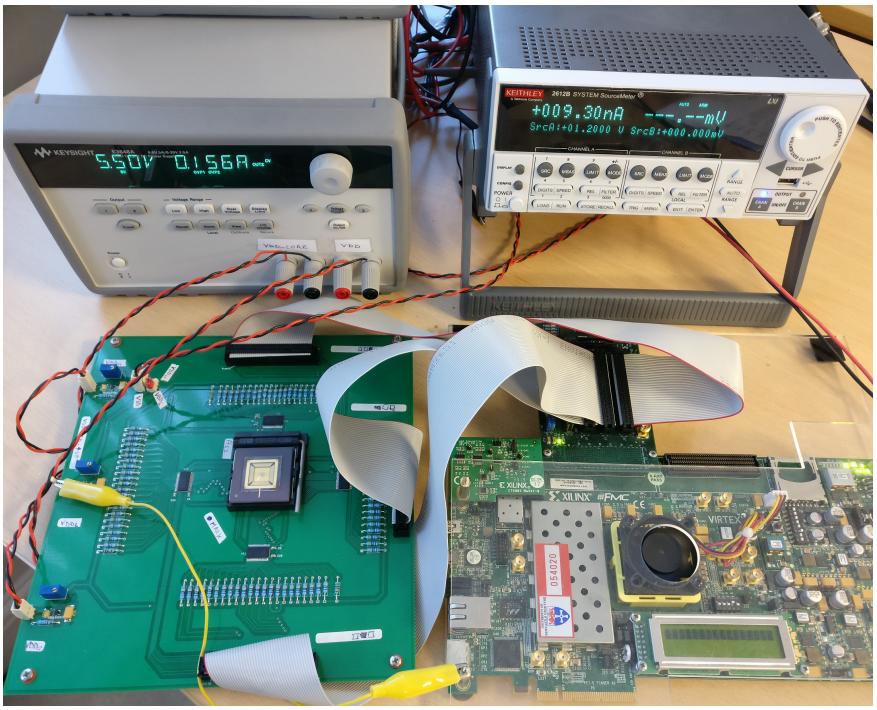
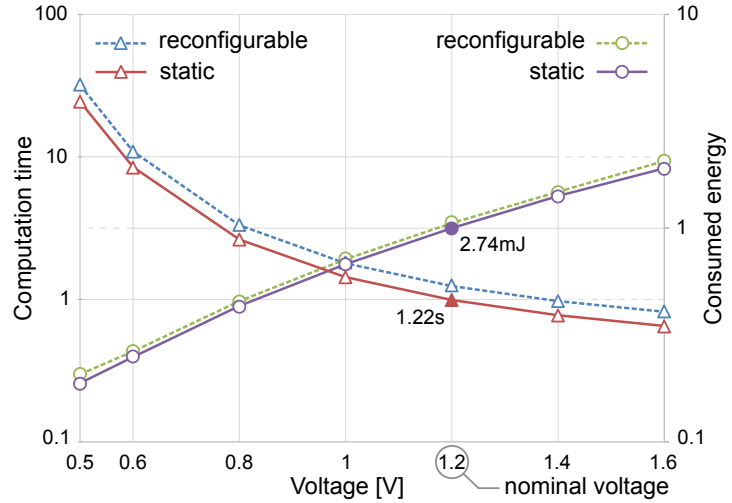


Figure 5.16: Testbench setup.

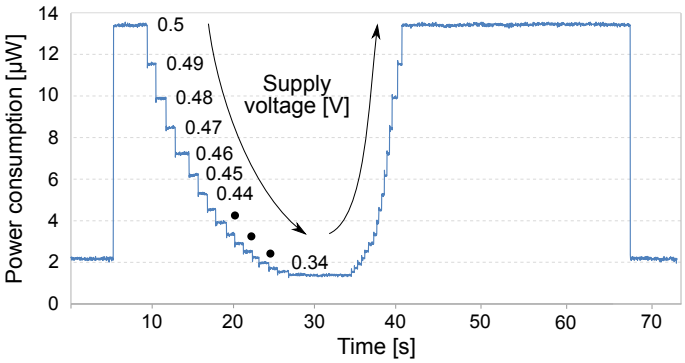
the normal mode, an input data stream is supplied via the in port and the results are produced at the out port at every iteration. In the random mode, a series of count random numbers is generated using a linear-feedback shift register (LFSR) based on a user-defined seed. A checksum of the output stream is calculated in the accumulator and a single data item is produced after all generated data is processed. This mode is essential for accurate measurements of the chip performance and energy consumption, as it removes the overheads for interfacing the chip to the testbench environment. The produced checksum is validated against the output of the OPE behavioural model initialised with the same seed and count parameters.

The chip floorplan and its main components are shown in Figure 5.15b. It was fabricated using TSMC 90nm CMOS technology for low-power applications via Europractice [111].

A custom test PCB was developed to interface the packaged chip with a Xilinx Virtex 7 FPGA board [112]. A series of experiments was run in the random mode for a stream of 16M LFSR-generated numbers, at supply voltages from 0.3V to 1.6V. The computation time was measured by the FPGA with 1ms precision, the power was monitored using



(a) Computation time and energy consumption at different voltages.



(b) Power consumption at changing supply voltage.

Figure 5.17: Experimental results for ordinal pattern encoding chip.

KEITHLEY 2612B SYSTEM source meter [113], with 1nW accuracy. The testbench setup is shown in Figure 5.16.

Experiments at varying voltage supply

The chip is fully asynchronous and can therefore operate in a wide range of voltages, dynamically adapting its speed. The computation time and energy consumption are characterised in Figure 5.17a for supply voltages from 0.5V to 1.6V. The length of the reconfigurable pipeline (dashed lines) is set to the maximum value and matches that of the 18-stage static pipeline (solid lines). Both the computation time and the consumed energy are normalised to the corresponding measurements of the static pipeline at the

nominal voltage of 1.2V (the reference values are 1.22s and 2.74mJ, respectively). As expected, the lower the voltage the slower, but at the same time more energy-efficient, is the circuit. The energy consumption of the reconfigurable implementation is slightly higher (5% overhead) due to the additional control logic for managing the pipeline configuration. The high computation time of the reconfigurable pipeline (36% overhead) is due to an inefficient implementation of the synchronisation between the stages using a daisy-chain C-element structure. This can be significantly improved (estimates overhead below 10%) using a tree-like C-element structure, see Section 5.2.3.

Another experiment demonstrates the capability of asynchronous pipelines to operate correctly at an unstable voltage supply, down to the near-threshold values. Figure 5.17b shows the power consumption of the reconfigurable OPE pipeline (with all 18 stages activated) during a single LFSR-generated experiment. At the very beginning (the left side of the graph), the voltage is set to 0.5V, the circuit does nothing, and the power consumption is due to the leakage current. Then, the up spike represents the beginning of the computation. Throughout the experiment, we gradually decreased the supply voltage down to 0.34V, which corresponds to the threshold voltage of the cells used for technology mapping (below this voltage the chip stops functioning as the difference between the low and high voltage levels is no more recognised). At this voltage level, the chip operation is frozen – all the gates within the digital circuit keep their logic state, the chip can be left at this voltage for hours with no progress being made. When the voltage is raised up again the circuit completes the remaining part of the computation (down spike) correctly.

Experiments at varying pipeline depth

All configurations of the reconfigurable pipeline (from 3 to 18 stages) were exercised and functionally verified at 0.5-1.6V. Figures 5.18, 5.19 and 5.20 describes the behaviour of the OPE chip under varying pipeline depth in terms of computation time, power consumption, and consumed energy, respectively. We show the analysis to the range of supply voltages $\{0.5, 0.8, 1.2, 1.6\}$ for improving the readability of the shown diagrams.

Figure 5.18 shows the computation time of the chip. When the pipeline depth is lower or equal than 6 stages (region 1), the computation time is limited by the accumulator (see

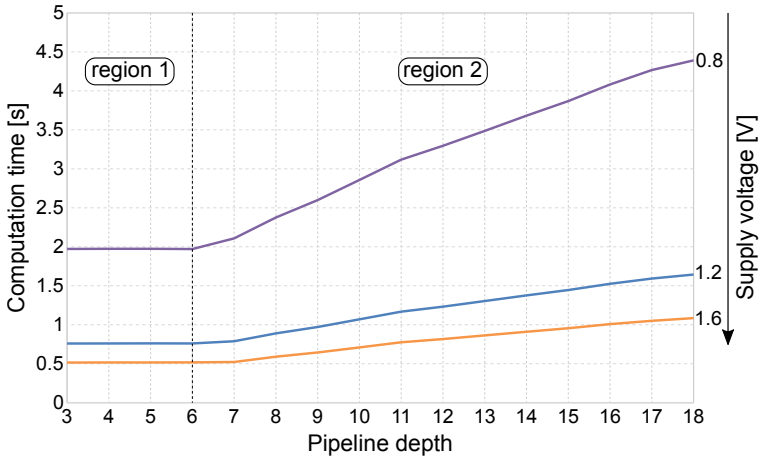


Figure 5.18: Computation time at varying voltages and pipeline depths.

Figure 5.15a), whose size is fixed by the maximum OPE result size. Reducing the number of stages below 6 does not reduce the computation time. In region 1, the computation time is only scaled down exponentially by the supply voltage reduction. On the other hand, when the pipeline depth is higher than 6 (region 2), the computation time of the chip is limited by the synchronisation time between the pipeline stages, whose delay is directly-proportional to the pipeline depth due to the daisy-chain of C-elements, see Section 5.2.3. In this region, the slope of the increment is reverse-proportional with the supply voltage. In this diagram, the experiments corresponding to the supply voltage of 0.5V are omitted for the sake of readability: the computation time is 19.4s in the region 1, and grows up to 42.4s when the pipeline depth is 18.

Figure 5.19 shows the power consumption of the chip. It increases with the pipeline depth up to the *minimum latency* point (which falls at 6/7 stages depending on the supply voltage), and subsequently decreases. The minimum latency point represents the situation in which the computation time of the accumulator and of the synchronisation time between pipeline stages are balanced, resulting in a higher throughput. In other words, data tokens can move faster within the pipeline, exercising more parts of the chip concurrently, and thus maximising the power consumption. Since the diagram scale does not allow to read the power consumption of the experiments corresponding to 0.5V supply voltage, we report for convenience the related data points in Table 5.5.

