Newcastle University, UK

School of Electrical and Electronic Engineering (EEE)

Compositional Circuit Design with Asynchronous Concepts

PhD Thesis

Jonathan Richard Beaumont

March 2018

Abstract

Synchronous circuits are pervasive in modern digital systems, such as smart-phones, wearable devices and computers. Synchronous circuits are controlled by a global clock signal, which greatly simplifies their design but is also a limitation in some applications. Asynchronous circuits are a logical alternative: they do not use a global clock to synchronise their components. Instead, every component reacts to input events at the rate they occur. Asynchronous circuits are not widely adopted by industry, because they are often harder to design and require more sophisticated tools and formal models.

Signal Transition Graphs (STGs) is a well-studied formal model for the specification, verification and synthesis of asynchronous circuits with state-of-the-art tool support. STGs use a graphical notation where vertices and arcs specify the operation of an asynchronous circuit. These graphical specifications can be difficult to describe compositionally, and provide little reusability of useful sections of a graph. In this thesis we present Asynchronous Concepts, a new design methodology for asynchronous circuit design. A concept is a self-contained description of a circuit requirement, which is composable with any other concept, allowing compositional specification of large asynchronous circuits. Concepts can be shared, reused and extended by users, promoting the reuse of behaviours within single or multiple specifications. Asynchronous Concepts can be translated to STGs to benefit from the existing theory and tools developed by the asynchronous circuits community.

Plato is a software tool developed for Asynchronous Concepts that supports the presented design methodology, and provides automated methods for translation to STGs. The design flow which utilises Asynchronous Concepts is automated using Plato and the open-source toolsuite Workcraft, which can use the translated STGs in verification and synthesis using integrated tools. The proposed language, the design flow, and the supporting tools are evaluated on real-world case studies.

Acknowledgments

I would like to thank my supervisor, Andrey Mokhov, for his wisdom and guidance throughout my postgraduate studies. He introduced me to graphical design methods, and helped me shape what has become the subject of this thesis, and I am very grateful. I would also like to thank Danil Sokolov, who has greatly helped my skills and knowledge in this field and in software design, and Alex Yakovlev, who often helped me expand my understandings and bring this into my research.

I also extend my thanks to my colleagues Alessandro de Gennaro, Vladimir Dubikhin and Adiran Wheeldon. We have worked together and discussed many topics over the years, and from them all I have learned many things. I hope they continue to be successful with their research and future careers.

To my friends, whom I have had the pleasure of spending time with since I began my studies. I thank you for all of your support and the fun times we have had. To Chris Dunn, thank you for your hospitality, and to Ashley Matthews, thank you for always being up for a chat, and your interest in my work. I also thank my family for their support throughout my studies.

Finally, and most importantly, I am thankful to Elie Howes for all of her love, support and patience throughout my PhD. She was there to help me at difficult times, and to share in the good times. Her encouragement has helped me with the research and with the writing of this thesis, and I am very grateful.

This research was supported by EPSRC research grant 'A4A: Asynchronous design for Analogue electronics' (EP/L025507/1) and the Royal Society Research Grant 'Computation Alive: Design of a Processor with Survival Instincts'

Contents

Abstract i

Acknowledgments ii

Contents iii

List of Figures vi

List of Tables xi

Publications xii

1 Introduction 1

1.1 Current usage of asynchronous technology 2

1.2 Motivation for a new design method 4

1.3 Contribution 6

1.4 Thesis layout 7

2 Technical background 9

2.1 Finite State Machines 9

2.2 Petri Nets 12

2.3 Signal Transition Graphs 15

2.3.1 STG design flow 17

2.4 Digital Circuits 18

2.5 Monoids 21

2.6 Boolean functions 23

2.6.1 Converting any Boolean function to CNF 24

2.6.2 Converting CNF functions to DNF functions 26

2.7 Haskell syntax 27

2.8 Summary . . . 28

3 Asynchronous Concepts 30

3.1 Circuit-specific concepts . . . 31

3.1.1 Signal-level concepts . . . 31

3.1.2 Gate-level and Protocol-level concepts . . . 40

3.2 Generalising to multiple inputs . . . 52

3.3 High-level concept functions . . . 55

3.3.1 Boolean function concepts . . . 55

3.3.2 Bubble transformation . . . 57

3.3.3 Dual transformation . . . 63

3.3.4 Enable transformation . . . 66

3.4 Set-Reset latch example . . . 70

3.5 Abstract concepts . . . 73

3.6 Summary . . . 77

4 Asynchronous Concepts design flow 79

4.1 Design approach . . . 80

4.2 Generating concepts from set and reset functions . . . 83

4.3 Process mining for Asynchronous Concepts . . . 84

4.4 User generated libraries of concepts . . . 86

4.5 Asynchronous Concept translation . . . 87

4.6 Combining concept specifications . . . 89

4.6.1 Sequential template . . . 90

4.6.2 Concurrent template . . . 90

4.6.3 Choice template . . . 91

4.6.4 Complex combinations . . . 92

4.7 Verification . . . 92

4.8 Synthesis . . . 94

4.9 Summary . . . 95

5 Automation of the design flow 97

5.1 Plato . . . 100

5.1.1 Translation to STG . . . 101

5.1.2 Translation to State Graphs . . . 107

5.1.3 Generating concepts from Boolean functions 112
5.2 Workcraft 116
5.2.1 Plato integration 116
5.2.2 PGminer integration 118
5.2.3 Verification and synthesis tools 119
5.3 Summary 123

6 Case Studies 125
6.1 Case Study 1: A simple buck controller 125
6.1.1 Combining the scenarios 131
6.1.2 Simulation and verification 132
6.1.3 Synthesis of a speed-independent controller 134
6.2 Case Study 2: Multiphase buck controller 135
6.2.1 Single phase block diagram 137
6.2.2 WAIT element 141
6.2.3 Full specification 144
6.3 Summary 151

7 Related Work 154

8 Conclusions 165
8.1 Main contributions 165
8.2 Future research and development 168

A Multiphase buck controller components 174
A.1 High-load handler 174
A.2 Token-control 176
A.3 Delay 177
A.4 MERGE element 179
A.5 Under-voltage handler 181
A.6 Zero-crossing handler 183
A.7 Over-current handler 186
A.8 Transistor Delay Controller 187
A.9 Full system translated STGs 189

Bibliography 193

List of Figures

1.1 Interconnect examples . . . 3
1.2 Example of an STG with hidden patterns of behaviour . . . 5
1.3 Equivalent concept specification for the STG in Figure 1.2 . . . 5

2.1 Abstract FSM of a stopwatch . . . 10
2.2 An example FSM with signals as transition conditions . . . 10
2.3 An FSM featuring 3 concurrent transitions . . . 11
2.4 The elements of a Petri Net . . . 12
2.5 A stopwatch PN model . . . 13
2.6 Constructs for modelling concurrency and choice in PNs . . . 14
2.7 An example of a system with concurrency modelled as a PN . . . 14
2.8 A PN featuring 3 concurrent transitions . . . 15
2.9 STG constructs for modelling choice and concurrency . . . 16
2.10 A system featuring concurrency in STG form . . . 16
2.11 An STG with 3 concurrent signal transitions . . . 17
2.12 Two possible implementations of a 3 input C-element . . . 19
2.13 Digital circuit converted to an STG . . . 20
2.14 Two formations of signal transition loops . . . 20
2.15 An STG using loop formation of signal transition loops . . . 21
2.16 AND-OR, or AO22 circuit icon . . . 21
2.17 Equivalent STGs for an AO22, both featuring an error . . . 22

3.1 Example of interface in a digital circuit . . . 33
3.2 First example specification with t as an output . . . 34
3.3 Second example specification with t as an internal . . . 35
3.4 Concepts and the translated STG for interface and initial state . . . 36
3.5 Translated STG with a single causality concept . . . 38

3.6 Translated STG containing AND causality . . . 38
3.7 Translated STG showing OR causality . . . 40
3.8 Buffer circuit . . . 41
3.9 Translated STG of a buffer . . . 42
3.10 Inverter circuit . . . 42
3.11 Translated STG of an inverter . . . 43
3.12 The translated STG of a handshake, and its resynthesized version . . . 44
3.13 C-element circuit . . . 45
3.14 Translated STG of a C-element . . . 46
3.15 OR gate circuit . . . 46
3.16 Translated STG of an OR gate . . . 47
3.17 AND gate circuit . . . 47
3.18 Translated STG of an AND gate . . . 48
3.19 XOR gate circuit . . . 48
3.20 Translated STG of an XOR gate . . . 50
3.21 Translated STG of a mutual exclusion concept . . . 51
3.22 Mutual exclusion element (metastability filter omitted for clarity) . . . 51
3.23 Translated STG of a mutual exclusion element . . . 52
3.24 Commonly used gates that we have not specified before . . . 57
3.25 Translated STG of `example7` . . . 58
3.26 Translated STG of `example8` . . . 59
3.27 Translated STG of `example9` . . . 60
3.28 STG and circuit generated from `example10` . . . 61
3.29 STG and circuit generated from `example11` . . . 62
3.30 Example circuit to apply `dual` to . . . 64
3.31 The dual of Figure 3.30 . . . 65
3.32 Mutual exclusion elements with and without enable . . . 67
3.33 STG of a mutual exclusion element with an enable signal . . . 69
3.34 Circuit implementation of a mutual exclusion element with enable . . . 69
3.35 Translated STG of `example19` . . . 71
3.36 Translated FSM of `example19` . . . 72
3.37 Synthesized set-reset latch circuit . . . 73

4.1 Translated STG of `example20` . . . 82

4.2 Synthesized circuit of example20 83
4.3 Synthesized circuit from example21 84
4.4 InvertedGates user concept file, with reusable concepts 87
4.5 A concept specification ready for translation 88
4.6 STG and state graph translations of example23 89
4.7 Example of sequential combination 90
4.8 Example of concurrent combination 91
4.9 Example of a choice combination 92
4.10 example24 translated to an STG 93
4.11 Technology mapped AND gate 94
4.12 A circuit synthesized without technology mapping 95

5.1 A diagram showing the interoperability of the software tools. 98
5.2 3-input NOR gate concept specification file 99
5.3 behaviour concept in atomic form 104
5.4 Translated STG with initial states inserted 106
5.5 Translated STG with one read arc inserted 107
5.6 Fully translated STG 107
5.7 example25 translated to a state graph 112
5.8 3-input NOR gate concept file generated by Bool-to-Concepts 115
5.9 Plato dialogue for authoring concepts 117
5.10 Plato include dialogue in Workcraft 117
5.11 Completed translation, and timing diagram of a concept specification . . 118
5.12 PGminer import dialogue 119
5.13 STG specification of a simple buck controller 120
5.14 Workcraft dialogue showing multiple satisfied verification properties . . . 121
5.15 Custom property dialogue for MPSAT 122
5.16 Dialogue indicating the custom property is satisfied 122
5.17 Synthesized circuit from the STG seen in Figure 5.13 123

6.1 Buck converter and its informal description. 126
6.2 Concept specification for the ZC absent scenario 128
6.3 STG translated from the zcAbsent concept and resynthesized 129
6.4 Concept specification for the ZC late scenario 129
6.5 STG translated from the zcLate concept and resynthesized 129

6.6 Concept specification for the ZC early scenario 130
6.7 STG translated from the zcEarly concept and resynthesized 131
6.8 Combined STG for a buck converter, featuring all three scenarios 132
6.9 STG for a buck converter, with some redundant arcs removed 133
6.10 Combined STG for a buck converter, featuring all three scenarios 134
6.11 Complex gate synthesis result of full STG 135
6.12 Technology mapped implementation of the simple buck controller 135
6.13 Schematic of a multiphase buck converter [1] 136
6.14 Block diagram of a single phase of the controller [1] 138
6.15 A WAIT element 141
6.16 Concept specification for a WAIT element 142
6.17 Resynthesized WAIT element STG translated from concepts 142
6.18 Figure 6.17 synthesized 143
6.19 Implementation of a WAIT element using an ME element 143
6.20 WAIT element specified using an ME element 143
6.21 STG of a wait element implemented using an ME element 144
6.22 Module name and imports for the activation specification 145
6.23 hlh_block concept 145
6.24 Connecting components with internal signals 146
6.25 tc_block concept 146
6.26 Concept specification for the activation section 147
6.27 Module name and imports for the charging specification 148
6.28 uv_block concept 149
6.29 dc_block concept 149
6.30 Concept specification for the charging section 150
6.31 Imports and module name for a single phase concept specification 151
6.32 Declaration of all signals in a single phase of the multiphase buck 152
6.33 Concept specification for a single phase of the multiphase buck 153

8.1 Two temperature control circuits, with a hazardous input signal 169
8.2 The circuit and models of a toggle circuit. 170
8.3 STG and state graph of a toggle circuit with CSC resolved 171

A.1 High-load handler component concept specification 175
A.2 HLH STG translated from concepts. 175

A.3 HLH STG resynthesized. 176
A.4 Token control component concept specification. 176
A.5 TC STG translated from concepts. 177
A.6 TC STG resynthesized. 177
A.7 Delay component concept specification. 178
A.8 Delay STG translated from concepts. 179
A.9 Delay STG resynthesized. 179
A.10 MERGE component concept specification. 180
A.11 MERGE STG translated from concepts. 181
A.12 Simplified state graph of a MERGE element 181
A.13 UVH component concept specification. 182
A.14 UVH STG translated from concepts. 183
A.15 UVH STG resynthesized 183
A.16 ZCH component concept specification. 184
A.17 ZCH STG translated from concepts. 185
A.18 ZCH STG resynthesized, with differing ZC signalling timings 185
A.19 OCH component concept specification. 186
A.20 OCH STG translated from concepts. 187
A.21 OCH STG resynthesized 187
A.22 DC component concept specification. 188
A.23 DC STG translated from concepts. 188
A.24 DC STG resynthesized 189
A.25 Translated STG of the activation section. 190
A.26 Translated STG of the charging section. 191
A.27 Translated STG of the full multiphase buck specification. 192

List of Tables

2.1 Truth table for function $(a \wedge b) \vee c$ 26

3.1 The hierarchy of interface concepts 34
3.2 The duals of gates in the library 66

5.1 OR-causality lists by effect 104
5.2 Causalities in DNF form 105
5.3 Causalities and source encoding 111

7.1 A comparison of Asynchronous Concepts with similar methods 156

Publications

Conference Papers

J. Beaumont, A. Mokhov, D. Sokolov and A. Yakovlev, "Compositional design of asynchronous circuits from behavioural concepts", 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)
doi: 10.1109/MEMCOD.2015.7340478

J. Beaumont, "Plato: a tool for behavioural specification of asynchronous circuits", 2017 17th International Conference on Application of Concurrency to System Design (ACSD), Zaragoza, Spain, 2017, pp. 68-73.
doi: 10.1109/ACSD.2017.14

Book chapter

A. Mokhov, J. Carmona, J. Beaumont. "Mining Conditional Partial Order Graphs from Event Logs", M. Koutny, J. Desel, J. Kleijn (Eds.): Transactions on Petri Nets and Other Models of Concurrency XI in Lecture Notes in Computer Science,
vol. 9930, pp. 114-136, Springer, 2016.
doi: 10.1007/978-3-662-53401-4_6

Journal Article

J. Beaumont, A. Mokhov, D. Sokolov, A. Yakovlev, "High-level asynchronous concepts at the interface between analogue and digital worlds", Special Issue on Circuit and System Design Automation for Internet of Things in Transactions on Computer-Aided Design of Integrated Circuits and Systems.
doi: 10.1109/TCAD.2017.2748002

Technical reports and memos

J Beaumont "Modelling Digital Systems using Behavioural Fragments", Technical Report Series, async.org.uk, January 2015
`http://async.org.uk/tech-reports/NCL-EEE-MICRO-TR-2015-194.pdf`

J. Beaumont, "A survey of theory and practice in compositional design of asynchronous circuits", Technical Memo Series, async.org.uk, November 2015
`http://async.org.uk/tech-memos/NCL-EEE-MICRO-MEMO-2015-011.pdf`

J. Beaumont, "Automated translation of asynchronous concepts to Signal Transition Graphs", Technical Memo Series, async.org.uk, December 2016
`http://async.org.uk/tech-memos/NCL-EEE-MICRO-MEMO-2016-013.pdf`

Chapter 1

Introduction

Digital systems are used in most industries in this day and age, such as to aid in diagnosis and treatment in the medical field. Many individuals will also use a digital system on a daily basis, as the availability of computers, smart phones, wearable technology, and connected-home devices grows.

These digital systems are primarily designed using synchronous technology. In a synchronous system, the clock signal oscillates at a regular frequency and is used to synchronise components. The clock period is used to determine the length of time that operations must take place in, such as the storing of an input to the circuit. However, there are several disadvantages to synchronous systems. Many operations can complete in a much shorter time than the clock period, but must wait to start a new operation for the remainder, meaning there can be periods of inactivity. Synchronous systems are also always active, polling for changes to inputs, increasing their power consumption.

Asynchronous circuits are a class of digital circuits which do not use a clock signal. Without a clock signal, asynchronous circuits immediately perform an operation as soon as the inputs change. This can reduce the length of time a circuit is waiting between operations, making the system faster compared to synchronous systems [2]. This also means that the system does not need to constantly test for changes in the inputs as synchronous circuits do. Thus, the time and power consumption of asynchronous circuits can be lower than that of their synchronous counterparts.

There are a few examples of fully asynchronous systems in academic literature, but even fewer in the industrial world. For example, *Fulcrum Microsystems* was a company which designed asynchronous networking solutions, and was bought by *Intel* [3]. This may influence the use of asynchronous technology in future Intel chips, but even with proof that this is beneficial in areas such as networking, wide adoption of asynchronous circuits has yet to come to fruition.

Asynchronous technology has yet to be widely adopted for a few key reasons. Historically, synchronous designers have used textual based design methods, using languages such as *Verilog* and *VHDL*, which describe systems based on clock signals. Asynchronous design methods tend to use behavioural models which are very different, specifying systems graphically using vertices and arcs, the differing ways in which these interconnect indicating different behaviours. Graphical methods are therefore unfamiliar and unattractive for industry, as it would require the retraining of designers, and the development of new Electronic Design Automation (EDA) tools that aid in the design of asynchronous circuits. This can be costly and time consuming, and mean the development of the following systems is relatively slow, which for an ever growing market of digital systems, is not ideal for industry.

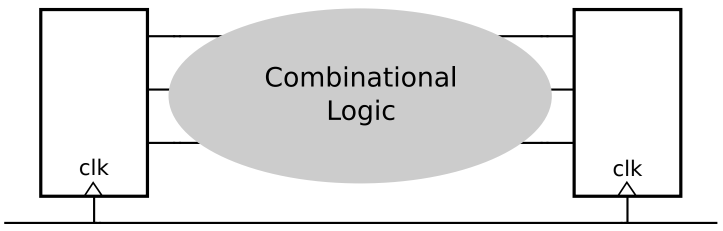
To address these issues, and realise the full potential of asynchronous systems, we need to develop design methods, and associated software tools, which provide a simpler switch from synchronous design to asynchronous design, that the digital system industry will be more willing to use. This thesis introduces such a design method, using a new language, *Asynchronous Concepts*, and some software tools which aid in automating many steps in the design process.

In this chapter we outline the current state of asynchronous technology in industry, and our motivation and proposed methodology for designing asynchronous circuits, discussing the contribution, aiming to combat the issues faced with existing methodologies. We also detail the layout of this thesis, briefly discussing the main topics of each chapter.

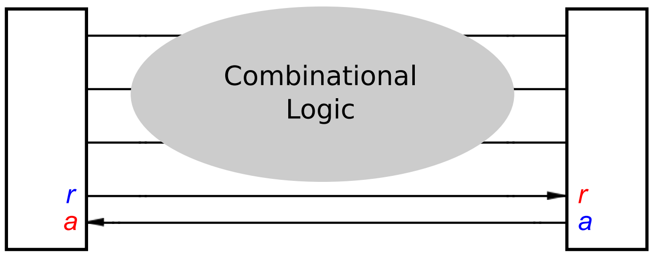
1.1 Current usage of asynchronous technology

Some asynchronous theories have become popular in recent years, when including these with existing synchronous technology. As a circuit becomes physically larger, ensuring the clock reaches the whole system can become difficult, as clock skew can cause timing disparities between different sections of the circuit. Communication between subsystems can therefore be difficult, and thus asynchronous handshakes have come into use with these systems.

Figure 1.1a is a simplified diagram of how subsystems in a fully synchronous circuit connect. The sending device (left) will output signals, which passes through combina-



(a) Example of synchronous interconnect



(b) Example of asynchronous interconnect

Figure 1.1: Interconnect examples

tional logic, and is stored by the receiving device (right) when the clock signal goes high. The outputs from the combinational logic may be ready before the clock, and thus there can be wasted time.

Figure 1.1b is a similar diagram, this time using the asynchronous handshake protocol to interconnect between devices. In this case, the left device sends a request (r) signal to the right device when the signals are output to the combinational logic. The request signal is delayed enough to allow for the signals to propagate through the combination logic, then as soon as this logic outputs the new values, and the request signal reaches the right device, this will send an acknowledge (a) signal back to the left device, indicating that the data has been received. When this acknowledge is received, the request will be dropped and the left device can begin again when ready. This means that these subsystems can move on as soon as they are able, not needing to wait.

This extends to more than just combinational logic, however. This example may contain sequential logic or other clock based systems between these devices. These can feature their own clock signal, rather than a distributed clock signal, but each subsystem continues to communicate asynchronously. Each subsystem can then operate at their own fastest possible speed, and communicate with other subsystems as soon as they are ready, increasing the speed of these primarily synchronous systems. Asynchronous techniques such as this can be applied to the circuit, component, or subsystem level as is the case in this example, with this type of system is known as *Globally-Asynchronous*

Locally-Synchronous (GALS) [4].

A new direction of research for asynchronous technology is analogue and mixed-signal systems, the process of using analogue signals with digital circuits, reacting to the changes of these analogue signals dynamically. This is necessary with the recent increase in availability of mobile and autonomous applications, such as smart phones, wearable electronics and self-powered Internet-of-Things nodes, where it is essential to have intelligent timing control and power regulation [5][6][7]. An on-chip power management system is an illustrative example: it relies on analogue circuitry for power regulation and conversion, and its behaviour is characterised by many operating modes with complex interplay and high-level decision logic that is digitally controlled.

Asynchronous circuits are event-driven, i.e. they react to changes in a system at the rate they occur [8]. This makes them particularly useful for interacting with analogue world, where the ability to quickly respond to non-digital input, e.g. a measurement of something in the environment, which the circuit in question needs to react to once this measures above a threshold value.

1.2 Motivation for a new design method

Various methods for designing asynchronous systems exist [9], including behavioural models which specify the low-level interaction between the system and the environment, such as Finite State Machines (FSMs), Petri-Nets (PNs) and Signal Transition Graphs (STGs) [10][11][12]. This has spawned multiple Electronic Design Automation (EDA) tools, which serve to make the design process using these methods simpler.

In this thesis we introduce a new language, Asynchronous Concepts, for the specification of asynchronous circuits. These are introduced as a method of describing patterns of behaviour when specifying asynchronous circuits, using a language-based approach. This is aimed at providing a textual method like the existing languages used in a synchronous design, but providing the ease of use for common asynchronous constructs. All concepts can be composed, producing new concepts which describe new behaviours, and all concepts can be reused, so a pattern can be used multiple times, and be built-upon as the pattern evolves and as asynchronous circuits are used more often.

The existing asynchronous design methods do include common patterns, but can become lost as a specification becomes larger, and more patterns are included. For example, Figure 1.2 contains a simple STG which features hidden patterns of behaviour.

Signals r1 and g1 behave like a buffer, with input r1 and output g1. This is also the case for r2 and g2. Signals g1 and g2 are also mutually exclusive. The equivalent concept specification for this STG is found in Figure 1.3

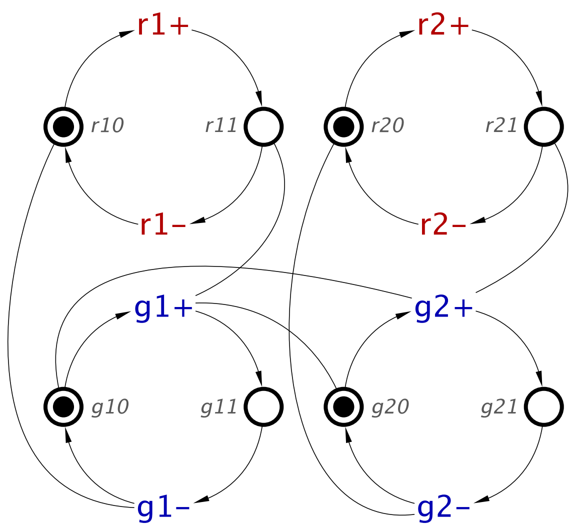


Figure 1.2: Example of an STG with hidden patterns of behaviour

```
example1 = behaviour <> initState <> interface
  where
    behaviour = buffer r1 g1 <> buffer r2 g2 <> mutex g1 g2
    initState = initialise0 [r1, r2, g1, g2]
    interface = inputs [r1, r2] <> outputs [g1, g2]
```

Figure 1.3: Equivalent concept specification for the STG in Figure 1.2

The behaviours in this system can be described easily in the concept specification, and the words used are indicative of the behaviour, but the STG hides these, making it harder to understand the interactions of these signals. Moreover, any useful behaviours which are specified in the STG cannot be easily reused, and they must be rebuilt in another specification. The concept equivalent can be referenced from another concept specification, importing this specification, or even saving any valuable new patterns in a user-generated library.

This example is that of a *Mutual Exclusion element* (ME element). This device arbitrates between two requests for a single resource, granting only one of these requests access at on time. The STG and concept specification do not specify the behaviour of any environment that might affect such a device. An STG that features some assumptions about the environment can be created, which may show more clearly the operations of the ME element. More information and concept specification for this example will be discussed in more detail in Section 3.1.2.

A basic library of circuit-specific concepts can be used to produce a behavioural specification, or be used to create further libraries, containing concepts which are used regularly throughout a design. Every concept can be composed with any other concept, forming another concept, to provide a full specification composed of any level of concepts. The different levels of concepts are:

- Signal-level concepts based on one or more signal events causing one or more other signal transitions,
- Gate-level concepts which describe the behaviours of logic gates and the signals included in these and
- Protocol-level concepts which describe constraints and interactions of multiple signals as often used useful constructs.

A composition of concepts ensures that each signal has relevant behavioural patterns applied to it. Any concepts can then be reused in future designs.

1.3 Contribution

The major contributions of this thesis are:

- A domain-specific language, *Asynchronous Concepts* [13][14]. This language is used to specify the behaviours of asynchronous circuits, describing the interactions of signals in systems at various levels.
- A library of circuit-specific concepts. This includes concepts at all levels, which describe the common protocols and gates which apply to asynchronous circuits. It also provides transformations which can change concepts to provide a much wider range of behaviours.
- We propose a design flow for designing asynchronous circuits using concepts. A design can be started from a blank specification, use existing concept specifications, using the Boolean set and reset functions from a circuit, or through process mining an existing circuit.

- To aid in this design flow, we provide *Plato* [15][16], an open-source EDA tool which implements the language of Asynchronous Concepts and includes the circuit-specific library. This tool compiles concepts specifications to check for errors, providing information to aid in corrections and features algorithms to translate concept specifications to STGs and state graphs.

Plato is integrated into the open-source EDA software suite *Workcraft* [17][18]. This supports multiple modelling formalisms, including FSMs, PNs, state graphs and STGs, and can automatically insert a translated concept specification from Plato in state graph or STG format for viewing, simulation and further operations.

Translating a concept specification to an STG means this can then be verified against certain properties, to ensure that it can be synthesized to produce a logic gate implementation. There are several software tools which can automatically verify and synthesize an STG specification, such as *Petrify* [19], *MPSAT* [20] [21], both of which are also integrated into Workcraft, allowing the design flow to be carried out entirely within one software suite, with a Graphical User-Interface (GUI).

We have developed a method of converting Boolean expressions into Concepts. This extends the outreach of concepts, allowing existing circuits to be used in conjunction with other concept specifications. A Boolean function can also be used in line with other concepts. This process will be explored.

We also provide a method of discovering specifications by observing real systems. Process mining performed by another tool integrated into Workcraft, *PGminer* [22][23], which can be used in order to identify concurrency and event ordering from the simulation traces of such a system. From this noted concurrency, we can then automatically generate a list of concepts forming a specification, and this could be optimised, finding higher-level concepts from the list of lower-level concepts.

Through transformations of concepts, a wider range of behaviours can be described which include inverted signals, the dual of a system, or applying enable operations to any specification. The concepts language can be expanded upon to allow the specification of asynchronous systems of many different types with further research and development.

1.4 Thesis layout

This thesis is organised as follows:

Chapter 1 - Introduction. In this chapter we briefly discuss the motivations for the thesis, and summarise the contributions, namely the Asynchronous Concepts language, the design flow using this language, and the software tools to support it.

Chapter 2 - Technical Background. We discuss the existing modelling formalisms of Finite State Machines, State Graphs, Petri Nets and Signal Transition Graphs. We explain the design methodology of STGs, and the shortcomings, and we discuss digital circuits and how they can be translated to STGs. We introduce Monoids, an algebraic structure on which concepts are built, as some definitions of monoids are used when discussing concepts. Finally, we introduce some of the syntax of Haskell, which is used to implement Plato, and write concept specifications, and as such, the syntax is used often throughout this thesis.

Chapter 3 - Asynchronous Concepts. This chapter introduces the Asynchronous Concepts language: Circuit-specific asynchronous oncepts and a library of signal-, gate- and protocol-level concepts will be explained, as well as generalized multi-input concepts for some of these gates. This library includes some high-level concept functions which can provide other methods of specification and a wider range of uses for concepts. We also discuss abstract concepts as a base for the development of new concepts for different types of asynchronous circuit.

Chapter 4 - Asynchronous Concepts Design Flow. This chapter will introduce the design flow using Asynchronous Concepts, providing a work-through from authoring concepts, to synthesis of a logic gate implementation. We will also explain how process mining and Boolean set and reset functions can be used within this design flow.

Chapter 5 - Tool Support. In this chapter, we will discuss the tool Plato, its integration with Workcraft, explain the algorithms for translating concept specifications to Signal Transition Graphs or state graphs, and discuss the process mining tool, PGminer.

Chapter 6 - Case Study. Here we use an example of a WAIT element to show how concepts can be used to specify this, and the real-world example of an asynchronous multi-phase buck controller to show how a full asynchronous system such as this can be specified using concepts.

Chapter 7 - Related Work. This chapter discusses other asynchronous design methodologies and EDA tools, and how the contributions of this thesis fit into this research area.

Chapter 8 - Conclusions. This will be a summary of the contributions as discussed in this thesis, and future research areas for Asynchronous Concepts and Plato.

Chapter 2

Technical background

In this chapter, we will discuss some existing models that can be used to specify asynchronous circuits, the advantages and disadvantages of the models themselves and their design flows. The models featured in this chapter are Finite State Machines in Section 2.1, Petri nets in Section 2.2 and Signal Transition Graphs in Section 2.3. We will also discuss digital circuits, what they represent, and the conversion of digital circuits to Signal Transition Graphs, and the uses of this in Section 2.4.

Monoids are a mathematical construct, important for the contributions of this thesis, and we introduce these in Section 2.5. Boolean functions are also used multiple times in this thesis, and these are discussed in Section 2.6, as well as some algorithms used to convert Boolean functions between various forms. Finally, we will discuss the syntax of the programming language Haskell, which is used heavily, as this is the language that Asynchronous Concepts are built in, Section 2.7.

2.1 Finite State Machines

Finite State Machines (FSMs) have many uses, from the modelling of a system at a higher level, such as a vending machine or a stopwatch, to the specification of a digital circuit, both synchronous and asynchronous. They hold an overall state for the modelled system which at higher levels abstracts the complex implementation details, and at lower levels shows the system state and all possible events from this state [24][25].

An FSM is comprised of states, and transitions between those states. Each transition has a condition applied to it, which must be satisfied for the transition to be enabled, allowing the system to move from one state to another.

In more abstract FSMs these conditions can be labels, simply indicating that some event in the environment of the model can occur. For example, Figure 2.1 contains an FSM of a simple stopwatch. In this, the states are labelled "*reset*", "*stop*", "*run*" and "*lap*". There are transitions between these states, indicated by the arcs connecting them, and the conditions on these transitions are labels, including "*lap_button*", "*reset_button*" and "*start_stop_button*". These indicate which buttons on the stopwatch must be pressed in order to change state. To move from the "*stop*" state to the "*run*" state, the "*start_stop_button*" condition must be satisfied, which in the physical system requires that the button has been pressed. Buttons are used to transition between all states, except the transition from states "*reset*" to "*stop*". The condition applied to this is an "ϵ", or empty, requiring no conditions for this transition to be enabled. This means that as soon as the stopwatch has been reset to 0, it immediately transitions to the stop state, preparing it to start timing from 0 once again.

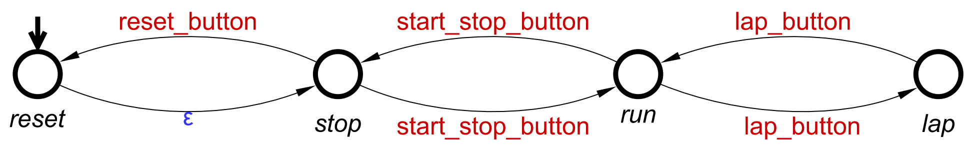


Figure 2.1: Abstract FSM of a stopwatch

For lower-level systems such as asynchronous circuits, where signal interactions need to be specified, we can also use FSMs. A variant of FSMs exists which specifically uses signal transitions as the conditions for state transitions, based on the signal types of input, output and internal. These are known as state graphs, and the states can be labelled with binary codes representing the values of the system's signals in that state.

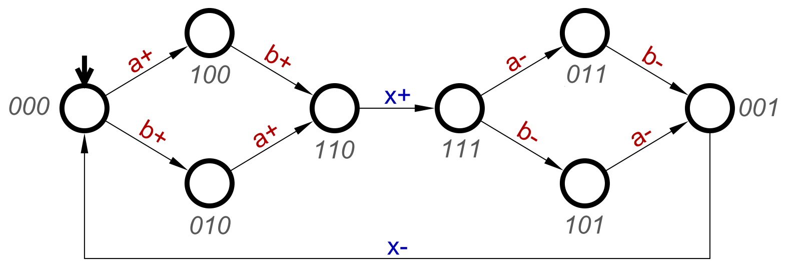


Figure 2.2: An example FSM with signals as transition conditions

Figure 2.2 contains a state graph featuring three signals, `a` and `b` are inputs and `x` is an output. Each state is labelled with the encoding of the state, in the order abx. When all signals are low, the encoding is 000, when a+ has occurred and the system transitions to state 100.

A property of state graphs and state encodings is *Complete State Coding* (CSC). This states that each state with differing behaviour has a different encoding. In this way, each state can be referenced separately, and the circuit synthesized from this will react differently to each state. If two or more states with the same encoding but differing behaviours occur, then the signals will be the same in each of these states. Logic would not be able to distinguish and perform the different behaviours. Therefore, the encoding of a state graph is important.

This example shows that the condition for each state transition allows only one signal to transition, so each state change indicates that one signal has transitioned. In the event that two signals can transition concurrently, such as in this example where both a and b must transition high (+) for x to transition high, then the state graph indicates that either order of a and b transitioning is possible, either a then b, or b then a. This is called *interleaving semantics*.

However, when the number of concurrent signal transitions increases, this can create a large and complicated state graph. If we add another input signal, c, which transitions concurrently with a and b, and must also be required to transition high for x to transition high, the result is found in Figure 2.3.

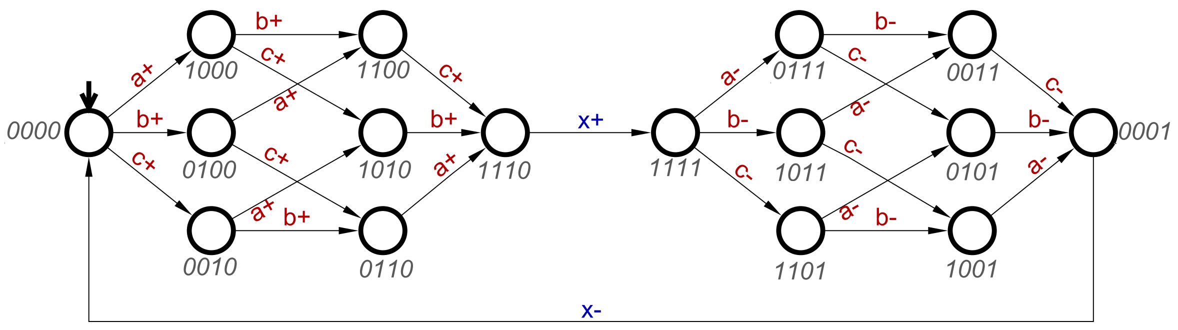


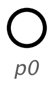
Figure 2.3: An FSM featuring 3 concurrent transitions

The concurrency in this example now causes much larger sections with multiple arcs crossing over each other, making the state graph harder to understand. This will only increase as the number of concurrent events increases, and is known as the *State Explosion* problem [26]. If there are n concurrent events, the number of states needed in the state graph to ensure that every possible order in which the events can occur is modelled becomes 2^n.

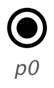
Concurrency is therefore difficult to model with FSMs which makes them undesirable for specifying larger asynchronous circuits with many signals, which often feature concurrency. However, in some cases, it may be useful to view the intricacies that a system may present, which can be more easily viewed in state graph form, which, due to its wide-range of uses, is more well-known.

2.2 Petri Nets

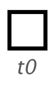
Petri Nets (PNs) are a mathematical model, introduced in [27], and are commonly used for modelling systems with a high degree of concurrency, and are therefore useful for specifying asynchronous systems. Unlike FSMs, a PN does not aim to show the state of the system being modelled at any point, and so does not deal with interleaving semantics which removes the issues associated with state explosion. Similar to FSMs however, PNs can be used to model systems at various levels, from high-level whole system designs, to low-level circuit designs.



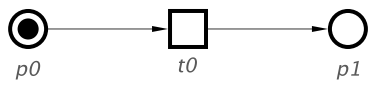
(a) A Petri Net place



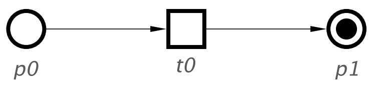
(b) A place containing a token



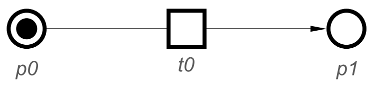
(c) A Petri net transition



(d) $p0$ contains a token, enabling $t0$



(e) $t0$ has fired, producing a token in $p1$



(f) $p0$ connects to $t0$ via a read-arc

Figure 2.4: The elements of a Petri Net

A PN is comprised of several elements. These include *places* (Figure 2.4a), which can hold a *token* (Figure 2.4b), *transitions* (Figure 2.4c) which consume tokens from places which connect to the transition and produce tokens for places this transition connects to, and *arcs* which are directed, connecting places and transitions (Figure 2.4d). Double-arcs can also be used, known as *Read-arcs* (Figure 2.4f). These arcs connect a place to a transition, the transition being enabled only when a token is contained within the place. When the transition fires, this token is not consumed, but the transition can still produce a token.

A transition can only be enabled when all places which connect to it contain a token. Using the example in Figure 2.4d, $t0$ is enabled as $p0$ contains a token. When $t0$ consumes the token, it is said to have *fired*, and will pass a token to all places which follow it, in this case, $p1$ (Figure 2.4e). Also included in a PN model is the *initial state*. This is the placement of tokens in a system when the system begins, which indicates which transitions are able to fire first. The formal definition of Petri Nets can be found in [28].

With PNs, we can model a high-level system like a stopwatch as we have with FSMs (Figure 2.1). The PN version can be found in Figure 2.5. This PN is not too different from the FSM version, but there is no concurrency in this system. The differences become more apparent when concurrency is a factor, as in the case of Figure 2.2.

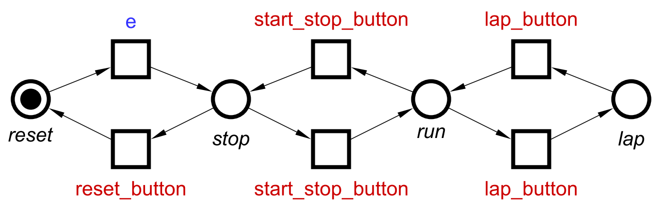


Figure 2.5: A stopwatch PN model

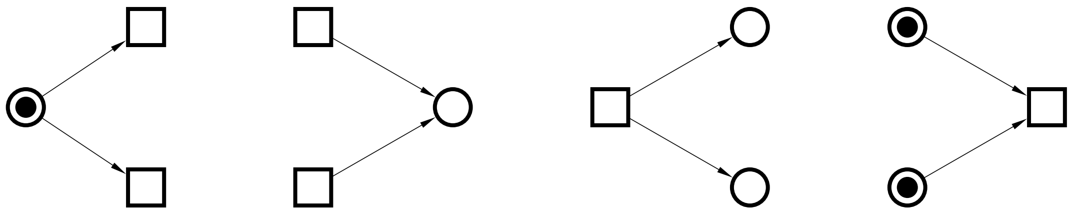
Using the elements of a PN there are multiple constructs available to model some useful information about the system, such as where concurrency begins and where there are choices of transitions. These are:

- **Choice** (Figure 2.6a) - Where there are multiple choices of transition following a place, only one of which can consume the token, meaning only one branch then runs.

- **Merge** (Figure 2.6b) - When one of the free choice branches has completed, the token is passed into a place, ready to move onto the next section.

- **Fork** (Figure 2.6c) - A transition begins a section with multiple concurrent events, all places following the fork transition receive a token, allowing each branch to run independently of the others.

- **Join** (Figure 2.6d) - When all concurrent branches have completed, each will have a token in its final place, allowing the join transition to fire, ending this concurrent section.



(a) Free choice (b) Merge (c) Fork (d) Join

Figure 2.6: Constructs for modelling concurrency and choice in PNs

For the example as seen in Figure 2.2, we can create a PN form of this using the fork and merge constructs to indicate concurrency between signals a and b. The resulting Petri net is shown in Figure 2.7.

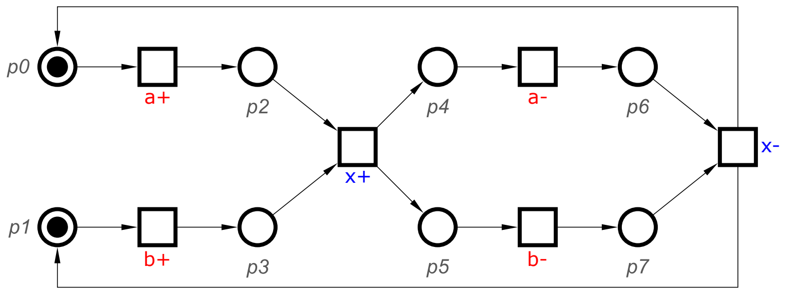


Figure 2.7: An example of a system with concurrency modelled as a PN

In comparison to Figure 2.2, the PN is quite different. The PN does not include both possible orders of a and b transitioning, instead there is a fork from x- to $p0$ and $p1$, leaving us with one branch where a+ can fire, and one where b- can fire, allowing either to fire first. Both of these transitions must fire however in order for the join from $p2$ and $p3$ to x+ which can only then fire. Following x+, another fork occurs to places $p4$ and $p5$, for two branches allowing a- and b- to fire in any order. $p6$ and $p7$ then join at x-, as both are required for this to fire.