Figures 5.20a and **5.20b** show the energy consumption of the chip per computation

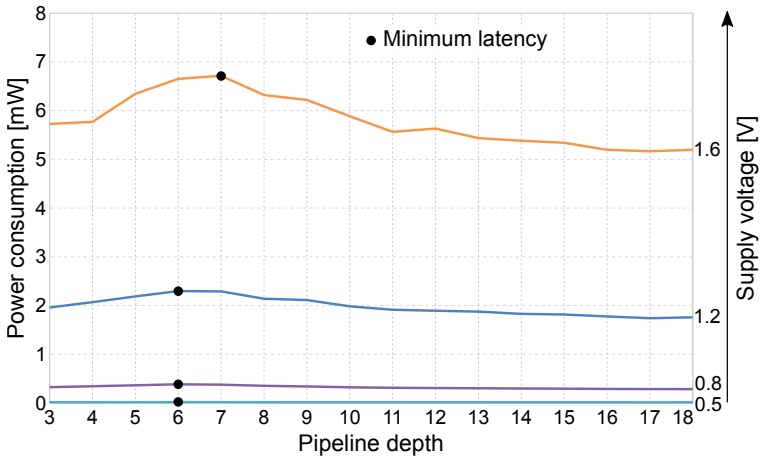


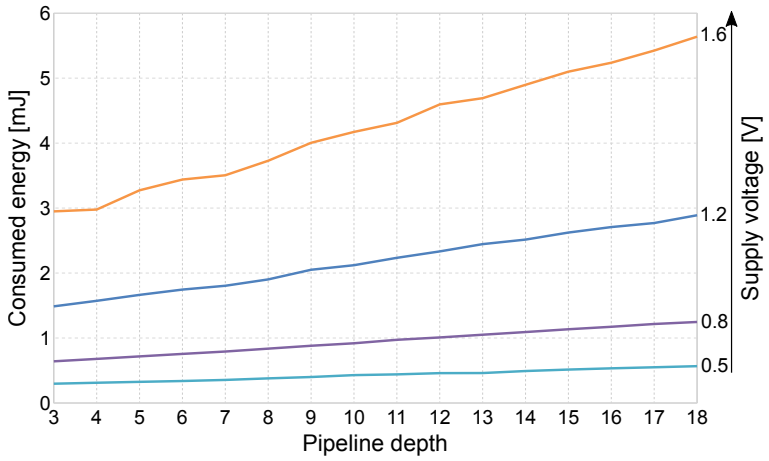
Figure 5.19: Power consumption at varying voltages and pipeline depths.

Pipeline depth	Power cons. [mW]	Pipeline depth	Power cons. [mW]
3	15.2	11	14.6
4	16.0	12	14.3
5	16.8	13	14.1
6	17.5	14	13.8
7	17.3	15	13.7
8	16.3	16	13.5
9	15.7	17	13.3
10	15.4	18	13.3

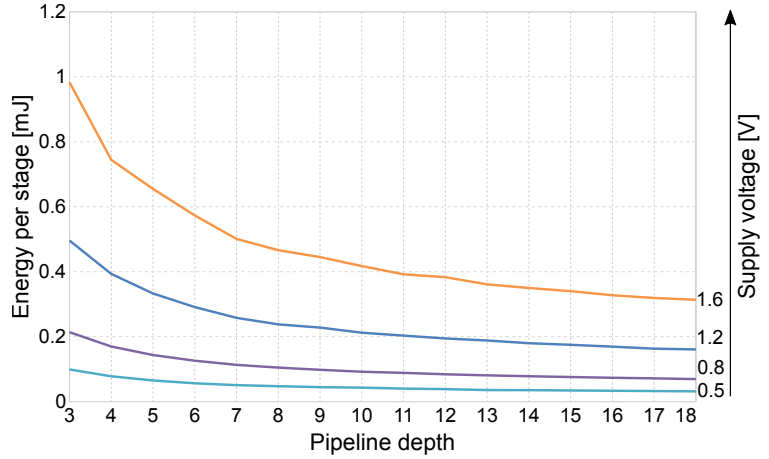
Table 5.5: Data points of the 0.5V line in Figure 5.19.

and per active stage, respectively. The former increases linearly with the depth of the pipeline, the slope of its increment is directly-proportional to the supply-voltage. The latter, in turn, decreases exponentially with the depth of the pipeline, i.e. the amount of work done per energy grows, resulting in a higher chip efficiency. Notice, however, that the pipeline depth is determined by the workload in applications that make use of OPE, rather than by the energy efficiency.

Figure 5.21 summarises these experimental results. It shows the computation time, power consumption and consumed energy at the nominal voltage 1.2V during four LFSR-generated experiments: with the pipeline depth set to 6, 10, 14 and 18. The power consumption is higher when the depth is set to 6 (minimum latency), and decreases with the increment of the pipeline depth. The computation time grows linearly with the increase of the pipeline depth (region 2 of Figure 5.18). The consumed energy is shown within the area delimited by the power consumption lines: one can see that it



(a) Energy per computation.



(b) Energy per computation per active stage.

Figure 5.20: Energy statistics at varying voltages and pipeline depths.

also increases with the pipeline depth. When the chip is waiting for the test to start, the power consumption is due to the leakage current of 15μW.

Summary of the results

The experiments demonstrate the high degree of flexibility and resilience of the fabricated OPE accelerator: it supports flexible window size (via reconfiguration of the pipeline depth) and can operate at a variable supply voltage (thanks to its asynchronous implementation). The cost of the reconfigurability is 5% in terms of power consumption

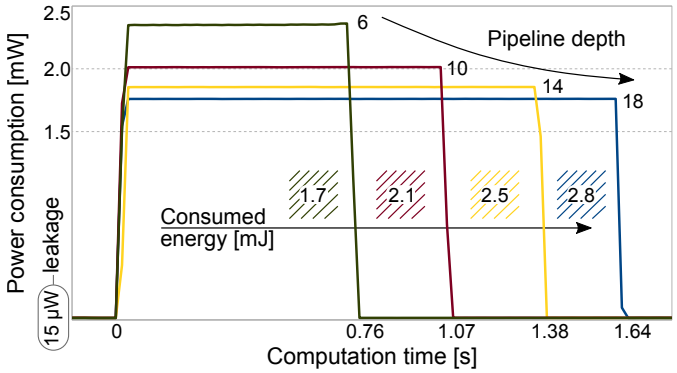


Figure 5.21: Time, power, and energy per computation at different pipeline depths at the nominal supply voltage 1.2V.

and 36% in terms of performance (can be improved to 10% in a future prototype).

5.2.5 Related work and conclusion

We described a methodology for designing and implementing *asynchronous reconfigurable pipelines*. While these have not been extensively studied in the past, there are a number of approaches available in literature for supporting other types of pipelines:

- *Synchronous pipelines* have been widely studied [114], and are supported by the mainstream EDA tools. As an example of a formal model for specifying, optimising and verifying reconfigurable synchronous pipelines see xMAS [115].
- *Asynchronous non-reconfigurable pipelines* have been also extensively studied. Engineers can rely on Static Dataflow Structures [25], reviewed in Section 3.4, for their design and optimisation. In [116], Nowick and Singh discuss how these pipelines can be implemented on a circuit using different styles.
- *Machine learning networks* are an important case of reconfigurable pipelines. These can be described by Google's TensorFlow [117], which combine Google's hardware *tensor processing units* to derive distributed processing systems.

To conclude, we summarise this research. We used the developed Dataflow Structures formalism to elaborate a general methodology for designing reconfigurable asynchronous pipelines. We showed a set of DFS-based pipeline stage templates that can be composed to achieve the desired circuit functionality. Afterwards, we proposed two

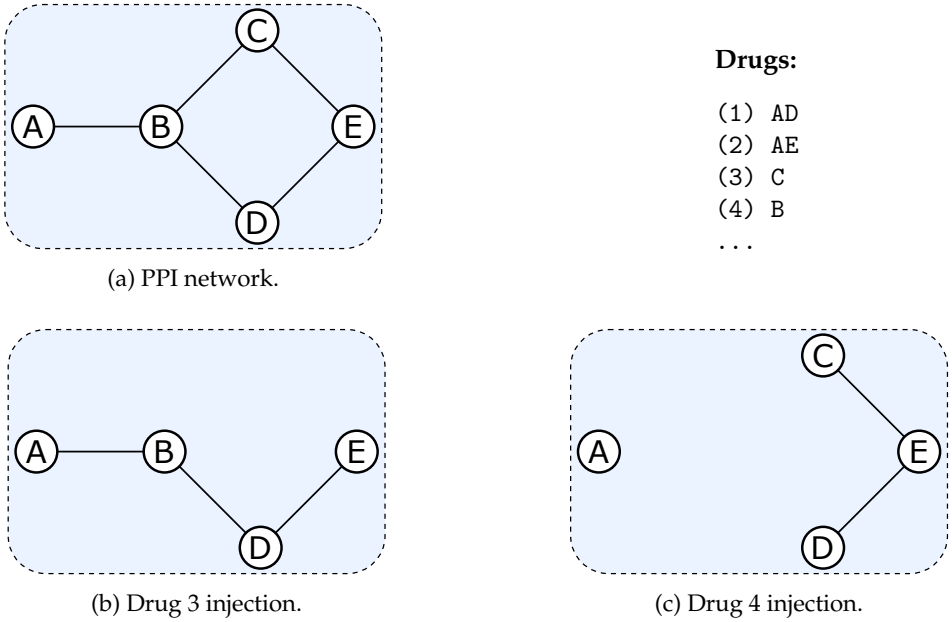
possible implementations for the available DFS set of nodes, which can be employed to implement Delay-Insensitive 4-phase dual-rail static and reconfigurable self-timed logic. We finally evaluated one of our proposed implementations by fabricating a reconfigurable accelerator for ordinal pattern encoding, and compared it to its static implementation counterpart for studying the characteristics of asynchronous dynamic reconfigurability. We relied on the conditional partial order graphs formalism for designing the reconfiguration control logic.

Our experimental results show that the reconfigurability overhead of our implementation is 26% in terms of area, 36% in terms of performance (10% expected for the improved presented implementation, whose evaluation is left for future research) and 5% in terms of power. The produced chip, as expected, is highly resilient to power supply variation, and can deliver a range of EQ implementation scenarios.

5.3 FPGA accelerator for drug discovery

So far, we validated the proposed scenario composition algorithm by designing different types of control architectures, and the presented Dataflow Structures formalism by designing a reconfigurable pipeline of an asynchronous processor. For these applications, we made use of *specification* and *implementation* scenarios. In this section, on the other hand, we consider the domain of network analysis and show that *activation scenarios* are instrumental to deal with large-scale systems, such as the networks in the domain of computational drug discovery.

This section is divided as follows. Section 5.3.1 follows up our motivational discussion (in Section 2.4) to introduce the reader to the area of computational drug discovery. Section 5.3.2 presents the developed accelerator that we prototyped on FPGA. Section 5.3.3 describes the scenario-based model of the drug discovery process, which we used to design part of the accelerator. Experimental results and final discussion are in Sections 5.3.4 and 5.3.5, respectively. Part of the content of this section has been published in [36–38].



(a) PPI network.

(b) Drug 3 injection.

(c) Drug 4 injection.

Figure 5.22: A PPI network, its possible drugs, and networks resulting from drug injection.

5.3.1 Introduction to computational drug discovery

Following the motivational discussion in Section 2.4, we give an example for introducing the reader to the domain of *computational drug discovery* [39], see Figure 5.22. For an in-depth technical introduction to the topic, we refer the interested reader to [39, 118].

Figure 5.22a shows an example of a protein-protein interaction (PPI) network [49] comprising 5 proteins $\{A, B, C, D, E\}$, which interact with each other according to the arcs that interconnect them. A PPI network is an undirected unweighted graph that models the functions of a biological system. On the right-hand side of the PPI network, four possible drugs are shown: e.g. drug 3, when injected into the network, disables protein C and leaves the network as in Figure 5.22b; drug 4, on the other hand, disables protein B and leaves the network as in Figure 5.22c. Intuitively, drugs, when injected in a biological system, modify its structure by attacking and disabling some of its proteins.

Real-life biological PPI networks are dense with interconnections: they can contain up to 20,000 nodes and 500,000 arcs. These networks are *resilient* to drug perturbation, i.e. the removal of one or more nodes does not modify the functionality of the system radically, as proteins are still able to interact with each others via the remaining paths and perform usual system functions. For example, the PPI network in Figure 5.22b, derived

by the removal of protein C, is still able to behave regularly as 4 out of 5 proteins can still interact to each others through the available arcs. On the other hand, the network obtained by the injection of drug 4 (see Figure 5.22c) is more disrupted, as A cannot interact with the remaining proteins due to the resulting lack of connections.

For quantifying the effectiveness of a drug x on a given PPI network G, the experts in this field rely on the so called *impact* measure (I_x), which is defined as:

$$I_x = \frac{|\theta_n(G) - \theta_0(G)|}{\theta_0(G)} \tag{5.1}$$

where θ_0 is the *Average Shortest Path* (ASP) of the *non-perturbed* PPI network (i.e. no drugs injected), and θ_n is the ASP of the PPI network where n proteins are removed. The ASP θ is widely used to measure network resilience [119], and it is defined as:

$$\theta(G) = \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{j \neq i}^{N} D(p_i, p_j) \tag{5.2}$$

where N is the number of proteins, and $D(p_i, p_j)$ is the distance between two proteins[2]. A drug x is said to be more effective than a drug y if: $I_x > I_y$. In the example in Figure 5.22, drug 4 is more effective than drug 3 as $I_4 = 0.59$ and $I_3 = 0.04$, i.e. drug 4 can better disrupt the functionality of the disease represented by the PPI network in Figure 5.22a.

In software, classifying each possible drug according to the shown impact measure is extremely time consuming for the following two reasons: (1) computing the average shortest path of a graph requires calculating the all-pairs shortest path of a network (for example by using the *breadth-first search* algorithm [52]), which is a computationally expensive task; (2) PPI networks cannot be stored entirely in processor cache due to their size. Thus, the processor has to communicate with the main (slow) memory multiple times to process a network.

To overcome these issues, we present a hardware-software infrastructure to accelerate the process of drug discovery. Our solution uses FPGAs as support for hardware processing, which has a number of advantages in comparison to other solutions that use many-core/cluster-based systems [120] or GPUs [121]:

[2] $D(p_i, p_j) = 0$ if p_j cannot be reached from p_i.

- FPGAs are more cost effective than other types of hardware supports.
- FPGAs allow a direct mapping between network elements and physical silicon structures (i.e. vertices can be mapped to flip-flops and edges to interconnect paths, as will be discussed shortly). This increases both the scale and performance of drug discovery analysis compared to other types of hardware supports that need more complex abstraction implementations for modelling PPI networks.
- FPGAs are programmable, so a single device can be used to analyse multiple networks, unlike ASICs. This is particularly useful if the underlying network is frequently updated (e.g. due to acquisition of new data).

The next section presents the prototyped accelerator for drug discovery.

5.3.2 The presented accelerator

General architecture

An architectural overview of the accelerator is shown in Figure 5.23. At the core, the accelerator consists of an *in silico* instance of a PPI network, synthesised by mapping vertices to individual memory elements (flip-flops) and edges to combinational paths between these elements, see Figure 5.24. The resulting hardware graph is encapsulated by the control circuitry to enable/disable selected vertices, coordinate computation, and read shortest-path computation results. An on-chip software processor (Nios II) is also included to communicate with the host computer and provide an Application Programming Interface (API) for drug discovery.

In our experiments, we used the *Altera DE4* FPGA board [122] as in the configuration shown in Figure 5.23, and the *Xilinx Virtex 7* board [112] (where the Nios II is substituted by the Microblaze software processor). Our experimental results on the two boards only differ by the utilisation factors of hardware resources, as the Xilinx board contains more resources. Thus, we will only show the results collected by means of the Altera board.

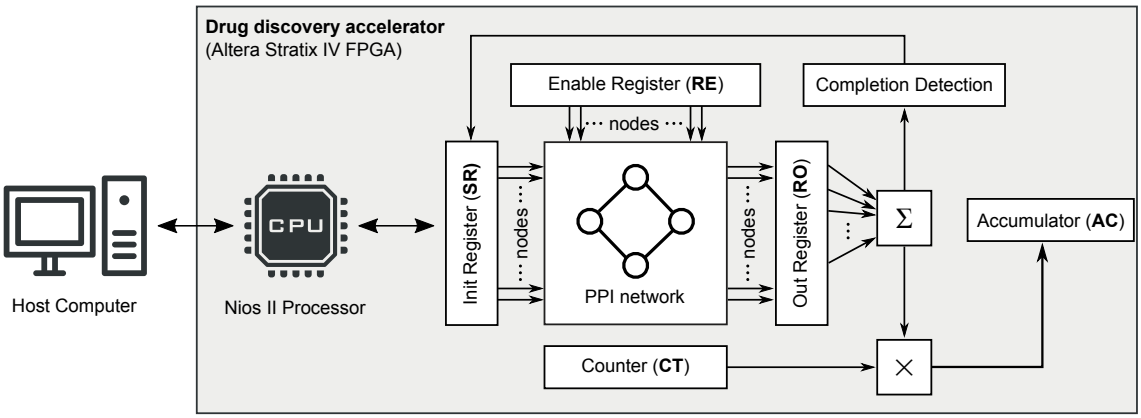


Figure 5.23: Overview of the hardware-software infrastructure for accelerating drug discovery.

Graph traversal in hardware

The basic idea behind representing graphs using flip-flops and combinational paths is that we wish to perform graph traversal by propagating logic high values between flip-flops. The logic state of each flip-flop therefore indicates whether a given vertex has been visited (logic high) or not (logic low). To propagate a visited state between flip-flops, we OR the outputs of all vertex neighbours and use it as an input to the vertex flip-flop. This mapping scheme is illustrated in Figure 5.24, and has been automated in the developed tool named FANTASI (i.e. FAst NeTwork Analysis in SIlicon) [123], where a PPI network (in the form of a XML-based file format) is used to generate the hardware representation of the graph and the architecture for its drug discovery analysis automatically.

Using this hardware representation, shortest path calculation from a starting vertex S is performed as follows. Initially, all vertex flip-flops except for S are reset (indicating an unvisited state). On the first clock cycle following the initial state, the visited state of S propagates to its immediate neighbours. The newly-visited vertices then propagate this state to their own neighbours in the following cycle and so on until the graph is fully traversed (i.e. when all vertices have been visited). In short, this computation is a classic breadth-first search where each iteration is performed in a clock cycle and involves visiting flip-flops by changing their state to logic high.

Note that the maximum number of clock cycles required to traverse the graph is equal to the graph *diameter* (i.e. the shortest distance between the two most distant vertices),

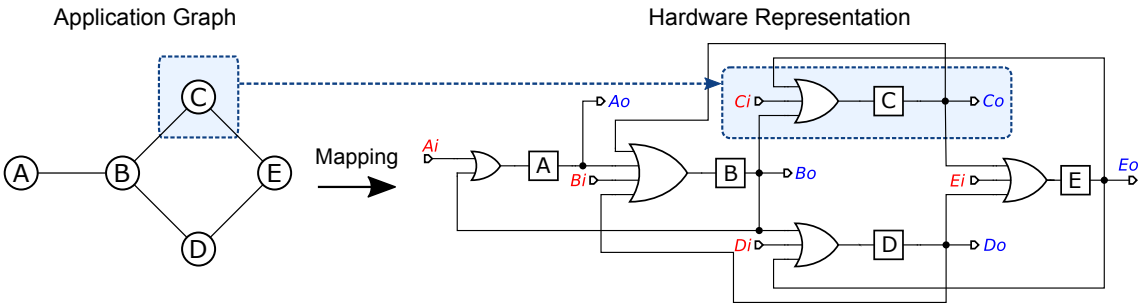


Figure 5.24: Mapping a graph to a digital circuit for implementing on an FPGA.

which is often very small for real-life graphs. For example, biological networks in our case study comprise thousands of vertices yet their diameter is typically around 5 due to the *small-world phenomenon* [124]. These networks can therefore be traversed in few clock cycles, which is faster than a single memory access on a commodity computer. This forms the basis for the significant acceleration factors reported in Section 5.3.4.

Calculating average shortest path

The accelerator is designed primarily to compute the average shortest path θ. For a better correspondence with hardware, we reformulate Equation 5.2 for computing θ as follows:

$$\theta(G) = \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{k=1} k \times C(v_i, k) \tag{5.3}$$

where $C(a,k)$ is the number of vertices at a distance k from vertex a. In this case the inner loop terminates when $C(v_i,k) = 0$ since this implies $C(v_i,h) = 0$ for $h > k$.

We now describe in more details how the accelerator computes θ, see Figure 5.23. The graph circuit interfaces with three registers: an initialisation shift register (**SR**), an enable register (**RE**) and an output register (**RO**). Register **SR** initialises vertex values at the beginning of each traversal while **RE** enables/disables selected vertices and **RO** detects which vertices have been visited during the current traversal step. Additionally, a counter **CT** maintains the step count during each traversal. Computing θ involves N traversals, each amounting to calculating the inner sum in Formula 5.3. During each step (of each traversal), the number of logic-high values in **RO** (vertices reached at a considered traversal step) is multiplied by **CT** and the result is added to an accumulator

AC. Each traversal is completed when **RO** $= 0$ (i.e. when no new vertices are visited). After N traversals, each starting from a different vertex, the value held in **AC** is read by the Microblaze processor and divided by $N(N-1)$ in software to obtain θ. Register **SR** initialises the graph in preparation for a traversal operation. As discussed earlier, all vertices except for the starting vertex S are initialised in an unvisited state. The value of **SR** is therefore a one-hot encoding of the index of S.

The accelerator is designed to simulate the process of drug injection, i.e. graph vertices can be disabled (modelling the proteins attacked by a drug) and θ can be recomputed in order to determine the impact of a drug on the network. Register **RE** provides this functionality; it is an N-bit register that can be prepopulated by the user (via API calls). Any 0 bit entries in this register will disable the corresponding vertices during the traversal process, effectively removing them from the graph. In Section 5.3.3, we describe alternative approaches to reconfigure PPI networks for simulating drug injections, and explain the reasons for which we opt for the one described in this paragraph.

The accelerator is controlled by a host computer; computations are started, monitored and their results are read via API calls. This provides a programmatic interface enabling the accelerator to be used as a step in an automatic quantitative workflow involving manipulating a base graph via vertex removal and evaluating the impact of drugs by re-calculating θ. Using the developed tool [123], an input graph can be converted into VHDL code and synthesised into an FPGA within the accelerator framework. Therefore, developers can read a graph description, synthesise and implement the accelerator, and then use it to analyse the graph, all while remaining within the same programming environment.

5.3.3 The scenario-based model of drug discovery

As anticipated in Section 2.4, one can model a PPI network and its drugs as activation scenarios and use them to derive control architectures for simulating the process of drug injection. In this section, we describe a CPOG-based approach to model this process, which inspired the choice of the described approach to network reconfiguration.