The way concurrency is shown in a PN is much clearer than in a state graph, and this becomes more apparent as the number of concurrent events increases. If we now add an input signal c in again, as in Figure 2.3, then for a PN another branch is added in each fork, where as the state graph adds many more states. This PN can be viewed in Figure 2.8.

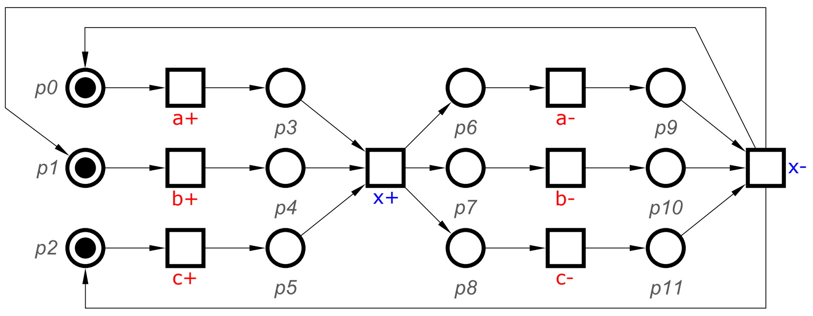


Figure 2.8: A PN featuring 3 concurrent transitions

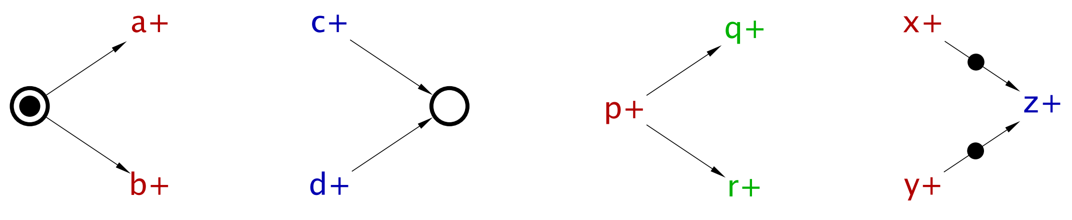
PNs can be used on much larger-scale systems, both high-level and at circuit-level, such as with asynchronous circuits. However, as can be seen in Figures 2.7 and 2.8 there are a multitude of places and transitions which can make a larger design look cluttered. To try and combat this issue, and create clearer models for larger specifications of circuits, Signal Transition Graphs which are discussed in Section 2.3.

2.3 Signal Transition Graphs

Signal Transition Graphs (STGs) are domain-specific Petri nets, used for specifying asynchronous circuits [19]. The transitions of STGs are solely the event of a signal transitioning, either high or low [10][12]. STGs can be used to model the environment that a circuit reacts to, the *input* signals, the intermediate signal changes within the circuit, *internal* signals, and the *output* signals which are the circuits reaction to the environment, and conventionally, input, output and internal signals are identified by their colour, red, blue and green respectively. Each signal can transition either high, indicated by the $+$ operator, or low, indicated by the $-$ operator.

As STGs are derived from PNs, they feature all of the elements of PNs, including tokens, arcs and read-arcs. Places are also a feature, however in an STG there can be *implicit places*, which allows tokens to be contained within an arc connecting two transitions. This removes the number of places that a single model contains, which can make for a clearer STG. The choice and merge constructs in a circuit can be indicated in the same way as PNs, but implicit places change how concurrency constructs, fork and join, are indicated.

Figure 2.9 contains the constructs for choice, merge, fork and join in STGs. Free choice (Figure 2.9a) and merge (Figure 2.9b) are very similar to those of PNs, with a place passing a token to only one branch of the system, and a chosen branch returning this



(a) Free choice (b) Merge (c) Fork (d) Join

Figure 2.9: STG constructs for modelling choice and concurrency

token to a place when merging, to be passed onto the next section. Fork (Figure 2.9c) and join (Figure 2.9d) however feature transitions connecting to transitions. In these cases, the places which in a corresponding PN would need to be connected between these transitions, are implied as part of the arcs which connect them. This is shown in the join figure, which contains a token on each arc, both of which being present allows `z+` to fire. For the fork example, once `p+` has fired, a token will be available on all following arcs.

An STG can be used to model a system such as the stopwatch example, as it is derived from PNs, but their usage is aimed at much lower-level specifications for circuits, at signal-level. For STGs we will use the concurrency example, which can be seen in state graph form in Figure 2.2, and PN form in Figure 2.8.

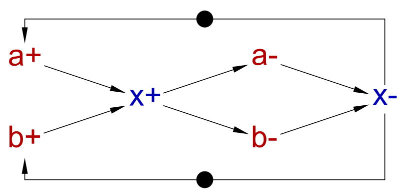


Figure 2.10: A system featuring concurrency in STG form

Figure 2.10 is very similar to the PN form, but without the places this simply shows the causality between signal transitions, one signal transition causing another, and so-on. The initial state is indicated by the tokens contained by the arcs from `x-` to both `a+` and `b+`.

If we add in a signal `c` as before, which is also required to transition concurrently with `a` and `b` for `x` to transition in the same direction, the result is similar to the PN equivalent, Figure 2.8, but clearer still. Figure 2.11 shows the STG for this.

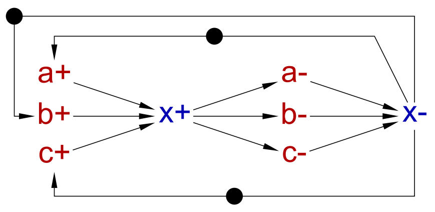


Figure 2.11: An STG with 3 concurrent signal transitions

2.3.1 STG design flow

STGs are commonly used for the specification of asynchronous circuits, and as such feature a design flow, taking a specification, verifying this and then synthesizing it for an implementation. In this section, we will discuss the ordering of this design flow, and the tool support available. We will also discuss disadvantages that are faced using this approach.

A specification using STGs is often started with a blank page, and transitions, places and arcs are added in manually as behaviours and interactions of the system being specified are covered. *Workcraft* is a software suite which provides a GUI for modelling and specifying systems with multiple interpreted graph models, including FSMs, state graphs, PNs and STGs [17][18]. This software can be used to create STGs visually.

Both during and following a specification being prepared, simulation can take place using Workcraft. This aids in the design process, allowing a user to simulate as they create, and can determine where their design does not operate as expected, allowing them to fix errors as they work.

For an STG to be considered correct and usable for synthesis, it needs to satisfy certain properties, which are discussed in Section 4.7. Workcraft also features integrated back-end tools to automatically perform verification, such as *MPSAT* [20] [21]. MPSAT tests whether a specification satisfies these properties, and can also verify for any custom properties a user chooses. If a system fails verification, this must be fixed in order for the system to be synthesized.

Fixing or debugging these verification errors becomes more difficult the larger an STG becomes. Finding the relevant area can become difficult, with a multitude of transitions, places and arcs interconnected which may need to be changed. This can also make comprehension difficult for a user who did not design this system initially.

When an STG specification has passed all the necessary verification, and is determined, through simulation, to work as desired, this specification can then be synthesized. This process determines the Boolean equations which describe the output (and internal) signals as functions of the environment. These functions can then be used to determine a set of logic gates which satisfy the equation, providing a logic gate implementation which can be used in a circuit design.

Using the example from Figure 2.11, we can synthesize this. We start by finding what causes the output, signal `x`, to rise. From the STG we can see that this requires that `a+`, `b+` and `c+` have all occurred. Thus, for the output to go high we provide the boolean function, known as the *set* function, $set : a \wedge b \wedge c$. For the output to go low, known as the *reset* function, we look at the STG and find that this occurs after `a-`, `b-` and `c-` have all occurred, therefore the reset function will be $reset : \overline{a} \wedge \overline{b} \wedge \overline{c}$.

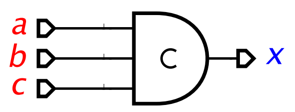
These functions can then be checked against a library, which contain the set and reset functions of various gates, and is used to map these functions onto existing gates. In this case, the result features at least one C-element. Synthesis can again be performed automatically by some tools, such as MPSAT and *Petrify* [19], which are integrated into Workcraft. The possible results of this synthesis can be found in Figure 2.12.

The specifications generated by this design flow tend to become monolithic, and any features of one specification which may be useful in future specifications are difficult to identify in a large specification and re-use. Therefore, a future design must again begin with a blank page, which makes future designs slower. While STGs are commonly used for design of asynchronous circuits, the lack of reusability, and the difficulty in comprehending, editing and debugging specifications can make this design flow undesirable.

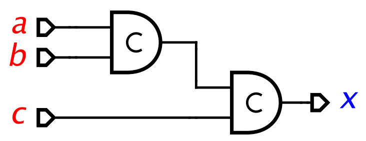
2.4 Digital Circuits

Digital circuits are circuits with digital signals, signals that have two states, high or 1 and low or 0. These circuits contain gates, including standard gates such as AND gates, OR gates or C-elements, and other non-standard gates which have specific behaviours to combine signals in desired ways, providing outputs indicating these combinations. Digital circuits are synthesized from specifications, such as those in STG form, and Boolean functions determined during synthesis can be mapped onto standard gates, contained in a library, or simply be stated based on set and reset equations.

Workcraft features a digital circuit plugin, where a user can create a digital circuit and simulate it. A digital circuit can also automatically be synthesized from a specification



(a) Three input C-element



(b) Two, two input C-elements

Figure 2.12: Two possible implementations of a 3 input C-element

and imported into Workcraft for further work.

Figure 2.12 contains two possible digital circuits which can be synthesized from the concurrency example as seen in Figure 2.11. Figure 2.12a is a three-input C-element that synchronises three signals. However, C-elements are commonly two-input gates, and as such, it could also be synthesized to produce a circuit as in Figure 2.12b, which features two C-elements. One of these gates synchronises input signals a and b, and the other synchronises the output of the first C-element, and the third input signal c. The output signal of this second C-element is the output of the circuit x. These two circuits are not equivalent, however, unless certain assumptions about how the environment, the inputs signals, behave are given.

Within Workcraft, it is also possible to convert a digital circuit directly to an STG. This allows a new specification to be used in conjunction with the specification of an existing circuit. The conversion process is direct, simply capturing the causalities between signal transitions. If we now convert either synthesized digital circuit from Figure 2.12, we will find that the STG generated correctly captures the operations of the circuit which produce the output x, but is not as compact as the equivalent STG, shown in Figure 2.11.

The converted STG features the input signals on the left, and the output signals on the right. Each signal is arranged in a formation which includes their high $(+)$ and low $(-)$ transition, as well as two places which are connected between these signals, labelled "s_HIGH" and "s_LOW", where s is the signal in question. These places are used to show the state of the respective signals, the LOW holding a token when the signal is 0, and $HIGH$ holding a token when the signal is 1.

These arrangements are used to ensure that each signal's transitions alternate in the system, allowing a high transition only after a low transition, and vice versa. This ensures that the *consistency* property is satisfied.

The places of the input signals are connected to transitions of the output signal, x via

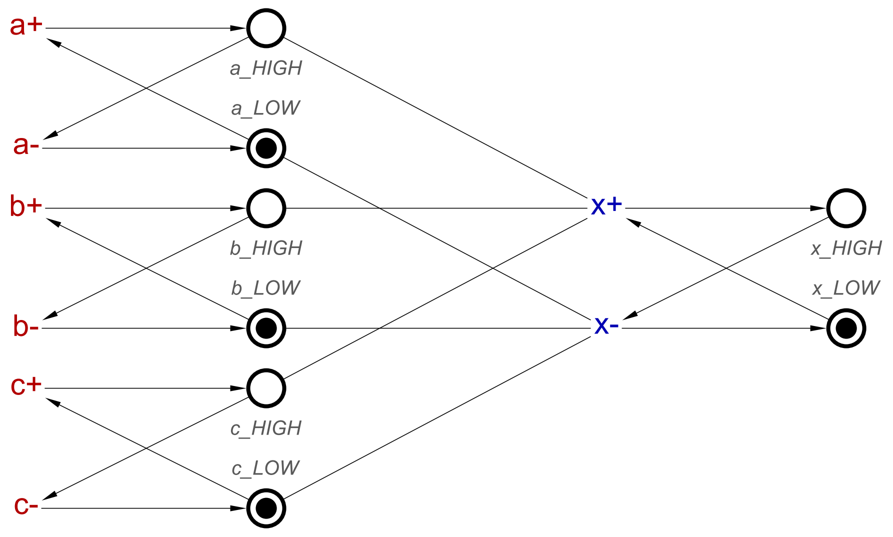
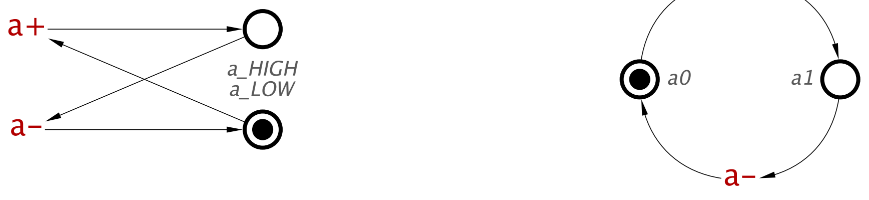


Figure 2.13: Digital circuit converted to an STG

read-arcs. These are used so that for a signal transition to be enabled, all of the places to which it connects must hold a token. The transition can then fire, but the read-arcs will ensure that none of the tokens are consumed, so no signals are then blocked from firing.

The arrangement of signal transitions and places in this converted STG is useful for capturing the causalities clearly, and ensure that no signal is ever blocked from transitioning. There are two forms of this notation; the *Z formation*, used when converting digital circuits to STGs using Workcraft as in Figure 2.13. The other form is a circular, *loop* formation, shown in Figure 2.14b, which will be commonly used in this thesis, as this formation is generated automatically by automated concept translation.



(a) "Z" formation of a signal transition loop (b) "Loop" formation of a signal transition loop

Figure 2.14: Two formations of signal transition loops

The example digital circuit converted to an STG using the loop formation can be viewed in Figure 2.15. This STG is equivalent to that in Figure 2.13. This also features *proxy places* connecting the `x` transitions to the places of the input signals, which indicate that a connection is there, but hidden, as the number of arcs could make the STG cluttered.

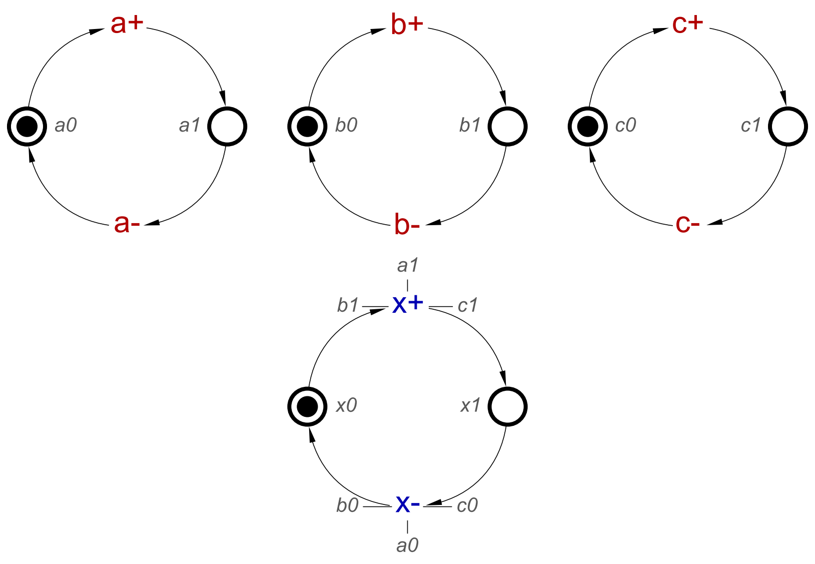


Figure 2.15: An STG using loop formation of signal transition loops

This example is fairly simple, and the resulting STGs in either form can be understood relatively easily. We now introduce another example which is more complex, an AND-OR gate, the circuit of which is shown in Figure 2.16. This gate serves to indicate when either both of the inputs `a` and `b`, or both of the inputs `c` and `d` have transitioned high.

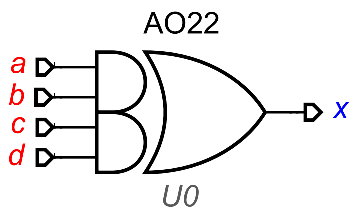
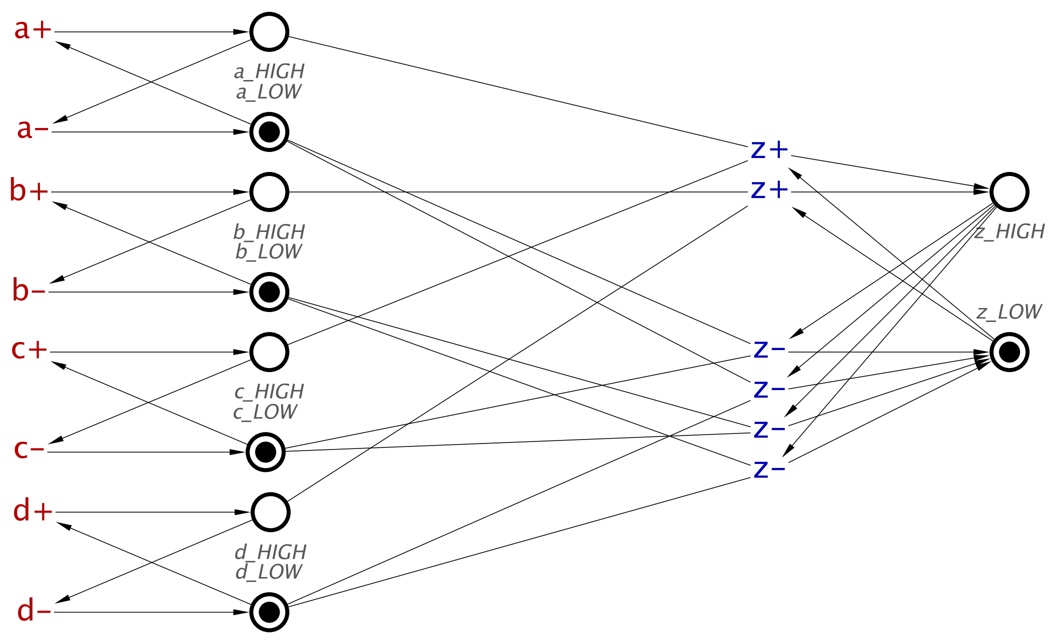


Figure 2.16: AND-OR, or AO22 circuit icon

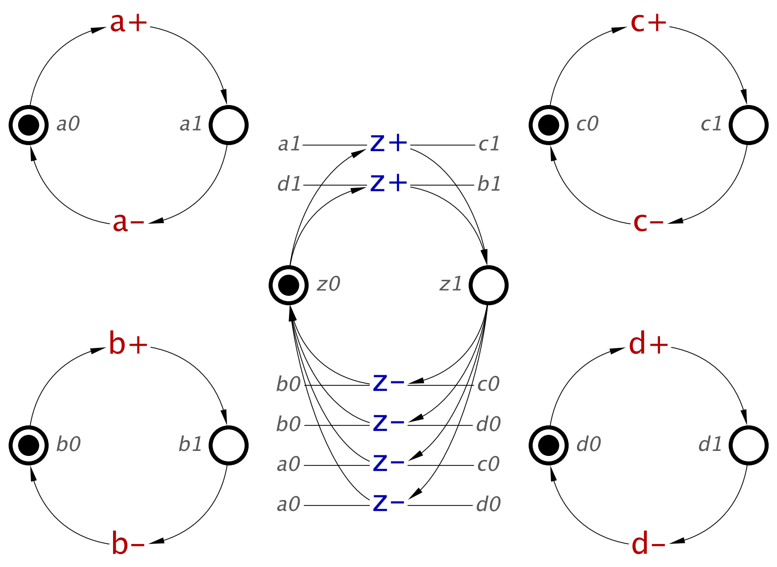
To show how difficult it can be to debug a more complex STG, we provide the converted STGs in both Z formation, in Figure 2.17a, and loop formation in Figure 2.17b. However, both of these examples contain the same errors, meaning it does not act as an AND-OR gate. Can you spot these errors in either of these STGs?

2.5 Monoids

Monoids are neither a graph formalism, nor a design method. Monoids are a structure in abstract algebra, on which the basis of Asynchronous Concepts is formed [29]. As such, in this section, we introduce monoids, listing and explaining some definitions and



(a) Z formation of an AO22 converted to an STG, with an error



(b) Loop formation of an AO22 converted to an STG, with an error

Figure 2.17: Equivalent STGs for an AO22, both featuring an error

notations that are used in this thesis. The information presented here will be built upon when describing the abstract base of concepts in Section 3.

We use $\mathbb{B}$ to denote the set of Boolean values $\{0, 1\}$. Given two Boolean functions $f : X \to \mathbb{B}$ and $g : X \to \mathbb{B}$ with the same domain X, we lift Boolean operators (disjunction $\wedge$, conjunction $\vee$, implication $\Rightarrow$, etc.) in the usual manner: $h = f \vee g$ means $h(x) = f(x) \vee g(x)$ for all $x \in X$,, etc. Furthermore, $\mathbf{0}$ and $\mathbf{1}$ stand for constant Boolean functions that discard their input and return values 0 and 1, respectively.

A monoid is a set M and a binary operation $\diamond : M \diamond M \to M$ satisfying two axioms:

- Identity: $e \diamond a = a \diamond e = a$ for any $a \in M$ where $e \in M$ is the identity element of the monoid.
- Associativity: $a \diamond (b \diamond c) = (a \diamond b) \diamond c$ for all $a, b, c \in M$.

Monoid is the simplest mathematical structure that captures the notions of emptiness and composition. Asynchronous Concepts introduced in Chapter 3 form commutative monoids: they have identity elements corresponding to empty specifications, and can be composed to build complex concepts from simpler ones. The order of composition does not matter, i.e., the concepts commute: $a \diamond b = b \diamond a$ for all $a, b \in M$.

2.6 Boolean functions

Boolean functions are used regularly throughout this thesis, as they feature heavily in design methodologies for asynchronous circuits. Here we briefly introduce some notations for Boolean functions which we will use.

Boolean functions consist of variables, constants and logic operations. The constants are the logic values of *true* or *false*, represented by 1 and 0 respectively. Variables can take on one of these values, but can be *negated* or *inverted*, which provides the opposite value than the current value for this variable. For example, a variable x can have value 0 or 1. If we it is said to have the value of 1, but is negated, indicated as $\overline{x}$, then the value is 0.

$$x = 1, y = 0$$
$$\overline{x} = 0, \overline{y} = 1$$

Logic operations are mathematical operations performed on the values of the variables. The two key operations used in this thesis are AND ($\wedge$) and OR ($\vee$). These take two

Boolean values, and depending on the operator, combine in certain ways. AND will return 1 if *both* input values are 1, and will return 0 otherwise. OR will take in two values and output 1 if *at least one* of the input values is 1, returning 0 when both inputs are 0.

Asynchronous Concepts utilise Boolean functions in several forms, and in this section we will discuss these. Two forms primarily used are *Conjunctive Normal Form* (CNF) and *Disjunctive Normal Form* (DNF). We use both forms in order to represent behaviours in Asynchronous Concepts in some way, and thus need to be able to convert between these. However, not every Boolean function is in either of these forms initially, and thus we need to be able to convert any function into one of these forms, which can then be converted to the other form if necessary.

CNF can be described as a form where the top-level logic operation is an AND. There are sub-functions within a CNF function which feature OR operations of variables. All sub-functions are ANDed at the top-level [30]. CNF functions are used to represent concepts, and thus, we provide an algorithm to convert from any Boolean function to CNF, discussed in Section 2.6.1.

DNF is the dual of CNF, the top-level logic operation is OR. Sub-functions are all ANDs of variables, and all sub-functions are ORed at the top-level [31]. DNF functions are used less than CNF in the language of Asynchronous Concepts, but are still used within the design flow, and we include a conversion method from CNF to DNF, discussed in Section 2.6.2.

2.6.1 Converting any Boolean function to CNF

Algorithm 1 details how a function is converted into CNF form. Consider the following example Boolean function:

$$function = (a \wedge b) \vee c$$

Using this function, we will follow Algorithm 1, and perform the operations manually to show the operation of this algorithm.

The algorithm begins by generating a table of all possible combination of true (1) and false (0) values for all the variables, a, b and c. Each of these combinations are then applied in turn to the function and evaluated, finding the true or false resulting value of the whole function.

Algorithm 1 Algorithm to convert a Boolean function to CNF

```
function CONVERT-TO-CNF(function)
    //Generate a list of all combinations of true/false for all variables
    values ← genVals(variables) in function
    for all values do
        //Evaluate each function. If it is true, ignore it.
        if evaluate(function, value) is false then
            for each value do
                //For the value of each variable
                if true then
                    //If true, OR the negated variable with the function
                    newFunction ← newFunction OR (NOT variable)
                else if false then
                    //If false, OR the non-negated variable with the function
                    newFunction ← newFunction OR variable
                end if
            end for
            //AND the new function with the function in CNF form.
            //This provides the top-level AND
            cnfFunction ← cnfFunction AND newFunction
        end if
    end for
    //Simplify for a more compact function.
    cnfFunction ← simplify(cnfFunction)
end function
```

If the first set of values for the variables is to set all variables to 0, then it is evaluated as follows. First, by replacing all of the variables with their value:

$$(a \wedge b) \vee c = (0 \wedge 0) \vee 0$$

The evaluation of this function with the given values will be 0. The truth table in Table 2.1, shows all combinations of values for the variables, and the evaluation values. The algorithm then aims to use any values which are *false* to generate the function in CNF form.

This section of the algorithm then uses the value of each variable used in the evaluation and uses it to generate a small function. Each variable is then inverted with respect to the values stated, and OR with the remaining function. For example, for *false false false*, the following occurs when generating the small function:

$function$ is empty

$a = 0$, therefore $function \vee a$

$function = a$

$b = 0$, therefore $function \vee b$

$function = a \vee b$

$c = 0$, therefore $function \vee c$

$function = a \vee b \vee c$

This is then ANDed with the whole CNF form of the set function. After this $cnfFunction$ will contain $a \vee b \vee c$ only. Each individual function for the different combinations can be found in Table 2.1.

Table 2.1: Truth table for function $(a \wedge b) \vee c$

a value	b value	c value	evaluation value	sub function if 0
0	0	0	0	$a \vee b \vee c$
0	0	1	1	N/A
0	1	0	0	$a \vee \overline{b} \vee c$
0	1	1	1	N/A
1	0	0	0	$\overline{a} \vee b \vee c$
1	0	1	1	N/A
1	1	0	1	N/A
1	1	1	1	N/A

The full CNF function, following the completion of the evaluation of all possible combinations of values, will include all of these sub functions ANDed as follows:

$$cnfFunction = (a \vee b \vee c) \wedge (a \vee \overline{b} \vee c) \wedge (\overline{a} \vee b \vee c)$$

This function can be used in it's current form. However, as the number of variables increases, the size of the function may also increase, leading to a higher-level of complexity than necessary. We therefore simplify the function on line 16. The simplified CNF function will be:

$$cnfFunction = (a \vee c) \wedge (b \vee c)$$

2.6.2 Converting CNF functions to DNF functions

While different in form, the CNF and DNF functions are similar and related. A CNF function can be converted to DNF using the Cartesian Product. Each sub-function in a

CNF function is part of the Cartesian product, each ORed variable in these is ANDed with each variable in every other sub-function. Each sub-function produced is then ORed, which is the top-level, producing the DNF form.

Using the CNF function from the previous example, we can convert this to DNF. Applying the Cartesian product results in the following function:

$$dnfFunction = (a \wedge b) \vee (a \wedge c) \vee (c \wedge b) \vee (c \wedge c)$$

This is a correct DNF function, but as with the conversion to CNF, if there are many variables, this could be much longer and more complex. We can simplify this however, for example, $(c \wedge c)$ simplifies to c. The simplified DNF function for this function is:

$$dnfFunction = (a \wedge b) \wedge c$$

This result is the same as the function we initially converted to CNF. This helps to show that these conversions are correct, but also that we can use Algorithm 1 in order to convert DNF functions, as well as any other Boolean functions to CNF. This is is the process used by tools developed for Asynchronous Concepts, as discussed in Chapter 5

2.7 Haskell syntax

Plato, the tool which includes the circuit-specific concept library and performs the automatic translation, is written in the functional programming language Haskell [15][16]. It compiles concept specifications, which themselves are Haskell files, and thus use Haskell syntax.

The examples of the Asynchronous Concepts language used within this thesis are therefore written primarily using Haskell syntax, which we discuss in this section in order to ensure that these examples can be fully understood. Some implementation details of the language, such as how certain concepts are derived, and the algorithms which perform various operations on concept specifications, will also be discussed, and use this syntax, as well as some standard Haskell functions. These will be identified and discussed further when used.

Primarily, Asynchronous Concepts are written as functions in Haskell, which may take in one or more parameters and return a concept, composable with other concepts. Functions in Haskell do not require parentheses or comma separations when passing in parameters. For example, a function which adds two integers can be defined as:

```
add x y = x + y
```

`x` and `y` are the parameters in for this function, and do not use parentheses. If a function such as `add` is used, it will simply be referenced in the form of `add x y`. Concept functions are used in the same way.

Some functions can also be used as a binary-operator, with one argument either side of the operator. A function such as this must only have two parameters, however. `add` for example, can be replaced with the operator `+`, and `add x y` can be performed by using `x + y`. In Plato, we add a few operators for our functions where appropriate. These will be explained in Chapter 3.

Lists are also commonly used in Asynchronous Concepts, for listing signals and events. Lists are denoted using square brackets, using commas for separation of each element in the list. For example:

```
[a, b, c]
```

Finally, Haskell does not support postfix notation, where functions or operators can be applied to parameters written before this function or operator. This occurs often in Plato, as the conventional notation for signal transitions is `a+` and `a-`, as is shown in STGs (Section 2.3), with `+` indicating a rising transition, and `-` indicating a falling transition.

Due to this, we include in Plato the functions `rise` and `fall`, which take one parameter, of the form of `rise a`. This correctly replaces the postfix operators, but can become confusing. For clarity, throughout the beginning of this thesis we will continue to use the postfix operators, in the form `a+` and `a-`.

2.8 Summary

We have discussed the background of the design of asynchronous circuits in this chapter. This includes FSMs and PNs which are higher level models for systems which do not necessarily feature interactions at the signal level. Derived from FSMs and PNs is state graphs and STGs respectively, which do show interactions for circuits at the signal level.

state graphs have some disadvantages however, such as not being able to show large levels of concurrency, with state explosion leading to large and complex state graphs. This leads to them not being used as commonly for asynchronous circuit design, thus we turn our focus to STGs.

STGs are most commonly used for designing asynchronous circuits, and multiple software tools exist to automate the verification and synthesis of these models. The design method using STGs is mostly monolithic however, meaning that larger designs become harder to debug, and useful sections cannot be easily reused in other designs. These disadvantages are what we aim to combat with the proposed methodology.

Digital circuits are also discussed, which the design flows mentioned in this thesis are aimed at creating, through specification, simulation, verification and finally synthesis, which provides us with a digital circuit implementation. Digital circuits can be used to automatically generate an STG, which can give us some details about the signal interactions.

These generated STGs can be in one of multiple forms, most notably the Z and loop formations, which indicate the state of an individual signal, using read arcs to connect places in a formation to transitions which require a signal in this state.

Monoids are the mathematical construct on which Asynchronous Concepts are based, and we introduce them to explain the idea that a concept is a monoid and is composable with another concept, forming a concept, which again is composable, and so on.

A brief background on Boolean functions is explained, as throughout this thesis we use Boolean functions. The causality captured by Asynchronous Concepts will be in CNF form, but for translation to a modelling formalism, we will need them in DNF form. Manipulation of Boolean functions is used often with the proposed design flow, and as such we need to be able to convert functions of any form to CNF, and CNF to DNF, and we introduce our methods of converting these.

Finally, we discuss Haskell syntax. Haskell is a functional programming language which has been used to implement the domain-specific language of Asynchronous Concepts, and as such, concepts themselves are written using this syntax. In this thesis, we will use many examples of concept specifications, and as such, some information on the syntax may help the reader understand how the concept specification works.

Chapter 3

Asynchronous Concepts

Asynchronous Concepts are a domain-specific language which was introduced [13] to provide a fully compositional design method which is highly reusable. These concepts are used to describe the behaviour of signals in asynchronous systems at signal-, gate- and protocol-level. These behaviours can be used to ensure that a resulting specification contains all of the behaviours applied to each signal, without the need to fiddle with a multitude of arcs, places and transitions which can occur with the monolithic approach of Signal Transition Graphs (Section 2.3.1).

This chapter is the main contribution of the thesis. It contains the definitions of Asynchronous Concepts in Section 3.1. This introduces how the library of circuit-specific concepts is built, and their use as a behavioural specification language for asynchronous circuits. We discuss the different levels of concepts; signal-, gate- and protocol-level through a library of circuit-specific concepts. This provided library includes concepts to describe some common behaviours and gates for asynchronous circuits. Through these we will show how reusable and composable concepts can be.

Asynchronous Concepts can be used to specify some standard logic gates with two inputs. However, these can have any number of inputs, thus they do not need to be defined as having only a certain number. We therefore provide some concepts for multiple input gates, discussed in Section 3.2. These use generalised forms of the standard gate-level concepts which can therefore apply to any number of inputs.

In some cases, a more complex system may be used as part of a specification, for which Boolean expressions are known. Rather than manually determine a concept specification, these functions can be passed to a higher-level concept and used in-line with other concepts. This provides flexibility to concepts, if a particular part of a system has been specified previously, or the implementation already exists, the Boolean expressions can be used instead of performing the lengthy procedure of respecifying this in concept form.

In practice, the protocols and gates we introduce are not always used as defined, and some variations on these can be used, for example, a gate may include an inverted input or output. It is possible to capture a variation as another concept definition, but this may need to be done for multiple variations on the same gate, which leads to lengthy and time-consuming specifications. It may be simpler to use the same concept, but with some transformations performed on the concepts involving an affected signal or signals. This extends the scope of the circuit-specific library, for example allowing for a standard gate-level concept to include the inverted output version, or the dual. These higher-level concepts and transformations will be discussed in Section 3.3.

Finally, this chapter discusses *Abstract Concepts*. These are the base level of concepts which identify the simple behaviours that a set of concepts should describe. Abstract concepts can be used as a starting block with the intention of using them to develop new domain-specific languages for Concepts for the purposes of specifying different types of asynchronous circuit. These are discussed in Section 3.5.

3.1 Circuit-specific concepts

This section discusses the library of circuit-specific concepts which have been defined and are included the tool Plato, covering signal-, gate- and protocol-levels. We will discuss their implementation, and their uses within asynchronous circuits.

These concepts are used to describe the behaviours of asynchronous circuits, and such, the alphabet of this domain is based on digital signals, and the transitions of these, the rising (+) and falling (-) transitions. A global state can be determined at any point during the operation of such a system based on the latest transitions of each signal. This information is used to determine causalities, as discussed later in this chapter.

3.1.1 Signal-level concepts

Signal-level concepts form the atomic level of Asynchronous Concepts. These can be composed to produce higher-level concepts, but cannot be decomposed further. Any specification can be composed wholly of signal-level concepts, but this can lead to long and complicated specifications. However, some of these concepts will be necessary for every specification, as they describe the basic information required for any circuit: *Interface* and *Initial State*.

Composition

Throughout this chapter we will be using compositions of concepts in order to define new concepts. The operator used for this is: <>. Every concept is a monoid (Section 2.5) and thus every concept can be combined using the <> operator.

Composition is used to combine initial states, the invariant and the interface of multiple concepts. This can have some interesting interactions, for example, if two subsystems feature common signals, but defined with different interfaces in each, then there is a hierarchy defined which determines the post-composition interface of each signal (see *Interface Concepts*).

Behaviours are also combined when composed. This allows a designer to specify one important behaviour at a time, and compose several to produce a specification which features all behaviours interacting together, as discussed in the *AND Causality* section.

Every concept featured in this chapter is composable, and the result produces a concept which again can be composed, and so on. This means that a designer can derive a concept which features many behaviours, composed of many concepts, and refer to this simply by a given name, and compose this with other concepts. The behaviours then composed, without needing to define them all once again. This reusability, and the highly compositional nature of Asynchronous Concepts will be seen throughout this chapter, as we discuss the derivation of each concept in the circuit-specific library.

Interface concept

Interface is important for specifications, detailing how the device being specified interacts with the outside world. Signals can be classified as either *input*, *output* or *internal*. This can help to quickly identify errors (e.g. an input transition is caused by a hidden internal transition), and reuse existing tools for circuit simulation, verification and synthesis. Signal type information is also used in the algorithm for automated translation of concepts to STGs (Section 5.1.1)

Input signals generally identify signals that come into the system from the environment. Outputs are those signals that are produced by the system and used to affect the environment. Internals are signals which are the outputs of part of the system, such as a gate, but are used simply to affect other parts of this system, and are not provided paths to the environment.

As interface is specific to circuits, which need to feature the types of each signal, we therefore introduce a primitive Interface concept. This can have one of four values: Input, Output and Internal as expected, but also Undefined. Each signal in the system is Undefined initially, until it is defined as any other type. This is helpful for the translation process, as a signal without a type cannot exist in a specification.

From this primitive, we then derive three concepts to describe signal types. These are `inputs`, `outputs` and `internals`. Signals which are of each of these types are listed. For example, if we have a system containing three input signals, `a`, `b` and `c`, one output signal, `z`, and a single internal signal, `t`, then these concepts would be defined as follows:

```
inputs [a, b, c] <> outputs [z] <> internals [t]
```

Note that interface concepts are commutative: these concepts in any order would produce an identical specification.

These concepts could be used for a specification featuring two C-elements, one with inputs `a` and `b`, and output `t`, the second takes as inputs `c` and `t` and outputs `z`. `t` could be used by another subsystem but never be available to affect the environment. The circuit diagram for this can be seen in Figure 3.1.

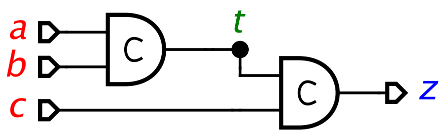


Figure 3.1: Example of interface in a digital circuit

For the interface concept, we impose a set of rules in the event that multiple concept specifications are combined. Signals which occur in one specification can occur in another, both with their own interface concept. These rules define what the resulting interfaces are in the event that two differing interface concepts for a single signal are composed, this is in the form of a hierarchy.

Table 3.1 details the results when two interface concepts are composed. It indicates that two of the same concepts composed will produce the same interface type concept. If

◇	Input	Output	Internal
Input	Input	Output	Internal
Output	Output	Output	Internal
Internal	Internal	Internal	Internal

Table 3.1: The hierarchy of interface concepts

the two concepts differ, the hierarchy indicates that internal is the highest type, followed by output, with input as the lowest. The intuition is as follows:

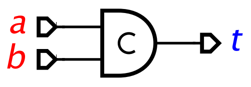
- If a signal is an input in one component of the system, but is an output in another component, then in the composition it will be an output.
- An internal signal is similar to an output signal in the sense that it is driven by the circuit (not the environment), but it is hidden, i.e. not accessible via the circuit interface. Once a signal is hidden and declared internal it cannot be revealed.

Thus, an input composed with any non-input type will become the non-input type. An output composed with an internal will become an internal, but remain an output if composed with an input, and an internal composed with anything will remain an internal signal.

If we take the example from Figure 3.1 and split it into two specifications, the first featuring a single C-element with signals defined as inputs, a and b, and the output, defined as such, t. The result can be seen in Figure 3.2

```
inputs [a, b] <> outputs [t]
```

(a) t is defined as an output



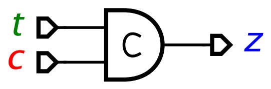
(b) Circuit diagram

Figure 3.2: First example specification with t as an output

The second specification takes c as an input signal, t as an internal signal, both of which are inputs to the C-element, signal z as the output. The second specification is a continuation of the first, using t as an internal signal, and therefore forcing it to be an internal signal instead of an output, usable by the system, but not viewable by the

```
inputs [c] <> outputs [z] <> internals [t]
```

(a) `t` is defined as an internal



(b) Circuit diagram

Figure 3.3: Second example specification with `t` as an internal

environment. The interface concepts and the circuit diagram for this specification can be viewed in Figure 3.3.

Composing both of these specifications would mean the signal `t` will be defined as internal, and the full circuit diagram would be as seen in Figure 3.1.

Initial state concept

Initial state is another important concept needed for specification of asynchronous circuits, as it indicates what the first transition of each signal and the first transition of the system is. Without the initial state of a system being known, no signal transitions can occur, and the specification cannot be used in further processes, such as combination with other specifications, verification and synthesis.

We provide the `initialise` concept. This takes a signal as an argument, and a Boolean value, $\mathbf{0}$ or $\mathbf{1}$. This therefore applies the selected initial value to the signal in question.

`initialise` a $\mathbf{0}$ <> `initialise` x $\mathbf{1}$

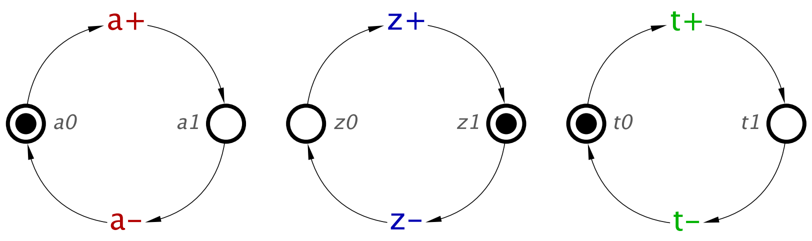
For convenience, we introduce two concepts to list signals by initial state: `initialise0` and `initialise1`. Listing the signals by those which are initially $\mathbf{0}$ and those which are initially $\mathbf{1}$ is a shorter method than using a single `initialise` concept for each individual signal, which for systems with more than a handful of signals, can lead to a large concept specification.

```
initialise0 [a, b, c] <> initialise1 [x, y, z]
```

Both interface and initial state must be declared in each specification, which will then allow a specification to be used in further process, such as translation to an STG or state graph, and in synthesis. From this concept onwards, we will use translated STGs from the given concepts to show how the concept in question affects the STG.

```
initialise0 [a, t] <> initialise1 [z] <>
inputs [a] <> outputs [z] <> internals [t]
```

(a) Concepts specification for interface and initial state



(b) Translated STG showing interface and initial state

Figure 3.4: Concepts and the translated STG for interface and initial state

Figure 3.4a contains a concept specification consisting of only initial state and interface concepts. These are the minimum requirements for translation of a specification to occur, thus we provide the translated STG in Figure 3.4b. The algorithm to perform this translation is discussed in Section 5.1.

The initial state concepts determine the placement of tokens in a resulting STG. `initialise0 [a, t]` translates to, in Figure 3.4b, signals `a` and `t` containing tokens in their 0 places (`a0`, `t0`). Signal `z` contains a token in its 1 place (`z1`), as the concept for this is `initialise1 [x]`. This gives the indication that `a` and `t` can initially transition high, and that `z` can transition low. This is the effect that the initial state concepts have on a resulting STG.

The interface concepts are `inputs [a]`, `outputs [z]` and `internals [t]`. The types of the signals are indicated by their colours in the STG as discussed in Section 2.3; red identifies an input, blue identifies an output and green identifies an internal signal. Thus, from the translated STG in Figure 3.4b we can determine that `a` is an input, `z` is an output and `t` is an internal signal.

For clarity, all concept examples for the remainder of this chapter will omit the interface and initial state concepts. While they must be included for a correct translation to an STG, we do this to focus primarily on the concepts we introduce and discuss. Some signal types may be stated in the text, but we assume that all initial states are **0**.

Causality concept

With the interface and initial state concepts, we also need concepts to describe the interactions between signals in a specification. A *causality* concept we use to refer to a concept which specified the interactions between signal transitions. We say that a transition $effect \in E$ causally depends on transition $cause \in E$. A causality function is provided:

```
causality cause effect
```

$effect$ can occur only in states that are after $cause$. For example, if an input signal `a` transitioning high *causes* an output signal `z` to transition high, we say that `a` transitioning high is the *cause* event and that `z` transitioning high is the *effect* event. This can be written using the `causality` concept:

```
causality a+ z+
```

Causality has an operator for convenience and clarity: `~>`. This is used in the form `cause ~> effect`, where cause and effect are signal transitions. Using the above example, we can instead state the causality concept as:

```
a+ ~> z+
```

The causality operator will be used throughout the rest of the thesis, instead of the function.

Figure 3.5 contains the translated STG for this concept. This STG includes an arc, connecting place `a1` with the signal transition `z+`. This is a *read arc*, a double-ended arc which in this case, will allow `z+` to transition only when `a1` contains a token. A token will be placed in `a1` only after `a+` has occurred, moving the token from `a0` to `a1`. `z+` can then occur, however this will not consume the token from `a1`, as this would block `a` from transitioning further. Instead, it simply checks for the token in order to allow `z+` to occur.

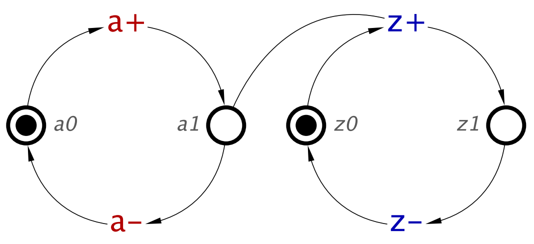


Figure 3.5: Translated STG with a single causality concept

AND causality

AND causality occurs when more than one causality concepts with a common effect transition is composed. The cause transitions combine, requiring all of the cause transitions to occur before the effect can occur. For example, in a system containing three signals, inputs a and b, and output z, we have the following concepts:

a+ ~> z+ <> b+ ~> z+

One of these causality concepts indicates that for z+ to be excited, the system must be in a state where a+ has already occurred. The other states that z+ is excited in states where b+ has occurred. Composing these concepts results in a system where for z+ to be excited, the system must be in a state where both a+ and b+ has already occurred.

The AND-causality in this example is viewable in the translated STG, Figure 3.6, where z+ features read arcs connecting it to both a1 and b1. This ensures that a+ and b+ have both occurred, indicated by tokens in these places which will allow z+ to occur.

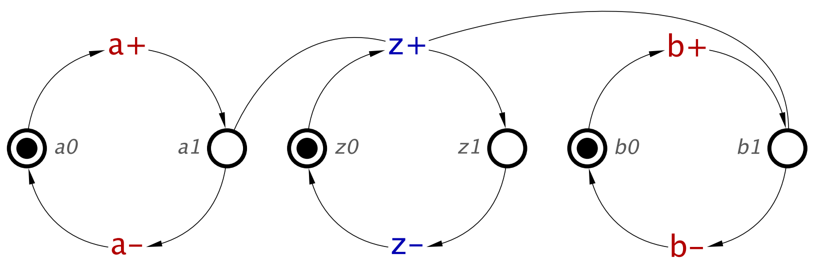


Figure 3.6: Translated STG containing AND causality

For ease of use, and smaller specifications, we also provide an operator for multiple cause events causing a single effect event. This operator is ~&~>. This is used by listing the cause events before the operator, for the single effect after the operator, such as [cause1,cause2, cause3, ...] ~&~> effect. The example above can therefore be rewritten as:

```
[a+, b+] ~&~> z+
```

These two concept representations are equivalent.

OR causality

OR causality is the dual of AND causality. Where AND causality is a collection of *required* causes for an effect, OR causality is a collection of *possible* causes for an effect. A minimum of one of the possible cause transitions is required to occur for the effect transition to occur.

For example, if there are two input signals, a and b, and *at least* one of these is required to transition high in order for an output signal, z to occur. We can describe this behaviour using the orCausality function, providing a list of the *possible* cause transitions, and the single effect transition:

```
orCausality [a+, b+] z+
```

In words, this states that *either* a rise in a, or a rise in b is required for the rising transition of signal z to be excited.

We introduce a new operator for OR causality: ~|~>. The use of this is to list the possible causes for an effect, such as [cause1,cause2, cause3, ...] ~|~> effect. one or more of the listed causes can occur in order for the effect to occur. The example above which uses the orCausality function for this OR causality concept can therefore be rewritten as:

```
[a+, b+] ~|~> z+
```

A translation of OR-causality to an STG provides an interesting result, containing multiple effect transitions. If we translate the example concept to an STG, the result will be as shown in Figure 3.7. The output, z, has two separate rising transitions, both are connected to the places in the loop, allowing either to transition and the signal to continue transitioning thereafter. Each of these z+ transitions are connected via read arc to one of the possible cause transitions. The left z+ transition connects to a1, requiring only a+ to have occurred for this to transitions, the right z+ transition connects to b1, and therefore only requiring that b+ has occurred in order for this to transition. Whichever z+ transition fires will pass the token from z0 to z1.

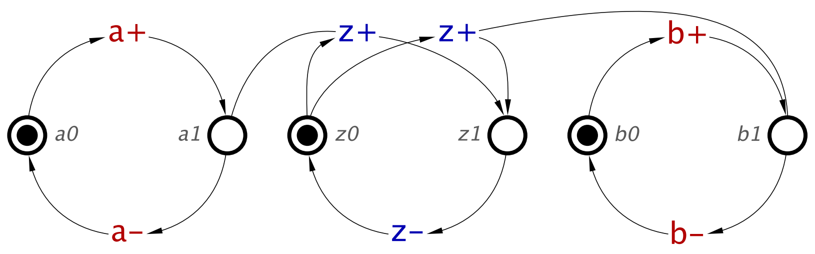


Figure 3.7: Translated STG showing OR causality

Never concept

The never concept is used to express a state of the system which violates the invariant. For example, let's say that signals a and b are mutually exclusive, i.e. we do not want these signals to both be high at the same time.

The never concept is used to list the signals which must not be in certain states at the same time. So, for the mutually exclusive signals a and b, we provide the concept:

```
never [a+, b+]
```

This concept states that any state where a+ and b+ have occurred, or any state where a and b are both 1 violates the invariant, and should not be allowed.

This concept does not produce any places, transitions or arcs in a translated STG in order to block this state from being reached, but it is important nonetheless for the purposes of verification. The information contained in never concepts can be passed into other tools to automatically verify that this state is not reachable. This process will be discussed in Section 4.7.

3.1.2 Gate-level and Protocol-level concepts

The signal-level concepts discussed in Section 3.1.1 are atomic, these cannot be broken down further. A concept specification can be authored entirely from these concepts, but for larger specifications this can make such a specification difficult to comprehend.

Using the example we have used before of a 3-input C-element, with inputs a, b and c and output z. One possible concept specification for this is:

```
example2 = outRise <> outFall <> init <> interface
  where
    outRise   = [a+, b+, c+] ~&~> z+
    outFall   = [a-, b-, c-] ~&~> z-
    init      = initialise0 [a, b, c, z]
    interface = inputs [a, b, c] <> outputs [z]
```

This is a large specification for a simple 3-input gate, even when using the shorter notation for AND causality and initial state. If this is to be used as just one part of a system, it is far too large to define each time.

As part of the library of circuit concepts, we define some derived concepts which specify commonly used gates and protocols, composed of the signal-level concepts, and other gate and protocol concepts. These can make describing the behaviours of asynchronous systems more convenient, and prove the reusable nature of concepts.

Buffer concept

A buffer is a simple gate, containing one input and one output. The output reacts to the input, following its transitions; when the input transitions high, the output transitions high, and vice versa.

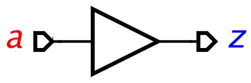


Figure 3.8: Buffer circuit

The concept specification of a buffer in this library is composed of two causality concepts. A user can then specify their initial states and the desired interface. The causality concepts for a buffer are defined as follows:

```
buffer a z = a+ ~> z+ <> a- ~> z-
```

For each of `z+` and `z-` there is a single cause transition. The STG for a buffer can be found in Figure 3.9. This features a read arc connecting `a1` to `z+`, satisfying the first concept, `a+ ~> z+`. A second read arc connects `a0` to `z-`, which satisfies the second concept `a- ~> z-`.

Note that the buffer concept does not restrict the behaviour of the input, it only specifies the behaviour of the output in relation to the input. The input signal can transition at

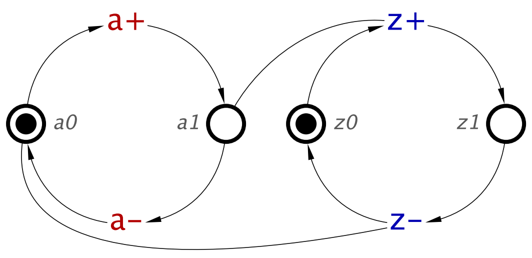


Figure 3.9: Translated STG of a buffer

any time with no respect to the output signal. To Restrict the environment the designer can use a separate concept.

This buffer concept can now be reused in any concept specification. `buffer a z`, replacing the signals `a` and `z` with the chosen input and output signal names from the target specification will automatically specify the buffer behaviour between the two chosen signals.

Inverter concept

An inverter is similar to a buffer, it has one input and one output. However, in this case, the output transitions the opposite way to the input, the output inverts, or negates, the input. When the input transitions high, the output transitions low, when the input transitions low, the output transitions high. Inverters are a very standard gate as signals are often inverted in digital circuits.

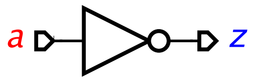


Figure 3.10: Inverter circuit

Similar to the buffer concept specification, the inverter specification contains two concepts, the effect transitions for the output signal are the opposite however.

```
inverter a z = a+ ~> z- <> a- ~> z+
```

Figure 3.11 contains the translated STG of the inverter concept. The concept does not contain any concepts for the interface or the initial state, thus for this STG we assume that the input is signal `a`, the output is signal `z`. We also assume the initial states for both signals are 0, however, this means that the `z-` transition will immediately be able to occur, as `a0` will initially contain a token.

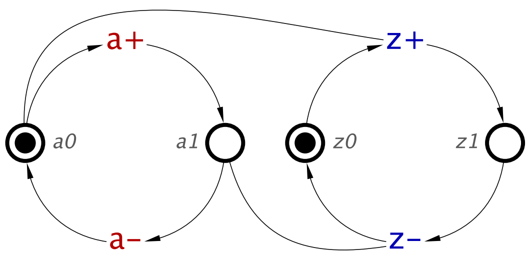


Figure 3.11: Translated STG of an inverter

As the system initialises, the state of a will become 0, and the output of the buffer, z will transition high as a result. A specification may include initialise1 [z] to automate this behaviour, but it is not required. This concept can be reused similarly to the buffer concept.

Handshake concept

A handshake is a protocol, often used in asynchronous circuits to communicate between devices or components of a circuit, as shown in Figure 1.1b. One device may send a request, indicated by setting a signal high, the receiving device will send an acknowledgement of receiving this request by setting one of its own signals high. When the requesting device has completed its requested operation it will set this signal low, and the receiving device will acknowledge this by setting its acknowledgement signal low as well.

It is ideal to include a concept specifically for handshakes, as it is a very common pattern used in asynchronous systems. A handshake consists of two signals, commonly known as request, r, and acknowledge, a.

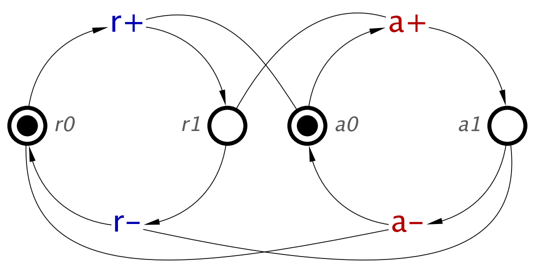
```
handshake r a = r+ ~> a+ <> a+ ~> r- <> r- ~> a- <> a- ~> r+
```

The concept for a handshake is composed of four concepts. r+ ~> a+ shows the receiver acknowledging the request. a+ ~> r- shows that following the acknowledge, that the request will be rescinded some time after. r- ~> a- indicates that following the request being removed, the acknowledgement will also be set low. Finally, a- ~> r+ is the reset of the handshake, allowing the process to start again when the requester requires.

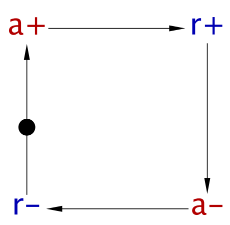
These concepts are descriptive of the behaviour of a handshake, yet it can be expressed in another way. Instead of signal-level concepts, these can be replaced by gate-level concepts, describing the behaviour of a handshake on at a gate-level. The concepts `r+ ~> a+` and `r- ~> a-` are the same concepts as featured in a buffer concept. The other two concepts, `a+ ~> r-` and `a- ~> r+` are the same concepts as featured in an inverter concept. Thus, the handshake concept can be redefined.

```
handshake r a = buffer r a <> inverter a r
```

This example shows that using concepts, a specification can be expressed in multiple ways, depending on the preference of an author. It also shows the reuse of concepts, in this case it is reusing concepts from a library, but this can apply to concepts written by a user, stored in their own libraries of concepts.



(a) Translated STG of a handshake



(b) Resynthesized handshake STG

Figure 3.12: The translated STG of a handshake, and its resynthesized version

The translated STG (Figure 3.12a) features mirrored read arcs. The connection between `r1` and `a+` and the connection between `a0` and `r+` are mirrored showing that when signal `r` goes high, `a` will go high after, and when `a` goes low, `r` will go high after, resetting the handshake. The read arc between `r0` and `a-`, and the mirrored read arc between `a1` and `r-` show that when `r` goes low, `a` will follow, and when `a` goes high, `r` will go low after, when its request has completed.

Figure 3.12b is an equivalent handshake STG, produced from the STG in Figure 3.12a by *Resynthesis*. Resynthesis can be used to find a smaller model by decomposing a model and recomposing it of selective components [32]. This produces a much more compact handshake, which is clearer and more recognisable as a handshake pattern.

The handshake concept can be reused, replacing `r` and `a` with signals from the target specification which act as request and acknowledge signals.

C-element concept

C-elements are commonly used in asynchronous circuits, as they are used to synchronise signals, the output going high when *all* inputs are high, the output going low only when *all* input signals go low.

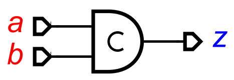


Figure 3.13: C-element circuit

For the circuit-specific concept library, we provide a 2-input C-element. For this example, the inputs are `a` and `b`, the output is `z`. The concept specification for a C-element is as follows:

```
cElement a b z = a+ ~> z+ <> b+ ~> z+ <> a- ~> z- <> b- ~> z-
```

This concept features AND causality, both `z` transitions requiring both `a` and `b` to transition first, `a+` and `b+` for `z+`, `a-` and `b-` for `z-`.

As with the handshake concept, there is another way of representing a C-element. If we look at the concepts by common cause transitions, we see that `a+ ~> z+` and `a- ~> z-` are the concepts for a buffer of `a` and `z`. Similarly, `b+ ~> z+` and `b- ~> z-` form a buffer of `b` and `z`. Thus, we can redefine a C-element concept.

```
cElement a b z = buffer a z <> buffer b z
```

This form of the specification for a C-element does not show the implementation, but shows the behaviour. Through the AND-causality between these two buffer concepts, we describe that `a` and `b` both transitioning high or low will cause the same transition in `z`.

The translated STG can be seen in Figure 3.14, and shows this AND causality. Looking at one input signal and the output signal shows the similarities to the translated buffer STG (Figure 3.9).

This concept can be reused for any concept specification, replacing the inputs with the signals to be synchronised, and the output with the signal used to identify this synchronisation. Using the three-input C-element example as we have used before 2.12, we can now define this using this `cElement` concept.

```
example3 a b c t z = cElement a b t <> cElement t c z
```

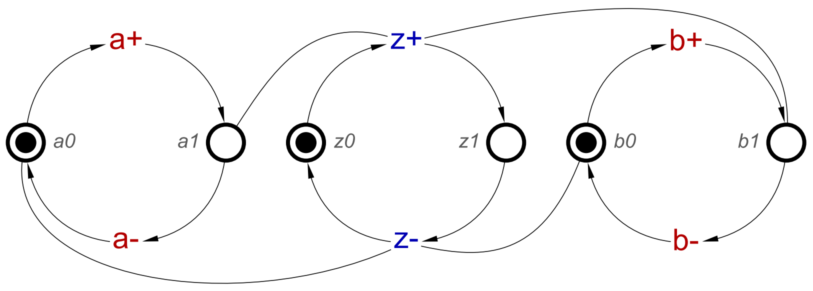


Figure 3.14: Translated STG of a C-element

OR gate concept

An OR-gate is a standard logic gate used in digital circuits, and it features OR causality. The output of an OR gate goes high when *at least* one of the inputs has gone high, and the output only goes low when all of the inputs are low.

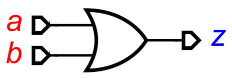


Figure 3.15: OR gate circuit

The circuit-specific concept library contains a 2-input OR gate. For this example, we have input signals `a` and `b`, and output signal `z`. The concept specification for an OR gate is:

```
orGate a b z = [a+, b+] ~|~> z+ <> [a-, b-] ~&~> z-
```

This specification is composed of OR causality for the set phase of the gate, and AND causality for the reset phase. For `z+` to occur, either `a+`, `b+`, or both must have occurred. For `z-`, both `a-` and `b-` must have occurred.

Both the OR and AND causalities are evident in the translated STG, found in Figure 3.16. It can be seen that `a1` and `b1` connect to separate `z+` transitions, indicating that these are the two possible causes of `z+`. For `z-`, there is only one transition, connected via read arc to both `a0` and `b0`, which is the AND causality requiring both input signals to be low for the output to go low.

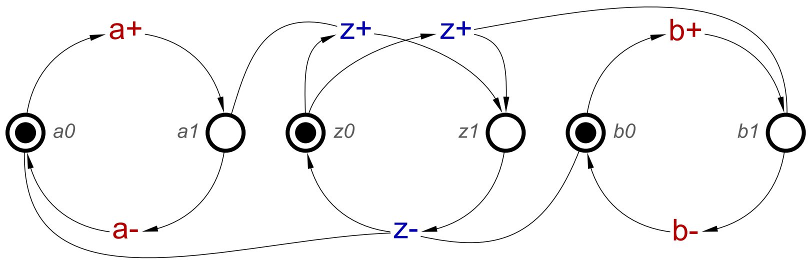


Figure 3.16: Translated STG of an OR gate

AND gate concept

AND gates are similar to OR gates, however, their operation is the dual. The output goes high when all input signals are high, going low when any of the input signals go low. Therefore, the AND gate also features OR causality.

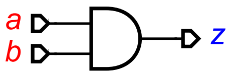


Figure 3.17: AND gate circuit

As with the C-element and OR gate, the AND gate concept featured in the circuit-specific library is a 2-input AND gate. The signals for our example are a and b as inputs, z as the sole output. The concept specification is:

```
andGate a b z = [a+,b+] ~&~> z+ <> [a-,b-] ~|~> z-
```

The set and reset phases of the AND gate are the dual of those for the OR gate, with AND causality required for z+, both a+ and b+ must have occurred, and OR causality used for z-, needing at least one of a- and b- to have occurred.
Figure 3.18 contains the translated STG for this AND gate concept specification. Comparing this to the OR gate STG (Figure 3.16), these STGs look like they have been mirrored around the central horizontal axis. The AND causality in this STG is apparent for z+, being connected to a1 and b1, thus requiring both a+ and b+ to have already occurred. In this STG, the OR causality affects the z- transition, providing two separate transitions, one connected to a0, so it can transition low after a has, and one connected to b0, so it can transition low when this has. This concept can now be used for any

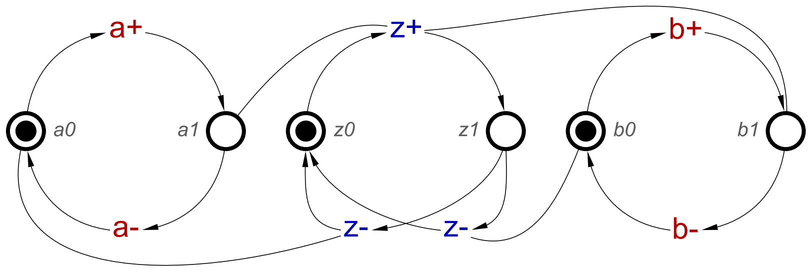


Figure 3.18: Translated STG of an AND gate

three signals, using the concept `andGate a b z`, replacing `a` and `b` for the inputs, and `z` for an output.

As stated, an AND gate is dual of an OR gate. Therefore we can simplify the AND gate concept further:

```
andGate a b z = dual (orGate a b z)
```

This uses the `dual` function which inverts every cause and effect transition in a given concept, thus swapping the OR and AND causalities in the OR gate concept, providing an AND gate. The dual function will be discussed in further detail in Section 3.3.3.

XOR gate concept

An XOR gate, or *Exclusive OR* gate is a slightly more complex logic gate. This gate sets its output high when exactly one of the inputs are high. i.e. If the inputs are all high or all low, the output is low.

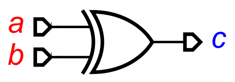


Figure 3.19: XOR gate circuit

The XOR gate concept included in the circuit-specific library is a 2-input gate. The two input signals are, `a` and `b`, and the one output is `z`. For an XOR gate, the best way to derive a concept is to look at the set and reset functions, the Boolean functions which define what causes the output to rise, the set function, and the output to fall, the reset function. The set function for an XOR gate is: $(a \wedge \overline{b}) \vee (\overline{a} \wedge b)$. In words, this is `a` being high and `b` being low, or `a` being low and `b` being high.

This can not directly be described with concepts, as the composition of concepts <> is an AND ($\wedge$) operation. Asynchronous Concepts do not feature an OR ($\vee$) operation. As discussed in Section 2.6 this function is in Disjunctive Normal Form (DNF), identified as having OR operations at the top-level. In order to describe this in concepts, and use the composition operator, we need to convert this function in DNF to *Conjunctive Normal Form* (CNF), which uses top level ANDs, as concepts do. Section 2.6 discusses this conversion in greater detail.

The result of the conversion of this function is: $(a \vee b) \wedge (\overline{a} \vee \overline{b})$. We can now use this to generate the concepts for an XOR output rising.

```
outRise = [a+, b+] ~|~> z+ <> [a-, b-] ~|~> z+
```

Now, we can perform the same operation for the reset function, which in DNF is: $(a \wedge b) \vee (\overline{a} \wedge \overline{b})$. Converting this to CNF produces: $(a \vee \overline{b}) \wedge (\overline{a} \wedge b)$. Producing the concepts for this we find:

```
outFall = [a+, b-] ~|~> z- <> [a-, b+] ~|~> z-
```

With the outRise and outFall functions, we can now produce the full concept for an XOR gate.

```
xorGate a b z = outRise <> outFall
```

The translated STG for this XOR concept can be viewed in Figure 3.20. Note that for both z+ and z-, there are two transitions. For the z+ transitions, one requires both a1 and b0 to contain tokens, and the other requires a0 and b1 to both contain tokens, as expected from an XOR gate. Similarly, one z- transition needs both a0 and b0 to contain tokens, the second needs a1 and b1 to contain tokens.
It is a generally useful technique to use the set and reset functions to generate concepts for a gate, as the gate itself may require a large and complex concept specification, so we offer the option to include these functions as concepts in a concept specification, or to generate a specification from the set and reset functions.

We provide some high-level concepts which can use Boolean functions, and we can use one to simplify the XOR gate concept further, complexGate. This will be discussed in more detail in Section 3.3.1. This function takes a set and reset function, and an output signal. This will then be used in translation to produce an STG. Therefore, we can derive the following concept for an XOR gate:

xorGate a b z = complexGate $((a \wedge \overline{b}) \vee (\overline{a} \wedge b))$ $((a \wedge b) \vee (\overline{a} \wedge \overline{b}))$ z

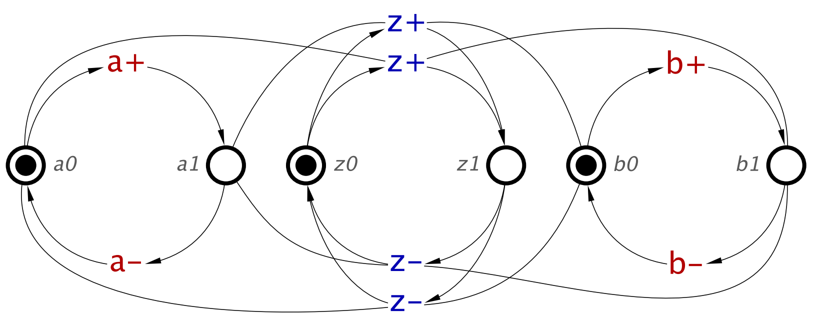


Figure 3.20: Translated STG of an XOR gate

Mutual exclusion concept

Mutual exclusion, or *mutex* for short, is a protocol between two signals which allows only one to be high at a time, these signals are said to be *mutually exclusive*. This exists for multiple reasons, for example, when a resource only allows access to one user at a time, if two users request access at the same time, mutual exclusion ensures that only one is granted access, the other gaining access when the first has finished.

For the concept library, we provide a two signal mutual exclusion concept. For this example, we have two output signals, as the signals to request access in such an example would be outputs of a system. All signals are stated to be initially 0.

```
mutex x y = x- ~> y+ <> y- ~> x+ <> never[x+, y+]
```

The causality concepts `x- ~> y+` and `y- ~> x+` require that for one of the signals to transition high, the other must have already transitioned low. These themselves guarantee that these signals will be mutually exclusive, but the `never` concept is included for automatic verification that these signals are mutually exclusive and are not both initially 1.

Figure 3.21 is the mutex concept translated to an STG. This model shows that for `x+` to transition, `y0` must contain a token, meaning that `y` must be low. This is the same for `y+` to occur, where `x0` must contain a token. Verifying this STG for a possibility where `x` and `y` are both 1 will prove that this is an unreachable state. This concept can now be included in any specification where signals are needed to be identified as mutually exclusive.

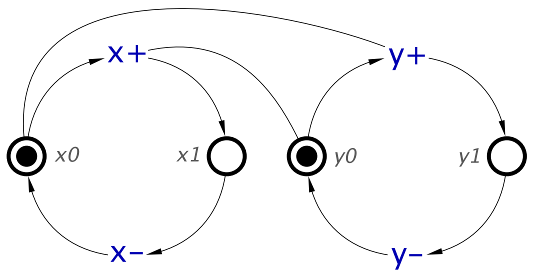


Figure 3.21: Translated STG of a mutual exclusion concept

Mutual exclusion element concept

A mutual exclusion element is a gate that implements the mutex protocol. It incorporates inputs in order to control the transitions of the mutually exclusive outputs. This gate was used as an example in Section 1.2, in the introduction, to show how an STG may be more difficult to understand, but concepts can be more indicative as they describe the behaviours.

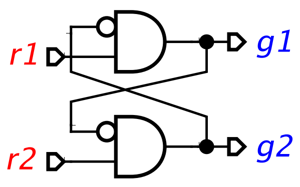


Figure 3.22: Mutual exclusion element (metastability filter omitted for clarity)

This gate has two inputs, `r1` and `r2`, which are request signals. They are inputs to this gate, coming from devices which request access to a resource. The outputs, `g1` and `g2` grant access to this shared resource, but only one at a time. When `r1` goes high, `g1` will pass the access, only if `g2` is not already high. When `r1` goes low, the requesting device has finished with the resource, relinquishing its hold of it, `g1` will then go low, allowing access to `g2`, if `r2` is high. This is the same for `r2` and `g2`. The concept specification for this gate is as follows:

```
meElement r1 r2 g1 g2 = buffer r1 g1 <> buffer r2 g2 <> me g1 g2
```

We use buffers in the specification of this gate as the behaviour of the grant signals is to react to their respective request signals, and leads to a more compact specification.

The mutual exclusion serves to make sure that the grant signals do not try to provide access to the shared resource at the same time.

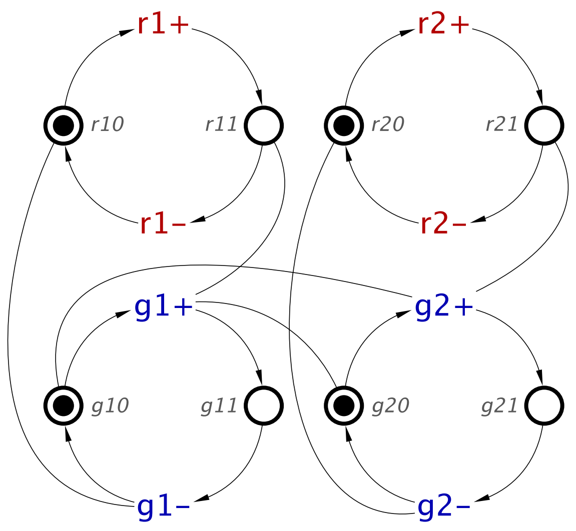


Figure 3.23: Translated STG of a mutual exclusion element

Figure 3.23 contains the translated STG for this concept specification. There are three areas to note. If we just take the signals `r1` and `g1`, and the read arcs connecting them, this is expected to be similar to a buffer, which compared to the STG of a buffer, as shown in Figure 3.9, is the same, but rotated. This is the same if we compare just the read arcs between `r2` and `g2`.

Now, taking just signals `g1` and `g2` and the read arcs between these two, we can see that this looks the same as the Figure 3.21, the mutual exclusion STG. These three separate views of this STG show the different effects the three concepts have, but composed produce a new gate-level concept, a mutual exclusion element.

3.2 Generalising to multiple inputs

In many cases, a standard logic gate may be used with more than two inputs. It is possible to use a gate as we have defined it in Section 3.1, but many internal signals will be included, which can make the resulting STG larger and more complex. Therefore, we introduce some concepts which allow any number of inputs, for a single output, for some standard gates. These are included in the circuit concept library, to supplement the existing gate concepts.

As discussed in Section 2.7, this generalisation to multiple inputs takes advantage of several Haskell functions, which aim to automate the process of defining gates with any number of inputs. These generally are used to iterate over any number of provided signals, and compose all of these. The Haskell functions in question will be discussed when used.

Multiple input C-element concept

As stated in Section 3.1, a C-element can be defined as a composition of buffers, making it a more compact concept. Using this, we can easily define a C-element which has any number of inputs. For example, a 3-input C-element can be defined as:

```
example4 a b c z = buffer a z <> buffer b z <> buffer c z
```

Adding another input will simply add another `buffer` concept. We can therefore define a new concept, `cElementN`, where N stands for any number of inputs. This takes a list of inputs, and a single output, and creates a `buffer` concept from each input to the output signal. The implementation of `cElementN` in Haskell is as follows:

```
cElementN ins out = mconcat (map (`buffer` out) ins)
```

The functions `mconcat` and `map` are Haskell functions, useful for this implementation. `mconcat` composes a list of monads, and `map` will apply each element of a list, in this case each element of `ins` to the function inside the brackets, in this case, `buffer`. This therefore produces a `buffer` concept for each element of `ins` as in the input, with the output for all being `out`. All of these buffer concepts are then composed by `mconcat`.

To produce the above concept for a 3-input C-element, we can use:

```
cElementN [a, b, c] z
```

As an example to show that this concept produces an equivalent 2 input C-element, we can use the concept `cElementN [a, b] z`. Using this, the internals of `cElementN` will produce the following composition of concepts:

```
buffer a z <> buffer b z
```

This is the same as the defined concept for a two input C-element.

Multiple input OR gate concept

OR gates are defined as a composition of OR causality and AND causality, listing the rising possible input transitions as OR causality for the output to rise, and all the required falling input transitions as AND causality for the output to fall. Manually, a three input OR gate concept would be defined as:

```
example5 a b c z = [a+, b+, c+] ~|~> z+ <> [a-, b-, c-] ~&~> z-
```

To define an OR gate concept for any number of inputs, it is a case of taking a list of the inputs, and defining all of their rise transitions as a possible causes for the output rising transition, and all the input falling transitions as required causes for the output falling transition. This concept is called `orGateN`. The implementation of `orGanteN` in Haskell is as follows:

```
orGateN ins out = map ins+~|~> out+ <> map ins-~&~> out-
```

This uses `map` once again. In this case, it iterates over all of the transitions for every input signal, provided in the parameter `ins`, applying the rise function, + to every signal. These are listed as OR causality for the output signal, `out` to rise. Similarly mapping all of the falling transitions for the inputs signals, `ins`, and setting these as AND causality for `out-`.

Again, to show this concept works for a two input OR gate, we can use the concept `orGateN [a, b] z`. The resulting composition from the use of `orGateN` will be:

```
[a+, b+] ~|~> z+ <> [a-, b-] ~&~> z-
```

These concepts are the same as for the derived two input OR gate.

Multiple input AND gate

A two input AND gate is defined as the dual of a two input OR gate. Therefore, for the multiple input AND gate, or `andGateN`, the concept implemented in Haskell as follows:

```
andGateN inputs output = dual (orGateN inputs output)
```

In this concept, `inputs` is a list of all input signals and `output` is a single output. Since a dual OR gate is an AND gate, the dual of a multiple input OR gate will be a multiple input AND gate. How the `dual` function works, and an example showing that a dual OR gate is an AND gate is discussed in Section 3.3.3.

3.3 High-level concept functions

The circuit-specific concepts derived in Section 3.1 can be used to design many circuits. However, in some cases the set and reset functions for a specification or implementation to be included may be known, and it would reduce design time, and increase ease, to use these in a specification. We introduce some functions which can use these Boolean functions in Section 3.3.1.

The concepts we have introduced in their current form may not describe every behaviour that is possible in a circuit. Through some concept transformations, we can find other behaviours which can provide many further uses for the underlying concepts. These transformations are aimed at reusing the existing concepts, and avoiding the need to specify each possible circuit explicitly, which can lead to needlessly long and complex concepts specifications and libraries. The transformations we introduce are `bubble` (Section 3.3.2), `dual` (Section 3.3.3), and `enable` (Section 3.3.4).

3.3.1 Boolean function concepts

When designing an asynchronous system using concepts, it is possible that an existing circuit will be used as part of the new system. While concepts can be used to define the behaviours within these specifications, the existing circuit may be large and complex, meaning that deriving the concept form of this can be time consuming.

However, the synthesis of this circuit will have produced a set and reset function, which are Boolean functions describing how the input signals are combined in order to cause the output to rise, from the set function, or fall, from the reset function.

We therefore introduce functions for the concepts language, which take Boolean functions and return a concept. First of all, we introduce `function`. This takes a single Boolean function and a signal transition which this boolean function causes. This is used in the form:

$$\texttt{function}\ \ (Boolean\,function)\ \texttt{x+}$$

Where $Boolean\,function$ is any function using AND, OR and NOT operations with standard operators, and where `x+` is any signal transition, rising or falling, which is caused by the Boolean function.

Using the example once again of a 3-input C-element, the output z will rise when all inputs, a, b and c are high. Therefore, the set function of z can be defined as $a \land b \land c$. function will be used for this example as follows:

```
function (a ∧ b ∧ c) z+
```

This does not complete the 3-input C-element specification however. We still need to define the concept for the reset function. For the output to fall, all inputs must have fallen. As such, the reset function can be defined as $\overline{a} \land \overline{b} \land \overline{c}$. We can now define function for this reset function, which because it produces concepts can be composed with any other concept. As such, we can compose both the set and reset functions for the 3-input C-element:

```
function (a ∧ b ∧ c) z+ <> function (ā ∧ b̄ ∧ c̄) z-
```

These concepts will correctly produce a C-element with three inputs, but requires two separate concepts for each function. For convenience we also include complexGate, which takes a set function, a reset function and the signal which is caused to transition by these functions. This is used in the form:

```
complexGate (setfunction) (resetfunction) x
```

Where $setfunction$ and $resetfunction$ are the Boolean functions for setting the output high and low respectively, and x is the signal whose transitions are caused by these functions. Note that in this case, we provide just the signal not the transition.

As with function, complexGate can be composed as with any other concept. Therefore, we can define this 3-input C-element fully:

```
example6 = complexGate (a ∧ b ∧ c) (ā ∧ b̄ ∧ c̄) z <> inits <> interface
  where
    inits = initialise0 [a, b, c, z]
    interface = inputs [a, b, c] <> outputs [z]
```

If only a set function is known, then this can be used to generate a *combinational gate*. This type of gate uses the set function to set the output high, but does not use a reset function, instead using the negation of the set function for this. Using complexGate to specify an AND gate, this would be as follows:

$$\texttt{complexGate}\ (a \wedge b)\ (\overline{a \wedge b})\ \texttt{z}$$

For when a combinational gate is to be generated with many more signals and a more complex set function, we provide a simpler concept, `combinationalGate`. This is implemented as follows:

$$\texttt{combinationalGate set out} = \texttt{complexGate}\ (set)\ (\overline{set})\ \texttt{out}$$

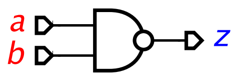
An AND gate can therefore be specified more simply using this concept:

$$\texttt{combinationalGate}\ (a \wedge b)\ \texttt{z}$$

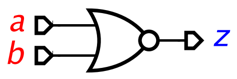
As part of the Plato tool, Boolean functions can also be used to generate a full concept specification, which can be saved as a file. This can then be used to generate an STG, or be used as part of another specification. This will be discussed further in Section 5.1.3.

3.3.2 Bubble transformation

Some gates may have multiple combinations, where inputs or outputs are inverted. These behaviours are important, but can mean that for each type of gate there can be multiple separate specifications to cover each of these combinations. We have specified an AND gate, but often an inverted output AND gate is used, more commonly known as a *NAND gate*. Similarly, a *NOR gate* is also often used, which is an inverted output OR gate.



(a) NAND gate



(b) NOR gate

Figure 3.24: Commonly used gates that we have not specified before

Figure 3.24 contains a NAND gate and a NOR gate. These are indicated as such due to them featuring *bubbles* on the outputs of their gates, small circles which are not featured on their non-inverting counterparts (see Figures 3.17 and 3.15).

We therefore provide a transformation function, `bubble`, to invert a single signal's polarity, in order to provide gates such as NANDs and NORs to a user, without necessarily needing to provide a separate concept for these in the library.

This function is important, as with any gate there can many possible combinations of inverted inputs and outputs. If we have a 3 input gate for example, then there is four signals, three inputs and an output. This means that there can be $2^4 = 16$ possible combinations of inverted and non-inverted signals. bubble can be used to avoid needing to specify 16 separate gates in a library. A NAND gate can be specified in concepts, using the bubble function as follows:

```
example7 a b z = bubble z (andGate a b z)
```

This in itself produces a concept, which can be composed with any other concept, as expected. The translated STG is shown in Figure 3.25. Note that this STG, unlike an AND gate (Figure 3.18) has two c+ transitions, and one c- transition. As expected of a NAND gate, for c+ to occur, either a or b must be 0, and for c-, both a and b must be 1. This is the inversion of an AND gate.

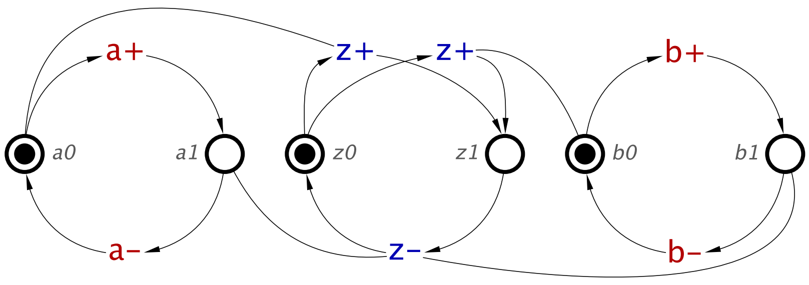


Figure 3.25: Translated STG of example7

However, note that in Figure 3.25 the place c0 contains a token. This being an inverting gate, and all initial states are 0, this means that as soon as the system starts up, c+ can occur. This is the correct operation, but it is ideal to capture that the initial state of c in this case is 1. Thus, the bubble function also will invert the initial state of the chosen signal. With the example of a NOR gate, we will show how bubble can be used to invert the signal and its initial state.

```
example8 a b z = bubble z (orGate a b z <> interface <> initialState)
  where
    interface    = inputs [a, b] <> outputs [z]
    initialState = initialise0 [a, b, z]
```

In this specification, we have defined the interface and initial state and composed these within the example8 concept, along with orGate a b z. This whole composition then has the bubble function applied to it for signal z. This will invert the transitions for signal z in the OR gate concept, and change the initial state of z from 0 to 1, but not affect the transitions or initial states of signals a or b. The interface is also unaffected by the bubble function, as the inversion of an interface is not a meaningful transformation. This is what provides us with a NOR gate.

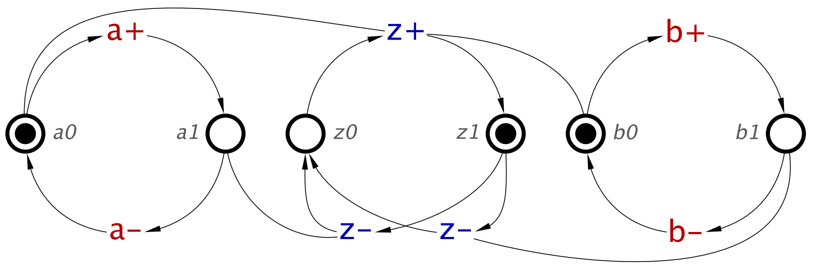


Figure 3.26: Translated STG of example8

The translated STG of this NOR gate specification (Figure 3.26) contains a token in place z1, indicating that the initial state has been changed by the bubble function. Note also, that this is also unlike its non-inverted counterpart, seen in Figure 3.16. In this case, either a or b must be 1 for z- to occur, but both must be 0 for z+ to occur. This is the expected operation of an inverted OR gate.

The bubble function also applies to the invariant, and will invert any states of the inverted signal that have been declared, using the never concept, to violate the invariant. This is to ensure that, using the example of two mutually exclusive signals, if one of these is "bubbled", then its inactive state will now be 1, so the state of both of these signals being 1 will be entirely reachable, when one state is active and the other is inactive.

We can demonstrate this using the mutex element. If we invert one of the grant signals, in this example g2, if we do not also invert the invariant for this signal, then when g1 is active and g2 is inactive, then this would violate the original invariant, stated to be never g1 and g2 being high at the same time, as g1 is high when active, and g2 is high when inactive. The concept specification of this is as follows:

```
example9 r1 r2 g1 g2 = bubble g2 (meElement r1 r2 g1 g2
                                  <> interface <> initialState)
  where
    interface    = inputs [r1, r2] <> outputs [g1, g2]
    initialState = initialise0 [r1, r2, g1, g2]
```

This function will take a standard mutual exclusion element, and invert transitions of output g2, so when r2 transitions high, this will be able to transition low, its initial state will be changed from 0 to 1, and the invariant will change from forbidding g1+ and g2+, to g1+ and g2-.

The translated STG, Figure 3.27 indicates that this is the case, requiring g1- to have occurred before g2- can occur, and g2+ to have occurred to allow g1+ to occur.