The goal of computational drug discovery is to select a set of *promising* drug candidates (i.e. which have a high impact on a PPI network) to be tested biologically

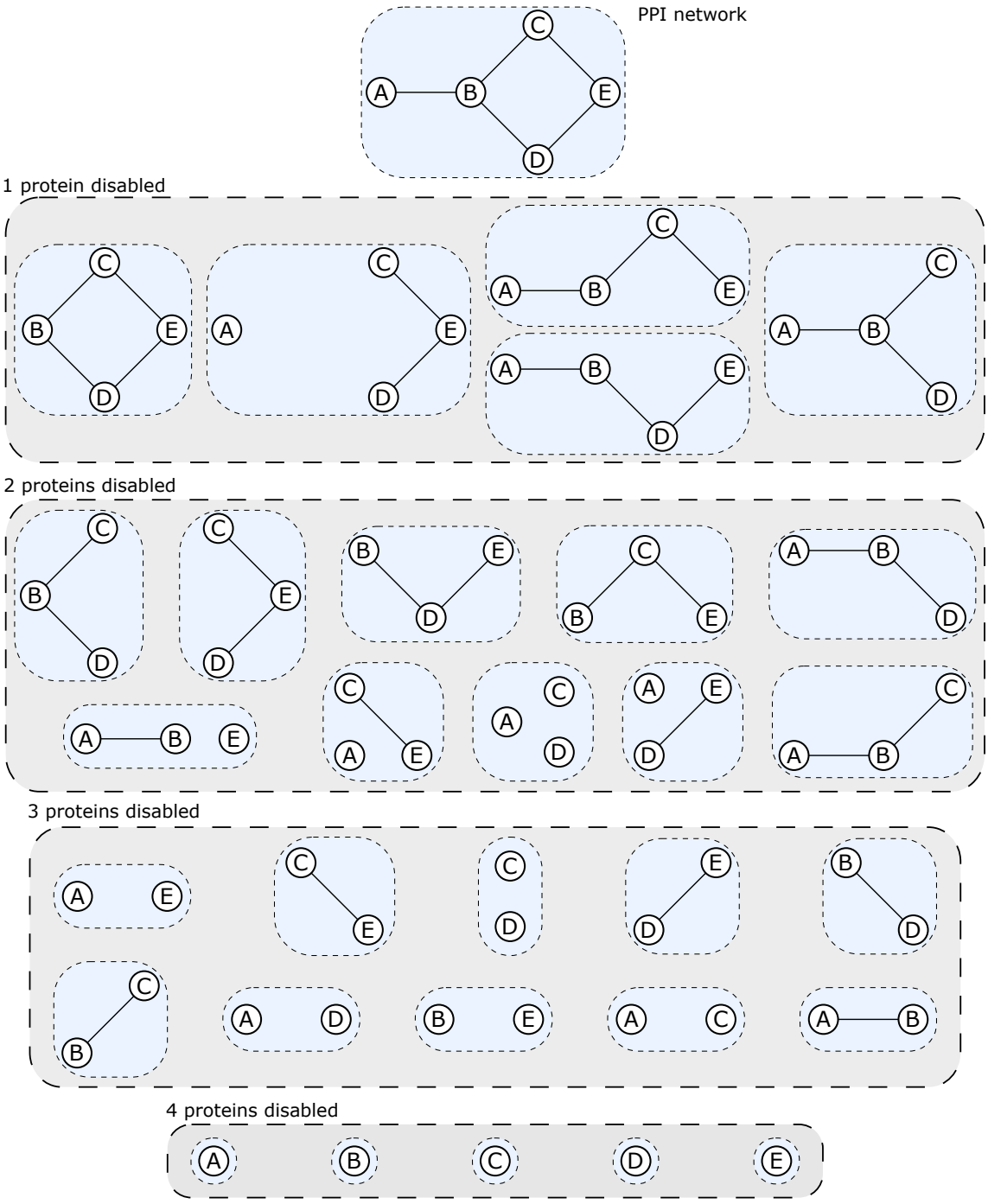
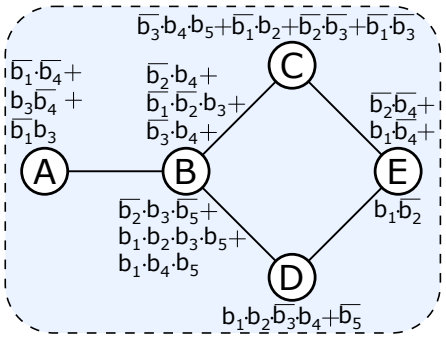
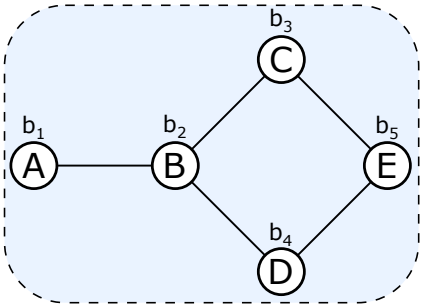


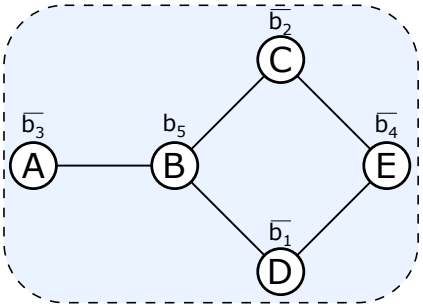
Figure 5.25: A PPI network (on top), and all its possible drugs in the form of activation scenarios.



(a) The heuristic composition technique.



(b) Single-literal composition.



(c) SAT-based composition.

Figure 5.26: The reconfiguration structure of the random-based drug discovery as CPOGs.

in wet-labs. This moves part of the workload for testing drugs to the earlier process of automated selection, and results in a higher drug discovery efficiency [50].

There are several approaches for the selection of promising drugs. The trivial approach is a *random-based* technique, and consists of disabling random subsets of proteins on a network (to simulate the process of drug injection) and calculating their corresponding impact. In this approach, one might want to be able to inspect all possible drugs (= all combinations of disabled proteins). As an example, Figure 5.25 shows the PPI network used in the previous sections (on top), and all its possible drugs grouped in terms of the number of proteins that they disable. Every drug is represented as an activation scenario, which includes the proteins and interactions that are still active after its injection.

Using the CPOG automated composition features, it is possible to derive hardware controllers for reconfiguring PPI networks. Figure 5.26 shows three CPOGs derived by three of the composition techniques that we compared in Section 5.1, where:

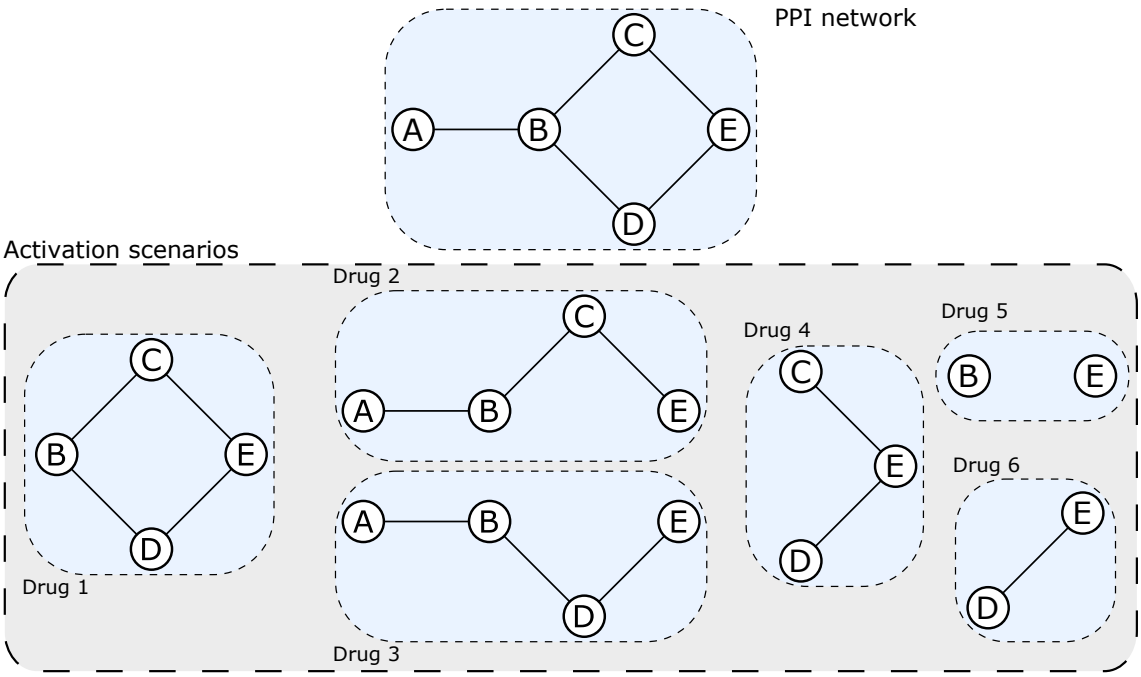
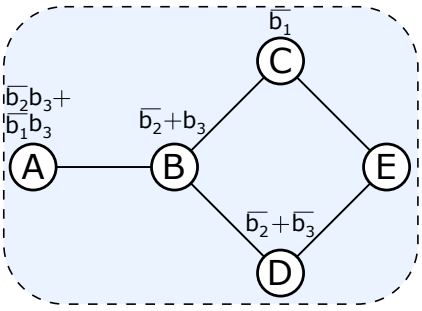


Figure 5.27: A PPI network (on top), and a set of *activations scenarios*: the networks that can be induced by a list of realistic drug candidates.

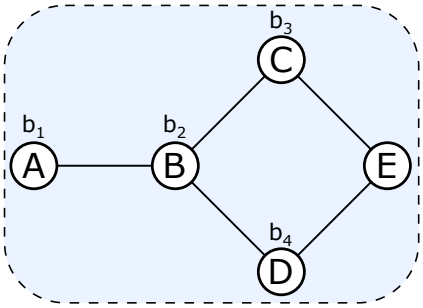
- Proteins are enabled if the paired Boolean *conditions* are satisfied, otherwise they are disabled.
- The network structure is determined by the injected drugs, which are encoded by the *codes* selected during the scenario composition step.

Our proposed heuristic composition algorithm does not find an efficient CPOG, as the conditions for controlling the proteins are not trivial (see Figure 5.26a) and introduce unnecessary complexity for reconfiguring the PPI network at runtime. On the other hand, the single-literal and the SAT-based algorithms (see Figures 5.26b and 5.26c, respectively) find more efficient CPOGs that have conditions composed of one independent variable per protein, even though the SAT-based solution has some of these variables inverted. These two latter algorithms are better suited to this random-based type of analysis, where proteins need to be switched on and off in any combination to enable one to simulate all internal activation scenarios of a network.

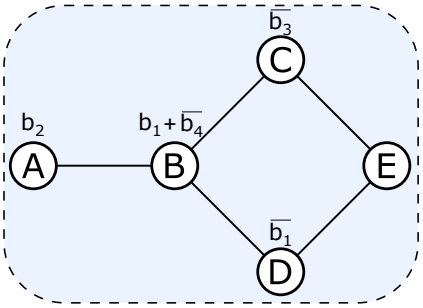
A different approach to the selection of promising candidates is the so called *drug library-based*. In this approach, libraries of *realistic* drug candidates – i.e. drugs that



(a) The proposed composition technique.



(b) Single-literal composition.



(c) SAT-based composition.

Figure 5.28: The reconfiguration structure of the library-based drug discovery as CPOGs.

are known to be realistically available and injectable in a system as they do not entail dangerous side effects, for example – are simulated and their drugs ranked in terms of their impact. As an example of a library of realistic drugs, see Figure 5.27.

Using the previous scenario composition techniques, we derived the three CPOGs in Figure 5.28. Notice that there are no drugs in the library that affect protein E, which is in fact always enabled in the shown CPOGs. The single-literal composition technique, whose CPOG is in Figure 5.28b, is the most area effective as proteins are controlled by independent variables and are enabled (disabled) by a logic 1 (0). The proposed and the SAT-based CPOGs, on the other hand, need a further combinational layer to be reconfigured according to the library of drugs in Figure 2.6.