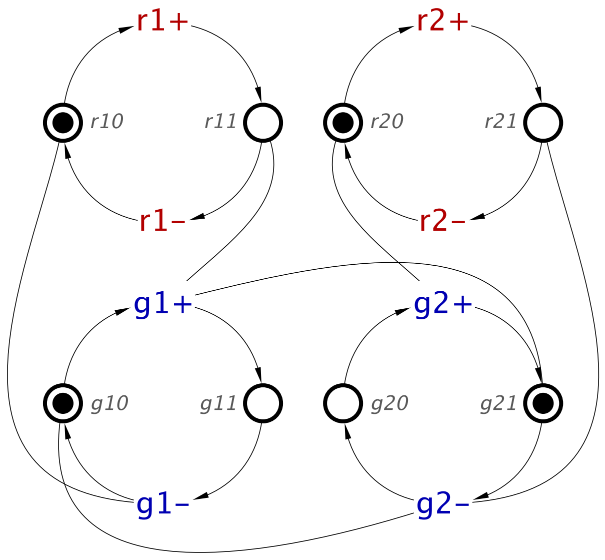


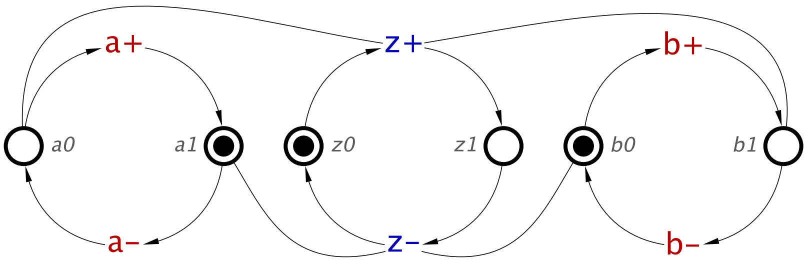
Figure 3.27: Translated STG of example9

The bubble function does not only apply to output signals. In many cases there may be an inverted input signal, but the gate output is non-inverted. Bubble is used in the same way, but by naming an input signal. With the example of a C-element, we can specify the following:

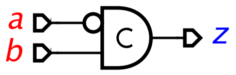
```
example10 a b z = bubble a (cElement a b z
                 <> interface <> initialState)
  where
    interface    = inputs [a , b] <> outputs [z]
    initialState = initialise0 [a, b, z]
```

This will mean that in order for z to go high, a will need to be low and b will need to be high. The initial state of a will also be 1, while b and z will be 0. This can be seen in Figure 3.28a.

The synthesized circuit produced from this STG is shown in Figure 3.28b, and features a bubble on the a input, as expected from the specification, and the translated STG.



(a) Translated STG of example10



(b) Synthesized circuit of example10

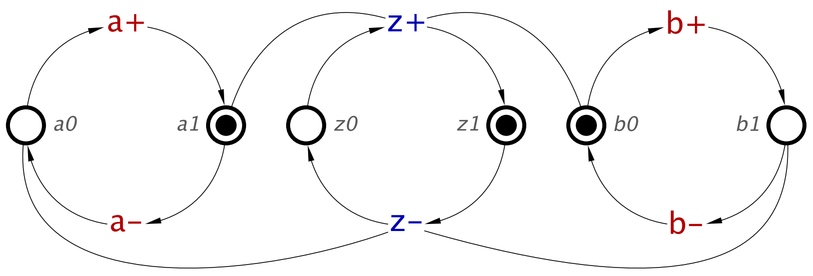
Figure 3.28: STG and circuit generated from example10

It is possible to combine the bubble function, applying it to one signal in a system and then applying it to another of the result of the first bubble. Using the above concept specification for a C-Element with one input inverted, if we bubble z as well:

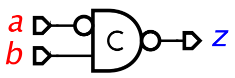
```
example11 a b z = bubble z (bubble a (cElement a b z
                <> interface <> initialState))
  where
    interface    = inputs [a , b] <> outputs [z]
    initialState = initialise0 [a, b, z]
```

The translated STG from this, shown in Figure 3.29a, will feature the inversion of a as with Figure 3.28a, but this time also features the inversion of z, which is initially 1, and to be set high requires a to be high and b to be low, and to be set low, requires a to be low and b to be high. The circuit for this (Figure 3.29b) features a bubble both on one of the inputs and the output.

Note: bubble is a commutative transform, meaning that regardless of the order in which multiple signals have bubble applied to them, the result will be the same. With example11, we have applied bubble a to the C-element, interface and initial states. We then apply bubble z to the result of this. If we swapped these bubble functions, applying bubble z to the C-element, interface and initial state, and then apply bubble a to the result, the produced STG and therefore the synthesized gate



(a) Translated STG of `example11`



(b) Synthesized circuit of `example11`

Figure 3.29: STG and circuit generated from `example11`

would be the same as in Figure 3.29.

Due to the commutativity of the bubble transform, we therefore include the `bubbles` transform. This is a more convenient form of `bubble`, which takes a list of signals to be inverted, avoiding the need to have multiple `bubble` functions and parentheses. `example11` can therefore be rewritten as:

```
example11 a b z = bubbles [z, a] (cElement a b z
                  <> interface <> initialState)
  where
    interface    = inputs [a , b] <> outputs [z]
    initialState = initialise0 [a, b, z]
```

As a simple example, showing that the bubble functions works as expected, we will aim to show the occurrence in a concept, when the `bubble` function is applied to the same signal twice. For this example, we use a C-element, with the output inverted twice. In concepts this is:

```
example12 a b z = bubble z (bubble z (cElement a b z))
```

First, we break the C-element concept into the signal-level concepts.

```
example12 a b z = bubble z (bubble z ([a+, b+] ~&~> z+ <>
                                      [a-, b-] ~&~> z-))
```

Now, with the first `bubble`, we invert each `z` transition, removing this function.

```
example12 a b z = bubble z ([a+, b+] ~&~> z- <> [a-, b-] ~&~> z+)
```

For the second bubble, we do the same again, inverting all z transitions.

```
example12 a b z = [a+, b+] ~&~> z+ <> [a-, b-] ~&~> z-
```

The result of this is the same as a non-inverted C-element. Thus, inverting the same signal twice acts as expected, and causes no inversion of this signal in the result.

3.3.3 Dual transformation

The dual transformation is based on the principle of duality for Boolean functions. This principle states that each operator and variable in a Boolean function has a *dual*, and that the result of the dual function will be the dual of the original function [33].

For a function $f(x)$, the dual can be described as: $d(x) = f'(x')$. In words, this states that a dual of a function is the inversion of the original function with inverted variables and constants. For example, using the complement axiom for AND, $A \wedge \overline{A} = \mathbf{0}$, stating that the AND of any variable and its complement is $\mathbf{0}$, if we find the dual of this by first inverting the variables and constants ($\overline{A} \wedge \overline{\overline{A}} = 1$) then inverting the operator, an AND ($\wedge$) becomes an OR ($\vee$) and vice versa. The result is $A \vee \overline{A} = 1$, which itself is an axiom for the complement under OR, stating that any OR operation on a variable and its complement is $\mathbf{1}$.

This principle is particularly useful in *dual-rail* logic [34], which is commonly used in low-power digital circuits. In dual-rail logic, each signal has two wires, and each bit is instead represented by two bits; a logic 0 is represented by 01 and logic 1 is represented by 10. Therefore, one wire for each signal represents the actual logic value, and the second represents the dual. These wires can in-fact feature four states: 01 and 10 as we have already discussed identify values 0 and 1 respectively, but there can also be 00 and 11. In dual-rail logic, 00 is used to identify the system as idle, not having any inputs, and 11 is used to show an illegal output, meaning an error has occurred.

Therefore, with these two wires the digital circuit can perform a logic operation on the non-inverted input wire, providing an output. This can then be checked to be correct by performing the dual logic operation on the inverted input wire. If the output is 01 or 10, we know that the circuit has performed successfully. If this is 00 or 11, either the logic operation has not completed or an error has occurred and the output can be ignored until a valid output has been produced.

In low-power digital circuits, dual-rail logic is commonly used, as the output not being the expected 01 or 10 identifies that the operation of this circuit has not yet completed, or there is an error. The circuit can then continue to wait for a correct 01 or 10 output. This makes dual-rail logic more dependable to produce correct output [35].

We provide the dual function for Asynchronous Concepts, which allows a user to define the operations performed on input signals to provide an output, and then provide the dual for this with ease, to allow the design of circuits such as those featuring dual-rail logic simpler.

The dual function behaves similar to the bubble function, but applies the inversions to the transitions, initial states and never concepts of *all* signals in the concept specification. This function can be used in multiple ways. To demonstrate that this function produces the dual correctly, we will first show the dual of an OR gate. This is specified in concepts as follows:

```
example13 a b z = dual (andGate a b z)
```

If we break the gate-level concept of `andGate` down to the atomic signal-level concepts, the specification will be:

```
example13 a b z = dual ([a+, b+] ~&~> z+ <> [a-,b-] ~|~> z-)
```

As stated, the dual function will invert every transition of a given concept, so the resulting specification after the dual function is:

```
example13 a b z = [a-, b-] ~&~> z- <> [a+,b+] ~|~> z+
```

This concept specification is now the same as an OR-gate, with either `a+` or `b+` needing to have occurred to allow `z+`, but both `a-` and `b-` needing to have occurred for `z-`.

Rather than on individual concepts, the dual function can also be used to provide the dual of entire specifications. For example, if we have the circuit as shown in Figure 3.30, we can apply `dual` to this entire circuit.

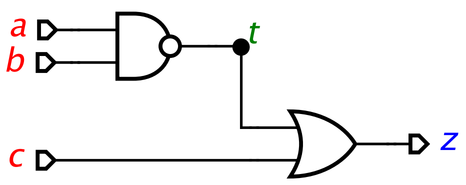


Figure 3.30: Example circuit to apply `dual` to

The concept specification for this circuit is:

```
example14 a b c t z = gate1 <> gate2 <> interface <> initialState
  where
    interface    = inputs [a, b, c] <> outputs [z] <> internals [t]
    initialState = initialise0 [a, b, c, z] <> initialise1 [t]
    gate1        = bubble t (andGate a b t)
    gate2        = orGate t c z
```

Now, if we apply the dual to this specification, as follows:

```
example15 a b c t z = dual (example14 a b c t x)
```

Which when synthesized produces the circuit as seen in Figure 3.31

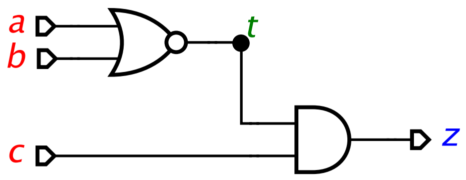


Figure 3.31: The dual of Figure 3.30

All of the initial states of the signals have been inverted, meaning that a, b, c and z are initially 1, and t is initially 0. Therefore, we can treat each signal as if it has been inverted from the original gate.

First, gate1 before the dual generates a NAND gate. The function of this NAND gate is: $\overline{t} = a \wedge b$. Applying the dual to this results in: $t = \overline{a \vee b}$, a NOR function.

gate2 before dual generates an OR gate. The function of this is $z = \overline{t} \vee c$. We have stated before (Section 3.1.2) the dual of an OR is an AND, so for this specification the dual function is: $\overline{z} = t \wedge \overline{c}$. Therefore, the dual of this has been found successfully with concepts.

Some standard gates that are derived in this circuit concept library have interesting duals, and these are detailed in Table 3.2.

To demonstrate that a dual transformation can produce the dual of a gate, and the dual of this is the original gate, we will use the example of an AND gate. As has been discussed, an AND gate is the dual of an OR gate. And this is how an AND gate has been defined, as:

Gate	Dual
Buffer	Buffer
Inverter	Inverter
C-element	C-element
ANDgate	ORgate
ORgate	ANDgate
XORgate	XNORgate

Table 3.2: The duals of gates in the library

```
andGate a b z = dual (orGate a b z)
```

For this example, we will apply `dual` to the `andGate a b z`. And this is the same effect as applying `dual` twice to an OR gate. Expanding this initially will produce:

```
example16 a b z = dual (dual (orGate a b z))
```

Now if we expand `orGate a b z`

```
example16 a b z = dual (dual ([a+, b+] ~|~> z+ <> [a-, b-] ~&~> z-))
```

If we apply one of the dual functions to this, we get:

```
example16 a b z = dual ([a-, b-] ~|~> z- <> [a+, b+] ~&~> z+)
```

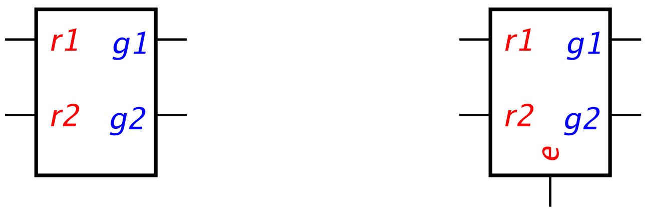
The AND causality and OR causality in this has now switched, so what was once and OR gate is now and AND gate. Let's apply the second dual:

```
example16 a b z = [a+, b+] ~|~> z+ <> [a-, b-] ~&~> z-
```

This specification is now the same as an OR gate, demonstrating that applying a dual to an AND gate, which is in itself is a dual of an OR gate, results in an OR gate.

3.3.4 Enable transformation

While the gates we have derived can be functional on their own, in some circumstances, their operation may be subject to certain conditions. For this transformation, we concern



(a) Without enable (b) With enable

Figure 3.32: Mutual exclusion elements with and without enable

signals which *enable* a circuit, that is, the polarity of one signal determines whether the output of the circuit can change, be it one gate or the output of an entire circuit.

In practice, an enable signal can be used to select between multiple devices, or be used to only allow a circuit's output to change when chosen by the environment, such as when it the output is ready to receive the changes. Commonly, therefore this enable signal is applied only to output signals.

As with `bubble` and `dual`, this transformation can avoid the need to derive many concepts with similar operations, but slight changes, based on inverted signals, or including an enable signal. Therefore, we introduce `enable`.

This transformation takes a transition, which is the transition of the enable signal which indicates that the gate is enabled. This can allow for both *active high* and *active low* enables, where the enable signal in question must be high or low to indicate an enabled state. It also takes in a signal, which cannot transition without the system being enabled. Finally, it takes in the concept, containing the circuit, gate or protocol to which this enable applies. An example of the use of `enable` is:

```
enable e+ x (example17 a b x)
```

Where `e+` is the enable transition, `x` is the signal that requires the enable transition and `example17` is a concept, which features the signal `x` as an output, so regardless of the operation of `a` and `b`, the output cannot transition without the enable signal having transitioned first. `enable` applies the enable transition, `e+` in this example, to both the rising and falling transition of the given signal, `x`. This ensures that the output signal only transitions following the enable transition.

A real use-case example of this transformation is with a mutual exclusion element. Figure 3.32 contains mutual exclusion element circuits with the implementation hidden.

Figure 3.32b should react the same as Figure 3.32a as long as signal e remains high, but the outputs will not change when e is low.

Note that there are two outputs of a mutual exclusion element g1 and g2. Enable is a commutative transformation, meaning that we can compose two enabled concepts in any order, applying enable to each signal correctly. For this example we need to include an enable transformation for each of the output signals, which will be in the form:

```
enable e+ g1 (enable e+ g2 (meElement r1 r2 g1 g2))
```

Here, the second enable, enable e+ g2, will be applied to the meElement concept. The result of this is also a concept, which will have the first enable applied to it, enable e+ g1. This results in another concept which can be composed as usual.

It is possible that circuits may have multiple outputs, all of which will not transition without a single enable signal. Therefore, we also include enables, which takes a list of signals to apply enable to. With this, we can derive a full concept specification for a mutual exclusion element with an enable.

```
example18 r1 r2 g1 g2 e = enables e+ [g1, g2] ( meElement r1 r2 g1 g2)
                          <> initState <> interface
  where
    initState = initialise0 [r1, r2, g1, g2, e]
    interface = inputs [r1, r2, e] <> outputs [g1, g2]
```

The resulting STG from translating this concept specification is found in Figure 3.33. This is very similar to the STG as seen in Figure 3.23, which is an STG for a mutual exclusion element without an enable signal. This STG includes an extra signal transition loop for signal e, with read-arcs connecting the rising and falling transitions of g1 and g2 to the place e1, which indicates when the circuit is enabled.
Taking this STG and synthesizing it produces the circuit as shown in Figure 3.34. Comparing this to Figure 3.22, the circuit for a mutual exclusion element without an enable signal, we find that they both feature the structure of two AND gates, with each taking one input and the opposite output signal inverted.
However, for the inputs, there is a difference. This time, r1 does connect to the AND gate to produce g1, but through another AND gate. The other input to this AND gate is e. If e is low, the output of this gate will be 0, regardless of the polarity of r1, so it

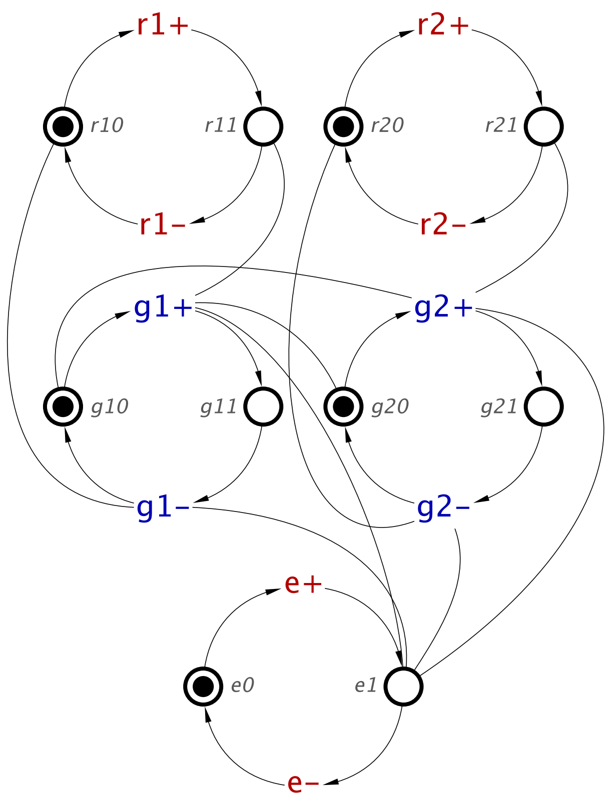


Figure 3.33: STG of a mutual exclusion element with an enable signal

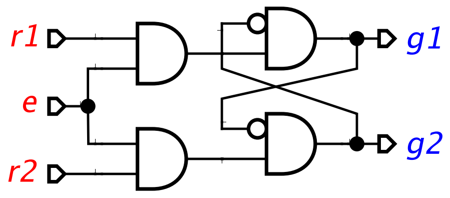


Figure 3.34: Circuit implementation of a mutual exclusion element with enable

cannot cause a change in `g1`. If `e` is high, then the output of this AND gate is whatever the value of `r1` is, which can then propagate to `g1`. This is the same for `r2`.

The enable signal in this example is used to determine whether the input signals can reach the logic which affects the output signals. If it is low, then no change can occur, if it is high, then the input signals determine whether the outputs change. This is the desired operation of the enable signal. This shows how effective the `enable` and `enables` transformations can be used, simply being applied to concepts to change their operation, without the need to derive a new concept specifically to include an enable signal.

3.4 Set-Reset latch example

A set-reset latch is an interesting example, which exercises many of the concepts we have discussed, and some intuition to provide a concept specification for a set-reset latch, but shows some problems with concepts which can be improved upon with future research.

A latch is a device used in digital circuits to store the value of an input signal, outputting this once stored. There are many types of latches, the differences between them are how to control the storing of the input signal. In a set-reset latch, there are two control signals for the latch, *set* and *reset*, which set the output high, and set the output low respectively, thus storing a high or low value.

First, let us describe the operation of a standard set-reset latch. This consists of four signals, two input signals, `s` for set and `r` for reset, and two output signals, `q`, which contains the value of the stored signal, and `nq`, which is the negated output of the stored value.

If we describe the behaviour of the latch just based on the non-negated output `q`, we can do this with the `complexGate` function. For `q` to rise, `s` must be high, and for `q` to fall, `r` must be high. This easily gives us the set and reset functions, and the concept is as follows:

```
example19 = complexGate (s) (r) q
```

This concept causes the desired change in `q`, but there are issues that can occur with a specification such as this: if `s` and `r` are both high at the same time, then what is the change in `q`?

Figure 3.35 is the translated STG for this example. Notice that if `s` and `r` are both high, then `q` can transition both high and low, depending on it's current polarity, but as long as both input signals remain high, `q` can continuously transition both high and low. This is more clearly shown in the state graph format of this STG, found in Figure 3.36.

In this state graph, we have highlighted the important arcs which show the issue when both inputs are high, i.e. when the states are 110 or 111. These arcs are *optional* [36]. Unfortunately, STGs do not support optional arcs, and neither do Asynchronous Concepts. These cannot specify what occurs in these situations. It may be possible to include optional arcs in Concepts, but this will be a challenging process, and is an opportunity for further research.

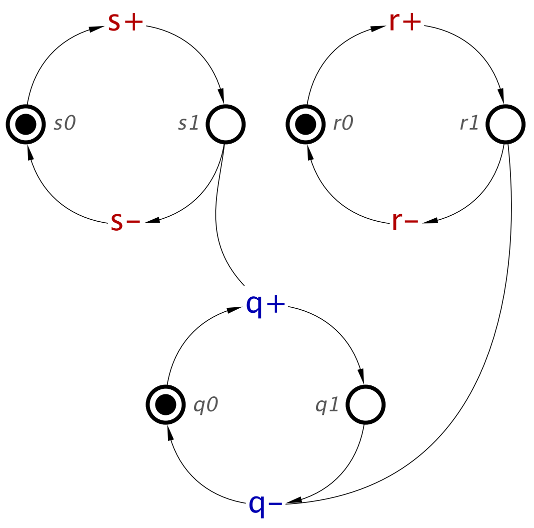


Figure 3.35: Translated STG of `example19`

Instead, with concepts, we aim to try and block access to the states where both input signals are high. Ideally, we would apply mutual exclusion to `s` and `r`, but this is a hard restriction to try and place on the environment, which we can specify for, but not control. Instead, we use a `never` concept, which can then be verified to ensure that a state where both the set and reset signals are high cannot be reached. Adding this to the concept specification becomes:

```
example19 = complexGate (s) (r) q <> never [s+, r+]
```

This now ensures that `q` only rises when `s` is high and `r` is low, and falls when `r` is high and `s` is low. There can still be an issue when an input signal transitions high, then low before the output has transitioned in the correct way. For example, if `s` rises and falls before `q` transitions high, then the latch is not correctly storing the value. We can therefore add two more concepts to ensure this does not occur. This concept is now the specification for one output signal of a latch, which we name `srHalfLatch`.

```
srHalfLatch s r q = complexGate (s) (r) q <> never [s+, r+]
                                <> q+ ~> s- <> q- ~> r-
```

With these final two causality concepts, the output must transition before the inputs can transition low. These impose a constraint on the environment, which is not ideal. However these are not acting as a block for the system entering a state, but they simply imply timings on the signal transitions.

With `srHalfLatch` we can now use this to derive a concept for a full set-reset latch, `srLatch`, using `srHalfLatch` as a base. The full latch will use all four signals, `s`, `r`,

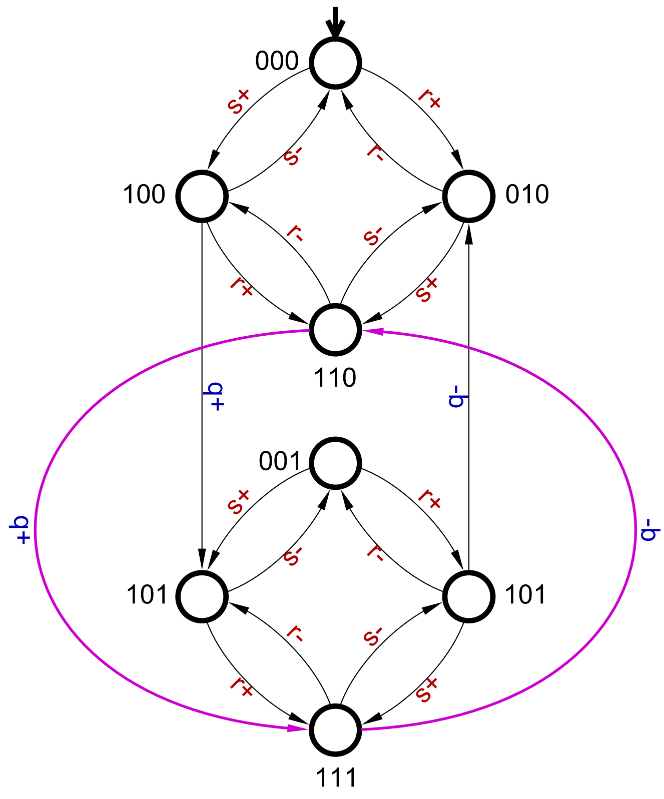


Figure 3.36: Translated FSM of `example19`

`q` and `nq`. Since the behaviours for `s` are already described in `srHalfLatch`, we can include this for `q`.

`nq` however is somewhat different. Since this is the negation of `q`, we can say that for it to rise, `r` must be high, and for it to fall, `s` must be high. The behaviours of `nq` are the same as with `q`, just with `r` and `s` swapped, thus we can reuse `srHalfLatch` and simply swap the two input signals. So far, the concept can be defined as follows:

```
srLatch s r q nq = srHalfLatch s r q <> srHalfLatch r s nq
```

Due to the use of `srHalfLatch` we include the `never` concept stating that `s` and `r` can never be high at the same time, and the concepts we used to state the timings that either input signal will only transition low after the output has changed, we also need to ensure that the initial state does not violate the invariant as explained in the `never` concept. We do this by forcing the initial states of signals `q` and `nq` to be 0 and 1 respectively.

```
srLatch s r q nq = srHalfLatch s r q <> srHalfLatch r s nq
                 <> initialise0 [q] <> initialise1 [nq]
```

Again this is not ideal, as it is entirely possible for a latch to have an initial state with `q` and `nq` as 1 and 0 respectively, but we need to impose this as these signals need to have *differing* initial states. Again, further improvements and research can be applied to concepts to allow these signals to have any initial states, providing they are different.

We can now synthesize a specification using `srLatch`, providing the environment restrictions are made. The resulting circuit can be found in Figure 3.37. This concept can now be used as any other concept is, composing it with other concepts or applying transformations to it.

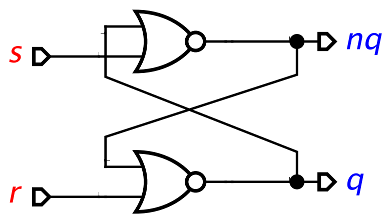


Figure 3.37: Synthesized set-reset latch circuit

3.5 Abstract concepts

Abstract Concepts are the base level of concepts that we use as building blocks for developing *domain specific concepts*, such as those related to asynchronous circuits (Section 3.1).

The alphabet of abstract concepts is the same as with that of circuit-specific concepts, a set of signals. These signals can have either a high polarity, 1, or low polarity, 0. This means there are a finite set of states in such a system, containing every possible combination of high and low polarities for each signal. While not all states may be reachable, the maximum number of states therefore is 2^n, where n is the number of signals.

The global state of a system can be determined at any point, using the polarities of all signals at that point. This provides an encoding, using 1 or 0 for each signal which can be used to identify each state. The current state of a system can be used to determine whether certain events are enabled or excited. An event in a system is the transition of a signal.

With this information, we can describe the basic concepts that make up abstract concepts, with the parameters of finite sets of states S and events E.

Initial state concept

The initial state concept captures all possible (or *permitted*) initial states of the system. In the most general form it is a function

$$\text{initial} : S \to \mathbb{B}$$

that given a state $s \in S$ returns 1 if s is an initial state and 0 otherwise. In practice this concept is often realised as a membership test of a set of initial states $I \subseteq S$, i.e. $\text{initial}(s) = s \in I$. However, we prefer the functional form because it is more abstract and permits other, often more efficient realisations. Note that $\mathbf{0}$ and $\mathbf{1}$ have natural interpretations as initial concepts: they correspond to systems with no initial states, and systems where any state can be initial, respectively. Initial state concepts form a commutative monoid with the identity element $\mathbf{1}$ and the composition operation $\wedge$.

Using the example of a 2-input AND gate circuit, if the initial state of this is set as 000, identifying that all signals are initially low, then passing this into the initial function will return:

$$\text{initial}(000) = \mathbf{1}$$

indicating that this is the initial state. However, if some transitions have occurred in the system, and the current state is 110, then

$$\text{initial}(110) = \mathbf{0}$$

indicating that this state is not initial.

Intuitively, if a system comprises two subsystems then its initial state should satisfy constraints imposed by both subsystems, hence the conjunction operator $\wedge$. For example, suppose we compose two subsystems, both of which feature a signal, a. If in subsystem one, we set the initial state for a as $\mathbf{0}$, then in the second subsystem, a must either not have its initial state set, or must must also be set at $\mathbf{0}$ to satisfy the specifications for both subsystems. Setting the initial state for this signal to $\mathbf{1}$ in one of the subsystems and $\mathbf{0}$ in the other means the initial states are *inconsistent*, and neither specification can be satisfied until the same initial state is agreed upon for both specifications. If only one subsystem sets the initial state of a, then this state will apply to both subsystems after composition.

Excitation concept

The event excitation concept captures all states wherein a given event can occur (or is *excited*). This is useful to determine causalities, determining whether a cause event (or events) has occurred, which will enable an effect transition.

In the most general form it is a function

$$\text{excited} : E \times S \to \mathbb{B}$$

that given an event $e \in E$ and a state $s \in S$ checks whether e is excited in s.

With the 2-input AND gate example, if we are checking whether the output signal, z, can transition high, we can pass this, and the current state of the system into the excited function. If the current state is 011:

$$\text{excited}(z+, 011) = \mathbf{0}$$

indicating that the z+ transition is not enabled in this state, which is expected as for the output to transition high, it must first be low, and both inputs must be high. If we use this function with the state 110:

$$\text{excited}(z+, 110) = \mathbf{1}$$

Now the z+ transition is enabled.

In practice this concept is often realised using *interpreted graph models* such as Finite State Machines, state graphs, Petri Nets [19], STGs, Conditional Partial Order Graphs [37], and others. A partial application of the excitation function is often useful: $\text{excited}(e)$ captures all states where event e is excited; for example, if $\text{excited}(e) = \mathbf{0}$ then e is never excited or *dead*.

Event excitation concepts also form a commutative monoid with $e = \mathbf{1}$ and $\diamond = \wedge$. This definition corresponds to the *parallel composition* operation, a standard notion for many behavioural models [38].

Invariant concept

Some states may be impossible or undesirable during the normal system operation. To express this we use the *invariant concept*, which captures all *correct* or *permitted* states of the system. A typical use case for invariant concepts is to specify assertions or assumptions about the system state space, that may by verified via model checking and/or used for optimising the implementation. In the most general form an invariant concept is a function

$$\text{invariant} : S \to \mathbb{B}$$

that given a state $s \in S$ returns 1 if s is permitted by the invariant and 0 otherwise. Note that if for some state s the initial concept $\text{initial}(s)$ holds but the invariant $\text{invariant}(s)$ does not hold, then the specification is *contradictory* and cannot be satisfied by any implementation. We therefore assume that $\text{initial}(s) \Rightarrow \text{invariant}(s)$ holds for all $s \in S$.

Using the AND-gate example once again, there may be an environment constraint stating that one of the input signals, a, must not be low when the other signal b is high, leading to a system where a must transition high before b, and b must transition low before a. This behaviour can be captured by a concept, and using the invariant function, we can determine which states are permitted. The state of 100 for example, when passed into this function:

$$\text{invariant}(100) = \mathbf{1}$$

which indicates that a state where a is high, and b is low is permitted by the invariant. With the state of 011:

$$\text{invariant}(011) = \mathbf{0}$$

This state is not permitted by the invariant, and thus, should not be reachable.

Invariant concepts also form a commutative monoid with $e = \mathbf{1}$ and $\diamond = \wedge$. Intuitively, if a system comprises two subsystems then its states should be permitted in both of the subsystems.

Silent concept

One can derive other useful concepts from the three concepts described above, for instance,

$$\text{silent}(e, s) = \overline{\text{excited}(e, s)}$$

captures all states $s \in S$ when a given event $e \in E$ cannot occur. Furthermore, one can define other useful concepts that cannot be derived from the above, e.g., the *execution concept* capturing the effects that different events have on the system state.

Concept composition

All the above concepts are monoids, hence their combinations are trivially monoids too. It is convenient to consider triples of concepts $(\text{initial}, \text{excited}, \text{invariant})$ with $(\mathbf{1}, \mathbf{1}, \mathbf{1})$ representing the *empty specification*, and composition

$$(\text{initial}_1, \text{excited}_1, \text{invariant}_1) \diamond (\text{initial}_2, \text{excited}_2, \text{invariant}_2)$$

$$\text{defined as}$$

$$(\text{initial}_1 \diamond \text{initial}_2, \text{excited}_1 \diamond \text{excited}_2, \text{invariant}_1 \diamond \text{invariant}_2)$$

.

Note that the result of the composition of two non-contradictory specifications is always non-contradictory, that is if both $\text{initial}_1(s) \Rightarrow \text{invariant}_1(s)$ and $\text{initial}_2(s) \Rightarrow \text{invariant}_2(s)$ hold for all states $s \in S$, then $\text{initial}_1(s) \diamond \text{initial}_2(s) \Rightarrow \text{invariant}_1(s) \diamond \text{invariant}_2(s)$ holds too.

The process of composing concepts is the same as finding the *Synchronous Product*, or *Parallel composition*, of two of the same modelling formalism, such as Petri nets [38]. Both concept composition and the synchronous product. Initial and invariant states compose as expected, ensuring that these states are satisfied in the result as well as in the original models. In the event that there is an effect transitions with different cause transitions in each of the original models, then the synchronous product of these will require that all of the cause transitions occur before the common effect transition can occur. This is the same as concept composition.

3.6 Summary

This chapter has discussed the library of Asynchronous Concepts specifically for asynchronous circuits, showing the derivation of each concept, and the reusability inherent in the language. This library includes many standard signal interactions, logic gates and protocols that asynchronous circuits feature.

From this library, we have derived some generalisations which provide concepts that enable the specification of logic gates with any number of inputs. This avoids the need to derive a concept for a gate of every size a user may desire.

High-level concepts have been discussed, which allows reuse of existing circuits, using the Boolean expressions used to synthesize these. These concepts can be used in-line with any other concept, to allow easier specification of a system, instead of manually specifying this existing circuit once again.

The library also includes several transformations, which serve to provide a wider range of gates from the derived gates in the library, without the need to derive a separate concept for each individual version of a gate. Bubble transformations can be used so that the concept for any gate-level concept can be used when any of inputs or outputs of this gate needs to be inverted, and this can also apply to any signal-level or protocol-level concept.

The dual transformation can also produce a wide range of gates from specifications using the library of circuit-specific concepts. This can be useful for different asynchronous circuit types, such as dual-rail, where the dual can be applied to an entire specification, providing the dual-rail for such a circuit automatically.

Enable is a transformation that can turn any concept, at any level, or any concept specification into a system which features an enable signal, requiring that a signal be in a specified state in order to allow the system to change state. This again provides a wider range of uses for the circuit-specific library discussed in Section 3.1.

Using the interesting example of a set-reset latch, we use some of the discussed concepts to derive a concept for this gate. The information from this chapter can now be used in the explanation of the design flow using concepts. The concepts discussed will be used and built-upon to explain further contributions of this thesis.

Finally, this chapter discussed abstract concepts in Section 3.5. This is a form of concepts on which other concepts, including the circuit-specific concepts, can be built. This is aimed at providing a base on which many other types of Asynchronous Concepts can be built, aimed at other types of asynchronous circuit, such as mixed signal circuits, or large-digital data based asynchronous circuits.

Chapter 4

Asynchronous Concepts design flow

With the circuit-specific concepts derived, and information on high-order functions for these concepts, we can now discuss the method of designing an asynchronous circuit. This chapter covers the usage of Asynchronous Concepts when designing an asynchronous circuit, which is performed in a few steps.

- The design flow may begin with a description of the system behaviours, and from this, finding appropriate concepts to capture these behaviours. We discuss this in Section 4.1.
- The design could include existing circuits for which we know the set and reset functions, which we can use in-line with other concepts following a transformation pattern presented in Section 3.3.1. Alternatively we can generate an entire concept specification from these functions, which we will discuss in Section 4.2.
- If we are to include a circuit for which the set and reset functions are not known, we can instead generate concepts for these through process mining. This finds patterns of concurrency in the simulation traces for a circuit, which can be used to generate concepts. This technique is explained in Section 4.3.
- Following this, a designer will have generated multiple concepts. These will be used to generate a specification for the circuit, some of these may be used more than once within this specification and in future specifications. These may be stored in user libraries, which can be imported into concept specifications for an overall design, and we will outline this in Section 4.4.
- When a user has completed a specification, for it to then be used in further operations, it must be compiled, ensuring that the concepts are sound, and then

translated to a form usable by existing verification and synthesis tools. This step is discussed in Section 4.5.

- If the system behaviour is described by more than one specification they can be combined using *templates* introduced in Section 4.6.
- When a full specification has been produced for the circuit, it then needs to be verified. This process ensures the specification satisfies certain properties which means it can produce an implementation. These properties are discussed in Section 4.7.
- Finally, when the specification is verified, we then use this to find an implementation through synthesis, as discussed in Section 4.8.

4.1 Design approach

When preparing a design, a description of how the system should operate may be given, in lieu of having any previous versions to base the design on. For example this could be a description of how the signals interact, in both the circuit and the environment. From this, the concepts are then derived using signal-level concepts to describe the individual interactions, or gate-level and protocol-level concepts if there are appropriate behavioural patterns. There is no requirements on the concepts used, it is down to preference and the experience of each designer.

We present a simple example to indicate how we find a design, from the description of the interaction between the circuit and its environment. The description of this is as follows:

The circuit contains three signals, two inputs and one output. The output is set high when both inputs are low, and the output is set low when both inputs are high. The inputs become high when the output is high, and they become low when the output is low. The inputs are initially low and the output is initially high.

From this description, we can immediately define the interface and initial states for this circuit. For this, we will use `a` and `b` to represent the inputs signals and `z` for the output.

```
 interface = inputs [a, b] <> outputs [z]
initState = initialise0 [a, b] <> initialise1 [z]
```

Now we begin to describe the operations. Beginning with what causes the output to rise, both inputs must be low. In signal-level concepts this can be described as:

```
outRise = [a-,b-] ~&~> z+
```

The concept to describe what causes the output to fall is similar to this:

```
outFall = [a+,b+] ~&~> z-
```

There are also operations which cause the inputs to change, describing the environment. While the operations of the environment are not directly synthesized, it is important to model them, so simulations and formal verification will ensure the entire circuit works as expected. We define the environment concept as follows:

```
environment = z+ ~> a+ <> z- ~> a- <> z+ ~> b+ <> z- ~> b-
```

We now have the full specification, which is defined as follows:

```
example20 = outRise <> outFall <> environment <> interface <> initState
  where
    outRise     = [a-,b-] ~&~> z+
    outFall     = [a+,b+] ~&~> z-
    environment = z+ ~> a+ <> z- ~> a- <> z+ ~> b+ <> z- ~> b-
    interface   = inputs [a, b] <> outputs [z]
    initState   = initialise0 [a, b] <> initialise1 [z]
```

This concept specification is complete, and adequately specifies the circuit as described above. It can now be translated to an STG for simulation, verification and synthesis, or it can be stored in a user concept library. However, this is just one way of representing this circuit with concepts, and it may be represent it more easily using concepts from the provided circuit specific library.

If we take the operations which cause the output signal to rise and fall, we can simplify this. The description states

"*The output is set high when both inputs are low, and the output is set low when both inputs are high.*"

This means that for the output to transition, both inputs must have transitioned. This operation is the same as a C-element, however, the output transitions the opposite way to both inputs, indicating that the output of the C-element is inverted. The `outRise` and `outFall` concepts can be replaced by:

```
gate = bubble z (cElement a b z)
```

Then environment concept as described above is also quite long, itself composed of four signal-level concepts. Instead, if we look at the concepts for this, it may be noted that the input signals follow the transitions of the output signal. This is like the buffer concept, where the input, z in this case, causes the output, a and b, to transition in the same way. The environment concept can be redefined as:

```
environment = buffer z a <> buffer z b
```

The final concept specification with these simplifications is as follows:

```
example20 = gate <> environment <> interface <> initState
  where
    gate        = bubble z (cElement a b z)
    environment = buffer z a <> buffer z b
    interface   = inputs [a, b] <> outputs [z]
    initState   = initialise0 [a, b] <> initialise1 [z]
```

These two concept specifications are equivalent as seen by substituting the definitions of bubble, cElement and buffer. The decision of which concepts to use is based on the preference of each user. Either of these concept specifications can be translated to produce the same STG, as seen in Figure 4.1.

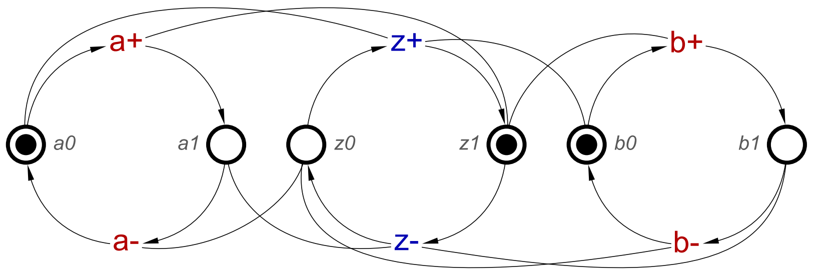


Figure 4.1: Translated STG of example20

The synthesis of this STG will lead to the circuit as viewed in Figure 4.2. This features a C-element with an inverted output, and buffers from the output to each of the input signals, as described in the concept specification. In this example, the behaviours described using the gate-level cElement and buffer concepts can be directly implemented. Each of the separate concepts which are defined to produce this concept can be seen in this circuit.

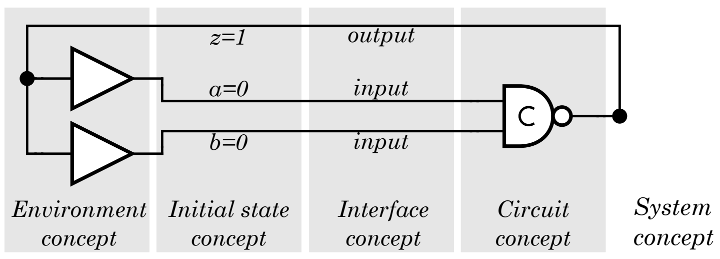


Figure 4.2: Synthesized circuit of example20

4.2 Generating concepts from set and reset functions

If the set and reset function of a circuit to be included in a design is known, then rather than manually derive the concept specification for this, the set and reset functions can be used, which saves time. In Section 3.3.1 we discussed the function and complexGate concepts which take in these Boolean functions and are used in-line with other circuit-specific concepts.

In some cases however, it may be necessary to view the concepts which the given set and reset functions generate. This may be for example required in order to use a certain part of it, to make minor changes manually, or to add further functionality to the specification. As such, we can generate a concept specification from these functions.

As an example for this section, we have an AND-OR gate, which features four inputs, a, b, c and d, the single output is z. The set function for this gate is: $(a \wedge b) \vee (c \wedge d)$. It is not required that a reset function be provided. In this event it can be assumed that the reset function is the negation of the set function, in this case: $\overline{(a \wedge b) \vee (c \wedge d)}$.

Passing the function to Plato, the output will be a full concept specification. The concept specification generated from providing the function for the AND-OR gate is as follows:

```
example21 = outRise <> outFall <> interface <> initState
  where
    outRise   = [a+, c+] ~|~> z+ <> [a+, d+] ~|~> z+
             <> [b+, c+] ~|~> z+ <> [b+, d+] ~|~> z+
    outFall   = [a-, b-] ~|~> z- <> [c-, d-] ~|~> z-
    interface = inputs [a, b, c, d] <> outputs [z]
    initState = initialise0 [a, b, c, d, z]
```

This specification may be used in this form, or a user can edit some of its operations, such as by applying a transformation to it. It may even be a requirement of the circuit that, for example, it operates exactly the same, except `a+` and `c+` both having occurred does not cause `z+`. It may have been noted that its operation is very similar to the AND-OR gate, so it was ideal to start with it as a base, and edit the functionality as appropriate.

This concept specification is translatable, and therefore may be verified and synthesized for an implementation as seen in Figure 4.3.

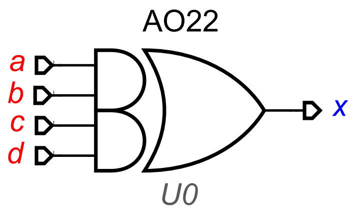


Figure 4.3: Synthesized circuit from `example21`

4.3 Process mining for Asynchronous Concepts

In the event that we do wish to include a circuit, but the set and reset functions are not known, it is also possible to generate a concept specification through process mining, or specification mining [39]. This takes a list of traces from simulations of the circuit, with multiple different orders of signal transitions, which form an *event log*. Mining these will extract concurrency and find behaviours in the circuit, which can then be captured using concepts.

For process mining, we developed a tool known as *PGminer* [22][23]. This takes in event logs, and automatically outputs Boolean functions, which as we know from Section 4.2, can be used to generate a concept specification automatically.

As an example, we have a system with two input signals, `a` and `b`, and one output signal `z`. We do not know what the set and reset functions of the circuit are. Instead, we simulate the circuit, recording all signal transitions. After some simulations, the following event log is produced:

$$a+ \quad b+ \quad z+$$
$$b+ \quad a+ \quad z+$$
$$a- \quad b- \quad z-$$
$$b- \quad a- \quad z-$$
$$a- \quad z-$$
$$b- \quad z-$$

Each line of this event log is its own trace, produced by a separate simulation. These traces have been split into what caused the output to transition high, and transition low. We separate these in order to determine separate causes.

The event log is now be passed into PGminer, which will extract concurrency and produce Boolean functions. These functions are, for each transition of the output, the combinations of concurrent events which are required for the given output transition. The given functions for this event log are:

$$z+ : a \wedge b$$
$$z- : (\overline{a} \wedge \overline{b}) \vee \overline{a} \vee \overline{b}$$

For `z+`, the traces have been mined, discovering that `a+` and `b+` are concurrent, as in one trace, `a+` occurs first, and in the other `b+` occurs first. However, `z+` only occurs after both `a+` and `b+`, so the function requires that `a+` AND `b+` have occurred.

With `z-` we have a few more traces. In two traces, both `a-` and `b-` occur, in either order, so this is included in the function as $(\overline{a} \wedge \overline{b})$. However, in two other traces, either `a-` occurs only, or `b-`. In these cases, the function ORs these requirements with the previous function, providing us with the full Boolean function.

With these functions, we can now use the method from Section 4.2 to generate concepts. This will simplify the functions, which causes no change for the `z+` function, but for `z-` will produce $\overline{a} \vee \overline{b}$. The concept specification generated from this will therefore be:

```
example22 = outRise <> outFall <> interface <> initState
  where
    outRise   = [a+, b+] ~&~> z+
    outFall   = [a-, b-] ~|~> z-
    interface = inputs [a, b, c, d] <> outputs [z]
    initState = initialise0 [a, b, c, d, z]
```

The gate used in this example is an AND gate, and the behaviours specified in the generate concept specification reflects this.

4.4 User generated libraries of concepts

Throughout the design process when using Asynchronous Concepts, a user may have generated several of their own concepts, which are used multiple times, or may have use for future specifications. It is possible to simply copy-and-paste these concepts, to use them as part of other compositions, but this leads to long specifications which are difficult to comprehend.

To combat this, a user can create their own concept library. This stores some useful and important concepts which a user chooses, and make the references to them simpler, and the concept specifications they are writing clearer, and less repetitive. The library can be imported during translations, and use other concepts libraries, including the circuit-specific concept library included with Plato.

For example, if a user frequently uses several gates not included in the provided library throughout their specifications, such as a NAND gate, rather than specify the NAND gate as

```
bubble z (andGate a b z)
```

each time, then a user library could derive a NAND gate concept.

```
nandGate a b z = bubble z (andGate a b z)
```

`nandGate` can then be used instead of the larger concept with the resulting concept being the same.

This feature allows a user to specify a concept which is complex only once, but use it many times. It allows a user to define their own gates and protocols which are useful for their designs, and not need to specify it each time. This promotes reuse, and reduces complexity in specifications.

A user library must include a *module name*. This is used as a reference when importing the chosen libraries. The module should be something which describes the concepts contained within, for example, if we created a library for inverted output gates, containing a NAND concept, NOR concept, inverted output C-element concept and so on, we could name this `InvertedGates`, and it could contain the concepts as shown in Figure 4.4

```
module InvertedGates where

import CircuitConcepts

nandGate a b z = bubble z (andGate a b z)

nandGateN inputs output = bubble z (andGateN inputs output)

norGate a b z = bubble z (orGate a b z)

norGateN inputs output = bubble z (orGateN inputs output)

nCElement a b z = bubble z (cElement a b z)

nCElementN inputs output = bubble z (cElementN inputs output)
```

Figure 4.4: `InvertedGates` user concept file, with reusable concepts

4.5 Asynchronous Concept translation

At this stage in the design flow, we will have a complete concept specification. This may be generated from different levels of Asynchronous Concepts from the circuit-specific library, while using concepts generated from the set and reset functions of an existing circuit, and importing concepts generated from process mining an existing circuit, all of which is used in conjunction with several concepts which a user has derived and stored in their own library.

For example, if we have the concept specification as found in Figure 4.5, we have used a `nandGate` concept as defined in the user generated library from Figure 4.4, the `InvertedGates` library, which has been imported. We also use signal-level `initialise` and `interface` concepts form the standard `CircuitConcepts` library included with Plato.

A concept specification cannot be used as is in any form. Concepts cannot currently be used directly for verification, to ensure that they satisfy the properties necessary for an asynchronous implementation, nor can a concept specification be used directly in synthesis, to produce an implementation which may be used to build a physical version of the circuit.

Concepts are in their infancy, and providing tools which verify and synthesize concepts is a time consuming process, and one which needs further research and development in order to produce. Instead, we provide a method of *translating* concepts to an equivalent

```
module Example23 where

import InvertedGates
import CircuitConcepts

example23 a b z = nandGate a b z <> interface <> initState
  where
    interface = inputs [a, b] <> outputs [z]
    initState = initialise0 [a, b] <> inintialise1 [c]
```

Figure 4.5: A concept specification ready for translation

specification in an existing formal specification method, namely Signal Transition Graphs (STGs) and state graphs.

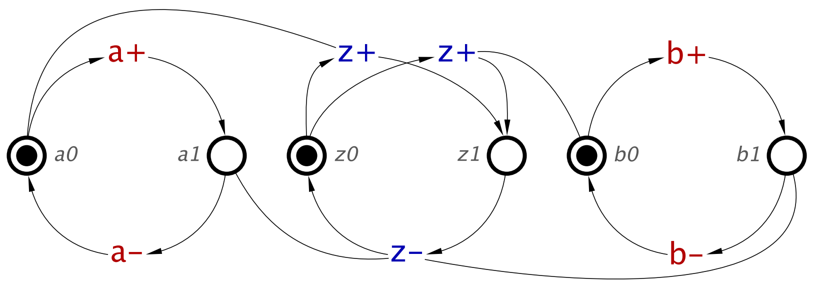
State graphs, a form of FSMs, provide a very low-level description of a system, as discussed in Section 2.1. This may be key to aid in the understanding of an asynchronous circuit for a designer, and thus a designer can translate a concept specification to a state graph, to view and simulate the system, making any changes they deem necessary to the concepts.

STGs provide a higher-level visualisation of a system and as discussed in Chapter 2, are much better at visualising the concurrency which is often a larger factor of asynchronous circuits. There are also several tools which automatically verify and synthesise STGs. These tools have a long history of research and development, and so are tried and tested in the field of asynchronous circuits, and are featured in several design flows and software suites. Translating a concept specification to an STG therefore means that a concept specification can be automatically verified and synthesize, removing the immediate need for tools specifically for concepts.

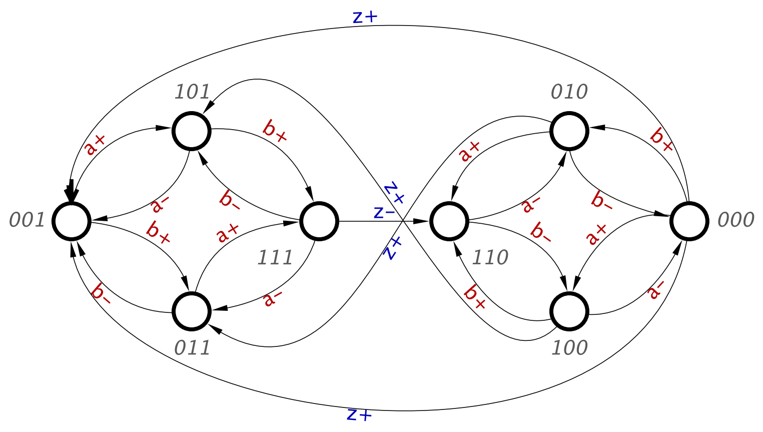
`example23` from Figure 4.5 will be translated into an equivalent state graph or STG, as seen in Figure 4.6.

It is not necessary that a designer first visualise the STG translated from a concept specification before they continue to verify and synthesize the system. A translated STG may be immediately passed to a tool which will perform these operations.

Translation is a key feature of Plato and will be discussed in Section 5.1. The operation is to call Plato, passing the file containing the target concept specification, as well as any of the files from which concepts are imported.



(a) Translated STG of a NAND gate, from `example23`



(b) Translated state graph of a NAND gate, from `example23`

Figure 4.6: STG and state graph translations of `example23`

4.6 Combining concept specifications

Some systems may be made up of multiple different concept specifications, each of which is a separate scenario which performs its own operation and has its own intricacies. In this case it may be beneficial to specify, translate, simulate and verify each scenario separately, resulting in an STG for each specification. Once all of these have been determined to be sound, these scenarios need to be combined for the full system specification.

With the STGs translated from concept specifications, we can combine specifications according to a few templates which state how these scenarios interact, when they run, and what determines which one is running at any time. These combinations must be performed at the STG level currently. Performing combinations at the concept level is a challenge for future research, and may be performed with the help of *Process Windows* [40].

4.6.1 Sequential template

Sequential combination is where an order is chosen for two or more scenarios, each running one after the other. This is, for example, useful for start-up scenarios, which initialise a system. This should be used for when there is a clear order for the specified scenarios to be run in. A token enters the first scenario, and is passed through its operations, and when this is complete, the token is passed directly into the next scenario.

This involves adding a place at the start of the first scenario, which will pass a token into this scenario when it runs. Between this scenario and the second, another place will be added to receive the token from the first scenario, and pass it into the second. A place will need to be added between each sequential scenario in this way. A place will also need to be included at the end of the final sequential scenario. This will take the token from the final scenario, signalling the end, but can also be used to handle tokens which should be passed to other scenarios combined with any other templates.

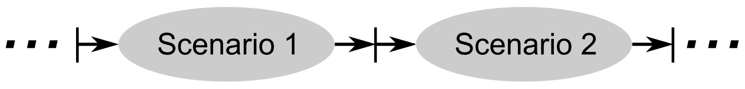


Figure 4.7: Example of sequential combination

4.6.2 Concurrent template

Concurrent combination allows scenarios to run at the same time as each other, both being able to begin at the same time, but for the system to move on, all of the concurrent scenarios must complete. For example, a circuit may feature two separate paths of logic, and when an input signal transitions, both of these paths will propagate this transition, but only when this has propagated through both will this cause a change to an output signal.

It may be necessary to limit how many of these scenarios run concurrently, by allowing only a certain number of tokens, one of which will be used by each scenario. This acts like a large interpretation of the fork and join concepts as explained in Section 2.3.

Each scenario will be provided with an input place, which receives the token from a fork. This token will then be passed through the scenario, and following completion, an output place will be added which will hold the token from this scenario until the join, which connects all of the concurrent scenarios, is enabled. When this fires, the token from each of these output places, one from each concurrent scenario, will be consumed.

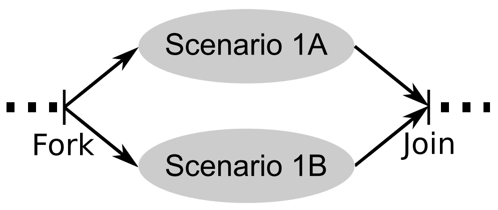


Figure 4.8: Example of concurrent combination

4.6.3 Choice template

Scenarios which are combined using the choice template are determined to only allow one to be active at a time, but there is a choice based on other knowledge about which scenario will run next. This is a larger scale choice and merge construct as discussed in Section 2.3. For example, if a mutual exclusion element (`meElement`) is used in a circuit, when both requests, `r1` and `r2`, arrive close to each other, a choice must be made about which grant signal, `g1` or `g2` must be set high, as due to the mutual exclusion, only one of these signals can be high at a time. When the choice is made, the circuit will continue to operate, and then merge, setting the first grant signal low once again in order to allow the second grant signal to be set high.

A choice place is used before all of the combined scenarios in this template, that connects to each of the first transitions in each scenario. This choice place contains a single token, which is passed into whichever scenario is chosen, blocking any other scenario from starting. When the running scenario is complete, the token is placed into a merge place which is connected to from all of the final transitions of each combined scenario. This token is then passed on for further operations in the system, or if the system contains only scenarios included in the choice, the choice and merge places may be one and the same.

The choice and merge places are used instead of providing a place at the beginning and end of each scenario, as with the sequential and concurrent combination templates. This ensures that only one of the choice scenarios can run at one time. The first element of each scenario in this case will be a transition, and only one transition will be able to fire, consuming the single token in the choice place.

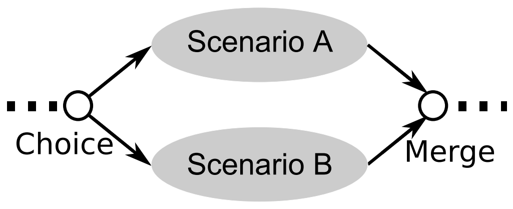


Figure 4.9: Example of a choice combination

4.6.4 Complex combinations

It is often the case that scenarios in a system will not only interact through one of these methods. It is possible to combine a specification of previously combined scenarios with another scenario or scenarios using another of these templates.

For example, if a system features a start-up scenario, and several other scenarios only one of which is run at a time, then these can be combined using the choice template, then a sequential combination can take place, with the start-up scenario as the first scenario in the template, and the previously combined set of scenarios as the second.

This provides multiple combination possibilities, accounting for scenarios running in multiple orders, and the combinations could be performed automatically, reducing the chance of error during this step [41].

The positioning of places included in each template ensures that each set of scenarios when combined can then be included in a combination template with other scenarios, or previously combined scenarios.

4.7 Verification

Before an implementation is found through synthesis, a specification needs to satisfy certain properties which determine whether it is implementable. These properties include:

- Signal consistency: in any trace the rising and falling transitions of each signal alternate.
- Deadlock freedom: no state is reachable from which no progress can be made.
- Complete State Coding (CSC): each state of the model with different behaviour has differing signal encodings to avoid problems during synthesis. Note that if the conflict is not *irreducible*, then it may be resolved automatically [42].

- Output persistence: No output signal can be disabled by any other signal transitioning.

Custom properties may also be checked, and in the event that a never concept is used in a specification, following the translation, we perform an automatic custom verification to check whether the states indicated in the never concept are unreachable.

As an example, we have a concept specification as follows:

```
example24 a b z = buffer a z <> buffer b z
                <> z+ ~> a+ <> z+ ~> b+
                <> inputs [a, b] <> outputs [z]
                <> initialise0 [a, b, z]
```

This is similar to a C-element, with buffers from the input signals, a and b to output signal z. Also included however is two environment concepts, which state that the output must be high for both inputs to go high. The translated STG of this concept specification is found in Figure 4.10.

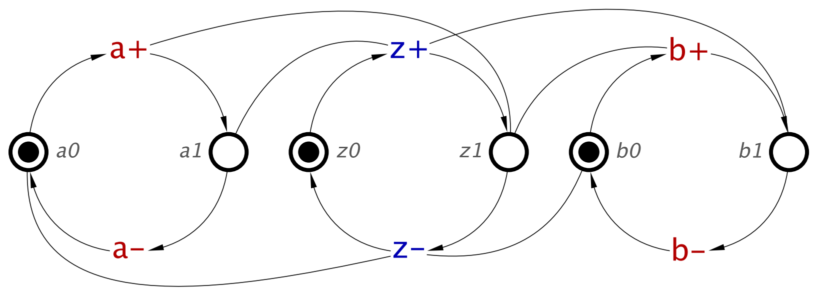


Figure 4.10: example24 translated to an STG

Note the read-arcs between a+ and z1, and b+ and z1. These require that z have transitioned high for a+ and b+ to occur. However, the read-arcs between a1 and z+, and b1 and z+ require that both inputs are high for the output to transition high.

This poses a problem. z+ will only occur if a+ and b+ have occurred, but a+ and b+ will only occur if z+ has occurred. Thus, this means that when the system has initialised, there are no possible transitions that can occur. This is a *deadlock*, and is one of the key properties that is verified, as a circuit will not be synthesized if there is no transitions that can occur. This is just one example of how an STG is verified.

There are several tools which perform the verification of STGs automatically and these tools will be discussed in further detail in Section 5.2.3.

4.8 Synthesis

Once a specification has been verified, the final step is to synthesize the specification. This process finds set and reset Boolean functions which describe the polarity and combination of signals which are needed to set the outputs high and low.

These functions can then be used in a number of synthesis methods. The most practically useful method of synthesis is *Technology Mapping*. This involves using a gate-library, containing a list of gates and the Boolean functions for these gates. The set and reset functions determined from an STG for synthesis will be mapped against the functions for the gates in the library, finding one or more gates which represent the determined functions.

Technology mapping is important in practice as a library can be provided which contains only the gates that are available in the type of logic being used, which can differ based on the technology processes used to create the set of gates. In the event that technology mapping cannot provide an implementation, the specification can be changed to ensure that only the gates in the library are used.

Synthesis is also often aimed at producing a *speed-independent* (SI) implementation. Speed-independence is a type of asynchronous circuit which will operate correctly regardless of any delays in the logic gates. This class of circuit ensures that an implementation will continue to work if the associated delays change at all, such as will occur with differences in power, and devices made using a different process technology [43].

If we have the STG for a 2-input AND gate (Figure 3.18), and we synthesize this, we will be provided with the set function of: $a \wedge b$. This is mapped against a gate library, which features an AND2 gate and has the registered function of $a \wedge b$. The function is therefore mapped onto an AND2 gate, and the digital circuit would be as seen in Figure 4.11.

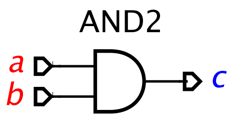


Figure 4.11: Technology mapped AND gate

We can synthesize without mapping, but this can lead to a digital circuit featuring logic gates which have multiple layers and do not exist in conventional libraries. Thus, the circuit will need decomposing in order to find multiple gates which are mappable, a process which is not always possible.

Figure 4.12 contains such a circuit. This is a gate, comprised of several separate gates, the outputs of which lead into the inputs of a gate in the next layer, but is not a standard gate in and of itself, and thus cannot be technology mapped.

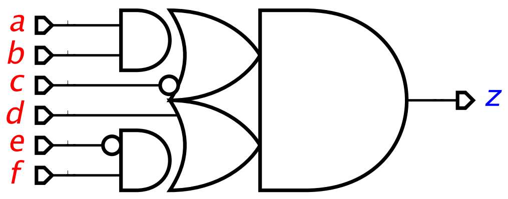


Figure 4.12: A circuit synthesized without technology mapping

Synthesis is also performed automatically by the same tools which perform verification automatically, which will be discussed in Section 5.2.3. Tools such as these will also perform some verification before attempting to synthesize, in order to avoid producing an implementation for a specification which is incorrect. Instead, an error will be given to the user, identifying the problem with the specification.

The implementation can then be built in the real-world, which will then be subject to further testing, to ensure that for its application, it functions as expected. This is necessary as the design process may not account for some circumstances which the circuit will come under when in use in the real-world.

4.9 Summary

We have introduced the design flow using Asynchronous Concepts in this chapter. This can begin from an informal description of the system to be specified, or it can begin with the Boolean set and reset functions used to identify the operation of an existing circuit, which can be used to find a concept specification. It may also begin with a system that only the inputs and output signals are known, and through simulation then process mining, a concept specification can be determined. Any of these methods can be used in conjunction with any other, if, for example, a system uses existing circuits with or without known set and reset functions, with some new operations, described informally.

Any concepts which are used multiple times throughout a specification, or specify behaviours which may be useful in future designs, can be stored in a user-generated concept library. This can be imported to a concept specification, and the necessary concepts simply used as any other concept, passing in the involved signals, and composed with other concepts.

Asynchronous Concepts cannot currently be directly simulated, verified or synthesized, and as such, must be translated to another form which can be used in these processes. Part of the design flow is compiling a concept specification, which identifies errors in the concepts, and when error-free, translates them to an existing modelling method, either STGs or state graphs. Each of these have their benefits, but STGs are the primary translation target, as these have a wealth of tools for simulation, verification and synthesis.

The concept specification may consist of multiple separate specifications, each of which specify a different mode or scenario, for example. In this event, the specifications need to be combined, which after translation they can using certain templates. These templates can combine STGs in certain ways, depending on how these scenarios interact.

After reaching a full specification, this must be checked to ensure it functions as required, and satisfies certain properties. Simulation can be performed on a translated STG or state graph, which allows a user to ensure that only the expected transitions can occur. STGs can be verified, ensuring that it can be used to derive an implementation.

STGs determined to be sound through simulation and verification can then be synthesized to find an implementation. This process finds Boolean functions which cause an output signal to be set (rise) or reset (fall). These functions are then used to find a set of logic gates which can produce the output signals from the input signals.

Chapter 5

Automation of the design flow

In previous chapters, we have introduced Asynchronous Concepts, and the associated design flow. Many of the processes which have been discussed may be done manually, but this is a time consuming process and is subject to human error, which adds further time to the design process. There are however several software tools designed to perform these operations automatically, which thus decrease the design time of a system.

In this chapter, we discuss several electronic design automation (EDA) tools. Some of these have been introduced recently, and provide uses specifically for Asynchronous Concepts, and as such, we will discuss the algorithms designed for these tools. Others tools were introduced previously, but their functionality is used in conjunction with concepts, and we will explain how they have been adapted for concepts and the design flow.

Plato, discussed in Section 5.1, is a software tool specifically for Asynchronous Concepts, and has multiple features. It contains a circuit-specific concept library, and automatically compile concept files, which identifies errors with syntax and concepts in the file. These are then translated to other modelling formalisms, and the algorithms will be introduced and explained. It can also be used to generate concepts from Boolean functions, which will also be discussed.

PGminer is a tool designed for the purpose of process mining, mining event logs of any system and producing Boolean functions. This tool can be adapted to work with simulation traces from existing systems, and we will discuss the tool and its integration with the design flow and Workcraft in Section 5.2.2.

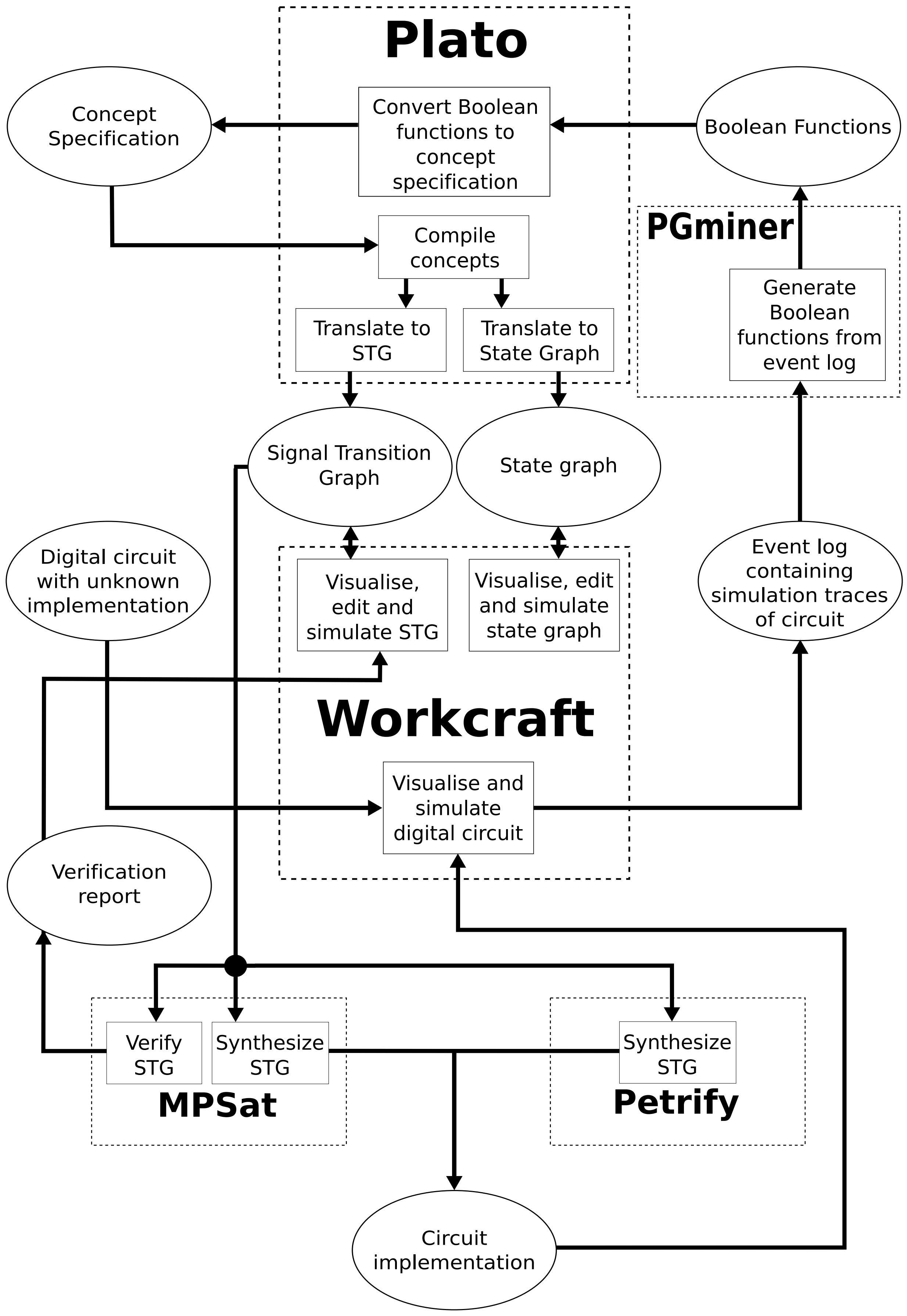

Plato
Concept Specification
Convert Boolean functions to concept specification
Boolean Functions
Compile concepts
PGminer
Generate Boolean functions from event log
Translate to STG
Translate to State Graph
Signal Transition Graph
State graph
Digital circuit with unknown implementation
Visualise, edit and simulate STG
Visualise, edit and simulate state graph
Event log containing simulation traces of circuit
Workcraft
Visualise and simulate digital circuit
Verification report
Verify STG
Synthesize STG
MPSat
Synthesize STG
Petrify
Circuit implementation

Figure 5.1: A diagram showing the interoperability of the software tools.

There are several tools which all work together but are not necessarily designed to work with each other, which means that there are some difficulties when using the results from one tool as the inputs to another. A method to combat this comes in the form of Workcraft. This software suite uses many tools as a backend, including Plato, and displays the results visually, and automatically pass these results into other tools. This creates a more user-friendly design environment, and we will discuss how these tools are integrated in Workcraft in Section 5.2.

Several tools exist which verify and synthesize STGs, and we use these for the STGs which are translated from concept specifications. These tools are commonly used for the STG design flow, and are used as part of the Asynchronous Concepts design flow. We will explain how they are used with concepts, and from within Workcraft in Section 5.2.3.

Figure 5.1 is a diagram of how the tools discussed in this chapter interoperate, to automate the design flow. It features the tools, and the features used in the design flow, as well as the intermediate data types which are used to communicate between the software.

As discussed in Section 2.7, the syntax of Haskell does not allow for signal transitions in Asynchronous Concepts to be stated in the form `a+` and `a-`. For clarity we have continued to use this form however in previous chapters. From this point on, as we will be using examples for tools which use Asynchronous Concepts, we will begin to use `rise` and `fall` for signal transitions in concept specifications. Signal transitions such as `a+` and `a-` will now be replaced by `rise a` and `fall a`.

Throughout this chapter, we will use a single example to show how the tools in question work, and in some cases using this example to show how an algorithm works. First we must provide a concept specification for this example. This example is of a 2-input NOR gate, which is enabled by a third input signal. The specification is found in Figure 5.2.

```
example25 a b e z = behaviour <> initState <> interface
  where
    behaviour = enable (rise e) z (bubble z (orGate a b z))
    initState = initialise0 [a, b, e] <> initialise1 [z]
    interface = inputs [a, b, e] <> outputs [z]
```

Figure 5.2: 3-input NOR gate concept specification file

This file includes more than just the concept specification, so we will discuss this. The actual concept specification, `example25`, will be compiled when passed to Plato. It features 4 signals, `a`, `b`, `e`, and `z`. This concept specification is composed of three concepts, `behaviour`, `initState` and `interface`.

Following this and the `where` are some *local* concept declarations. These concepts can only be used within this function, `example25`. The three concepts which are composed for this specification are declared here. `initState` and `interface` are named for their purpose, to declare the initial states and the interface of the signals respectively.

`behaviour` is the most important concept here. This is what describes the operation of the circuit. Note, that we have not imported the derived NOR-gate concept from Figure 4.4. This example describes the circuit using this manner in order to better indicate how the tools described in this chapter operate.

5.1 Plato

Plato [16] was introduced in [15] as an implementation of the domain-specific language of Asynchronous Concepts, written in the functional programming language Haskell. Its main function is to take a concept specification, importing all associated concepts, and *compiling* this. This notion of compiling a specification comes from the idea that STGs and state graphs are the *assembly language* of asynchronous circuits, and Plato compiles the higher-level language of concepts to produce lower-level STGs and state graphs.

The compilation takes place at the beginning of the translation process, where a concept specification is used to generate an equivalent STG or state graph. This uses the *Glasgow Haskell Compiler* (GHC) to perform this, as the domain-specific language of concepts is implemented in Haskell, and as such, a file containing a concept specification is in itself a file which features Haskell code, and uses the same syntax (Section 2.7).

Plato also has the feature of generating concepts from Boolean set and reset functions, similar to the `function` and `complexGate` concepts. Rather than using the concepts generated as part of a concept specification however, Plato will generate a separate concept specification consisting of the concepts generated from the Boolean functions. This can then be used in several ways.

5.1.1 Translation to STG

Translation from Asynchronous Concepts to STGs is the key feature of Plato, as currently, there is no tool which verifies and synthesizes a concept specification directly. This may indeed be possible, but developing such tools is a time consuming process, and is an opportunity for further research and development.

Therefore, we provide a method of translating concepts to existing formalisms which are more commonly used, and feature several commonly used tools which automatically verify and synthesize specifications.

This process is performed by calling Plato, passing in the file path which points to the concept specification file, and any file paths of concept libraries that this specification uses, excluding the circuit-specific library provided with Plato.

The algorithm to translate a concept specification to an equivalent STG is introduced in Algorithm 2. In this section, we will explain how the translation algorithm works using the example of the 2-input NOR gate with `enable`. This will show the intricacies of the translation algorithm.

The main function of this algorithm is *translate-stg* which is called by Plato. This takes in a concept specification, and can access the information contained within this, such as the list of cause and effect transitions, all signals in the system, and so on. This calls several other functions which are also included in the algorithm, and passes in this information for it to compile the concepts, and build the STG The first function this calls is *list-transitions*, passing in the list of effect transitions.

Regardless of the concepts used in the specification, be they signal-level or higher-level gate and protocol concepts, the algorithm uses the atomic signal-level concepts, as all concepts are made up of these. *list-transitions* takes each effect transition, and lists all of the cause transitions for them. However, there may be multiple lists, each individual list being a list of possible causes, in the form of OR-causality, and the collection of lists being all causalities in Conjunctive Normal Form (CNF), i.e. each list is composed with AND causality, but the sub-lists are OR-causality.

The `behaviour` concept from the specification for this example with all concepts in atomic form is found in Figure 5.3. The `initState` and `interface` concepts use only signal-level concepts, thus they are in their most atomic form, so we will omit them from this figure.

The algorithm takes this and then forms lists of the cause of each effect, the effects in

Algorithm 2 Algorithm for translation from Asynchronous Concepts to STGs

```
//Main function. Takes a concept specification which includes
//signals, cause and effect transitions and generates an STG.
function TRANSLATE-STG(concept-specification)
    transitions ← list-transitions(effects)
    loops ← create-loops(signals)
    loops-with-inits ← set-initial-states(signals, loops)
    stg ← connect-transitions(transitions, loops-with-inits)
end function

//Takes list of effects, performs the Cartesian product on the causes for each.
//Returns a list of cause transitions for each effect in DNF.
function LIST-TRANSITIONS(effects)
    for all effects do
        transitionList ← cart-product(causes(effect))
        transitions ← transitions + transitionList
    end for
end function

//Takes list of signals. Creates signal transition loops.
//This connects transitions and places for each signal, and sets the interface.
function CREATE-LOOPS(signals)
    for all signals do
        set-interface(signal)
        create-place(signal-name + 0)
        create-place(signal-name + 1)
        for all transitions of signal do
            if transition is rising then
                create-consuming-arc(0 place, transition)
                create-producing-arc(transition, 1 place)
            else if transition is falling then
                create-consuming-arc(0 place, transition)
                create-producing-arc(transition, 1 place)
            end if
        end for
    end for
end function
```

```
//With the transition loops, checks each signal's initial state concept.
//Sets a token in the correct place, 1 place for high, 0 place for low.
function SET-INITIAL-STATES(signals, loops)
    for all signals do
        if init-state(signal) is low then
            add-token(0 place)
        else if init-state(signal) is high then
            add-token(1 place)
        end if
    end for
end function

//With the list of transitions, connects the place signalling a cause transition
//to the effect transition, connecting all causalities.
function CONNECT-TRANSITIONS(transitions, loops)
    for all transitions do
        if transition is rising then
            for all effects of transition do
                create-read-arc (1 place, effect)
            end for
        else if transition is falling then
            for all effects of transition do
                create-read-arc (place 0, effect)
            end for
        end if
    end for
end function
```

```
behaviour = [rise a, rise b] ~|~> fall z
          <> fall a ~>rise z <> fall b ~> rise z
          <> rise e ~> rise z <> rise e ~> fall z
```

Figure 5.3: `behaviour` concept in atomic form

Table 5.1: OR-causality lists by effect

Cause transitions	Effect transition
`rise a, rise b`	`fall z`
`rise e`	`fall z`
`fall a`	`rise z`
`fall b`	`rise z`
`rise e`	`rise z`

this example being `rise z` and `fall z`. Table 5.1 contains these lists. For `fall z`, there are two lists. One contains the rising concepts for the input signals, `a` and `b`, for the NOR gate, and one contains only the enable transition. In CNF, these two lists can be stated as a Boolean Function: $(a \vee b) \wedge e$. When translating to an STG, we add transitions for `fall z` into the transition loop, but this requires a set of possible cause transitions for each of these `fall z` transitions. Thus we need to combine these two lists in a way which applies all OR causalities, and the AND causalities.

This operation is done with the *Cartesian Product*, which in the algorithm is called by *cart-product*. This function applied to lists will AND each element of one list with each element of all other lists, and OR each of these. We treat the OR-causality list of transitions as is, but we must also treat the `rise e` transition as its own list of one. Then, performing the Cartesian product, on both of these lists, we get two lists:

```
[rise e, rise a], [rise e, rise b]
```

This is a list of two possible combinations of transitions which must occur for `fall z` to occur, and each is applied to a separate `fall z` transition in the transition loop. This process converts the transition combinations into *Disjunctive Normal Form* (DNF), which for translation is the arrangement we require. The Boolean Function of this is now: $(e \wedge a) \vee (e \wedge b)$.

We must then perform this for `rise z`. For this transition, there are three separate AND causalities, and no OR-causalities. We therefore treat each as a single element list, and perform the Cartesian product. For these transitions we simply combine all three cause transitions, leading to a single list:

```
[rise e, rise a, rise b]
```

We now have a set of causality concepts in DNF form, as seen in Table 5.2. There are two possible `fall z` transitions, which are numbered, so they can be referenced separately but still identify a falling transition for `z`. All of these cause transitions, for the given effect transitions are now stored in a list.

Table 5.2: Causalities in DNF form

Cause transitions	Effect transition
`rise e, rise a`	`fall z/1`
`rise e, rise b`	`fall z/2`
`rise e, fall a, fall b`	`rise z`

translate-stg can now start to build the STG. It calls the function *create-loops* passing in all signals, which creates the signal transition loops, as were discussed in Section 2.4. For each signal in the system, it starts by setting the interface and creating places for the signal, naming these with the signal name and a 0 or a 1. Next, it connects all transitions of the signal to these places, rising transitions have a consuming arc from the 0 place to the transition, and a producing arc connects the transition to the 1 place. The falling transition connects the opposite, consuming arc from the 1 place to the transition, and a producing arc from this to the 0 place. This completes the loop for this signal, and the function will continue to do this for all signals, returning all of these loops to be used by *translate-stg*.

The next function called is *set-initial-states* and passes in these newly created signal transition loops, to prepare the initial states. This simply takes the initial state concept for each signal, placing a token in the 0 place if it is initially low, or placing a token in the 1 place if the state is initially high. Figure 5.4 contains the STG after the initial states have been inserted. For this example, all signals are initially 0, except the output signal, `z`, which is initially 1. Therefore all input signals will have a token in their 0 place, and `z` will contain a token in its 1 place. *set-initial-states* then returns the loops with the initial states.

Finally, the *connect-transitions* function is called, which takes in the initialises signal transition loops and the list of transitions. For each cause transition, it determines

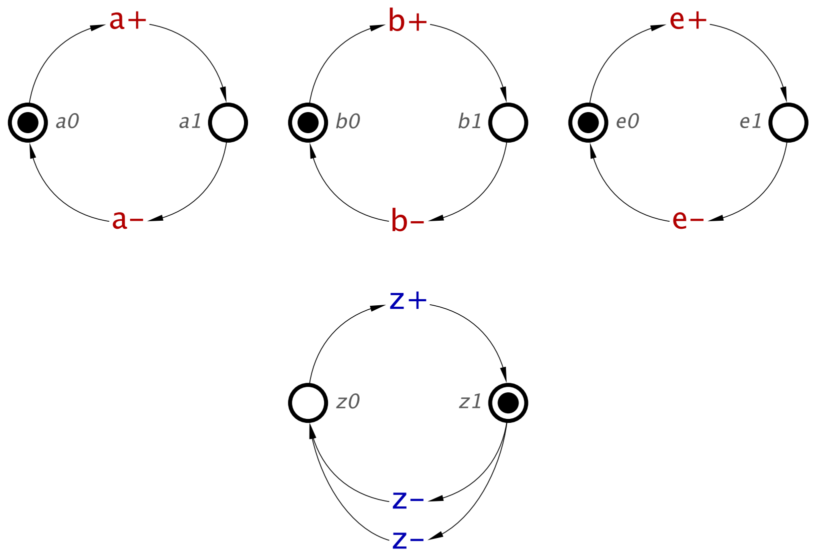


Figure 5.4: Translated STG with initial states inserted

whether it is a rising or falling transition. If it is rising, it then connects the 1 place of this signal to the effect transition with a read arc, and if it is a falling transition, it does the same with the 0 place of this signal. Because of the placement of the token in a transition loop, this is used to determine the polarity of this signal, and as such, can be used to determine whether the effect transition is enabled.

Looking at the list of causalities in Table 5.2, the first cause transition is `rise` e and the associated effect transition is `fall` z/1. The /1 reference in the signal is not identified in the STG, but is used to refer to one of the `fall` z transitions. This transition is then connected to the `e1` place, as seen in Figure 5.5.

This is then repeated for each transition in the system, next connecting `rise` a to `fall` z/1, and so on. When all read arcs are inserted, the translation is complete, and the resulting STG is equivalent to the concept specification. This STG returned to *translate-stg*, which can then print it. The completed STG for the NOR-gate with enable gate example is found in Figure 5.6.

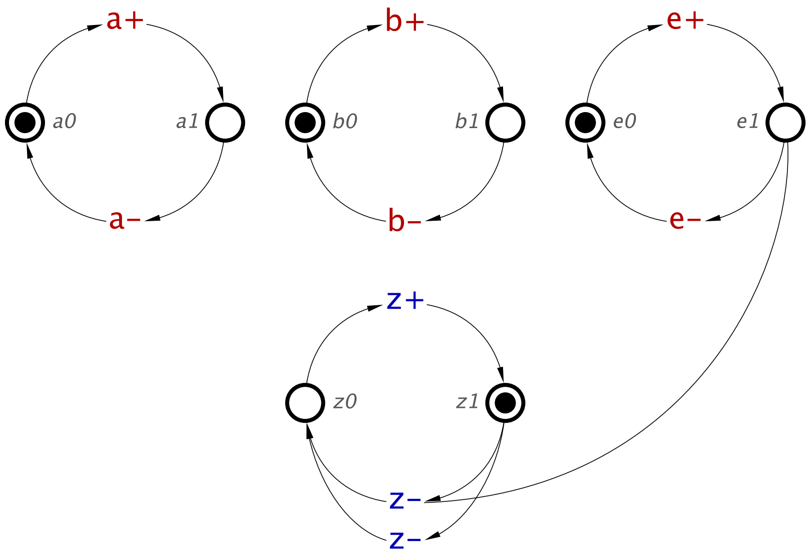


Figure 5.5: Translated STG with one read arc inserted

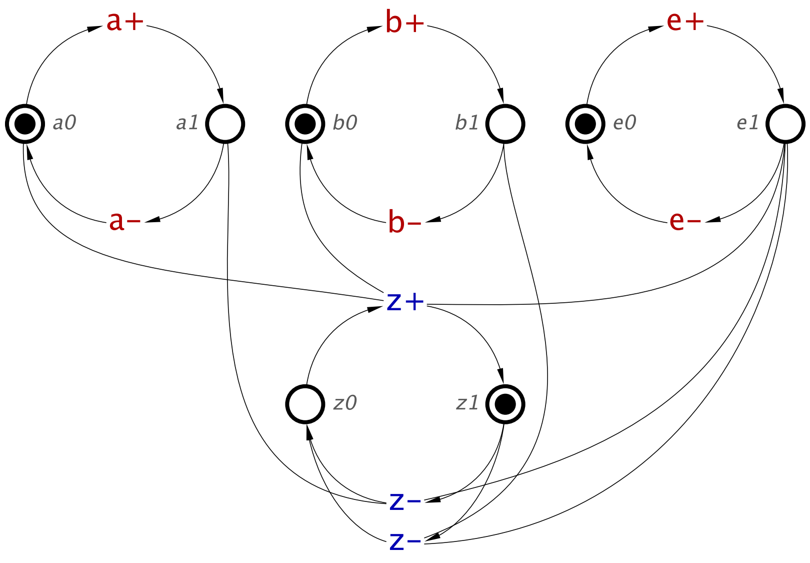


Figure 5.6: Fully translated STG

5.1.2 Translation to State Graphs

As discussed in Section 2.1, state graphs are undesirable when specifying asynchronous circuits, and there are fewer major verification and synthesis tools for state graphs compared to those for STGs therefore. However, state graphs may still be useful for viewing the operation of a system as they are at a low-level and so, it may be easier to understand signal interactions with an state graph.