The CPOGs derived by several low-scale PPI networks, such as the ones shown above, inspire the design of the hardware reconfiguration approach implemented in the prototype, which has the following characteristics:

- Reconfiguration features are provided for all proteins (as in Figure 5.26b). This enables the accelerator to simulate the injection of any drug and frees the PPI

network hardware mapping from the dependency with the used drug library (as many of these libraries are available for a PPI network). This is also important because the number of possible scenarios explodes exponentially for large-scale graphs, making thus infeasible to synthesise optimal reconfiguration structures without relying on simpler solutions that allow to potentially execute all of such scenarios without knowing them upfront.

- Each protein (modelled as a flip-flop) is controlled by an independent variable. This enables to encode a drug x by a code of size N (where N is the number of proteins of the network), with $x(i) = 1$ for all proteins i that are not affected by x, and $x(i) = 0$ otherwise. This is instrumental from the point of view of a user, who only needs to know the index of a protein to investigate its effects on a network.

- A drug is injected via software API calls (by populating the ER register by a drug code). This enables users of the accelerator to implement custom drug discovery algorithms in software. For example, one could implement the *hub-proteins based* approach, where proteins with high degree of interconnections are disabled first as they are more likely to have a higher impact on a system.

In addition, this approach to network reconfiguration is simple to implement, as the enable input of each flip-flop (protein) has to be only connected to its corresponding activation variable (a bit of the ER register).

5.3.4 Experimental results

We evaluate the developed drug discovery accelerator by synthesising and analysing six protein interaction networks on the Altera DE4 FPGA board[3]. The networks are used to evaluate and compare the effects of different drug candidates on complex cellular systems. The impact of each drug is evaluated by disabling selected vertices that the drug is known to perturb and then calculating θ. Our motivation was: (i) to compare the accelerator performance to a software implementation, and (ii) to test the scalability of our hardware prototype using benchmarks from an industrial application. Our

[3]The experimental results on the Xilinx Virtex 7 only differs by resource utilisation factors, and are thus not shown.

Table 5.6: Resource Utilisation and Performance Comparison for Six Protein to Protein Interaction Network Benchmarks on the Altera DE4 board (FPGA: Stratix IV EP4SGX230).

The **Resource Utilisation of the Network** entries show the resources used by the hardware representation (HW) of the considered PPI network only. Our biggest benchmark $n5$ cannot be synthesised into the FPGA, thus some of the table entries are missing (see $-$) and others were estimated (see values followed by $*$).

The **Resource Utilisation of the Prototype** shows the amount of resources used by the entire drug discovery prototype. The volume of logic utilization added by the extra control circuitry and the NIOS II software processor is not negligible, but it is not the cause of the network $n5$ synthesis failure. In fact, the Altera tool QUARTUS also fails in the attempt of synthesizing only $n5$.

The **Operating Parameters** show the power consumption of the prototype in the FPGA as estimated by the Altera tool *PowerPlay Power Analyser*. It also shows the maximum working frequency at which each network can be clocked (calculated without the extra accelerator logic), and the frequency that we fixed for the prototype. The prototype frequency and *processing cycles* (i.e. number of cycles needed to calculate θ) are used to determine prototype performance.

Finally, the **Performance** part of the table shows the obtained acceleration figures relative to a software implementation in C++. See Section 5.3.4 for further details on the experimental results.

Network	**n0**	**n1**	**n2**	**n3**	**n4**	**n5**
Vertices	3	15	87	349	1628	3487
Edges	2	42	804	6456	53406	115898
Resource Utilisation of the Network						
Dedicated registers (FFs)	3	15	87	349	1628	3487
Lookup Tables (LUTs)	3	29	250	1541	11054	24878
Logic Utilisation	1%	1%	1%	1%	8%	18%*
Average Interconnect Usage	$\sim$0%	$\sim$0%	$\sim$0%	0.4%	6%	16%*
Peak Interconnect Usage	$\sim$0%	$\sim$0%	1%	18%	79%	95%*
Resource Utilisation of the Prototype						
Dedicated registers (FFs)	1272	1362	1753	3089	9510	18850
Lookup Tables (LUTs)	1347	1453	1954	4192	18114	38759
Logic Utilisation	1%	1%	1%	3%	12%	–
Average Interconnect Usage	0.6%	0.6%	0.7%	1%	11%	27%*
Peak Interconnect Usage	15%	15%	18%	31%	90%	120%*
Operating Parameters						
Power Consumption (mW)	956	956	961	993	1238	–
Network Frequency (MHz)	$>$1000	$>$1000	426	195	122	–
Prototype Frequency (MHz)	100					
Processing Cycles (per network)	32	215	1206	4793	22772	48371
Performance						
Software Throughput (networks/sec)	10^5	$>10^4$	1176	56	3	$\sim$0.88
FPGA Throughput (networks/sec)	$>10^6$	$>10^5$	82918	20863	4391	2067*
Acceleration Factor	$\sim$10$\times$	$\sim$10$\times$	70$\times$	372$\times$	1463$\times$	2349$\times$*

benchmarks ranged from very small test networks to considerably large real-life PPI networks (3,487 vertices, 115,898 edges).

Table 5.6 summarises accelerator resource utilisation and performance for the six PPI networks. In the **Resource utilisation** side of the table, the *Dedicated Registers* instantiated on FPGA for the hardware representation of the networks match the number of vertices of the PPI networks, meaning that every protein is successfully mapped to a flip-flop register as expected (see Figure 5.24). We found that our biggest benchmark $n5$ cannot be synthesised into the FPGA. This is due to the high degree of *Edges* (protein interactions) of this network, which is not matched by the number of planar FPGA interconnections (see *Peak Interconnect Usage* entries). Many real-world networks have comparable degrees of connectivity and so we expected the scalability of our hardware implementation to be upper-bounded by FPGA interconnect density. Notice that the Altera tool QUARTUS could not provide an estimate of the *Logic Utilization* factor of the prototype that encapsulates $n5$. This factor provides an indication of how full the FPGA device is, and it is calculated during the *Place & Route* task of the fitting step of the synthesis process [125]. The failure might be an indication that also the synthesis tool capabilities fail in the attempt to synthesize such a large and highly interconnected biological network. Nevertheless, $n5$ is the largest within its class of proteins interaction networks used at our industrial partner e-Therapeutics, and so our hardware implementation and choice of FPGA device is sufficient for this particular application.

The **Operating Parameters** part of the table summarises the maximum working frequency found by the Altera QUARTUS tool for the HW networks (see *Network Frequency* values); the working frequency of the prototype (including the software processor) that we fixed to 100 Mhz (see *Prototype Frequency* value); and the number of cycles needed to calculate the average shortest path of each network (see *Processing Cycles* values). The network frequency found by the tool reduces when synthesising larger networks. This is because neighbouring vertices had to be mapped to more distant flip-flops to accommodate the entire network and the worst-case propagation delay had to be increased accordingly. Another effect of increasing network scale is that the number of cycles to calculate θ also increases since the all-pairs shortest path computations have

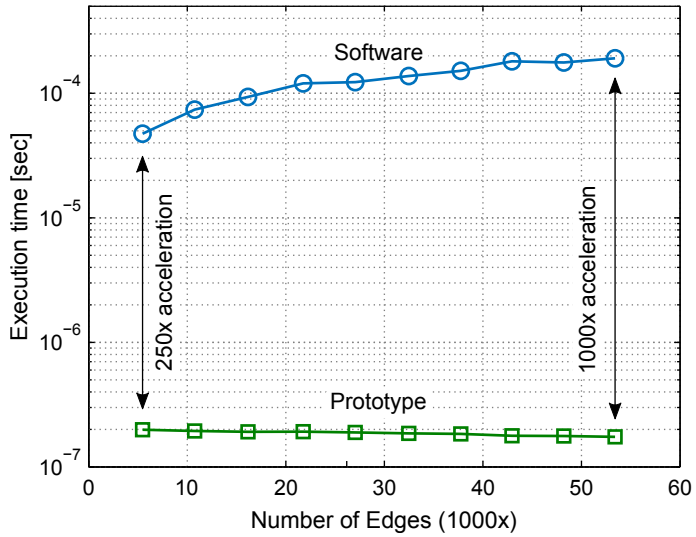


Figure 5.29: Execution time of a analysis run on the network $n4$ at varying number of edges.

to be repeated for a larger number of vertices.

In the **Performance** part of the table, we compare our reference software implementation (single-threaded implementation of the breadth-first search algorithm written in C++, running on Intel i7-6700HQ 2.60GHz CPU, 16GB RAM, 6MB cache) to the developed prototype (using 100 *Mhz* as working frequency). Notice that the acceleration of the prototype encapsulating $n5$ was estimated assuming that its frequency meets the fixed 100 *Mhz*. The throughput of average path calculations using our prototype was higher by 1-3 orders of magnitude depending on the network size. Even though larger networks require more cycles to calculate θ, the relative acceleration in comparison to the software reference was higher. This is a trend that we expected: our approach scales much better with respect to network size compared to a software implementation. This is because the diameter of the network decreases in bigger networks due to the increase of interconnection density. The performance benefit is therefore more prominent when processing larger networks.

This trend can be also observed in Figure 5.29, where we show the execution time of 10 different θ calculations on the network $n4$. In each of these calculations, a different number of edges (decided randomly) is enabled, from 10% to 100%. When the number of edges increases, the runtime of the software implementation also increases due to the higher number of internal paths that need to be inspected. Conversely, the runtime of

the prototype decreases since the higher number of edges reduces the diameter of the network. This is the cause of the higher acceleration, which increases from $\simeq 250\times$ (10% of the edges enabled) to more than $1000\times$ (100% of the edges enabled).

5.3.5 Related work and conclusion

The idea of using FPGAs to accelerate graph processing is not new. However, to the best of our knowledge, the existing approaches are all based on the Von Neumann architecture of storing a graph in the memory, and then operating on the graph via a set of processing cores that read and write to the memory. These approaches achieve acceleration by relying on memory structures that exploit the flexibility of FPGA [126, 127], or on cores optimised to perform specific types of graph calculations [128, 129]. Our approach is not general purpose as it can only traverse graph and compute average shortest paths, but it achieves much higher acceleration factors.

Another attempt to accelerate computational drug discovery is in [130], where the author describes two algorithms for processing PPI networks (1) on GPUs: graphs are divided into strongly connected sub-graphs whose average shortest paths are calculated concurrently using the Bellman-Ford algorithm [52]; and (2) on FPGAs: the problem of calculating average shortest paths (implemented by the breadth-first search algorithm [52]) is transformed to a matrix vector multiplication problem via linear algebra and solved on FPGA. Again, these approaches are constrained by the Von Neumann approach, and achieve an acceleration of less of $5\times$.

We presented a prototype for accelerating the process of drug discovery. The purpose of the prototype is to calculate average shortest paths of protein-protein interaction networks very efficiently for estimating the impact that drugs have on biological systems. In our approach, PPI networks are mapped to a hardware representation where proteins are modelled by flip-flops, and protein interactions by combinational paths between such flip-flops. PPI networks can be dynamically reconfigured, i.e. flip-flops can be disabled at runtime for simulating the injection of drugs. The design of the described reconfiguration structure has been supported by a scenario-based model of the process of drug discovery that relies on activation scenarios.

The experimental results highlight a consistent acceleration compared to a reference

approach for calculating average shortest paths in software. The acceleration is higher as PPI networks become bigger and denser of interconnections. The reason of such a big acceleration is due to the fact that PPI networks are mapped to silicon without any form of abstraction, thus each flip-flop can act concurrently and fully exploit hardware parallelism.

Chapter 6

Conclusions

This thesis presents new approaches and formal models for tackling the design of microelectronic systems using high-level scenarios. The proposed approaches have been applied to real-life case studies for evaluating the maturity of the considered scenario-based methodologies.

In this chapter, we summarise the main ideas developed and described along the thesis. Section 6.1 summarises key contributions, experimental results and limitations of the considered approaches. Section 6.2 outlines the future research areas that we believe are worth to investigate further.

6.1 Summary of the contributions

Scenarios for control synthesis with CPOGs: in Section 4.1, *we presented an algorithm for composing scenarios into efficient implementations, and applied this approach to the composition of Partial Orders into Conditional Partial Order Graphs*. The proposed algorithm uses similarities between scenarios to drive the scenario encoding process, which affects the characteristics of the final system implementation. Unlike existing composition approaches, the proposed solution supports *composition constraints*, which enable engineers to specify custom encodings and reuse existing IP blocks. In Section 5.1, we applied the proposed composition technique on an extensive set of benchmarks that

includes control architectures for different applications. Compared to existing scenario composition algorithms, the proposed technique finds equally good results (in terms of size of the derived implementations, and synthesis runtime) when dealing with small-scale benchmarks, and scales better when the number of scenarios of a system grows. Our experiments also highlight that the CPOG-methodology can achieve equally good results (in terms of size of derived implementations, synthesis runtime and supported models) in comparison to the classic FSM-based methodology, and better results than the STG-based methodology, which typically targets small-scale asynchronous controllers. The main current limitation of the methodology based on CPOGs resides at the scenario specification phase. Currently, scenarios of systems have to be specified by hand, which may be a long and error-prone process.

Scenarios for asynchronous design with DFSs: in Section 4.2, *we presented the new Dataflow Structures formal model, which extends the existing Static Dataflow Structures formalism with an additional set of nodes that can be used for modelling dynamic reconfiguration in asynchronous circuits*. We described formally the behavioural semantics of the new formalism, and showed how to translate DFS models to Petri nets, which enables designers to reuse the wide range of existing back-end tools for automated hardware synthesis and verification. In Section 5.2, we used the presented DFS formalism to elaborate a general methodology for modelling and implementing reconfigurable asynchronous pipelines. We also validated the methodology by fabricating an asynchronous dataflow accelerator for the ordinal pattern encoding. The chip implements both a *reconfigurable* (i.e. whose number of active stages can be selected at runtime) and a static pipeline; it was used (1) to characterise the overhead (in terms of area, power and performance) of asynchronous dynamic reconfigurability, and (2) validate the proposed Dataflow Structures formalism. Experimental results highlight that the implemented reconfigurable pipeline has a 26% area overhead, a relatively low energy consumption overhead (5%) but a fairly large speed overhead (36%) in comparison to the static pipeline. We described an alternative implementation for overcoming this issue. As expected, the energy consumption and speed of the chip have a quadratic dependency with the voltage supply. We also characterised the reconfigurable pipeline under a variable number of active stages. The data collected shows that the power consumption

depends on the amount of computation performed concurrently. Even though the energy consumption of the pipeline increases when more stages are active, the 'energy per stage' factor decreases. The current methodology is based on the presented DFS behavioural semantics, which can only be applied to circuits functioning with the 4-phase handshake protocol. The formalisation of newer semantics are left for future research.

Decomposing concurrent systems into scenarios with PWs: in Section 4.3, *we presented the new Process Windows formal model, which provides automated decomposition features to extract scenarios (or windows) from complex concurrent system specifications*. Every extracted window models a part of a system. Windows can interact with each other to model the behaviour of the whole system. In order to show that Process Windows can simplify the understanding of complex specifications, we applied the proposed methodology to low-scale examples in the context of asynchronous digital design, and process mining. The characteristics of the model have to be further characterised by an extensive benchmarking phase, which we leave for future research.

Scenarios for graph processing: in Section 5.3, *we presented a hardware accelerator for processing graphs coming from the domain of computational drug discovery, and prototyped it on FPGA*. Unlike existing approaches, a graph is not stored in memory but it is a digital circuit itself that can be simulated on silicon to collect data. The accelerator is application specific, i.e. it is designed to compute the all-pairs shortest paths of protein interaction networks, but it achieves much higher acceleration factors in comparison to other general-purpose solutions (two or three orders of magnitude). We described the process of computational drug discovery with a model based on scenarios (relying on CPOGs). This was essential to design the reconfiguration structure for disabling network nodes at runtime, and potentially being able to analyse any subgraph of the original graph. This approach is constrained by the amount of FPGA resources: very large networks including hundreds of thousands of interconnections cannot be synthesised on a single FPGA (see our biggest benchmark $n5$ in Section 5.3.4).

6.2 Future work

To further improve the **CPOG methodology** a number of recommendations for future research are given. (1) Automating the extraction of scenarios from high-level languages. Research on a specification language capable of extracting scenarios is currently ongoing [64]. (2) Parallel implementation of the presented composition algorithm, which is important to further improve the efficiency of scenario composition by exploring more solutions at no extra runtime cost. (3) Support for x-aware scenario encoding (with x being latency, power, energy, and other characteristics), which is important for making the methodology attractive to many practical domains.

The presented **DFS methodology** can be also improved with: (1) description of additional behavioural semantics for other popular asynchronous protocols, see [25]. (2) Development of a domain-specific language for describing reconfigurable dataflow pipelines textually. (3) Evaluation of the described *distributed reconfiguration* approach as implementation for reconfigurable asynchronous pipelines, and comparison to the *reconfigurable interconnections* approach that has been evaluated in this work.

Additional work can be done also for the presented **Process Windows methodology**. In this work, we showed how to enforce a set of properties for deriving choice-free scenarios. However, other types of properties (application dependent) can derive structurally different scenarios. Also, the developed algorithm for window decomposition have to be tested on a wider set of benchmarks in order to evaluate its applicability to real-life specifications.

Finally, the presented **prototype for drug discovery** can also be enhanced in a few different aspects. (1) The accelerator is meant to compute the average shortest path of encapsulated networks. It might be extended to enable other types of network-based calculation, e.g. network centrality. (2) The generation of network reconfiguration hardware structure can be automated by means of the CPOG synthesis features. This might be beneficial for some networks where internal parts are not to be disabled. (3) Supporting large graphs that do not fit in one FPGA [38].

Appendix A

The scenario-based specification of the ARM Cortex M0+ processor

The appendix shows (see Figure A.1) and describes (see below description) the scenario-based specification of the ARM Cortex M0+ processor. It consists of 11 scenarios that have been derived by analysing the instruction set architecture reference manual of the ARMv6-M [131], and modelling by hand sets of instructions that share similar functionality and addressing modes. The presented scenario specification models 61 out of 68 processor instructions. Below, we briefly describe the function and structure of every scenario.

Scenario 1 covers the *LDR (reg.)* instruction. It loads a specified register with a word located in memory, whose address is obtained from two registers (ALU $\rightarrow$ MAU). Afterwards, the next instruction is fetched (IFU).

Scenario 2 covers the *NOP* instruction, which increments the PC (PCIU $\rightarrow$ PCIU/2) and fetches the next instruction (IFU).

Scenario 3 covers the *POP* instruction. It loads multiple memory locations into specified registers (MAU), and then fetches the next instruction in the program memory (IFU).

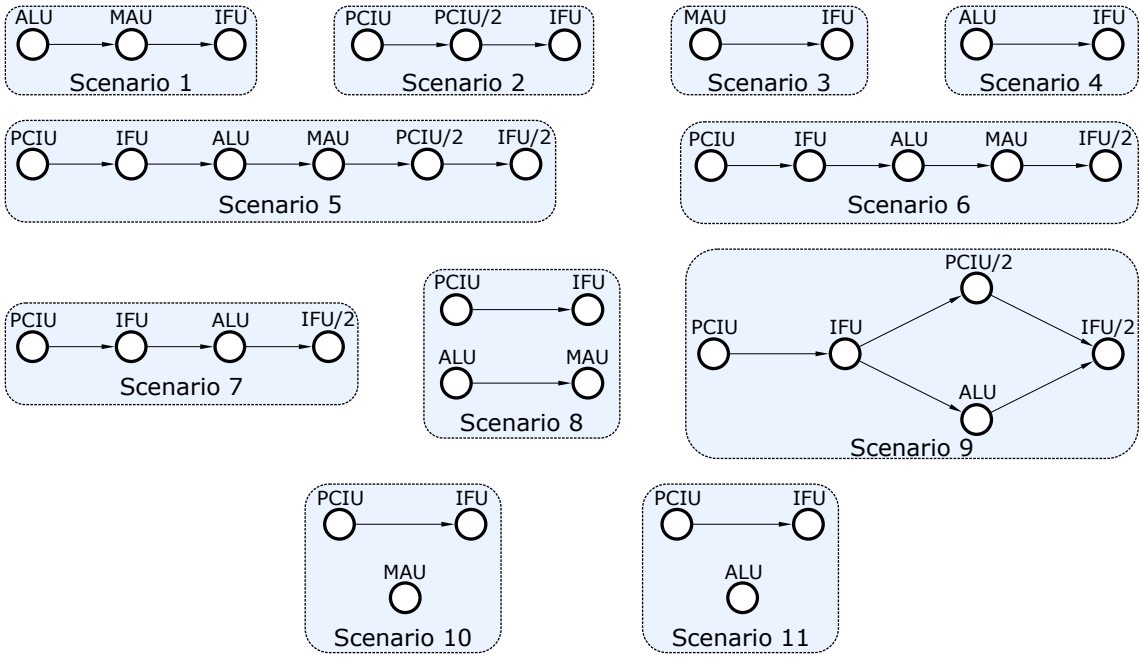


Figure A.1: The scenario specification of the ARMv6-M instruction set architecture.

Scenario 4 covers the branch with link and exchange (*BLX*) and branch and exchange (*BX*) instructions. They both branch to an address specified in a register R, conditionally changing the processor state (*ARM* or *Thumb* states). The ALU operation moves the content of R to PC. In the case of a *BLX* instruction, ALU also stores the PC content into the link register before branching.

Scenario 5 covers the load and store instructions with immediate addressing mode: *LDR (imm.)* and *STR (imm.)*. An immediate value is fetched from the IR (PCIU $\rightarrow$ IFU), and is used to compute the memory address for the load/store operation (ALU $\rightarrow$ MAU). Afterwards, the next instruction is fetched from the program memory (PCIU/2 $\rightarrow$ IFU/2).

Scenario 6 covers the *LDR (lit.)*. A immediate value is fetched from the IR (PCIU $\rightarrow$ IFU), and is added to the PC to compute the memory address of the word to be loaded into a specified register (ALU $\rightarrow$ MAU). Afterwards, the next instruction is fetched from the program memory IFU/2.

Scenario 7 covers the unconditional branch instruction *B*. The branch offset is specified as an immediate value, and is fetched from the IR (PCIU $\rightarrow$ IFU). Afterwards, the target branch is computed and moved into the PC, and the next instruction is fetched (ALU $\rightarrow$ IFU/2).

Scenario 8 covers the load and store instructions with register addressing mode: *LDR (reg.)* and *STR (reg.)*. Unlike the Class 5, the memory operation is executed (ALU $\rightarrow$ MAU) without the need for an immediate value, the fetch of the next instruction (PCIU $\rightarrow$ IFU) is therefore executed concurrently.