As part of Plato, we provide translation from Asynchronous Concepts to state graphs.

Algorithm 3 Algorithm to translate Asynchronous Concepts to state graphs

```
1: //Main function. Takes a concept specification. Uses the signals,
2: //and causalities to generate a state graph.
3: //Uses some functions from Algorithm 2
4: function TRANSLATE-STATE-GRAPH(concept-specification)
5:     consistent-spec ← add-consistency(signals)
6:     transitions ← list-transitions(effects)
7:     init-encoding ← get-initial-encoding(signals)
8:     state-graph ← encode-and-connect(transitions, signals)
9: end function
10:
11: //Takes in the signals, and provides consistency to the state graph, ensuring
12: //that each signal can transition high when it is low, and vice versa
13: function ADD-CONSISTENCY(signals)
14:     for all signals do
15:         set-interface(signal)
16:         concept-specification <> rise signal ~> fall signal
17:         concept-specification <> fall signal ~> rise signal
18:     end for
19: end function
20:
21: //Uses the initial states of each signal find the encoding of the initial state.
22: //This state is then designated as initial in the generated state graph
23: function INIT-ENCODING(signals)
24:     for all signals do
25:         if init-state(signal) is low then
26:             initState ← initState + 0
27:         else if init-state(signal) is high then
28:             initState ← initState + 1
29:         end if
30:     end for
31: end function
```

```
//For each effect, finds the source state based on the cause transitions,
//the destination state, and connects them with the effect transition.
function ENCODE-AND-CONNECT(transitions, signals)
    for all each effect-transition do
        for all cause-transitions(effect) do
            for all signals do
                if signal is cause-transition then
                    srcEnc ← srcEnc + polarity(cause-transition)
                else if signal is effect-transition then
                    srcEnc ← srcEnc + polarity(effect-transition)
                else
                    srcEnc ← srcEnc + dontCare(signal)
                end if
            end for
            allSrcStates ← replaceDontCares(srcEnc)
            for all allSrcStates do
                destState ← togglePolarity(effect-transition)
                connect(srcState, destState, effect-transition)
            end for
        end for
    end for
end function
```

While not as key a feature as translating to STGs, it is still useful, and the difference in usage is negligible. Plato is called as usual, passing in the file path for the concept specification file and any user-generated concept libraries that are used by this specification, again, excluding the circuit-specific concept library. To translate to a state-graph instead of an STG, a flag is added: `-f`, or `--fsm`.

The algorithm for this translation is presented in Algorithm 3. The main function is *translate-state-graph*, which again calls other functions, including *list-transitions*, as in Algorithm 2. This is because it is still necessary for the transitions to be converted to DNF form for translation. As such, we do not include *list-transitions* in this algorithm. The main difference in how these algorithms differ is in how the STG and state graph are built. We will explain the operation of this algorithm in this section, once again using the example of a NOR gate with an enable signal for the output.

The algorithm first calls *add-consistency*, passing in the signals in this concept specification. This defines the interfaces of all signals in the system, and adds the consistency for each signal, which for state graphs requires that for any signal transition, the opposite signal transition has occurred first. Thus, we add two concepts for each signal:

```
rise x ~> fall x <> fall x ~> rise x
```

Where `x` is any signal. These are simple concepts, but this ensures that for any transition in the system, it will only occur when the opposite transition has occurred. Including these concepts here means that they are included in in *list-transitions* conversion, which ensures that the state of a signal for an effect transition is a requirement. For signals which are never an effect transition, this ensures that they can transition freely, but a transition does not mean this signal stays in that state indefinitely.

For clarity, we will omit these consistency concepts from the worked-through example, as it adds minor details which do not add to the algorithm explanation.

Following this, *translate-state-graph* calls *list-transitions*, the same function as in Algorithm 2. This is because for translation to either STG or state graph, we need the causalities to be in DNF, and as such, we convert them in the same way. Therefore following this function call, Table 5.2 will be produced once again.

translate-state-graph will then call *init-encoding* to find the initial state for the state graph. Rather than placing tokens as in STG translation, this involves finding the encoding for this state. The encoding for all states is written using 0 to represent a signal being low, and 1 to represent a signal being high, and the order of signals is `a`, `b`, `e`, `z`. Thus, from the concept specification, the initial state for the two-input NOR gate with enable will be 0001. *init-encoding* returns this initial state which is used for the translated state graph.

The final function, *encode-and-connect* builds the state graph. Initially, for each effect signal transition, we produce a source encoding, the encoding of the states from which the effect transition occurs. This is done by taking the effect transition, and using the cause transitions, generate this state. It may be the case that not every signal in the system will be involved in this, and for these signals we apply a *don't care* value, represent by a '-' character.

For the first causality from Table 5.2, `rise` e and `rise` a causing `fall` z, we check for each signal in the system. The first signal is `a`, which is involved in this causality, and for the effect transition `fall` z, `a` must be high, so the source state requires that `a` be 1. Signal `b` is not involved in this causality, so the source state does not require `b` to be any state, so the polarity of `b` is set to don't care, or '-'. `e` is involved, and it is required that `rise` e has occurred, so the source state needs `e` to be 1. Finally, we have `z`, which is the effect signal, and due to consistency, it is necessary that `z` be high to allow a falling transition to occur, so we set the source state to include `z` at 1. The final source state for this causality is 1-11.

Table 5.3: Causalities and source encoding

Cause transitions	Effect transition	Source encoding
`rise e`, `rise a`	`fall z`	1-11
`rise e`, `rise b`	`fall z`	-111
`rise e`, `fall a`, `fall b`	`rise z`	0010

This is performed for every causality in *encode-and-connect*, and Table 5.3 contains this list of causalities with the source states. We use these encodings to find a list of all states where the effect transitions can occur, which is performed on line 46 of the algorithm. With just the first causality, we take the encoding 1-11 and remove the - states, converting them into all possibilities of 0 and 1, providing us with more states, with encodings which are used to reference actual states in the system. 1-11 expands into: 1011, 1111.

Finally, the algorithm generates a destination state encoding from each of these states, which is the state encoding of the state *after* the effect transition. The destination state of `fall z` from the source state 1011 will be 1010, and from the source state 1111, the destination state will be 1110. These states are then connected, adding the effect transition `fall z` as the event for the arc.

The remaining causalities are then translated to a state graph in the same way, and any states which have the same encodings as any states we have discussed may feature connections to or from other states due to the remaining causalities. This function will then return the translated state graph to *translate-state-graph* which will then output this.

Figure 5.7 contains the translated state graph for `example25`. This state graph is quite complex, for a simple circuit. It can be seen that z only transitions once e has transitioned, and either a or b for one of three possible z- transitions, or both a and b for the single z+ transition. States where e is low (the outer diamonds) have no z transitions.

Note that with the state graph translation algorithm, any CSC conflicts will be translated, and the functionality of the states with encoding conflicts will all be included all within the same state. This is due to how the algorithm builds the state graph using state encodings. As discussed in Section 2.1, CSC is important in a state graph for the purposes of synthesis. Unfortunately, the algorithm therefore will not produce an accurate state graph.

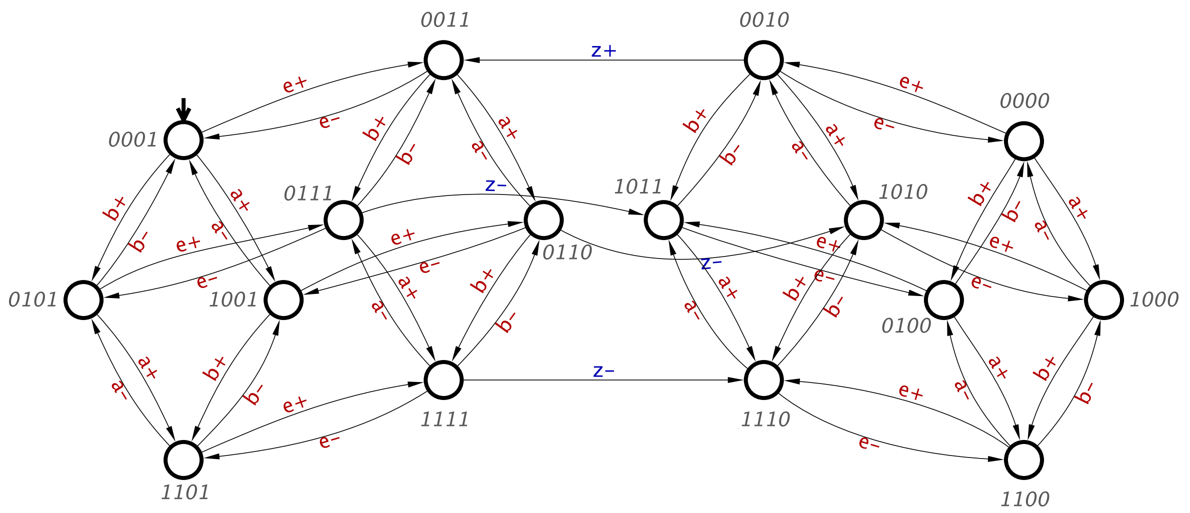


Figure 5.7: `example25` translated to a state graph

Since state graphs feature less tool support for verification, CSC conflicts will not be caught during the verification step, but this also means that there is less too support for synthesis of state graphs, and as such, CSC conflicts will not propagate to an implementation. This algorithm can be improved through future development, however, the major contribution of Plato is the translation of Asynchronous Concepts to STGs, so the state graph translation algorithm is not of the highest priority.

5.1.3 Generating concepts from Boolean functions

Boolean functions may be used directly in a concept specification with the `complexGate` and `function` high-level concepts (Section 3.3.1), or be used to generate a concept specification (Section 4.2). Both of these features are built into Plato, `complexGate` and `function` in the circuit-specific concept library, and a function called *bool-to-concept*.

In both cases, the Boolean functions may be in any form, and must be evaluated in order to find the CNF, which is used to generate concepts. Algorithms to convert Boolean functions into CNF are discussed in Section 2.6. CNF for the functions is necessary as this is the form in which concepts are represented in Plato.

Bool-to-concepts can take arguments of the set and reset functions or just a set function, a file path to a file in which to write the concept specification, and the name of the effect signal. If functions are not given, the user will be prompted to enter them, and the reset function can be left blank. In the event a reset function is not entered, the reset function will be treated as the inversion of the set function. If no output file path is given, the resulting concept specification will be printed to stdout.

Algorithm 4 Algorithm to generate concepts from Boolean functions

```
//Main function. Takes Boolean set and reset functions. Outputs a concept
specification.
//This converts the functions into CNF and uses this to generate concepts.
function BOOL-TO-CONCEPT(setFunc, resetFunc)
    if setFunc is empty then
        (setFunc, resetFunc) ← get-functions
    end if
    if resetFunc is empty then
        resetFunc ← invert(setFunc)
    end if
    allVars ← get-vars(setFunc, resetFunc)
    outVar ← get-output-var
    setCNF ← convert-to-CNF(setFunction)
    resetCNF ← convert-to-CNF(resetFunction)
    outputRise ← generate-concepts-text(setCNF, rise outVar)
    outputFall ← generate-concepts-text(resetCNF, fall outVar)
    print-concepts(fielpath, allVars, outVar, outputRise, outputFall)
end function

//Takes a function, and an output transition. Generates the concept for
//the output to rise or fall. Outputs this in text form.
function GENERATE-CONCEPTS-TEXT(function, output-transition)
    for all sub-functions in function do
        for all variables in sub-function do
            if variable is negated then
                causes ← causes + "fall" + variable
            else
                causes ← causes + "rise" + variable
            end if
        end for
        concepts ← concepts + "<>" + causes + "~|~>" + output-transition
    end for
end function
```

Algorithm 4 details how, given Boolean set and reset functions, a concept specification will be generated. We will use our NOR gate with enable example to detail the operation of this Algorithm. Suppose the functions given were as follows:

$$set : (\overline{a} \wedge \overline{b}) \wedge e$$
$$reset : (a \wedge e) \vee (b \wedge e)$$

The main function of this algorithm is *bool-to-concept*. As with the main functions in other algorithms, it calls other functions which perform useful functions, and return information for further functions. This function is passed a set and reset function, which may contain functions from calling this function, or may be empty. It begins by checking this, and if empty, calls *get-functions* which prompts the user to enter them.

Following this, it then checks the reset function, as this can be left empty by the user. If this is the case, the algorithm inverts the set function to use as the reset function, as performed on line 8.

When the set and reset functions are obtained, a list of all variables in both functions is created, resulting in a, b and e. *get-output-var* obtains the output variable name, which may be input by a user when calling the function, or be a default of out. For this example we will set it as z.

Next, *bool-to-concept* will convert the functions provided to CNF, using Algorithm 1. The resulting CNF functions, simplified will be as follows:

$$setCNF : \overline{a} \wedge \overline{b} \wedge e$$
$$resetCNF : (a \vee b) \wedge e$$

Now, we can begin to generate the text of concepts, using the function *generate-concept-text*. This is used in order to generate both the concept for the output to rise and to fall, and as such, passed into it is a function, either set or reset, and the output transition, either `rise z` or `fall z`.

generate-concept-text begins with a loop, working with each sub-function in the given function. A sub-function in this case is found between each top-level AND. Using setCNF this leaves us with three sub-functions, each a single variable; $\overline{a}$, $\overline{b}$ and e. For each of these, it will determine whether the variable is negated or, using the `fall` function to indicate a falling transition if so, or a `rise` function, indicating a rising transition, if not. The text of the transition is created and stored in a list fo each sub-function.

Once all variables in the sub-function have been used to generate a transition, we then create the concept in text-form for this, using OR-causality to the given effect transition of the output variable. Each sub-functions causes are composed with one another as expected for the specification.

Line 13 in *bool-to-concept* will reuse *generate-concepts-text* for *resetCNF*, creating a concept with the effect transition of `fall z`instead. Note, that we use OR-causality exclusively in this algorithm. For the purposes of automatic concept generation, we use only this form of causality, as if a sub-function contains only one variable, then this can be described as the only possible cause of the effect signal, having the same effect as AND causality, and we will refer to these cases as AND-causality in this example

Following this, we have created the concepts of *outputRise* and *outputFall*, which will be as follows:

```
outputSet = fall a ~> rise z <> fall b ~> rise z <> rise e ~> rise z
  outputReset = [rise a, rise b] ~|~> fall z <> rise e ~> fall z
```

With these, the behaviours are described, and a concept specification can be generated, and printed to the desired destination. We pass the necessary variables to the *print-concepts* function which will print a concept specification in the form as seen throughout this thesis, seen in Figure 5.8.

Note that when generating these concepts, for initial state concepts we assume that all signals in the system will be initially 0, and for the interface concepts, it is assumed the given output variable is the only output, and all variables in the functions are inputs. This specification is now complete, Figure 5.8 displayss the resulting file.

```
module Concept where

import CircuitConcepts

circuit a b c z = outputRise <> outputFall
               <> initState <> interface
  where
    outputRise = fall a ~> rise z <> fall b ~> rise z
              <> fall c ~> rise z
    outputFall = [rise a, rise b, rise c] ~|~> fall z
    initState  = initialise0 [a, b, c, z]
    interface  = inputs [a, b, c] <> outputs [z]
```

Figure 5.8: 3-input NOR gate concept file generated by Bool-to-Concepts

The assumptions of interface and initial state are not accurate, as from the set and reset functions alone it is not possible to derive what the interface and initial states of the given system are. For the purposes of automation, we include these default interface and initial state concepts in order to generate a specification which is immediately translatable, as Figure 5.8 is. In this way, a user can translate the specification, and discover this information through simulation of the circuit, or if the interface is known for example, can edit the generated concept specification themselves. This process is still quicker than that of manually deriving the concepts of a circuit for which only the set and reset functions are known.

5.2 Workcraft

Workcraft [17][18] is a software suite which provides a graphical user interface (GUI) for multiple graph-based modelling formalisms, including FSMs, state graphs, PNs and STGs. These graphs can be visualised and manipulated in a way which manually would be a lengthy and difficult process. It also features several algorithms and tools as a back end which automatically perform operations with these graph formalisms.

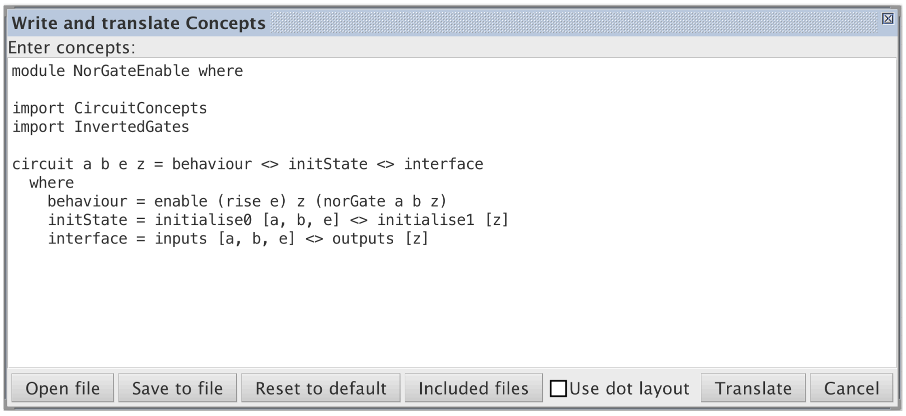
The aim of Workcraft is to add an ease of use to existing design methods, and automate many processes to also speed up designs. A graph can be designed from the ground up, adding and removing vertices and arcs, or be generated using a tool, or converted from another graph formalism. Graphs can be simulated to show the movement of tokens such as with PNs and STGs, and the traces from this be used to generate timing diagrams. A graph can be verified automatically at the click of a button, and in some cases, verification problems can be solved automatically. A graph can then be synthesized, and the resulting implementation be manipulated and synthesized from within Workcraft.

In this section, we will discuss Plato (Section 5.1) and PGminer (Section 5.2.2), and how they are integrated into Workcraft, for the Asynchronous Concepts design flow, which provides a GUI for this. We also discuss the processes of verifying and synthesizing an STG, and the tools which perform this.

5.2.1 Plato integration

Plato can currently be used to generate both STGs and state graphs, and thus, the plug in for both of these graphs features an option to generate them from Asynchronous Concepts. Both feature a "*Conversion*" menu, which contains a "*Translate concepts...*" option. Selecting this option provides a dialogue for the authoring of concepts, as seen in Figure 5.9. Note, this dialogue contains a different concept specification for the NOR gate with enable, importing the NOR gate from `InvertedGates` (Figure 4.4). This is used to show features of Workcraft integration.

This provides some options for authoring Asynchronous Concepts. Either, a specification can be written within this dialogue and be translated, without the need to save the file, or it may be saved for use later. An existing concepts file and be opened, edited, translated or saved. There are also options to reset the dialogue to a basic template for

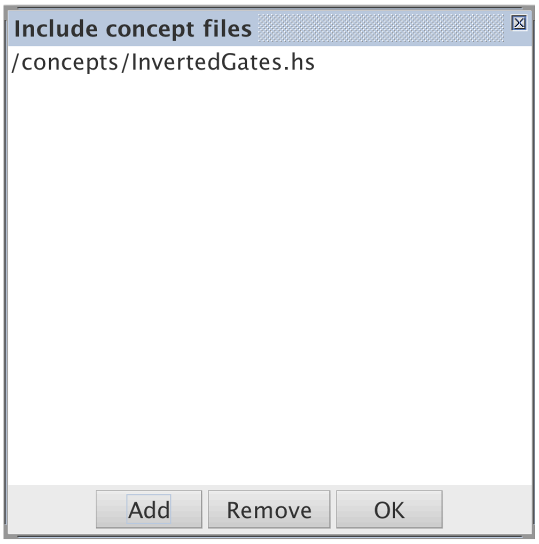


```
module NorGateEnable where

import CircuitConcepts
import InvertedGates

circuit a b e z = behaviour <> initState <> interface
  where
    behaviour = enable (rise e) z (norGate a b z)
    initState = initialise0 [a, b, e] <> initialise1 [z]
    interface = inputs [a, b, e] <> outputs [z]
```

Figure 5.9: Plato dialogue for authoring concepts

a concept specification, and to select a layout preference for the translated STG, either the loop notation, or use a dot description to set the layout using the tool *Graphviz* [44], another backend tool utilized by Workcraft.

A button labelled "*Included files*" is part of this dialogue, and when pressed, opens another dialogue, which is used to select files which must be included in the translation, such as those that are imported in the concept specification. As seen in Figure 5.10, we have included the `InvertedGates` file, which contains the `norGateN` concept.



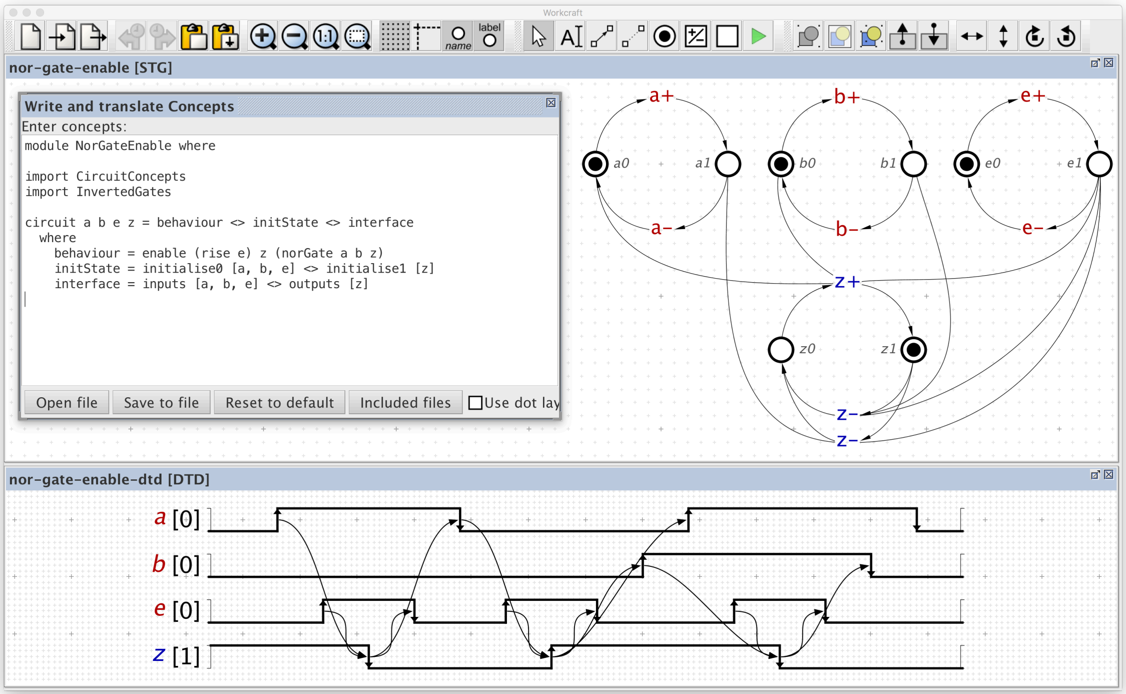
/concepts/InvertedGates.hs

Figure 5.10: Plato include dialogue in Workcraft

In the Plato preferences of Workcraft, concept files which are used regularly may be listed, in order to be included in every translation from concepts to either STGs or state graphs. This feature is intended for user-generated concept libraries, which may be often used, avoiding the need to ensure they are included in every translation.

Following the writing of a concept specification, one may choose to save the file, and then translate this. This will then automatically pass the necessary files, the concept specification and all included files, to Plato, which will generate an STG or state graph. The resulting graph will then be automatically imported to Workcraft, and laid out using the chosen methods, as seen on the top right of Figure 5.11.

The generated graph is then be simulated to ensure it works as expected. The traces from a simulation may be used to generate a timing diagram, such as at the bottom of Figure 5.11.



```
module NorGateEnable where

import CircuitConcepts
import InvertedGates

circuit a b e z = behaviour <> initState <> interface
  where
    behaviour = enable (rise e) z (norGate a b z)
    initState = initialise0 [a, b, e] <> initialise1 [z]
    interface = inputs [a, b, e] <> outputs [z]
```

Figure 5.11: Completed translation, and timing diagram of a concept specification

The generated STG can then be used in further operations such as verification and synthesis, which will be discussed further in Section 5.2.3.

5.2.2 PGminer integration

The general operation of process mining has been discussed in Section 4.3. This tool takes in an event log, and outputs Boolean functions, which are then be used to generate concepts using the *Bool-to-Concept* function of Plato (Section 5.1.3).

An event log is produced by simulating a circuit, or specification. This can be performed in Workcraft, producing an event log. This contains the events which cause the output to transition, such as the transitions which caused an output to transition high. One simulation of what cause transitions cause an effect transition is known as a trace, and a collection of traces forms an event log. We split up traces which cause an effect transition to rise and fall.

PGminer features its own plug in within Workcraft. This tool imports event logs and it process mines these to indicate any concurrency. There is a menu for "*Process mining*", which offers two options, one to extract the concurrency of the selected graphs in the current editor, or to import an event log. The second option opens the dialogue as shown in Figure 5.12.

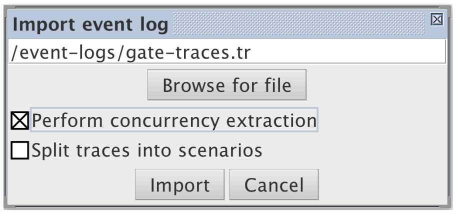

Import event log
/event-logs/gate-traces.tr
Browse for file
☒ Perform concurrency extraction
☐ Split traces into scenarios
Import Cancel

Figure 5.12: PGminer import dialogue

This dialogue allows a user to select an event log file from their file system, to extract the concurrency of this. In this case, we are extracting the concurrency of the set of traces from simulations of a circuit. "*Perform concurrency extraction*" being checked means that PGminer will extract the concurrency from this event log and import the results. "*Split traces into scenarios*" will, when importing the event log, name events which repeat in a single trace as different events, which helps to identify the number of times a single event occurs as part of the system operations.

Importing this file will pass the event log to PGminer, which will then perform concurrency extraction, and generate Boolean functions which represent the states of the signals required to cause the output transitions to rise and fall. These can then be passed into Plato, for Bool-to-concept to generate a concept specification for the given system.

5.2.3 Verification and synthesis tools

When we have produced a specification in Asynchronous Concept form, and translated it to an STG, before we can use it to find an implementation, it needs to be verified that the STG satisfies certain properties. Workcraft features back end tools to perform these automatically, using any STG, including those automatically translated from a concept specification. Note, state graphs can be used for simulation and some tools can perform synthesis with them but due to state explosion this can be a lengthy process, thus we will discuss STGs in relation to verification and synthesis.

To display verification and synthesis, we will use the example of a simple buck controller. The operation and behaviours of this circuit will be discussed in Chapter 6, but is a good example for showing how verification and synthesis are be performed within Workcraft. The STG for this circuit is seen in Figure 5.13.

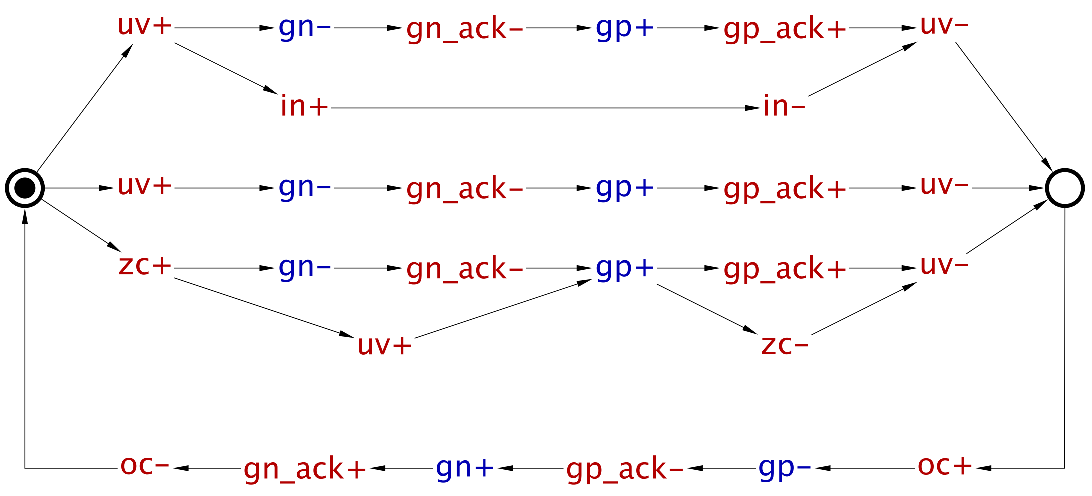


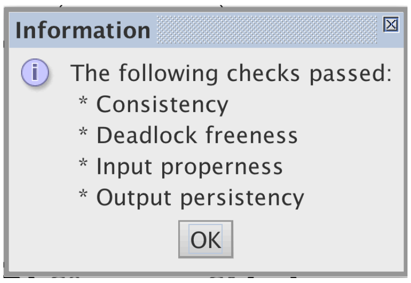
Figure 5.13: STG specification of a simple buck controller

These automatic verification and synthesis tools that Workcraft uses are Petrify [19] and MPSAT [20] [21]. They take an STG in '*.g*' format, and perform their operations. First of all we will discuss the process of verification.

The STG plugin of Workcraft features a "*Verification*" menu, from which the STG in the editor at that time can be verified automatically, determining whether the STG satisfies some properties, including consistency, deadlock freeness, input properness, output persistency and complete state coding. Workcraft will either state that the properties have been satisfied, or that there is a problem, and provide a trace which identifies how the property has not been satisfied.

Verifying the STG of the simple buck for all properties provides the dialogue as shown in Figure 5.14, indicating that all properties are satisfied.

MPSAT also provides the option to verify custom properties. This is used by Plato when a translated STG is imported to Workcraft. If a `never` concept is used, Plato passes this


Information

The following checks passed:
 * Consistency
 * Deadlock freeness
 * Input properness
 * Output persistency

Figure 5.14: Workcraft dialogue showing multiple satisfied verification properties

information to Workcraft, which generates a custom property to make sure that a the `never` concept is satisfied. For example, if the concept `never [rise x, rise y]` is included in a concept specification, this is used to generate a custom property, as shown in Figure 5.15.

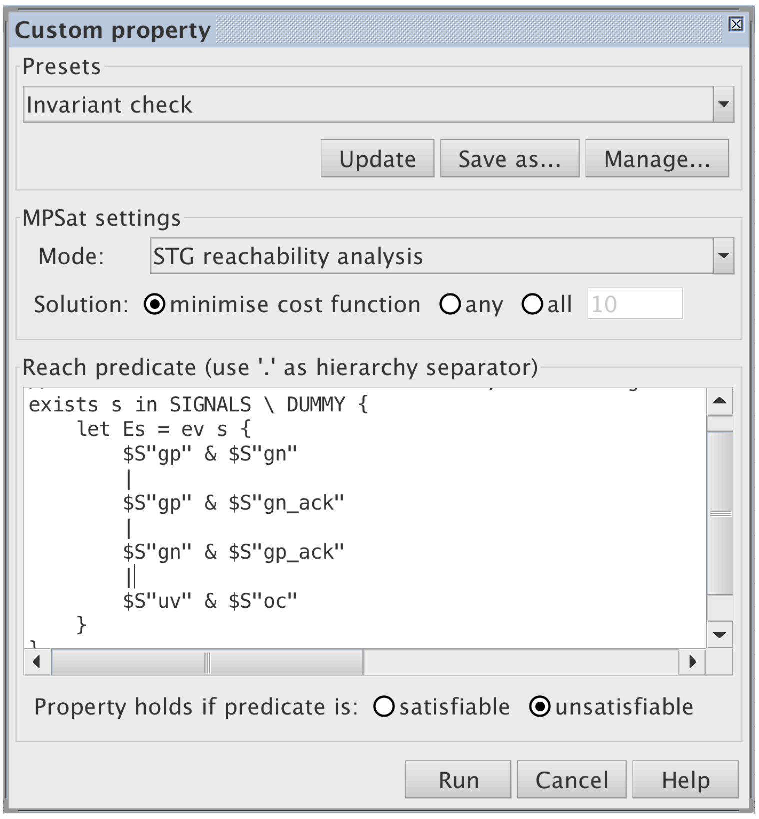
With the example of the STG in Figure 5.13, we need to also ensure that it is not possible for signals `gp` and `gn` to both be high at the same time, as they are mutually exclusive and switch PMOS and NMOS transistors respectively, and both being active at the same time will lead to a short circuit. As well as this, we should also ensure that `gn_ack` and `gp` are never high at the same time, and `gp_ack` and `gn` are never high at the same time, as these acknowledge signals indicate whether the corresponding transistor is active, so this can also be used to ensure that no short circuit occurs.

It must also be ensured that signals `uv` and `oc`, which are also mutually exclusive, are never high at the same time. We verify this with a custom property, as shown in Figure 5.15.

Running MPSAT and verifying this custom property for the given STG will provide a dialogue indicating whether the property is satisfied or not. For this STG `gp` and `gn` must never be high at the same time, and `uv` and `oc` must never be high at the same time, thus the mutual exclusions for these signals hold. The dialogue stating that this is satisfied is seen in Figure 5.16.

This feature may be used for many different types of custom property for a user to ensure that their own set of properties are satisfied.

If at any point, a user finds a problem with the STG in question, this can be fixed either at the concept specification or the STG level. Once all properties are satisfied however, an implementation will then be found through synthesis. Both Petrify and MPSAT automatically synthesize these, and through Workcraft, the Boolean functions


Custom property

Presets

Invariant check

Update | Save as... | Manage...

MPSat settings

Mode: STG reachability analysis

Solution: [✓ minimise cost function] any all 10

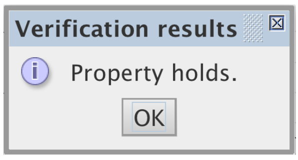
Reach predicate (use '.' as hierarchy separator)

```
exists s in SIGNALS \ DUMMY {
    let Es = ev s {
        $S"gp" & $S"gn"
        |
        $S"gp" & $S"gn_ack"
        |
        $S"gn" & $S"gp_ack"
        ||
        $S"uv" & $S"oc"
    }
```

Property holds if predicate is: satisfiable [✓ unsatisfiable]

Run | Cancel | Help

Figure 5.15: Custom property dialogue for MPSAT


Verification results

Property holds.

OK

Figure 5.16: Dialogue indicating the custom property is satisfied

that these generate for the implementation will be imported to Workcraft.

As discussed in Section 4.8, Workcraft uses these functions, and compares them against a library of gates in technology mapping. These libraries can be used to limit the set of possible gates usable by an implementation, and Workcraft features a library for these. Following the matching of functions to the gates, Workcraft will then produce an implementation, and visualise it in the circuit plug in.

The simple buck controller circuit will now be synthesized, as all verification properties, including our own check that mutual exclusions for some signals, are satisfied. We will synthesize this circuit using technology-mapping, and the resulting circuit is seen in Figure 5.17

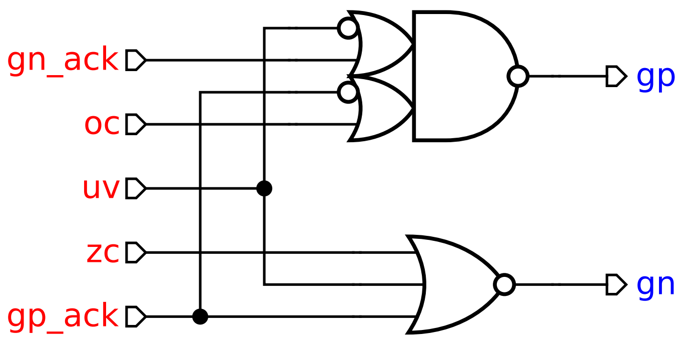


Figure 5.17: Synthesized circuit from the STG seen in Figure 5.13

5.3 Summary

Discussed in this chapter are several software tools. These tools serve to automate much of the design flow, discussed in Chapter 4. Automation of this can allow for a quicker design flow, as well as reducing the chance of error that can occur through performing many of these processes manually.

Plato is major contribution discussed in this chapter. This implements the domain-specific language of Asynchronous Concepts, and provides several functions for the design flow. It will compile a given concept specification, importing any concepts which a user has stored in a library, and aim to find any syntax or concept errors. It can then automatically generate an equivalent STG or state graph for the concept specification.

Plato can also be used to automatically generate concepts from Boolean functions. This can be from functions provided within a concept specification using `complexGate` or `function`, which will generate concepts and simply compose them with the rest of the specification. Alternatively, an entire concept specification can be generated from given Boolean functions, which can then be edited, built-upon, or imported into other concept specifications.

PGminer is a tool used to generate concept specifications from event logs, which are generated from the simulations of an existing state graph. This automatically provides Boolean functions which can be passed to Plato, to generate a concept specification.

Workcraft is a software suite, supporting several modelling formalisms, including those mentioned within this thesis. Plato, PGminer and some verification and synthesis tools are all integrated into Workcraft, providing the means to use one piece of software for an entire design using the Asynchronous Concepts design flow. It provides a GUI, allowing a user to author concepts, automatically translate to an STG or state graph, which is imported into Workcraft. From here, the model can be verified and simulated with a few mouse-clicks, avoiding the need to fiddle with command line tools.

The combination of these tools allows the design flow to be carried out more easily, automating some processes which can be time consuming. Workcraft provides a cleaner interface to be able to use all of these tools, one after the other, without any issues of incompatibility between these tools. This all aids in providing a simpler and quicker design flow.

Chapter 6

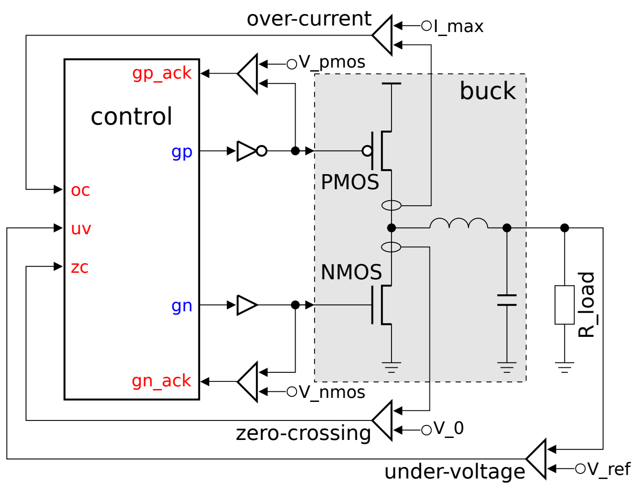
Case Studies

So far in this thesis, we have introduced a library of circuit specific concepts which can be used to design many asynchronous circuits. The design flow of Asynchronous Concepts has been discussed, explaining how to start a design from a blank page, and through to producing an implementation. We have also explained the tools which support concepts and the design flow, automating much of the design process.

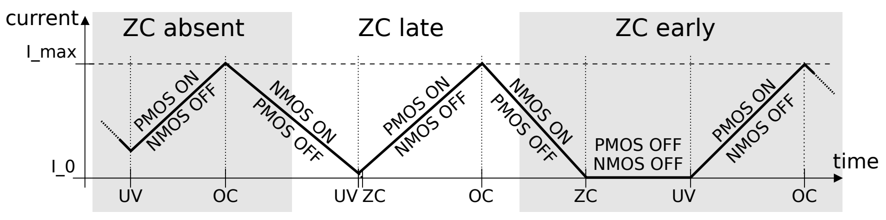
In this chapter, we will use all of this information to follow two case studies. These examples come from the power management domain [14] [1], specifically on-chip voltage regulators. The first example is quite simple, and the second is much more complex. We will briefly introduce the systems we are designing in each of these examples, and follow how one can design these using the Asynchronous Concepts design flow, creating a concept specification which uses concepts from a user generated library. We will translate these to Signal Transition Graphs, which we can simulate and verify to ensure are sound. We can then use these to synthesize for an implementation for each of these circuits.

6.1 Case Study 1: A simple buck controller

The first example is that of a *multi-scenario power regulator* [45]. A basic power regulator comprises an analogue buck and a digital controller, as shown in Figure 6.1a. The controller operates the power regulating PMOS and NMOS transistors of the buck (using `gp` and `gn` output signals) as a reaction to *under-voltage* (UV), *over-current* (OC) and *zero-crossing* (ZC) conditions (`uv`, `oc`, and `zc` inputs, respectively). These conditions are detected by a set of sensors that compare the measured current and voltage with



(a) Schematic (digital control in analogue environment).



(b) Informal description of three behavioural scenarios.

Figure 6.1: Buck converter and its informal description.

some reference values (V_ref, I_max, I_0). Note that in order to avoid a shortcircuit, the PMOS and NMOS transistors of the buck must never be ON at the same time. Therefore, the controller is explicitly notified (by gp_ack and gn_ack) when the power transistor threshold levels (V_pmos and V_nmos) are crossed.

The operation of a power regulator is usually described in an intuitive, but rather informal way, e.g. by enumerating the possible sequences of detected conditions and describing the intended reaction to these events, as shown in Figure 6.1b. The diagram shows that UV should be handled by switching the NMOS transistor OFF and PMOS transistor ON, while OC should revert their state – PMOS OFF and NMOS ON (**ZC absent** scenario). Detection of the ZC after UV does not change this behaviour (**ZC late** scenario). However, if ZC is detected before UV then both the PMOS and NMOS transistors remain OFF until the UV condition (**ZC early** scenario).

The informal specification of the buck converter defines three operating modes that require distinctive control scenarios. Requirements for each operating mode can be captured with a separate list of concepts and translated into scenario STGs. These can be subsequently combined to produce single STG for the whole circuit. During this process, it is useful to find any operations which occur between two or more operating modes, as these can then be reused.

ZC absent scenario

A circuit that handles buck operation in absence of ZC condition is specified as in Figure 6.2. The `zcAbsent` concept is a composition of buck charging and UV handling. Consider the charging function captured in the separate `chargeFunc` concept. It comprises several concepts: `interface` specifies the types of signals and `initState` defines their initial state; `gpHandshake` and `gnHandshake` specify the protocol on `gp`/`gp_ack` and `gn`/`gn_ack` interfaces respectively; `noShort` enforces a safety constraint to prevent a shortcircuit; `ocFunc` and `ocReact` define the interplay with OC condition; and `environment` captures the fact that OC and UV conditions never happen at the same time.

Note that the sequence of PMOS/NMOS switching during the charging cycle is the same for all operating modes, and therefore the `chargeFunc` concept can be reused in other scenarios. We have stated it as a separate concept in this concept file, as well as `uvFunc` and `uvReact` which describe the operations caused by UV. These can now be imported to other concept files and be used as it has in `zcAbsent`.