Scenario 9 covers arithmetical, logical and data transfer instructions with immediate addressing mode, e.g. *ADD (imm.)*, *LSR (imm.)*. An immediate value is fetched from the IR (PCIU $\rightarrow$ IFU), which is used for the selected operation (ALU). The latter can be executed in parallel with the program counter incrementation (PCIU/2), needed for fetching the next instruction from the memory (IFU/2).

Scenario 10 covers memory access instructions where multiple registers and memory locations are involved, e.g. *PUSH*, *LDM*, *STM*. Data transfer is meant to be done by the MAU operation, executed concurrently with the fetch of the next instruction (PCIU $\rightarrow$ IFU).

Scenario 11 covers arithmetical, logical and data transfer instructions with register addressing mode, i.e. *CMP (reg.)*, *MOV (reg.)*. The next instruction is fetched (PCIU $\rightarrow$ IFU) concurrently with the selected operation (ALU).

Bibliography

[1] R. R. Schaller, "Moore's Law: Past, Present, and Future", *IEEE Spectr.*, vol. 34, pp. 52–59, June 1997.

[2] R. Brayton and J. Cong, "Electronic design automation past, present, and future", July 2009. Available online at `cadlab.cs.ucla.edu/nsf09/NSF_Workshop_Report.pdf`, last accessed: 20/09/2017.

[3] Calma, "GDSII stream format manual", February 1987. Documentation No.: B97E060, release 6.0, available online at `bitsavers.informatik.uni-stuttgart.de/pdf/calma/GDS_II_Stream_Format_Manual_6.0_Feb87.pdf`, last accessed: 20/09/2017.

[4] C. Mead and L. Conway, *Introduction to VLSI Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1979.

[5] A. Sangiovanni-Vincentelli, "The tides of EDA", *IEEE Design Test of Computers*, vol. 20, pp. 59–75, Nov 2003.

[6] H. P. Sharangpani and M. L. Barton, "Statistical analysis of floating point flaw in the pentium processor", November 1994. Intel Corporation – Available online at `http://users.minet.uni-jena.de/~nez/rechnerarithmetik_5/fdiv_bug/intel_white11.pdf`, last accessed: 03/10/2017.

[7] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of

inefficiency in general-purpose chips", in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 37–47, ACM, 2010.

[8] D. Gajski, A.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, and P. Bricaud, "Essential issues for IP reuse", in *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pp. 37–42, IEEE, 2000.

[9] M. J. A. Sexton, "History of Intel Chipsets". Tom's Hardware. Available online at: `www.tomshardware.com/picturestory/784-intel-chipset-history.html`, last accessed: 22/09/2018.

[10] M. Wolf, "The Physics of Event-Driven IoT Systems", *IEEE Design & Test*, vol. 34, pp. 87–90, April 2017.

[11] C. LaFrieda, B. Hill, and R. Manohar, "An Asynchronous FPGA with Two-Phase Enable-Scaled Routing", in *Proceedings of the 2010 IEEE Symposium on Asynchronous Circuits and Systems*, ASYNC '10, (Washington, DC, USA), pp. 141–150, IEEE Computer Society, 2010.

[12] M. Rykunov, *Design of Asynchronous Microprocessor for Power Proportionality*. PhD thesis, Newcastle University, 2013.

[13] Sun Microsystems, "Datasheet UltraSPARC IIIi processor". Available online at `http://datasheets.chipdb.org/Sun/UltraSparc-IIIi.pdf`, last accessed: 13/10/2017.

[14] D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, and A. Yakovlev, "Benefits of asynchronous control for analog electronics: Multiphase buck case study", in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1751–1756, March 2017.

[15] A. D. Brown, J. E. Chad, R. Kamarudin, K. J. Dugan, and S. B. Furber, "SpiNNaker: Event-Based Simulation - Quantitative Behavior", *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, pp. 450–462, July 2018.

[16] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing", in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 105–117, June 2015.

[17] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future", *IEEE Design Test of Computers*, vol. 26, pp. 18–25, July 2009.

[18] "IEEE Standard System C Language Reference Manual", *IEEE Std 1666-2005*, pp. 01–423, 2006.

[19] D. J. Greaves, "Layering RTL, SAFL, Handel-C and Bluespec constructs on Chisel HCL", in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 108–117, Sept 2015.

[20] J. Beaumont, A. Mokhov, D. Sokolov, and A. Yakovlev, "Compositional design of asynchronous circuits from behavioural concepts", in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 118–127, Sept 2015.

[21] A. Alekseyev, *Compositional Approach to Design of Digital Circuits*. PhD thesis, Newcastle University, June 2014.

[22] G. Birkhoff, *Lattice Theory*. No. v. 25, pt. 2 in American Mathematical Society colloquium publications, American Mathematical Society, 1940.

[23] A. Mokhov, *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, 2009.

[24] A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, and A. Romanovsky, "Synthesis of Processor Instruction Sets from High-Level ISA Specifications", *IEEE Transactions on Computers*, vol. 63, pp. 1552–1566, June 2014.

[25] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[26] D. Sokolov, I. Poliakov, and A. Yakovlev, "Analysis of static data flow structures", *Fundamenta Informaticae*, vol. 88, pp. 581–610, 2008.

[27] A. de Gennaro, P. Stankaitis, and A. Mokhov, "A Heuristic Algorithm for Deriving Compact Models of Processor Instruction Sets", in *2015 15th International Conference on Application of Concurrency to System Design*, (Brussels, Belgium), pp. 100–109, June 2015.

[28] A. de Gennaro, P. Stankaitis, and A. Mokhov, "Efficient composition of scenario-based hardware specifications", *IET Computers Digital Techniques*, vol. 13, no. 2, pp. 57–69, 2019.

[29] D. Sokolov, A. de Gennaro, and A. Mokhov, "Reconfigurable asynchronous pipelines: From formal models to silicon", in *18th Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1562–1567, March 2018.

[30] A. de Gennaro, D. Sokolov, and A. Mokhov, "Design and Implementation of Asynchronous Reconfigurable Pipelines", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. To be submitted.

[31] A. Mokhov, J. Cortadella, and A. de Gennaro, "Process Windows", in *17th International Conference on Application of Concurrency to System Design (ACSD)*, pp. 86–95, June 2017.

[32] The tool SHUTTERS for Process Windows synthesis. GitHub repository: `https://github.com/tuura/shutters`.

[33] D. Sokolov, V. Khomenko, and A. Mokhov, "Workcraft: Ten years later", in *This asynchronous world. Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pp. 269–293, 2016. Available online: `async.org.uk/ay-festschrift/paper25-Alex-Festschrift.pdf`.

[34] The WORKCRAFT design environment. GitHub repository: `https://github.com/workcraft/workcraft`, see also the reference website: `www.workcraft.org`.

[35] C. Guo, W. Luk, and S. Weston, "Pipelined reconfigurable accelerator for ordinal pattern encoding", in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pp. 194–201, June 2014.

[36] A. Mokhov, A. de Gennaro, G. Tarawneh, G. Lukyanov, S. Mileiko, J. Scott, A. Yakovlev, and A. Brown, "Language and Hardware Acceleration Backend for Graph Processing", in *International Conference on Application of Concurrency to System Design*, (Verona, Italy), IEEE, 2017.

[37] A. Mokhov, A. de Gennaro, G. Tarawneh, G. Lukyanov, S. Mileiko, J. Scott, A. Yakovlev, and A. Brown, "Language and hardware acceleration backend for graph processing", in *Languages, Design Methods, and Tools for Electronic System Design - Selected Contributions from FDL 2017*, Springer, 2018. In Press.

[38] A. Brown, D. Thomas, J. Reeve, G. Tarawneh, A. de Gennaro, A. Mokhov, M. Naylor, and T. Kazmierski, "Distributed Event-based Computing", in *Parallel Computing '17 (ParCo)*, Advances in Parallel Computing, 2017. In press.

[39] M. P. Young, S. Zimmer, and A. V. Whitmore, "Chapter 3. Drug Molecules and Biology: Network and Systems Aspects", in *RSC Drug Discovery* (J. R. Morphy and C. J. Harris, eds.), pp. 32–49, Royal Society of Chemistry.

[40] A. Mokhov, A. Alekseyev, and A. Yakovlev, "Encoding of processor instruction sets with explicit concurrency control", *IET Computers Digital Techniques*, vol. 5, pp. 427–439, November 2011.

[41] D. R. Schertz, "Fault-Tolerant Computing: An Introduction", *IEEE Transactions on Computers*, vol. C-23, pp. 649–650, July 1974.

[42] M. Alioto, "Energy-quality scalable adaptive VLSI circuits and systems beyond approximate computing", in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 127–132, March 2017.

[43] V. K. Chippa, S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing: An integrated hardware approach", in *2013 Asilomar Conference on Signals, Systems and Computers*, pp. 111–117, Nov 2013.

[44] A. Doblander, A. Maier, B. Rinner, and H. Schwabach, "Improving fault-tolerance in intelligent video surveillance by monitoring, diagnosis and dynamic reconfiguration", in *Third International Workshop on Intelligent Solutions in Embedded Systems, 2005.*, pp. 194–201, May 2005.

[45] P. Wang, J. Zhang, and Z. Chang, "Fault Tolerance of Multiprocessor-Structured Control System by Hardware and Software Reconfiguration", in *2007 International Conference on Mechatronics and Automation*, pp. 3745–3749, Aug 2007.

[46] G. Chang, S. Maity, B. Chatterjee, and S. Sen, "A MedRadio Receiver Front-End With Wide Energy-Quality Scalability Through Circuit and Architecture-Level Reconfigurations", *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, pp. 369–378, Sept 2018.

[47] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenarios in system development: current practice", *IEEE Software*, vol. 15, pp. 34–45, Mar 1998.

[48] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks", *Rev. Mod. Phys.*, vol. 74, pp. 47–97, Jan 2002.

[49] W. Zhang *et al.*, "Network pharmacology: A further description", *Network Pharmacology*, vol. 1, no. 1, pp. 1–14, 2016.

[50] e-Therapeutics story, `https://www.etherapeutics.co.uk/our-story/`, last accessed: 15/01/2018.

[51] The *Partially Order Event Triggered Systems* (POETS) project website. Available online: `https://poets-project.org/`. Last accessed: 11/09/2018.

[52] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[53] A. Mokhov and A. Yakovlev, "Conditional Partial Order Graphs: Model, Synthesis, and Application", *IEEE Transactions on Computers*, vol. 59, pp. 1480–1493, Nov 2010.

[54] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers", *IEICE TRANSACTIONS on Information and Systems*, vol. E80-D No.3, pp. 315–325, 1997.

[55] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic synthesis of asynchronous controllers and interfaces*. Springer, Jan 2002.

[56] C. A. Petri, *Kommunikation mit Automaten*. PhD thesis, Universitt Hamburg, 1962.

[57] T. Murata, "Petri Nets: Properties, analysis and applications", *Proceedings of the IEEE*, pp. 541–580, Apr. 1989.

[58] L. Y. Rosenblum and A. Yakovlev, "Signal Graphs: From Self-Timed to Timed Ones", in *International Workshop on Timed Petri Nets*, (Washington, DC, USA), pp. 199–206, IEEE Computer Society, 1985.

[59] T. A. Chu, *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, 1987. MIT Laboratory for Computer Science.