The `zcAbsent` concept can now automatically be translated to an STG for use with simulation, verification and synthesis tools, and for combination with the other scenarios. We have resynthesized this STG into a form which is more compact as shown in Figure 6.3 using the MPSAT tool [32]. The STG translated from the specification while correct is large and complex, so for clarity we include a more compact version.

```
module ZCAbsent(uvFunc, uvReact, chargeFunc) where
import CircuitConcepts

zcAbsent uv oc zc gp gp_ack gn gn_ack
    = uvFunc uv gp gn <> uvReact gp_ack gn_ack uv
   <> chargeFunc uv oc zc gp gp_ack gn gn_ack

uvFunc uv gp gn = rise uv ~> rise gp <> rise uv ~> fall gn

uvReact gp_ack gn_ack uv = rise gp_ack ~> fall uv
                        <> fall gn_ack ~> fall uv

chargeFunc uv oc zc gp gp_ack gn gn_ack
    = interface <> initState <> ocFunc <> ocReact
   <> environment <> noShort <> gpHandshake <> gnHandshake
  where
    interface   = inputs [uv, oc, zc, gp_ack, gn_ack]
               <> outputs [gp, gn]
    initState   = initialise0 [uv, oc, zc, gp, gp_ack]
               <> initialise1 [gn, gn_ack]
    ocFunc      = rise oc ~> fall gp <> rise oc ~> rise gn
    ocReact     = fall gp_ack ~> fall oc <> rise gn_ack ~> fall oc
    environment = mutex uv oc
    noShort     = mutex gn gp <> fall gn_ack ~> rise gp
               <> fall gp_ack ~> rise gn
    gpHandshake = handshake gp gp_ack
    gnHandshake = handshake gn gn_ack
```

Figure 6.2: Concept specification for the ZC absent scenario

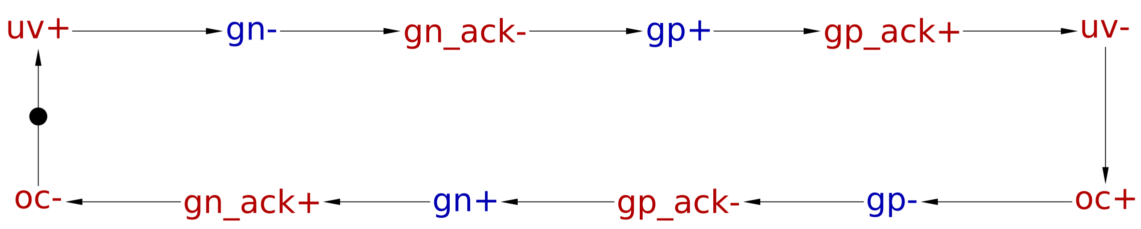


Figure 6.3: STG translated from the zcAbsent concept and resynthesized

ZC late scenario

If ZC condition is detected after UV, then buck operation is the same as in absence of ZC, i.e. ZC conditions can be ignored. This is captured by an additional zcLate concept, as seen in Figure 6.4

```
module ZCLate where
import CircuitConcepts
import ZCAbsent(uvFunc, uvReact, chargeFunc)


zcLate uv oc zc gp gp_ack gn gn_ack
    = zcLateFunc <> uvFunc uv gp gn <> uvReact gp_ack gn_ack gp
   <> chargeFunc uv oc zc gp gp_ack gn gn_ack
  where
    zcLateFunc  = rise uv ~> rise zc <> fall zc ~> fall uv
```

Figure 6.4: Concept specification for the ZC late scenario

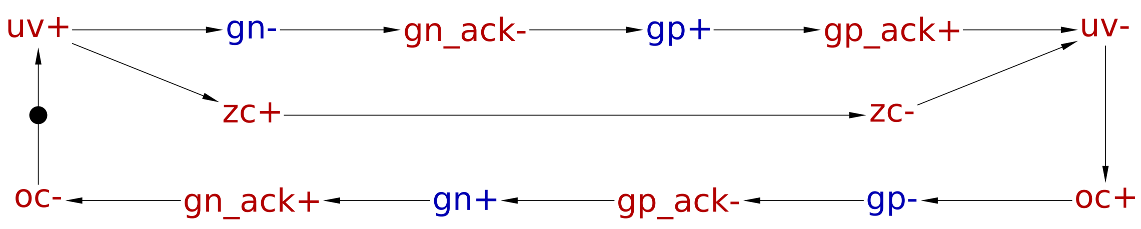


Figure 6.5: STG translated from the zcLate concept and resynthesized

The STG specification automatically produced from the zcLateScenario concept is shown in Figure 6.5. It looks similar to the STG in Figure 6.3 but features a concurrent branch for the zc signal. Note that the arc `rise zc ~> fall zc` is implied at translation time by the consistency provided by signal transition loops as discussed in Section 5.1.1.

Note that the concept specification of zcLate reuses most of the code from the zcAbsent concept, and imports this for the containing file using `import ZCAbsent`. This includes uvFunc and uvReact, as the operations that occur after UV signals does not change, and chargeFunc as the environment, safety constraints and operation after OC signals does not change. This removes the need to specify this again, as would be necessary for the monolithic STG approach. This makes the specification for this scenario much shorter.

Workcraft allows to copy and paste STGs, which mitigates the problem, but duplication and associated design problems remain [46] – for example, if the analogue environment needs to be amended, these changes need to be done consistently in all scenarios. With concepts, only one definition needs to be changed, which increases the productivity of the designer.

ZC early scenario

If ZC condition is detected before UV then it needs to be explicitly handled by the control circuit. This is specified with the concepts as seen in Figure 6.6.

```
module ZCEarly where
import CircuitConcepts
import ZCAbsent(chargeFunc)

zcEarly uv oc zc gp gp_ack gn gn_ack
    = zcFunc <> zcReact <> uvFunc' <> uvReact'
   <> chargeFunc uv oc zc gp gp_ack gn gn_ack
  where
    zcFunc    = rise zc ~> fall gn
    zcReact   = fall oc ~> rise zc <> rise gp ~> fall zc
    uvFunc'   = rise uv ~> rise gp
    uvReact'  = rise zc ~> rise uv <> fall zc ~> fall uv
            <> rise gp_ack ~> fall uv
```

Figure 6.6: Concept specification for the ZC early scenario

The obtained STG for the zcEarly concept is shown in Figure 6.7. Note this is similar in shape to the previous STGs, but ZC is the first condition to be signalled and thus, the gn and gn_ack signals go low, and both transistors are turned OFF, until UV eventually signals, where gp and therefore gp_ack will be set high, and zc will go low before uv goes low.

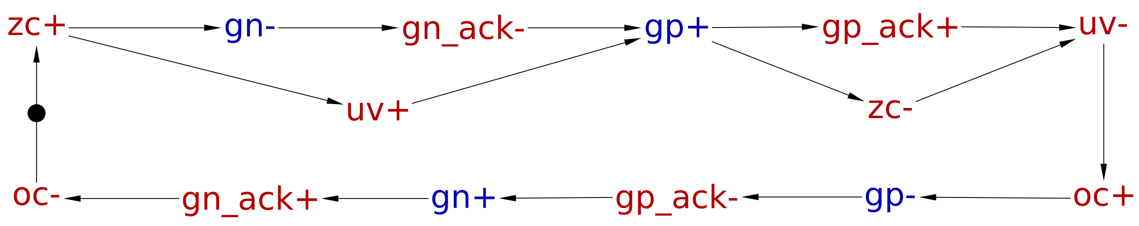


Figure 6.7: STG translated from the zcEarly concept and resynthesized

A lot of code was once again reused from the imported ZCAbsent module, namely the charegFunc concept. As with the zcLate concept, some of the main behaviours of the zcEarly concept are the same, thus it is not necessary to specify these behaviours again, providing a more compact concept specification.

6.1.1 Combining the scenarios

The scenario STGs can now be combined as described in Section 4.6 to produce a multi-scenario specification. As buck operating modes are active one at a time, the control scenarios are combined using the non-deterministic choice template.

Figure 6.8 is the combined STG. It contains all three scenarios as translated and seen in Figures 6.3, 6.5 and 6.7. However, this now leads to some redundant arcs, such as those connecting oc- and uv+, and in the ZC early scenario, the arc connecting oc- to zc+. These can be safely removed, producing the STG as seen in Figure 6.9.

Figure 6.8 shows the resulting STG specification. This can be further simplified manually by merging the common parts of the scenarios, thus producing a more compact model similar to that shown in Figure 6.10.

There is an explicit place in this model which holds a token initially and allows any of the scenarios to run. This free-choice place has no control over which scenario can run, and only allows one of them to run at a time.

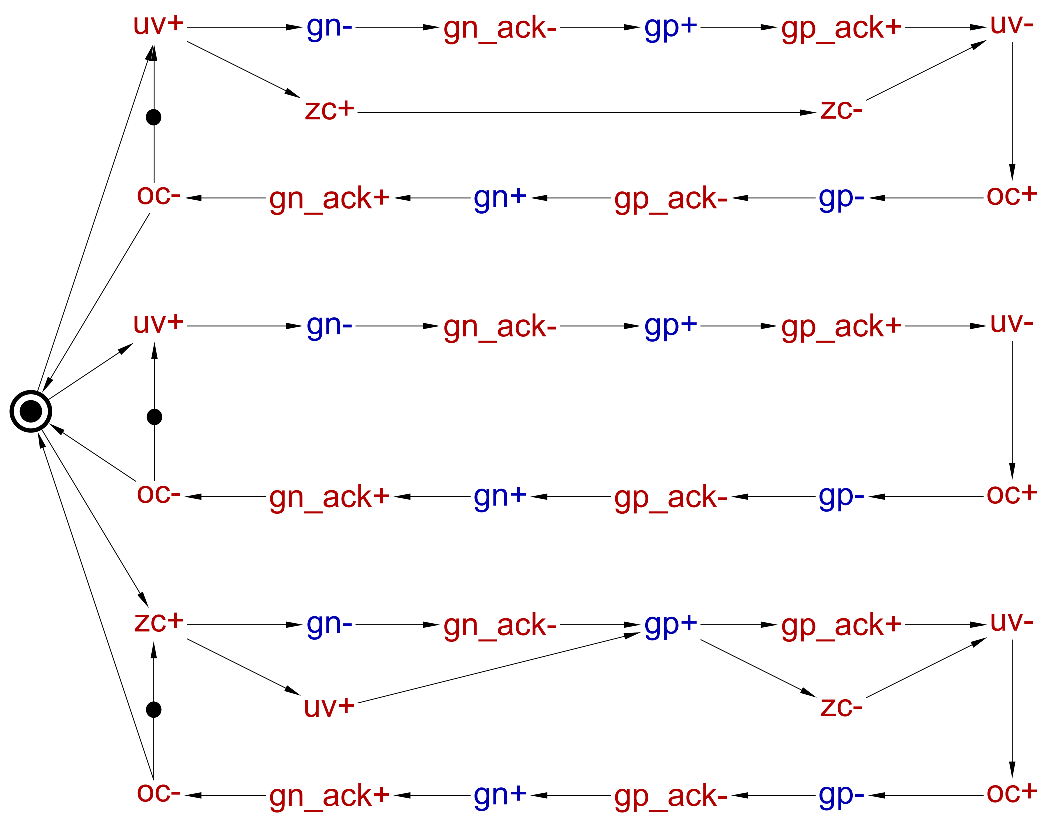


Figure 6.8: Combined STG for a buck converter, featuring all three scenarios

6.1.2 Simulation and verification

We can now simulate this STG in Workcraft, and test that it works as expected, and attempt to find any possible errors. Some points of error to look for during the simulation of this STG are:

- Unexpected enabled transitions - Any transitions which are enabled at in a state that it should not be.
- Each scenario is separate from each other - Each scenario is separate, and each is its own *process window* [40]. This means that each can be extracted from the full system STG.
- The STG will correctly return to the choice state - Once a scenario has run, the STG will correctly return to a state where any of the scenarios can run once again.

Assuming the STG simulations were successful, as discussed in Section 4.7, before we can synthesize a specification to find an implementation, we need to verify the STG to ensure

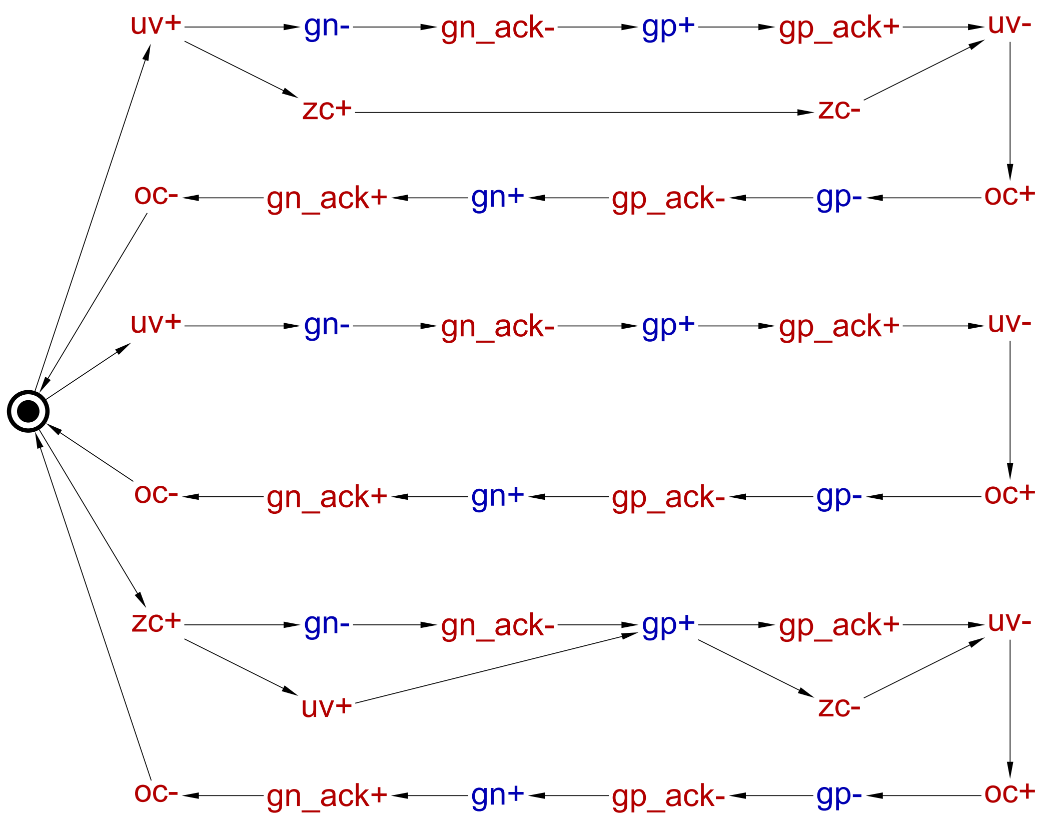


Figure 6.9: STG for a buck converter, with some redundant arcs removed

it satisfies implementability properties (Section 4.7). We can use MPSAT [20] [21] to verify the STG automatically.

Due to the safety constrains and the environment stated in the concept specifications of these scenarios, the translation process will also automatically run custom verifications. These ensure that the UV and OC conditions will never signal at the same time, as stated by the `mutex uv oc` concept.

This also applies to the `noShort` concept, which ensures no short circuit can occur in the system with the concept `mutex gp gn`, indicating these two signals can never be high at the same time. However this is not sufficient, as the `gp` and `gn` signals are buffered in order to switch the transistors, and thus there is a delay between a transition in one of these signals, and the associated transistor switching ON or OFF. This means that, while one of these signals may transition low to switch a transistor OFF, setting the other signal high may cause both transistors to be switched ON at the same time, before the first has switched OFF. For this reason, the signals `gp_ack` and `gn_ack` exist to indicate the state of the PMOS and NMOS transistors respectively. Therefore, along with the concept of `mutex gp gn`, we also include two other concepts, `gn_ack ~> gp`,

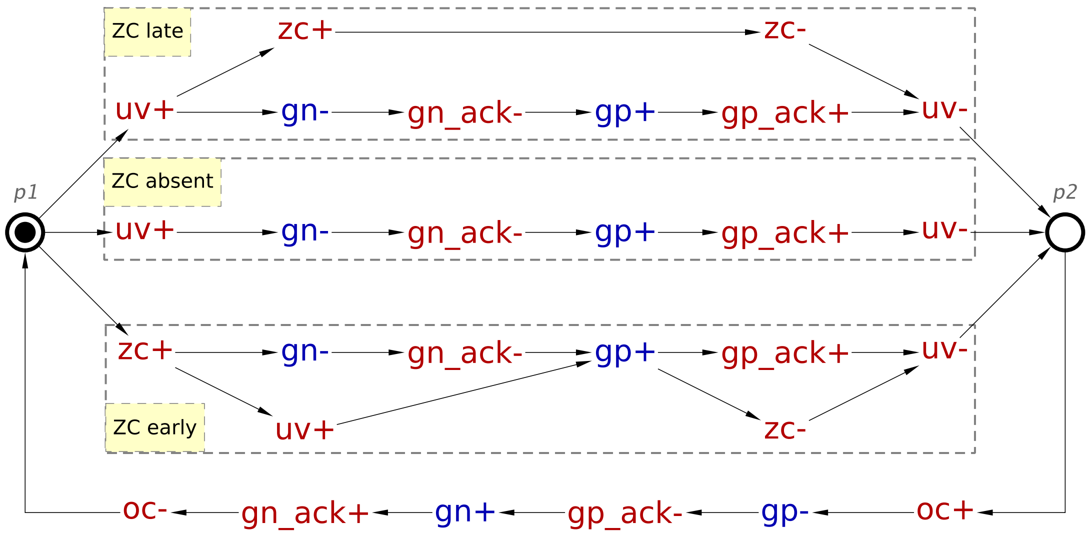


Figure 6.10: Combined STG for a buck converter, featuring all three scenarios

which stops the PMOS transistor from being switched ON until the NMOS transistor has is acknowledged to have been switched OFF, and gp_ack ~> gn, to stop the NMOS transistor from being switched ON until the PMOS transistor has been acknowledged to have been switched OFF.

In the event that the simulation testing fails, or one or more of these verification properties does not hold, the STG cannot be used for synthesis, as reported to the user. In this case a problematic concept can be fixed in a specification, and re-translated and combined with the others, and the simulation and verification process be run again. Or, the STG can be edited itself, however this means that the concept specifications will no longer be equivalent to the STG.

6.1.3 Synthesis of a speed-independent controller

The final step in this design flow is to synthesise the STG specification (Section 4.8), in this case, for a speed-independent controller. Circuits can be synthesised automatically using Petrify or MPSAT. For this example, Figure 6.11 shows the result of *Complex gate* synthesis of the buck STG.

Complex-gate synthesis does not use a gate library and yields Boolean functions of arbitrary complexity. These functions are often too large to be implemented by a single gate available in the gate library. Unfortunately, breaking up a complex-gate into smaller ones, when performed naïvely, generally yields an incorrect circuit – this happens due to the delays associated with the outputs of the newly introduced gates and can lead to

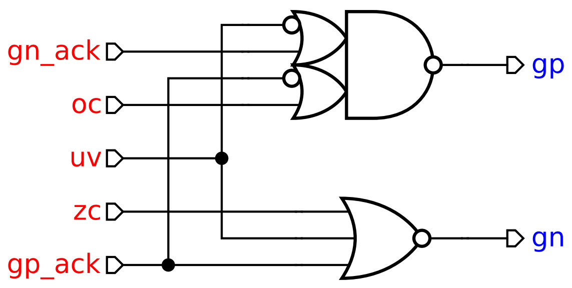


Figure 6.11: Complex gate synthesis result of full STG

glitches or deadlocks.

In fact, logic decomposition in the context of speed-independent circuits is a very difficult problem, that cannot always be solved. Petrify and MPSAT backend tools do a good job in many situations, but occasionally they fail to converge to a solution and a manual intervention by the designer is required.

We can also use Technology mapping to generate an implementation for this example. The result is found in Figure 6.12.

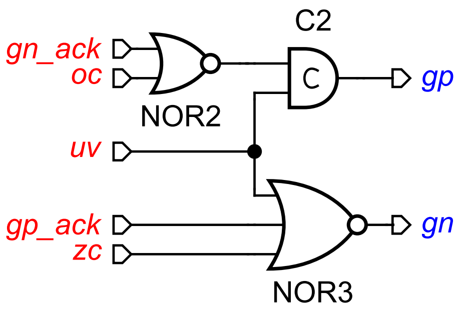


Figure 6.12: Technology mapped implementation of the simple buck controller

The gate labels correspond to the gate names in the library. More information on the synthesis of this example can be found in the tutorials of the Workcraft website [18].

6.2 Case Study 2: Multiphase buck controller

The second example is that of an *Asynchronous Multiphase Buck Controller*. Introduced in [1], this consists of multiple control circuits, each for one of multiple PMOS and NMOS transistors, each known as a phase. For reference Figure 6.13 contains the schematic of the multiphase buck converter, with the control system which we aim to design.

A single `uv` signal is input to all phases, which signals under-voltage, and causes the PMOS transistor to switch ON and the NMOS transistor OFF, as with the simple buck from Section 6.1, in the currently active phase. Each phase has it's own oc and zc signals, which indicate the over-current and zero-crossing conditions of each phase, and also switch the transistors the same as in the simple buck (OC switches PMOS OFF and NMOS ON, ZC switches both transistors OFF).

Each phase becomes active one after the other, with connecting signals passing a token indicating activity from one to the next. There is also a condition known as *high-load* (hl), indicating a sudden increase in power demand, which activates all phases at the same time, starting a charge as with the under-voltage, switching the PMOS transistor ON, and the NMOS transistor OFF.

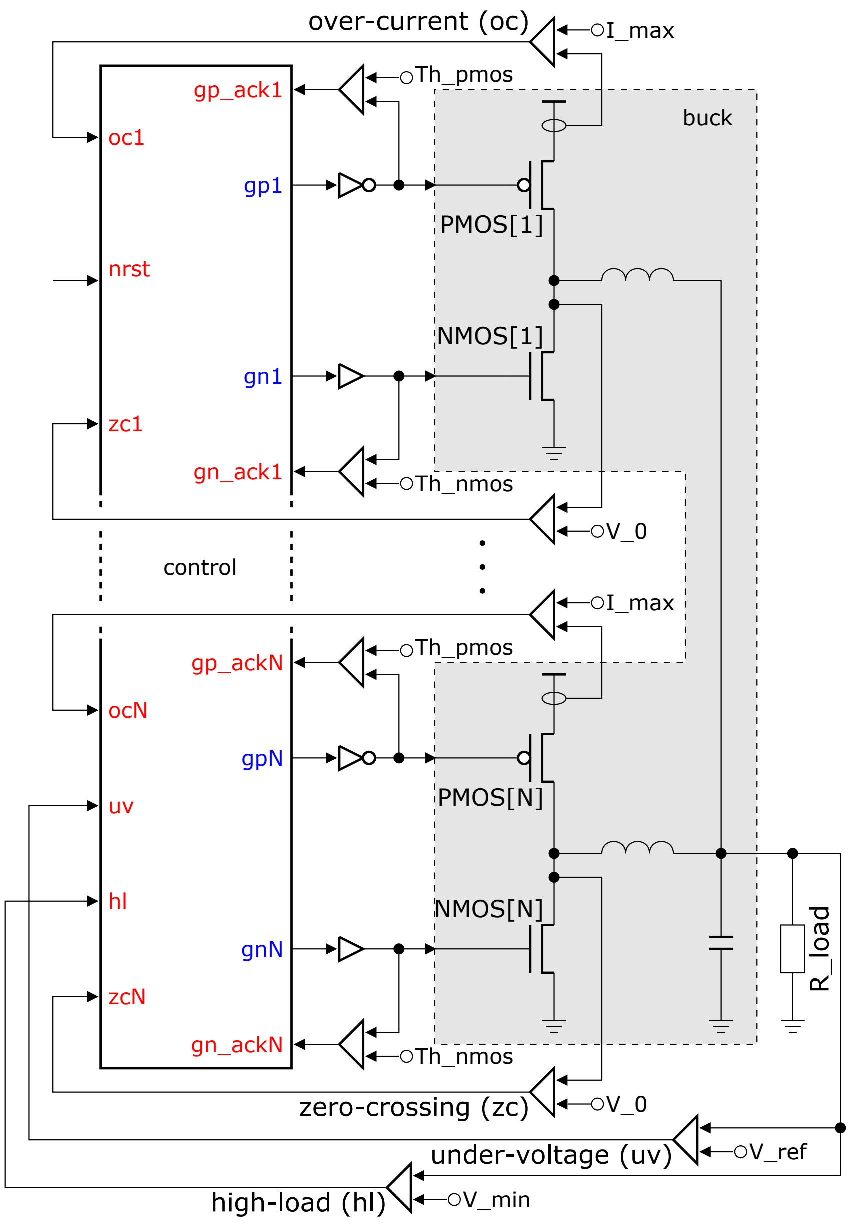


Figure 6.13: Schematic of a multiphase buck converter [1]

In this section, we will show how, using Asynchronous Concepts, we can specify the separate components of such a circuit, creating a library. We can then create concept specifications which import concepts from the library, and connect them to produce a full specification.

Note: As we are focussing on how to build a system specification from component parts in this section, for clarity we will not be showing the specification of every component. Appendix A contains concept specifications and STGs for each component not discussed in this section, and we will identify the relevant figures for each component. We will also be focussing on the authoring of this specification, as opposed to the verification and synthesis of this.

6.2.1 Single phase block diagram

We will begin by introducing a block diagram for a single phase of the controller, Figure 6.14, and briefly discuss each component. This diagram identifies how the components interconnect, useful for the full system specification.

Initially, it can be seen in this block diagram that there are two clear sections, *Activation* and *Charging*, which are connected by a handshake. We will discuss each section individually.

Activation section

- HL_WAIT (High-load WAIT) - This is a WAIT element, which will be discussed in Section 6.2.2. `hl` is a non-persistent analogue signal and thus cannot be used directly as an input to an asynchronous digital circuit. The WAIT element serves to *sanitize* this analogue signal, setting the digital signal `san` high when `hl` crosses the threshold indicating it is high, signalling the high-load condition.

- HLH (High-load handler) A.1 - As the high-load condition sets all phases to charge, this will take the `hl` signal (sanitized by HL_WAIT), and then request that this phase, and all phases, activate, and set the charging section into charge mode, switching the PMOS transistor ON and the NMOS transistor OFF.

- TC (Token control)A.2 - This takes a token from the previous phase, via `get`, which will then send a request to the merge element, which in turn will activate the charging section. The TC will also simultaneously send a request to the TOKEN_TIMER, which will ensure that the token is kept in this phase for a

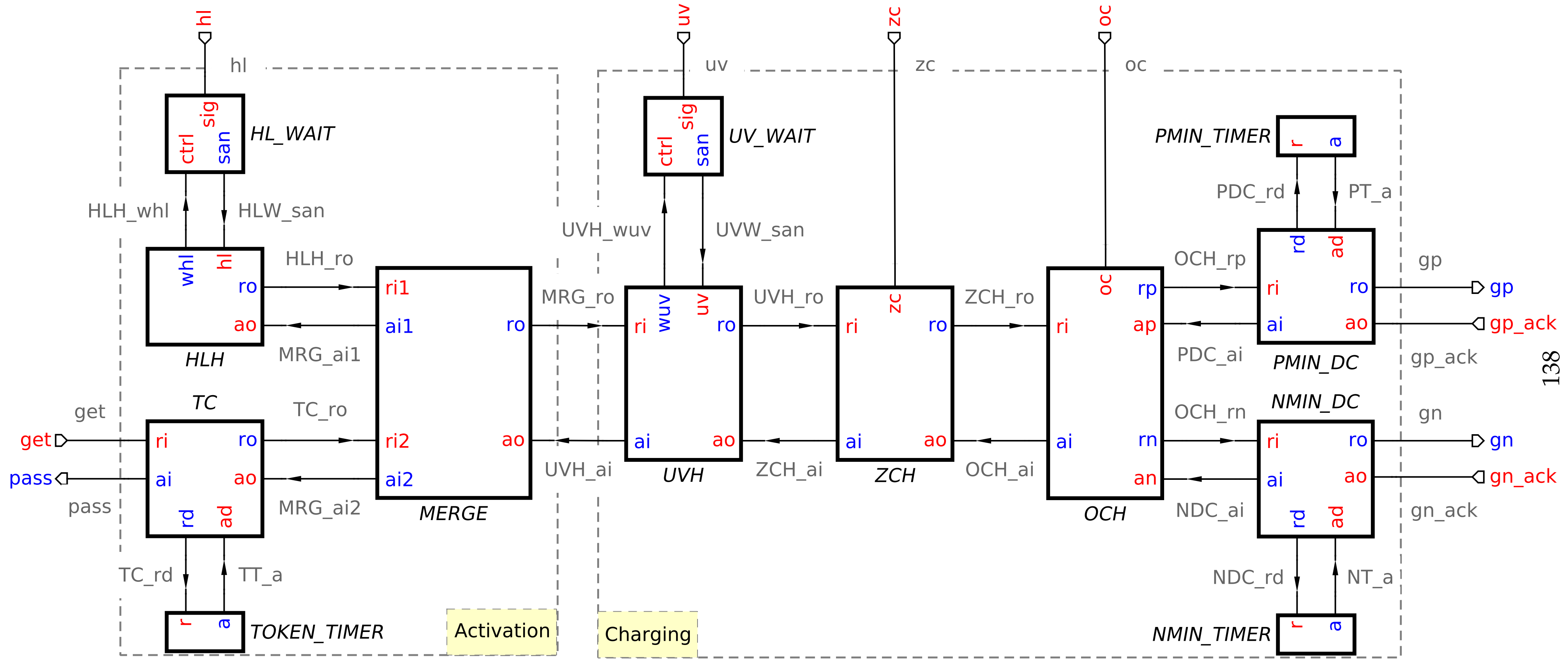


Figure 6.14: Block diagram of a single phase of the controller [1]

minimum amount of time. When the activation section is complete, the TC will receive an acknowledge, and when both this and the acknowledge from the TOKEN_TIMER are received, it will pass the token to the next phase, via the `pass` signal.

- TOKEN_TIMER (Delay) A.3 - The token timer is started as the token is received from `get` in order to delay the token being passed to the next phase via `pass`. Delays are used multiple times throughout a phase, and each operate in the same way to provide a delay.

- MERGE A.4 - This MERGE element aims to combine the request and acknowledge handshakes from HLH and TC, both of which aim to activate the phase, into one channel, which is passed to the charging section to activate it. This was, whether the circuit is in high-load condition, or the token is passed, the charging section will be activated.

The operation overview of the activation section is: Either a token will arrive at this phase, or the high-load condition will signal. Either or both of HLH or TC will send requests to MERGE, which will send a request to the charging section from one of these. When the acknowledge signal comes from the charging section, MERGE will send the acknowledge to one of HLH or TC, depending on which device sent the original request. If HLH receives the acknowledge, then this will remove the latched `hl` signal from the WAIT element, resetting it. When TC receives an acknowledge, it synchronises with the acknowledge from the delay, which indicates it has been a minimum length of time to hold the token, and then passes the token to the next phase.

Charging section

- UV_WAIT (Under-voltage WAIT) - A WAIT element (Section 6.2.2). `uv` is a non-persistant signal, similar to `hl` and must be sanitized to be used with the digital circuit. The output of this, `san` will then signal the under-voltage condition if `uv` is deemed to be high.

- UVH (Under-voltage handler) A.5 - When the activation section sends a request, this then checks the UV_WAIT for the under-voltage condition. If this is signalled, then the acknowledgement is sent back to the activation section, so this can then pass the token to the next phase (`ai`), and it sets the output request `ro` high, sending it to the ZCH module.

- ZCH (Zero-crossing handler) A.6 - `zc` may not signal, or it may signal after the request comes from UVH via `ri`. In either of these cases, the condition is ignored. `zc` may go high before any request comes from UVH however, and in this case the request, `ro` will go high, which is passed to OCH to switch both power regulating transistors OFF. When `ri` goes high, which signals that `uv` has gone high, this will then set `ro` low, to switch ON the PMOS transistor to fix the under-voltage condition.

- OCH (Over-current handler) A.7 - This component controls the switching of the transistors. The request from ZCH, via `ri` determines the switching of transistors. When `ri` goes high, the NMOS transistor is switched OFF, and when `ri` goes low, the PMOS transistor is switched ON. When the over-current condition is signalled by `oc`, then this will switch the transistors in the opposite way.

- PMIN_DC (Delay control for PMOS) A.8 - This component is used to switch the PMOS transistor ON for a minimum period of time. The request comes from OCH when it sets `rp` high, for `ri` to PMIN_DC. This will set `ro` high, to switch the PMOS transistor ON, and set `rd` high to start the delay (PMIN_TIMER). When the transistor is acknowledged as being switched ON by `gp_ack` and the timer has completed, signalled by `ad`, only then can the acknowledge be sent back to OCH.

- PMIN_TIMER (Delay) A.3 - When `rd` is set high from PMIN_DC this then will set `ad` high after a delay, the minimum period of time the PMOS transistor can be switched ON for.

- NMIN_DC (Delay control for NMOS) A.8 - This is the same as PMOS_DC, but switches the NMOS transistor according to the requests from the `rn` signal from OCH. It uses NMIN_TIMER to switch the NMOS transistor ON for a minimum period of time.

- NMIN_TIMER (Delay) A.3 - Similar to PMIN_TIMER, this will allow the NMOS transistor to be switched ON for a minimum period of time.

The operation overview of the charging section is: When a request comes from the activation section, UVH will then check for under-voltage. When this signals, it sends a request to ZCH. This can either pass the request straight to OCH, switching the NMOS transistor OFF and the PMOS transistor ON, if zero-crossing either signals after under-voltage, or does not signal at all. If it signals before under-voltage, then it sends a request to OCH to switch the NMOS transistor OFF, but waits for a request from UVH before removing the request, which switches the PMOS transistor ON.

OCH will switch the signals as requested from ZCH, but if the over-current condition is signalled, then it switches the PMOS transistor OFF and the NMOS transistor ON. OCH uses request signals `rp` to switch the PMOS transistor, and `rn` to switch the NMOS transistor. These signals are received by PMIN_DC and NMIN_DC respectively, which will switch the transistors according to the requests. If being switched ON, then a delay is used to ensure that the transistors are ON for a minimum length of time.

6.2.2 WAIT element

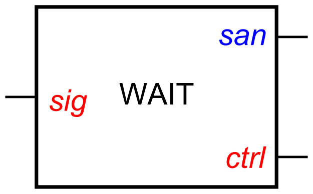


Figure 6.15: A WAIT element

A WAIT element is an *Asynchronous Arbitration Primitive* [47]. It is designed to allow a *hazardous* signal to be used as an input to an asynchronous digital circuit. Analogue signals are said to be *hazardous* to digital circuits, due to the fact that they do not have the two states, high and low, and as such a signal may be a value in the middle, and the value must be decided for use in the digital domain.

Another type of hazardous signal is one that is *non-persistent*. In the multiphase buck, the signals `hl` and `uv` are non-persistent. This is because these signals signal to every phase in the system, and these conditions can be solved by any phase. For example, when `uv` signals, the current phase will switch the PMOS transistor ON and the NMOS OFF, and the token will pass to the next phase. If the under-voltage condition stops signalling while the next phase is reacting to it and switching the transistors as necessary, then this may lead to an unspecified behaviour of the circuit.

Due to this non-persistence, the WAIT element must allow the input, `sig` to transition freely. The other signals, `ctrl` and `san` form a handshake, with `ctrl` being used to

set the device into *WAIT mode*, where it then waits for the input sig to transition high. When this occurs, this is captured by setting san high, sanitizing the non-persistent sig. At this point, the state of the input signal does not matter, and san will remain high until the ctrl signal is set low again, at which point san will transition low.
From this informal description of the operation, we can provide the concepts in Figure 6.16.

```
wait sig san ctrl = behaviour <> initState <> interface
  where
    behaviour = handshake ctrl san <> rise sig ~> rise san
    initState = initialise0 [sig, san, ctrl]
    interface = inputs [sig, ctrl] <> outputs [san]
```

Figure 6.16: Concept specification for a WAIT element

The translated STG of these concepts can be found in Figure 6.17. Note, we have resynthesized this to provide a clearer handshake. This correctly captures the behaviour, freely allowing sig to transition, but with san capturing the high transition of sig only when ctrl is high, when the system is in WAIT mode.

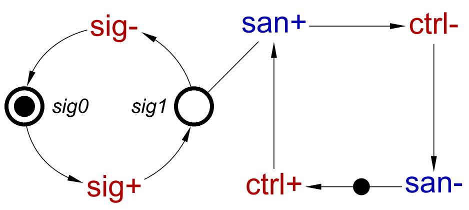


Figure 6.17: Resynthesized WAIT element STG translated from concepts

This specification unfortunately does not capture the real world operation of such an element, where it is possible for a conflict to occur when sig transitions low before san captures this. This can be seen in the synthesized circuit in Figure 6.18, where sig can transition high, which will need to propagate through the OR-AND gate in order for san to go high. Following this, san would hold the output high until ctrl was set low. However, if sig transitions low before this has propagated through the OR-AND gate, then the san would not transition high at all.

This issue, and the contrasting usefulness in sanitizing non-persistent signals in the multiphase buck controller is why the WAIT element is interesting and important.

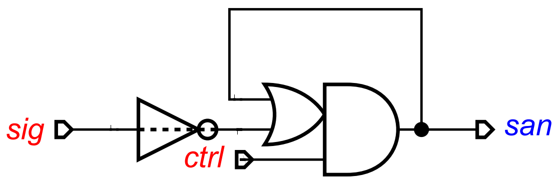


Figure 6.18: Figure 6.17 synthesized

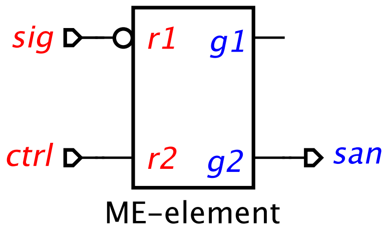


Figure 6.19: Implementation of a WAIT element using an ME element

This leads us to a more robust implementation. This implementation uses a mutual exclusion element, or ME-element (Section 3.1.2) as seen in Figure 6.19. The operation of this circuit is: While `sig` is 0, the input `r1` to the ME-element is high, which causes the `g1` to go high. When `ctrl` transitions high, setting the input `r2` high, `san` cannot go high from the ME-element output `g2` until `sig` transitions high, setting `r1` low and therefore `g1` low. This will then be held as long as `ctrl` is high, as while `san` is high, `g1` can never go high.

As the circuit-specific concept library already features an `meElement` concept, we can reuse this, producing the concept specification as seen in Figure 6.20

```
wait sig san ctrl g1 = behaviour <> initState <> interface
  where
    behaviour = bubble sig (meElement sig ctrl g1 san)
    initState = initialise0 [sig, san, ctrl, g1]
    interface = inputs [sig, ctrl] <> outputs [san] <> internals [g1]
```

Figure 6.20: WAIT element specified using an ME element

Note that in Figure 6.19 the `g1` output of the ME-element is not connected to an output pin. This output is not used, and therefore can be removed. However, in the concept specification we still include a signal `g1` which is specified as an internal signal. A limitation of concepts is that we cannot use a derived concept but ignore certain signals from it. Future development of concepts may provide a method of allowing certain behaviours of `g1` to be included, such as the mutual exclusion with `g2`, and therefore `san`, but ignore the signal itself, as it serves no purpose. For this case study, however we must include this signal. Specifying it as an internal signal means that we can still observe its behaviour in a translated STG, such as in Figure 6.21.

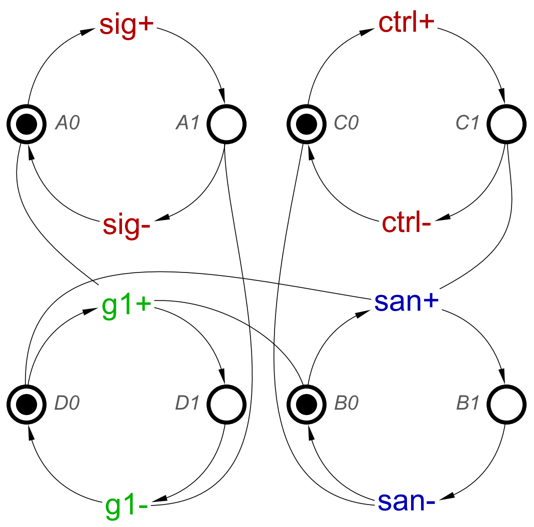


Figure 6.21: STG of a wait element implemented using an ME element

The WAIT element in the specification of the multi-phase buck will use the concept component variant which uses an ME-element, and the concepts we have provided in this section will be imported into the specification, along with the concepts featured in Section A.

6.2.3 Full specification

With concept specifications for each block in the diagram (Figure 6.14), we can now start to build a specification for the full multiphase buck controller. To begin with, we can start by deriving the specification for each of the larger sections, activation and charging.

Activation section specification

This section uses; a WAIT element, the high-load handler, the token control and a timer for the token control, and finally a Merge element. We can begin by importing these into the activation concept module, as in Figure 6.22

```
module Activation where
import CircuitConcepts
import WAIT
import HLH
import TC
import D
import Merge
```

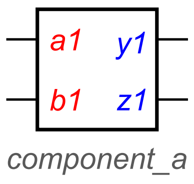
Figure 6.22: Module name and imports for the activation specification

Since the behaviours are all specified in the specific files, this specification becomes a description of how these components interconnect. To ease this process, we try to combine some of the components. For example, HL_WAIT is used simply to provide a sanitized `hl` signal to the HLH. Thus, we can produce a concept which connects these, Figure 6.23

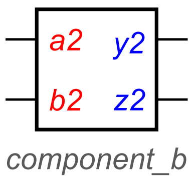
```
hlh_block hl san whl ro ao wait_i1 =
 hlwait <> hlh san whl ro ao <> interface
   where
     hlwait = wait hl san whl wait_i1
     interface = internals [san, whl]
```

Figure 6.23: `hlh_block` concept

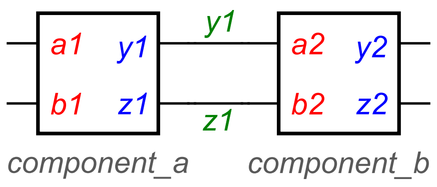
This concept will take in the `hl` signal from the environment, and sanitize it in the HL_WAIT component. This is controlled by the `whl` signal in the HLH component, and the sanitized signal is passed from HL_WAIT to HLH via signal `san`. These signals were defined as inputs and outputs to these components, but in this instance, can be declared as internal signals, as they do not come from the environment, nor are they output to the environment, they are simply used to connect these two circuits. For example, Figure 6.24 contains two components, *component_a* (6.24a) and *component_b* (6.24b). Both of these have two inputs and two outputs. If we connect the outputs of component_a to the inputs of component_b, then the outputs of component_a no longer affect the environment, and the environment no longer affects the inputs of component_b, we have control over the signals connecting these two components, and thus they are internal, as in Figure 6.24c.



(a) Component_a



(b) Component_b



(c) Internal signals connect them

Figure 6.24: Connecting components with internal signals

For this reason, we declare them as internals, which supersedes any previous interface declarations of input or internal, as discussed in Section 3.1.1. This block can now be used as part of the specification for the activations section.

We can now do the same for the token control and token timer, connecting these. The tc_block concept can be seen in Figure 6.25.

```
tc_block get pass ro ao rd ad tt_ro tt_ao tt_csc1 =
 timer <> tc get pass rd ad ro ao <> interface
   where
     timer = delay rd ad tt_ro tt_ao csc1
     interface = internals [rd, ad, tt_ro, tt_ao]
```

Figure 6.25: tc_block concept

tc_block takes in the get signal from a previous phase, and pass will output the token to the next phase when this has completed. Connecting the delay component to the token control means that the signals rd and ad are now internal signals, and as such are declared. We also declare the two other signals in the delay, named here tt_ro and tt_ao, as internals. These would normally be an output and an input respectively, but this part of the circuit is not included within the specification so we declare it as an internal. This block can also now be used in the specification for the activation section.