[60] A. Arnold, *Finite Transition Systems: Semantics of Communicating Systems*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1994.

[61] A. Mokhov and V. Khomenko, "Algebra of Parameterised Graphs", *ACM Trans. Embed. Comput. Syst*, vol. 13, pp. 143:1–143:22, July 2014.

[62] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with ISA-Formal", in *International Conference on Computer Aided Verification*, pp. 42–58, Springer, 2016.

[63] K. E. Gray, P. Sewell, C. Pulte, S. Flur, and R. Norton-Wright, "The Sail instruction-set semantics specification language", Technical report published by Cambridge University, 2017.

[64] G. Lukyanov and A. Mokhov, "Concurrency Oracles for Free", in *Proceedings of the Algorithms & Theories for the ANalysis of Event Data 2018 Workshop*, 2018.

[65] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice", *Artificial Intelligence*, vol. 187, pp. 52 – 89, 2012.

[66] N. Eén and N. Sörensson, *An Extensible SAT-solver*, pp. 502–518. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[67] R. Hamming, "Error detecting and error correcting codes", *The Bell System Technical Journal*, vol. 29, pp. 147–160, April 1950.

[68] The Conditional Partial Order Graph tool SCENCO. GitHub repository: `https://github.com/tuura/scenco`, see also the WORKCRAFT website: `workcraft.org/help/encoding_plugin`.

[69] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing", *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

[70] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vicentelli, "ESPRESSO-SIGNATURE: a new exact minimizer for logic functions", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, pp. 432–440, Dec 1993.

[71] B. L. Synthesis and V. Group, "ABC, a system for sequential synthesis and verification". Website available at: `eecs.berkeley.edu/~alanmi/abc/`, last accessed: 24/04/2019.

[72] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Sis: A system for sequential circuit synthesis", Tech. Rep. UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[73] A. Mokhov, M. Rykunov, D. Sokolov, and A. Yakovlev, "Design of Processors with Reconfigurable Microarchitecture", *Journal of Low Power Electronics and Applications*, vol. 4, no. 1, pp. 26–43, 2014.

[74] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.

[75] C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[76] V. Khomenko, M. Koutny, and A. Yakovlev, "Logic synthesis for asynchronous circuits based on Petri net unfoldings and incremental SAT", in *Proceedings. Fourth International Conference on Application of Concurrency to System Design, 2004. ACSD 2004.*, pp. 16–25, June 2004.

[77] V. Khomenko, "A usable reachability analyser". Technical Report, CS-TR-1140, Newcastle University, 2009.

[78] A. Bouakaz, P. Fradet, and A. Girault, "A Survey of Parametric Dataflow Models of Computation", *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, pp. 38:1–38:25, Jan. 2017.

[79] J. B. Dennis, J. P. Fosseen, and L. J. P., "Data Flow Schemas", Massachussets Institute of Technology, 1972. Available online at `https://link.springer.com/content/pdf/10.1007/3-540-06720-5_15.pdf`.

[80] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, 1993. AAI9431898.

[81] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow", *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, Sept 1987.

[82] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, "BPDF: A statically analyzable dataflow model with integer and boolean parameters", in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 1–10, Sept 2013.

[83] A. Saifhashemi and P. A. Beerel, "Observability Conditions and Automatic Operand-Isolation in High-Throughput Asynchronous Pipelines", in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation* (J. L. Ayala, D. Shang, and A. Yakovlev, eds.), (Berlin, Heidelberg), pp. 205–214, Springer Berlin Heidelberg, 2013.

[84] Wojcik and K.-Y. Fang, "On the Design of Three-Valued Asynchronous Modules", *IEEE Transactions on Computers*, vol. C-29, pp. 889–898, Oct 1980.

[85] J. de San Pedro and J. Cortadella, "Mining Structured Petri Nets for the Visualization of Process Behavior", in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, (New York, NY, USA), pp. 839–846, ACM, 2016.

[86] E. Best and R. Devillers, "Characterisation of the state spaces of live and bounded marked graph petri nets", in *Language and Automata Theory and Applications* (A.-H. Dediu, C. Martín-Vide, J.-L. Sierra-Rodríguez, and B. Truthe, eds.), (Cham), pp. 161–172, Springer International Publishing, 2014.

[87] E. Best and R. Devillers, "Synthesis of Bounded Choice-Free Petri Nets", in *26th International Conference on Concurrency Theory (CONCUR 2015)* (L. Aceto and D. de Frutos Escrig, eds.), vol. 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 128–141, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

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

[89] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev, "Deriving petri nets from finite transition systems", *IEEE Transactions on Computers*, vol. 47, pp. 859–882, Aug 1998.

[90] E. Boros and P. L. Hammer, "Pseudo-boolean optimization", *Discrete Applied Mathematics*, vol. 123, no. 1, pp. 155 – 225, 2002.

[91] The tool CATS for Process Windows extraction. GitHub repository: `https://github.com/upc-eda/Cats`.

[92] T. Philipp and P. Steinke, "Pblib – a library for encoding pseudo-boolean constraints into cnf", in *Theory and Applications of Satisfiability Testing – SAT 2015* (M. Heule and S. Weaver, eds.), (Cham), pp. 9–16, Springer International Publishing, 2015.

[93] D. Harel and P. S. Thiagarajan, "UML for Real", ch. Message Sequence Charts, pp. 77–105, Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[94] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Berlin, Heidelberg: Springer-Verlag, 2003.

[95] D. Fahland, "Oclets – scenario-based modeling with petri nets", in *Applications and Theory of Petri Nets* (G. Franceschinis and K. Wolf, eds.), (Berlin, Heidelberg), pp. 223–242, Springer Berlin Heidelberg, 2009.

[96] A. Polyvyanyy, M. La Rosa, C. Ouyang, and A. H. Hofstede, "Untanglings: A Novel Approach to Analyzing Concurrent Systems", *Form. Asp. Comput.*, vol. 27, pp. 753–788, Nov. 2015.

[97] M. Koutny and B. Randell, "Structured Occurrence Nets: A Formalism for Aiding System Failure Prevention and Analysis Techniques", *Fundam. Inf.*, vol. 97, pp. 41–91, Jan. 2009.

[98] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st ed., 1994.

[99] V. Khomenko, M. Koutny, and A. Yakovlev, "Detecting state coding conflicts in stg unfoldings using sat", in *Third International Conference on Application of Concurrency to System Design, 2003. Proceedings.*, pp. 51–60, June 2003.

[100] P. Kurup and T. Abbasi, *Logic synthesis using Synopsys*. Kluwer Academic Publishers, 1995.

[101] A. Mokhov, J. Carmona, and J. Beaumont, *Mining Conditional Partial Order Graphs from Event Logs*, pp. 114–136. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016.

[102] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 203–215, Feb 2007.

[103] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji, "A Comparative Study on ASIC, FPGAs, GPUs and General Purpose Processors in the $O(N^2)$ Gravitational N-body Simulation", in *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 447–452, July 2009.

[104] L. S. Nielsen and J. Sparsø, "Designing asynchronous circuits for low power: an IFIR filter bank for a digital hearing aid", *Proceedings of the IEEE*, vol. 87, pp. 268–281, Feb 1999.

[105] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee, "The design of an asynchronous MIPS R3000 microprocessor", in *Proceedings Seventeenth Conference on Advanced Research in VLSI*, pp. 164–181, Sept 1997.

[106] L. F. Cristfoli, A. Henglez, J. Benfica, L. Bolzani, F. Vargas, A. Atienza, and F. Silva, "On the comparison of synchronous versus asynchronous circuits under the scope

of conducted power-supply noise", in *2010 Asia-Pacific International Symposium on Electromagnetic Compatibility*, pp. 1047–1050, April 2010.

[107] K. M. Fant and S. A. Brandt, "Null convention logic^{TM}: a complete and consistent logic for asynchronous digital circuit synthesis", in *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP '96*, pp. 261–273, Aug 1996.

[108] R. Sovani, K. Haque, and P. Beckett, "Short word length NULL convention logic FIR filter for low power applications", in *2015 IEEE International WIE Conference on Electrical and Computer Engineering (WIECON-ECE)*, pp. 102–105, Dec 2015.

[109] A. Vakil, K. P. Jayadev, S. Hegde, and D. Koppad, "Comparitive analysis of null convention logic and synchronous CMOS ripple carry adders", in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pp. 1–5, Feb 2017.

[110] T. E. Williams, *Self-timed Rings and Their Application to Division*. PhD thesis, Stanford, CA, USA, 1991. UMI Order No. GAX92-05744.

[111] Europractice IC website, "*TSMC 90nm technology overview*". `https://www.europractice-ic.com/technologies_TSMC.php?tech_id=90nm`, last accessed: 04/07/2018.

[112] Xilinx Virtex-7 FPGA VC707 Evaluation kit. `https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html`, Last accessed: 28/06/2018.

[113] Keithley 2612B system source meter data sheet. `www.testequipmentdepot.com/keithley/pdfs/2600b_datasheet.pdf`, last accessed: 04/07/2018.

[114] Arvind and D. E. Culler, "Annual Review of Computer Science Vol. 1, 1986", ch. Dataflow Architectures, pp. 225–253, Palo Alto, CA, USA: Annual Reviews Inc., 1986.

[115] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification", *IEEE Design Test of Computers*, vol. 29, pp. 80–88, June 2012.

[116] S. M. Nowick and M. Singh, "High-Performance Asynchronous Pipelines: An Overview", *IEEE Design Test of Computers*, vol. 28, pp. 8–22, Sept 2011.

[117] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". White paper, Google Research, 2016.

[118] T. Ideker and R. Sharan, "Protein Networks in Disease", vol. 18, no. 4, pp. 644–652.

[119] P. Crucitti, V. Latora, M. Marchiori, and A. Rapisarda, "Efficiency of scale-free networks: error and attack tolerance", *Physica A: Statistical Mechanics and its Applications*, vol. 320, no. C, pp. 622–642, 2003.

[120] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing", in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1–11, Nov 2012.

[121] P. Harish and P. J. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA", in *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, (Berlin, Heidelberg), pp. 197–208, Springer-Verlag, 2007.

[122] Altera DE4 Development Board. `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=501`, Last accessed: 28/06/2018.

[123] FANTASI tool. GitHub repository: `https://github.com/tuura/fantasi`.

[124] J. Kleinberg, "The Small-world Phenomenon: An Algorithmic Perspective", in *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, (New York, NY, USA), pp. 163–170, ACM, 2000.

[125] Intel Corporation, "How do I interpret the Logic Utilization number reported in the Quartus II Fitter report?". Available online: `https://www.altera.com/support/support-resources/knowledge-base/solutions/rd05172012_146.html`, last accessed: 24/07/2018.

[126] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: Graphlet counting case study", in *2011 International Conference on Field-Programmable Technology*, pp. 1–8, Dec 2011.

[127] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martnez, and C. Guestrin, "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation", in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 25–28, May 2014.

[128] N. Kapre, "Custom FPGA-based soft-processors for sparse graph acceleration", in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 9–16, July 2015.

[129] M. Lin, I. Lebedev, and J. Wawrzynek, "High-throughput Bayesian Computing Machine with Reconfigurable Hardware", in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, (New York, NY, USA), pp. 73–82, ACM, 2010.

[130] A. Grivas, *High Performance Graph Analysis on Parallel Architectures*. PhD thesis, Newcastle University, 2016.

[131] ARMv6-M Architecture Reference Manual. ARM DDI 0419C (ID092410). 2010.