It may be necessary to translate the activation section on its own, to ensure that its operation is as expected. To do this, we can write a component specification, which uses the above derived hlh_block and tc_block. Due to the number of signals in this section, this will have a large number of signals in the concept declaration. Thus, we will split these up and label them. The labels are the same as those used to label the interconnections between components in Figure 6.14, however as per the requirements of Haskell syntax when using these as parameters, we us lower case characters.

```
component
 hl hlw_san wait_i1         -- HL_WAIT signals
 hlh_whl hlh_ro             -- HLH signals
 get pass tc_rd tc_ro       -- TC signals
 tt_a tt_ro tt_ao tt_csc1 -- Token timer signals
 mrg_ai1 mrg_ai2 mrg_ro mrg_i1 mrg_i2 -- Merge signals
 uvh_ai                     -- Signal from charging section
  = hlh_block hl hlw_san hlh_whl hlh_ro mrg_ai1 wait_i1
 <> tc_block get pass tc_ro mrg_ai2 tc_rd tt_a tt_ro tt_ao tt_csc1
 <> merge hlh_ro tc_ro mrg_ai1 mrg_ai2 mrg_ro uvh_ai mrg_i1 mrg_i2
```

Figure 6.26: Concept specification for the activation section

Figure 6.26 contains the activation section concept specification. While this may be complicated, it uses only three concepts in order to specify the section. The behavioural details are within the concepts, and signals which are shared between the concepts hlh_block, tc_block and merge will connect these components.

This can be translated to an STG, which can then be simulated, verified and synthesized if necessary. This STG can be viewed in Figure A.25. Due to the large number of signals, this produces a large STG, which may be harder to decipher as much information as the concepts which specify it. Many of the signals are internals, and this is due to there being many signals which are the output of one component and the input to another, and thus can be declared as internal signals.

Some internal signals also serve other purposes, such as signals used to resolve CSC conflicts, which for this case study have been solved previously. Some signals also are used as intermediate signals, such as a signal from the output of a logic gate to the input of another logic gate. This is due to a limitation of the Asynchronous Concepts language, which may be solved with future research and development.

We could have used a higher-level concept function, such as function or complexGate, however the behaviours we are trying to specify with the logic gates which use intermediate signals are not aimed at using the physical logic gates described. Instead, we are trying to capture some behaviours, which may or may not translate to the logic gates used.

Charging section specification

The components in the charging section are; a WAIT element, the under-voltage handler, the zero-crossing handler, the over-current handler, two transistor delay controllers, and two minimum time, or delays for the transistor delay controllers. We can begin this concept specification by importing other modules as seen in Figure 6.27.

```
module Charging where
import CircuitConcepts
import WAIT
import UVH
import ZCH
import OCH
import DC
import D
```

Figure 6.27: Module name and imports for the charging specification

This specification will also become a description of what components are in the circuit, and how they interconnect. We will combine the UV_WAIT and UVH components into one concept for ease, and we will also create a concept which combines a transistor delay controller and delay timer into one concept, which can then be used for both PMIN_DC and NMIN_DC.

The `uv_block` concept is specified as in Figure 6.28. This provides the WAIT element to sanitize the input signal `uv`, and this is controlled by `wuv`. It is then passed into the UVH component via the signal `san`. This means that the signals `wuv` and `san` are signals used to connect the two components, and thus are declared as internal signals.

For the transistor delay controllers, PMIN_DC and NMIN_DC we assume that the delays which are used to ensure that each transistor is switched ON for a minimum length of time are the same. Thus we can create a reusable concept, `dc_block`, which

```
uv_block uv wuv san ri ai ro ao uvw_i1 uvh_i1 =
 uv_wait <> uvh wuv san ri ai ro ao wvh_i1 <> interface
   where
     uv_wait = wait uv san wuv uvw_i1
     interface = internals [wuv, san]
```

Figure 6.28: uv_block concept

will take in the request signals from the OCH component, it will connect to a delay component, and have outputs for switching either the PMOS or NMOS transistor. We specify this block in Figure 6.29.

```
dc_block ri ai rd ad ro ao d_ro d_ao d_csc1 =
 timer <> dc ri ai rd ad ro ao <> interface
   where
     timer = delay rd ad d_ro d_ao d_csc1
     interface = internals [ri, ai, rd, ad, d_ro, d_ao]
```

Figure 6.29: dc_block concept

For either of the PMIN_DC or NMIN_DC components, it can be seen that this will work, removing the need to specify a similar concept for both. The delay is also included. We also define some internal signals which are the interconnecting signals between these two components.

We will now produce a component concept for translation of this charging section, for simulation and further use. Again, many signals are included in this specification, so we will label them using the labels provided by the block diagram, but with lower case charactes, as with the activation section. (Figure 6.14). This will use a combination of the concepts specified in this section, or concepts which are specified in Appendix A. The charging section concept specification can be viewed in Figure 6.30.

Once again this translates to produce a large STG, as there are many signals, particularly internal signals used for CSC resolution, intermediate signals, and component interconnections. However, it is not necessary that the translated STG be visualized. The STG can simply be immediately passed into another tool for verification and synthesis. The charging section STG can be found in Figure A.26.

```
component
 mrg_ro                             -- Signal from activation section
 uv uvw_san uvw_i1                  -- UV_WAIT signals
 uvh_wuv uvh_ro uvh_ai uvh_i1       -- UVH signals
 zc zch_ai zch_ro zch_nozc zch_i1   -- ZCH signals
 oc och_ai och_rp och_rn            -- OCH signals
 pdc_ai pdc_ai                      -- PMIN_DC signals
 pt_a pt_ro pt_ao pt_csc1           -- PMIN timer signals
 ndc_ai ndc_ai                      -- NMIN_DC signals
 nt_a nt_ro nt_ro nt_ao nt_csc1     -- NMIN timer signals
gp gn gp_ack gn_ack                 -- Signals to/from transistors
= uv_block uv uvh_wuv uvw_san mrg_ro uvh_ai uvh_ro zch_ai uvw_i1 uvh_i1
<> zch zc uvh_ro zch_ai zch_ro och_ai zch_nozc zch_i1
<> och oc zch_ro och_ai och_rp pdc_ai och_np ndc_ai
<> dc_block och_rp pdc_ai pdc_rd pt_a gp gp_ack pt_ro pt_ao pt_csc1
<> dc_block och_np ndc_ai ndc_rd nt_a gn gn_ack nt_ro nt_ao nt_csc1
```

Figure 6.30: Concept specification for the charging section

Complete specification

Finally, we can now produce a full specification. As shown in the block diagram (Figure 6.14) these two sections interconnect through two signals. Thus, there are several ways we can do this. For this case study, we will reuse a lot of the concepts from the activation and charging section, and produce a system specification. Initially, we must import all of the necessary modules (Figure 6.31).

As this is a system concept we must now declare the signals. This serves to make the system concept specification itself to be shorter, as the component specifications for activation and charging are quite long, and feature many signals. Note that these declared signals also all start with capital characters, unlike with the specifications for activation and charging. This is due to Haskell syntax once again, where signals that are declared in this way act like data types, which must be declared at least with the first character being a capital letter. The signals therefore begin with capital letters, but match the signal names as given in the block diagram for a single phase, Figure 6.14. The signals for this system, can be defined as seen in Figure 6.32.

```
module Concept where
import CircuitConcept
import WAIT
import HLH
import TC
import D
import Merge
import UVH
import ZCH
import OCH
import DC
import Activation
import Charging
```

Figure 6.31: Imports and module name for a single phase concept specification

We can specify the behaviour of a single phase of this multiphase buck, using previously derived concepts, as in Figure 6.33.

With the signals declared as they are, this makes for a clearer concept, only containing the concepts used to specify the behaviours. Note that in this, we have specified that the signals `MRG_ro` and `UVH_ai` are internals. This is because these signals are used to communicate between the activation and charging sections. As we are specifying the whole system, these are not needed to be viewed by the environment, so specifying them as internals supersedes any specifications of these signals as inputs or outputs.
This specification can now be translated to an STG, this can be viewed in Figure A.27.

6.3 Summary

In this chapter, we have shown how a specification can be built, using real examples. The first case study used a simple, multi-scenario buck controller. This included describing the behaviours of each scenario, using an informal description of the operation of each scenario. Between each scenario, there were common concepts, which were reused by importing the specification containing the reusable concept.

```
data Signal =
HL | HLW_san | WAIT_i1 |
HLH_whl | HLH_ro |                           -- HLH signals
Get | Pass | TC_rd | TC_ro |                -- TC signals
TT_a | TT_ro | TT_ao | TT_csc1 |            -- Token timer signals
MRG_ai1 | MRG_ai2 | MRG_ro | MRG_i1 | MRG_i2 |  -- Merge signals
UV | UVW_san | UVW_i1 |                     -- UV_WAIT signals
UVH_wuv | UVH_ro | UVH_ai | UVH_i1 |        -- UVH signals
ZC | ZCH_ai | ZCH_ro | ZCH_nozc | ZCH_i1 |  -- ZCH signals
OC | OCH_ai | OCH_rp | OCH_rn |             -- OCH signals
PDC_ai | PDC_rd |                           -- PMIN_DC signals
PT_a | PT_ro | PT_ao | PT_csc1              -- PMIN timer signals
NDC_ai | NDC_rd  |                          -- NMIN_DC signals
NT_a | NT_ro | NT_ao | NT_csc1              -- NMIN timer signals
GP | GN | GP_ack | GN_ack          -- Signals to/from transistors
deriving (Bounded, Enum, Eq)
```

Figure 6.32: Declaration of all signals in a single phase of the multiphase buck

Following this, we used the choice combination template (Section 4.6) to combine all three of these scenarios to provide a full system specification. This was then translated to an STG, and we simulated this to ensure that it works as expected. We then verified the specification, and synthesized it to produce a logic gate implementation, and synthesized with technology mapping to produce an improved implementation.

The next case study was more complex, a single phase of a multiphase buck controller. This had some similar features to the simple buck controller, with the same conditions occurring, and them being corrected in a similar fashion. However, each phase of this performs the same operation, and there can be many phases. There were two sections of this system, both of which were described.

To create a specification for this system, we used a block diagram, and specified the behaviour of each block in the Appendix. This is with the exception of the WAIT element, an interesting component which sanitizes potentially hazardous signals. We discussed this further because the concepts to describe the behaviours do not produce a particularly safe implementation, and as such, we use a mutual exclusion element to

```
system =
hlh_block HL HLW_san HLH_whl HLH_ro MRG_ai1 HLW_i1
<> tc_block Get Pass TC_ro MRG_ai2 TC_rd TT_a TT_ro TT_ao TT_csc1
<> merge HLH_ro TC_ro MRG_ro MRG_ai1 MRG_ai2 UVH_ai MRG_i1 MRG_i2
<> uv_block UV UVH_wuv UVW_san MRG_ro UVH_ai UVH_ro ZCH_ai UVW_i1 UVH_i1
<> zch ZC UVH_ro ZCH_ai ZCH_ro OCH_ai ZCH_nozc ZCH_i1
<> och OC ZCH_ro OCH_ai OCH_rp PDC_ai OCH_rn NDC_ai
<> dc_block OCH_rp PDC_ai PDC_rd PT_a GP GP_ack PT_ro PT_ao PT_csc1
<> dc_block OCH_rn NDC_ai NDC_rd NT_a GN GN_ack NT_ro NT_ao NT_csc1
<> internals [HLH_ro, MRG_ai1, TC_ro, MRG_ai2]
<> internals [ZCH_ro, OCH_ai, MRG_ro, UVH_ai]
```

Figure 6.33: Concept specification for a single phase of the multiphase buck

provide the implementation.

Following this, we generated component specifications of both sections, activation and charging, importing the component specifications from the relevant blocks, translating and displaying the result. In both cases, the STG was large and difficult to comprehend, but proves that an STG can be produced, which can then be used with verification and synthesis tools.

Finally, we created a system specification for the whole phase of the controller. This used declared signals, which in turn produced a more compact concept specification, and importing concepts reused from all of the previously specified blocks. This produced an STG, which was very large also. The size of these STGs was due to the large number of internal signals, which are used for the interconnecting signals between components, as well as for CSC resolutions, and due to limitations in the Asynchronous Concepts language, intermediate signals between gate-level concepts in specifications.

Chapter 7

Related Work

There are several methods of designing asynchronous circuits. These descriptions may represent a system in one of several ways, for example in a text form or as a graph. Each one has benefits for designing a certain type of asynchronous circuits, and therefore has several pitfalls for designing another type of circuit.

There also exists several design methodologies for breaking a system down into separate sections to be designed separately, similar to the simple buck controller example in Section 6.1. This sort of methodology we refer to as *Modular design*. This means that changes can be made to one module without affecting the functionality of others. Because each module will be smaller, finding an area to make changes will also be easier, unlike with the monolithic approach, where a large design can make finding a certain transition or read-arc fiddly.

As discussed throughout this thesis, Asynchronous Concepts are a language for describing an asynchronous circuit at various levels, from the low-level of signal interactions, to higher-level concepts which use Boolean expressions. All concepts can then be composed with other concepts to produce a specification, or simply concepts which can be reused. A specification can then be automatically translated to a form which is usable in automated verification and synthesis, making for a quick and clear design process, and allows reusability of any concepts used in a design.

In this chapter, we will discuss and compare multiple methods of asynchronous and modular design methods, including the methodology introduced in this thesis. We include Table 7.1 which contains key information based on several metrics of comparison. These metrics help us to discuss how design methods differ, and are as follow:

- **Asynchronous circuit support** - Do the methods feature support for asynchronous circuits?

- **Software tool support** - Which of these methods have some form of software which can assist in the design of a circuit? This metric is focussed on whether there is a tool which can automate the given methodology, and not on whether the tool has a GUI or is supported by multiple platforms etc.

- **Composition** - This metric shows whether a method allows for composition of smaller elements of a design, producing a larger element which can in-turn be composed.

- **Gate-level design** - Can the design methods allow for logic gates to be designed? Can these then be referenced to abstract the complexity of the gates when designing systems?

- **Signal-level design** - Is it possible to design a system based on events which occur in a system or the environment, such as signal transitions on inputs?

- **Protocol-level design** - Can the listed design methods allow for signal protocols, such as handshakes, to be described and then be used in abstract to avoid repetition of the causal relationships between these signals?

- **Design focus** - Asynchronous circuits can be 'little digital' focused, for example a control system, which interacts with and aims to control analogue circuitry using a control signals and sensors. Asynchronous circuits can also be 'big digital' focused, where they are aimed at data operations, with wires which are multiple bit widths.

Following this table we will discuss each of these methodologies in turn, comparing them to Asynchronous Concepts.

Name	Asynchronous support	Tool support	Composition	Gate -level	Signal -level	Protocol -level	Design Focus
Algebra of Parameterised Graphs	✓	✓	✓	✓	✓		Little digital
Algebra of Switching Networks	✓		✓	✓			Little digital
Balsa	✓	✓		✓		✓	Big digital
Biscotti	✓	✓		✓			Big digital
Caltech Synthesis Method	✓		✓	✓	✓	✓	Little digital
Communicating Hardware Processes	✓	✓	✓		✓	✓	Big digital
Cλash		✓		✓		✓	Big digital
Conditional Partial Order Graphs	✓	✓		✓	✓		Little digital
DI Algebra	✓		✓		✓		Little digital
Discriminators	✓		✓	✓	✓	✓	Little digital
Hierarchical Design of DI Systems	✓		✓	✓		✓	Little digital
Lava		✓		✓		✓	Big digital
Proteus	✓	✓		✓			Big digital
Resynthesis	✓	✓	✓		✓		Little digital
Snippets			✓				Little digital
Structural Design		✓					Modular
Tiempo	✓	✓		✓		✓	Big digital
Uncle	✓			✓			Big digital
Asynchronous Concepts (proposed method)	✓	✓(Plato)	✓	✓	✓	✓	Little digital

Table 7.1: A comparison of Asynchronous Concepts with similar methods

Algebra of parameterised graphs [48] has been introduced to overcome limitations of Conditional Partial Order Graphs (CPOGs), such as the lack of structural abstraction and composition methods, as well as the difficulty of formal analysis and verification. Similar to CPOGs, PGs target little digital systems and support signal and gate level modelling of asynchronous circuits. PGs have rudimentary support in Workcraft.

Algebra of Switching Networks [49] specifically addresses signal and transistor level design of little digital circuits. The key differentiating feature of this modelling approach is that both structure and behaviour of a system can be captured by the same mathematical expression and therefore both analysis and synthesis tasks can be achieved by rewriting the expression according to specific sets of rules. This modelling method supports various forms of composition, however there is currently no tool support.

The algebra of PGs, the algebra of switching networks, and Asynchronous Concepts have a similar idea, using a textual method of representing a graph. For PGs and switching networks, this becomes a form of equation, which may be simplified in this form, and become smaller than an equivalent graph, but not necessarily easier to comprehend. With concepts, the aim is to describe the operations in one of several ways, which is chosen by a designer, to help them create a specification which is easy for them to understand.

Balsa [50][51] is a design approach which features a *Register Transfer Language* (*RTL*) like language, similar to *VHDL* or *Verilog*, which aims to produce both data-driven and control circuits. This approach closely follows the process of the *Phillip's Tangram compiler* [52]. A specification written in this language is used to produce a circuit implementation in two steps. First, a Balsa program is converted into a format describing a network of handshaked components. This format can then be used for simulation, circuit diagrams and in the second step, which maps handshaked components on to library components for synthesis.

RTL languages are regularly used for synchronous design, thus a designer can adapt more easily to asynchronous design. These languages feature the ability to easily perform operations on multiple bits unlike the proposed approach, and uses programming constructs such as conditional statements for control. Specifying a control system can lead to a complicated program which can be difficult to comprehend, in comparison to concept specifications, which explicitly states signal interactions and the use of protocols. *RTL* languages do allow for reuse of modules, something we address with the proposed method, and this can speed up the design process. *Resynthesis* is commonly used for the optimisation of Balsa control circuits. However, in Balsa the set of predefined components is fixed, so a designer cannot easily introduce new components. This means Balsa does not support composition natively, as Asynchronous Concepts do.

Biscotti [53] is a C-like language which features '*forever*' blocks, in which code runs sequentially, but all blocks run concurrently to each other. This design method starts by specifying a circuit. This is then *compiled* into formats for use by various tools, such as Petri nets for verification in *Workcraft*, for optimisation and net list generation. If these stages are successful, then the circuit can be synthesized. This is designed for data-driven asynchronous systems.

Similar to *Balsa*, a C-like language can be easy to adapt to, as designers are likely to have programming knowledge. *Biscotti*, however is designed primarily for data-driven circuits, for specifying data operations which run in parallel, and communications between them. As with *Balsa* and RTL languages in general, specifying an asynchronous control system can become complex, with more signal interactions to describe, however, reuse of written code and modules, as with Balsa, Biscotti and Asynchronous Concepts, can help to produce a quicker design process.

The **Caltech Synthesis Method** [54][55][56] uses individual signal interactions, such as those for control systems, and these are specified using a regular expression style language, based on *Communicating Sequential Processes* (CSP). After these *programs* have been specified as a list of *processes*, they are then *compiled*, where a process is decomposed into a set of processes which are equivalent to the original. This occurs until all processes are in a simpler form that the compiler can continue to use. Next is *handshake expansion*, where handshaking replaces connections between each process. During this, some process orders may be changed which do not affect the operation, but may avoid issues such as deadlocks, this is known as *reshuffling*. Finally, *operator reduction* is performed to reduce the number of operators used in the new set of processes, by finding operators which can be described by other more standard operators. After this, the program will be synthesizable using a library of standard operations.

As with Asynchronous Concepts, the Caltech Synthesis Method is used to describe causalities at the level of signal transitions. Because of the CSP language, understanding a program can be complicated if there are more than a handful of signals, and while writing a specification is somewhat simpler than in that of an *RTL* language, reusing a CSP specification is not as simple, nor as simple as with the reuse of concepts and scenarios in the proposed method.

Communicating Hardware Processes (CHP) [57] is a programming language which is primarily used for designing asynchronous circuits. A program written in CHP consists of a fixed set of concurrent processes which communicate by messages. These processes are written separately, the code in each of which is usually sequential but some in-process concurrency is allowed, and a full system is produced from parallel composition of these processes. CHP processes use variables for data manipulation and for signal interactions, supporting standard programming constructs such as 'if..then' statements for example. This allows for selection of signals, useful for control systems. Processes do not share these variables however, and data is passed in messages through communication channels. There is a tool as part of CHP, called *CHPsim* which simulates CHP programs.

Similar to the Asynchronous Concepts language, CHP provides a language for specifying asynchronous circuits, different than those methods which use an RTL language, and CHP is popular for this reason. Specifying control through signal states is simpler, and data operations can be specified also. It is similar to Biscotti in that blocks of sequential code are written, and these all run concurrently, but CHP offers simpler methods of specifying the communication channels between these blocks. Reuse is therefore available in CHP, but a control system written in CHP and featuring many signals and interactions can still be difficult to specify, comprehend and debug.

Cλash is another tool which is implemented in Haskell. [58][59]. There are similarities to *Lava* and is focussed on *synchronous* data processing circuits. It has some cross-over features with Lava, such as builtin verification, and the ability for users to define functions. Cλash also features built in synthesis and simulation, avoiding the need to export VHDL, however this feature remains. This allows a simpler way of specifying synchronous circuits, which may be more natural for designers, however Haskell as a language features huge differences to programming languages like C, which may be an issue for adoption.

The implementation of Asynchronous Concepts also uses Haskell as the host language, but our focus is on asynchronous control circuits, therefore the designer only works with a very small subset of Haskell using predefined domain-specific primitives, i.e. no advanced Haskell knowledge is required. We use Haskell because it provides powerful functional programming abstractions, significantly simplifying our implementation, and allowing us to use algebraic approaches to the specification of event-interaction graph.

Conditional Partial Order Graphs (CPOGs) [37][60] target a class of systems that are comprised of multiple acyclic behavioural scenarios, such as microprocessors [61]. CPOGs are equipped with powerful scenario-level composition techniques that are automated in Workcraft. CPOGs can also be described in algebraic form (see *Algebra of* *parameterised graphs*). Structural composition of CPOGs is very limited and not automated at present. The CPOG model has been extended to model asynchronous circuits with cyclic scenarios [62] at the levels of signals and gates, however, automation in this context is also limited at present.

CPOGs, like Asynchronous Concepts, do provide the methods to describe signal interactions, and even gate behaviours, but not protocols, and the lack of composition means that a single CPOG must be used to describe a whole system, as opposed to an individual section which can be described separately and reused. Due to the conditions that determine which partial orders are active at a time, CPOGs can be used to describe the operations of devices like a processor, which depending on the polarity of a set of signals, will perform different operations. This makes CPOGs better suited to little digital systems which are composed of several defined circuits, and generate a structure to determine which circuit is active based on a set of control signals.

DI Algebra, introduced in [63], is a method of describing systems as algebraic equations, specifying causal relations between signal transitions, making it ideal for asynchronous control systems. Each equation represents an operation of the specification, and these can be composed and simplified for a more compact version. All equations can then be composed to find an equation for the whole specification and again simplified for a more compact version.

The proposed method is similar to DI algebra, however Asynchronous Concepts are described textually and as such, simplification does not occur at concept level, but during the composition and combination steps. To the best of our knowledge there are no tools or methodologies supporting compositional design of asynchronous circuits based on DI algebra and thus it is incompatible with the rest of our design flow and not suitable for use in industrial settings.

A **Discriminator** is a "black box" which contains a model, usually a PN, but to everything outside of the discriminator, only the input and output pins are known [64]. This allows each discriminator to contain a separate model of a function of an asynchronous system, which is designed and verified separately. Each discriminator can then be composed, connecting the inputs of some with the outputs of others, and vice versa. Some discriminators can be used multiple times in a specification, and even in other specifications.

This methodology has plenty of similarities to Asynchronous Concepts, in that a specification can be broken down into different functions, operating modes, or scenarios, specified separately and then composed, connecting outputs and inputs of each separate unit. Discriminators can be reused both within a specification and in further specifications, as can concepts. However, each discriminator is still a PN model, which may be problematic to specify if it is a large function, and difficult to comprehend for debugging, editing and future use. Behavioural specification using a language such as concepts can be beneficial in capturing behaviours in a manner that is easier to understand, and the lack of tool support for discriminators means that manual composition is necessary, which can lead to errors with larger specifications.

Hierarchical design of DI systems [65] provides a set of building blocks, either *Delay Insensitive* (DI) or *hybrid* (Non-DI) blocks, which are not necessarily individual logic gates. These are composed by describing the interconnections between the blocks, and this forms a module. Modules can then be composed with other modules in the hierarchy. Signal Transition Graphs are used in this method for specification of a circuit from blocks and modules, and this can be used for analysis of the circuit.

This has some similar ideas to Asynchronous Concepts. Both feature a lower level of specification (building blocks or concepts) which is used to create an STG specification (modules or scenarios) and these are then combined in some manner to produce a full system specification. The difference is that the building blocks provided in the hierarchical design method are set, and Asynchronous Concepts allows for users to define their own low level specification components.

Lava [66] is another tool written in the functional programming language Haskell, like Cλash, with its own associated design flow able to design data-driven or control circuits, and all design steps can be performed in Lava. It features several predefined functions, such as simple logic gates which can be used either as direct operations for circuits, or as part of user defined functions. A user can define a function in terms of inputs, operations on these inputs, and outputs. A circuit is defined as inputs, stored in variables, operations are performed on these using functions which can be sequential or parallel, and then variables are set as outputs. Lava has built in verification, using a parameter which defines verification property. This returns a logic equation which is automatically processed, returning a value determining whether the property is satisfied. Lava also features the ability to generate code for other languages, primarily VHDL, which can then be used by other tools for simulation and synthesis.

Due to Lava being a Haskell based language, it features similar issues to Cλash. It features fairly different syntax to languages used in other existing methods. Unlike

Cλash, Lava does not feature built in tools for synthesis and simulation, and as such, specifications must be converted into VHDL for these processes, a feature which is built in.

Proteus [67][68] introduces a design flow which uses *Pipelines*. Pipelines are the channels which pass data between stages of an asynchronous system which in some cases can cause *bottlenecks*, a major source of delay as data passage is slowed. Proteus is a tool which automatically analyses and optimises a pipelined system to reduce bottlenecks and delays. It takes in an RTL language or a CSP like specification. This is then synthesized, producing a net list which is then analysed and optimised to produce a new pipelined implementation which will have the best performance.

Proteus takes in a specification in the form of CSP or VHDL, which can be useful for designers who may prefer one language over another, however Proteus is a tool which analyses and optimises a previously designed pipelined system which are generally data-driven systems, where as Asynchronous Concepts are aimed at designing an optimised asynchronous control system from the ground up.

Resynthesis [32] is the process of decomposing a full model and recomposing it of selective components to reproduce a smaller model. This can be used to reduce the number of signals to connect two separate models for example.

Resynthesis requires full models which are decomposed. For the proposed methodology, we take a ground-up approach to design, starting with concepts to be composed, and producing specifications which can be combined for a full system. Resynthesis can be used at a later stage of the proposed approach, once the complete model of a system (or a subsystem) has been obtained, which in some cases can produce a more compact and more easily comprehensible STG from one which has been translated from Asynchronous Concepts.

Snippets [69] are smaller FSM models which are used to compose full FSMs of larger systems. Snippets describe the operation of a part of a system in terms of input and output alphabets, and in which ways these snippets can fail. When composed with other snippets they can produce a working system state graph model.

With Asynchronous Concepts we want to go deeper than snippets and compose a component from concepts which are responsible for capturing signal behaviours for system features, such as handshakes, mutual exclusion, synchronisation, etc. As discussed in Section 2.1, FSMs feature some disadvantages when specifying asynchronous circuits, and these apply to the snippets methodology too.

Structural Design [70] encompasses the re-usability of modular components. A component design can be used multiple times across full device designs in conjunction with several other circuit modules. These modules can be changed in some way without affecting how they are used in a full device and how they interact with other modules. This method aims at promoting reuse of circuit designs across different full systems, and reduces the need for redesign of correctly working systems for each new device.

The ideas of this method are similar to the ideas of Asynchronous Concepts, to reduce design time by reusing previously designed elements. However, this method is at a much higher level, using fully designed and tested components, where as we propose to allow reusability when modelling at circuit level, using composed concepts.

Tiempo [71] introduced a design flow which uses a tool called *Asynchronous Circuit Compiler*(ACC). This tool uses *Verilog* to model operations and communication channels between asynchronous entities, which are normally handshaked. The tool allows the use of several asynchronous architecture types aimed at data-driven circuits, i.e. pipelined, parallel, sequential etc. Synthesis uses libraries which contain asynchronous cells, and constraints, such as timing information, have to be specified for the tool. First, a netlist is produced and further constraints produced by the tool. A *place and route* tool then optimises and verifies the system based on the constraints, and a few necessary properties.

Tiempo uses Verilog as a specification language, another RTL language. This has some differences to design flows like *Balsa*, mainly in how it verifies and synthesises a specification, but the advantages and disadvantages are similar. These design methods are usually used for data-driven systems, making them unsuitable for control systems.

Uncle is introduced in [72]. It is a tool which uses a design approach aimed at producing an implementation using *Null Convention Logic* (NCL), a set of components which have a state similar to *precharge*. Each component starts in the null state where outputs and inputs are all null, which does not represent any data. It remains in this state until data is present on all inputs, at which point the component will output data based on the inputs. This data will be held on the outputs of the component until all inputs return to null, when the outputs will return to null. *Uncle* uses an RTL language. This tool then synthesizes the specification using a library of NCL gates which can be simulated and verified, ultimately producing an NCL implementation.

Uncle also uses an *RTL* language for specification of circuits, and these are discussed above. In Uncle, specifying a circuit is carried out in effectively the same way as other methods, the differences are in the synthesis and verification, where null convention logic is used which operates differently and requires different verification properties to standard logic types produced by other design tools.

Asynchronous Concepts, the proposed method, have many advantageous features, such as reuse, natural description, multiple level description and composition, and more. Several of these approaches feature similar ideas which make them beneficial in certain ways, but we believe fall down where the inclusion of one or more of these features could make an approach better. With the Asynchronous Concepts language, we have attempted to address these issues and make Plato not only a powerful tool to specify asynchronous circuits, but a method with as much ease-of-use as possible, particularly targeting the little-digital design domain. Asynchronous Concepts and Plato are also supported by industrial-strength open-source software toolsuite Workcraft.

Chapter 8

Conclusions

This thesis presents a new language for the behavioural specification of asynchronous circuits, and a design flow utilising this language. These aim to promote modular specification, and provide reusability of useful behaviours for future specifications. We provide an EDA tool, Plato, and a library of standard Asynchronous Concepts to aid in the design flow. It also automatically translates a specification to a form which can be used by multiple existing EDA tools for verification and synthesis. These all serve to provide a new compositional design methodology for asynchronous circuit design, to make them more attractive to industry.

In this chapter we will summarise the thesis, discussing the main subjects of each chapter, and outlining the main contributions (Section 8.1). We will then identify areas of future research for improvements to the project (Section 8.2).

8.1 Main contributions

In Chapter 3 we discuss **Asynchronous Concepts**. Starting with the abstract base, we discuss Asynchronous Concepts which introduce the notion of initial state, excitation and invariant. From these, we then build circuit-specific **signal-level concepts**, which specify behaviours in asynchronous circuits. These are atomic and can be composed to produce higher-level concepts. We explain how these are built on top of the abstract concepts. The causality concepts, built upon the abstract excitation concept, are composed to produce **AND causality**, or **OR causality** where a set of possible causes can be specified and composed with other causalities.

Gate- and Protocol-level concepts are then introduced. These are combinations of signal-level concepts, used to specify the behaviour of standard asynchronous logic gates, such as a C-element, and protocols, such as a handshake. We identify how these are derived using signal-level concepts, and how the concepts at this level can be reused to specify other gates. We also introduce several **generalised multiple input gates**, which can allow the behaviours of some gates to apply to any number of inputs, as opposed to the two-input gates we discussed previously.

We also introduce **high-level concept functions**, which provide extensions to the language, such as `function` and `complexGate` which allow Boolean functions to be used along-side concepts. We also provide some **transformations** to extend the range of behaviours a single concept can be used for. The `bubble` transformation allows signal transitions to be inverted in the given concept, such as inverting the output of a gate-level concept. `dual` is a transformation which provides the dual of a given concept by inverting all transitions, the initial state, and any states which do not hold for the invariant. Furthermore `enable` can provide a signal which determines whether the specified device outputs can transition.

All concepts and high-level functions introduced in Chapter 3 are included in a **circuit specific library** provided with the tool designed for use with concepts, Plato. We use several of these in an example when deriving a concept for a set-reset latch. This identifies some difficulties when trying to derive the concept, which indicates an issue with the language of concepts, an issue which we aim to fix with future research. Ultimately, we provide a concept, `srLatch`, which has some assumptions about the environment, but does specify the behaviour of this component.

Chapter 4 discusses the **Asynchronous Concepts design flow**. This can start with an informal description of the device to be designed, and these descriptions can be used to specify behaviours. Alternatively, it may use a previously defined circuit, and we can use the set and reset functions for this to generate a concept specification, or if no functions are known, we can use process mining to obtain one possible specification from the simulation traces. Of course, the specification may include a combination of new components, and previously designed components. Any useful behaviours or components can be stored in a **user generated concept library**. This can be imported by other concept files for use of these concepts multiple times in this specification, or in future specifications.

A concept specification is then compiled and **translated to an STG**. This process identifies errors in the concept specification for a user to correct, and then produces an

STG which can be visualised. If a system features several specifications, each of which may describe a separate scenario for example, then these can be combined at the STG stage using **templates**, which will add information which allows the STGs to interact according to some rules.

When a full system specification has been produced, this can then be simulated, to ensure that the system operates as expected. Verification can then be carried out, to ensure that the STG is implementable. This can then be synthesized, producing an implementation.

Chapter 5 introduces the tools which support the Asynchronous Concepts design flow. This includes **Plato**, a tool developed for concepts, which compiles and translates concept specifications, and implements algorithms for translation to either STGs or state graphs. We also provide the algorithm for the **Bool-to-Concept** feature of Plato, which takes Boolean functions, such as the set and reset functions of a component, and generates a concept specification for these.

This chapter also discusses the open-source toolsuite, Workcraft, which features a GUI in order to visualise graphical modelling formalisms, such as STGs, FSMs and state graphs. **Plato is integrated to Workcraft**, which allows the design flow to be performed entirely within Workcraft, where concept specifications can be written, saved, edited, and translated at the click of a button. The generated STGs and state graphs are automatically imported into Workcraft, for further operations. Workcraft also features integrated tools for simulation, verification and synthesis which can all be performed from the click of a button, further streamlining the design flow.

We then use the Asynchronous Concepts language, the associated design flow, as well as the supporting tools, in some case studies in Chapter 6. The first was the example of a simple buck controller, which featured three scenarios. Each was specified using asynchronous concepts, verified, and then all of these were combined. The STG was then verified and synthesized, producing one of several implementations.

The second case study in Chapter 6 was that of a multiphase buck controller. Specifically, we specified the behaviour of a single phase of this system, producing concept specification for each separate block. These were then composed in specifications for the two larger sections, activation and charging. Finally, we produced a single concept specification for the full system, using all of the concepts derived for these blocks, and translated an STG from this.

8.2 Future research and development

The contributions of this thesis discuss the current state of the language of Asynchronous Concepts and the associated tool Plato. However, further improvements may be made with additional research and development, to extend the usability of concepts, improve the design flow, and provide a tool with more features. These are discussed in this section.

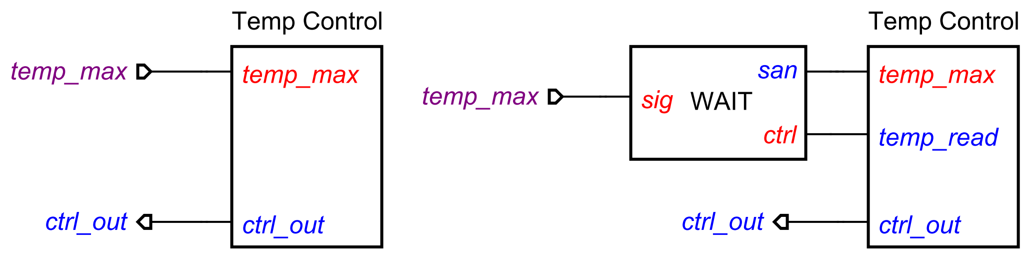
Currently, Asynchronous Concepts are aimed at specifying asynchronous control systems, and the library provided is specific for control circuits. However, asynchronous systems can be used for different types of digital systems. A future research area for concepts could be for expanding the language to be able to specify different types of asynchronous circuits.

For example, the circuit specific concepts could be expanded to include mixed signal systems [73] as these are a recent research area for asynchronous concepts. Analogue signals could be included, which have a numeric value associated with them, and have threshold values identifying whether this can be considered as high or low. This information could help the specification in expressing how this signal should be handled.

As with analogue signals, a hazardous signal could be identified in a concept specification. This may not require a numeric value to be associated with it, however it can aid in determining how behaviours this signal is involved with should operate. For example, a WAIT element, and other derived arbitration primitives, are designed to sanitize hazardous signals. If a hazardous signal is identified, the translation process could automatically include a WAIT element for this signal, to ensure that the issues this hazardous signal can present to a circuit are automatically prevented [47].

For example, if we specify a temperature control circuit, which controls a fan, as shown in Figure 8.1a. When the temperature has reached a threshold, `temp_max` signals, which turns on the fan to cool the environment. However, when the temperature is around the temperature threshold, it may fluctuate, meaning that `temp_max` may be hazardous, causing the fan control system to enter a state which may be problematic.

A concept specification which identifies `temp_max` as a potentially hazardous signal could instead automatically include a WAIT element. This, as discussed in Section 6.2.2, will sanitise the `temp_max` signal. The automation of this could also prepare the logic necessary for the control of the WAIT element. The result of this will be as seen in Figure 8.1b.



(a) A hazardous input signal (b) Using a WAIT element

Figure 8.1: Two temperature control circuits, with a hazardous input signal

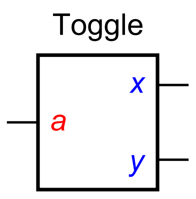
Asynchronous Concepts could be adapted to big-digital systems, which deal with data of multiple bit widths, and the operations on these. This could be built upon the abstract base, and even be used in conjunction with the circuit specific concepts, which can be used to specify the control for the big-digital circuit. An interesting case study would be to produce a library of Asynchronous Concepts which features dataflow components be used to design dataflow systems, such as in [74].

Similarly, we could derive libraries for other types of asynchronous system, and test their usefulness, to explore the range of uses that Asynchronous Concepts can be used for. For example, multi-way arbitrating components as discussed in [75], or Flat Arbiters, components which make decisions, as discussed in [76].

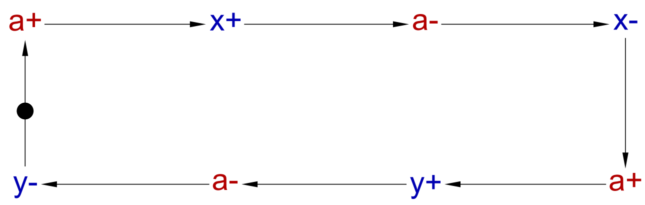
CSC conflicts are currently detected and corrected at the STG phase of the design-flow. This is often resolved by adding an internal signal to the STG, sometimes automatically. This therefore means that the concept specification, and the STG with the CSC conflict resolved, are no longer equivalent. To combat this, a method of CSC conflict detection could be added to the compilation and translation operations of Plato, in order to identify this to the user, and add the resolution to the concept specification. This way, the concept specification and the translated STG remain equivalent.

As an example we use a toggle circuit (Figure 8.2a). This has one input signal, a, and two output signals, x and y. After initialisation, when a rises, then x will rise. Following this, a will fall, and x will fall. When a rises again, y will this time rise, and fall after a falls. x and y will alternately rise when a falls.

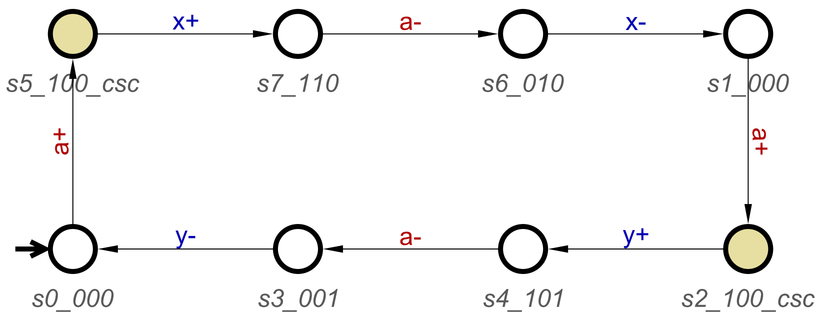
The STG of a toggle circuit can be found in Figure 8.2b. This is a simple STG, but it does not show the CSC conflicts which occur with this. Figure 8.2c does show this. There states which conflict are highlighted. These states occur because there is no information to suggest whether x or y should transition high next.



(a) Toggle circuit



(b) Toggle circuit STG



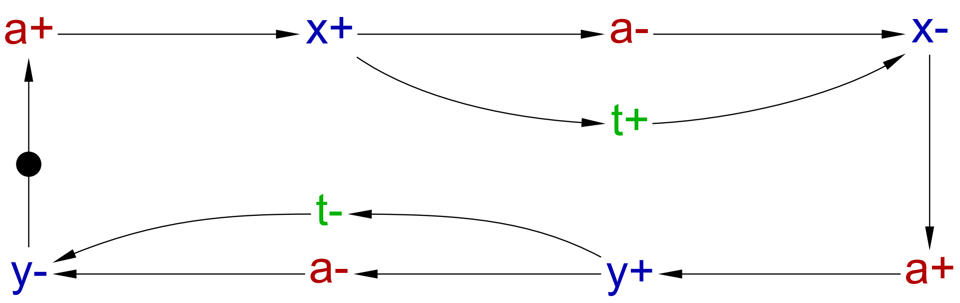
(c) Toggle circuit state graph

Figure 8.2: The circuit and models of a toggle circuit.

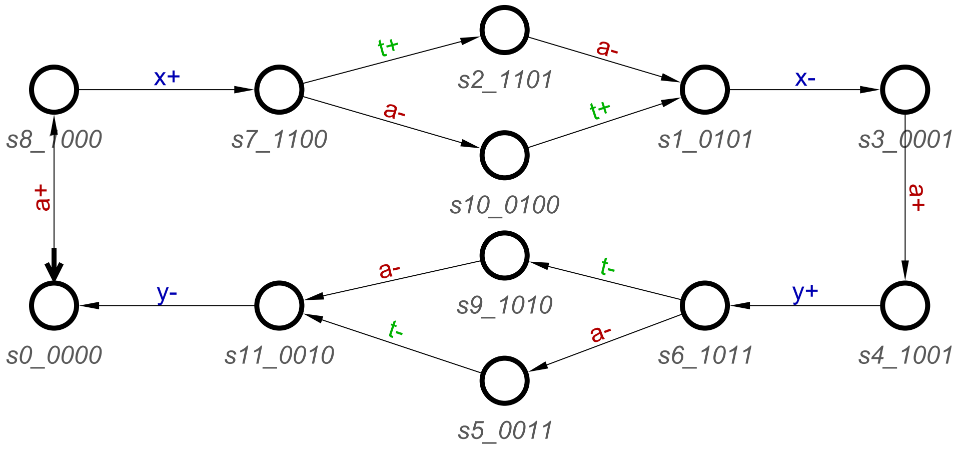
The circuit needs some form of memory in order to identify which of the output signals will transition next, and this can be in the form of an internal signal. This signal will in turn cause the encodings of one set of these states to be different, solving the CSC conflicts. Ideally, this would be performed automatically by Plato during compilation or translation, and this is an area of future development. The CSC resolved STG and state graph can be found in Figure 8.3.

As discussed as part of the set-reset latch example (Section 3.4), *optional* arcs can occur, where a system can continually transition between some states, until an alternate transition causes a change to a different state not featuring these optional arcs [36]. Being able to identify optional arcs in Asynchronous Concepts, and what operation should occur in this without making the assumptions about the environment as we have in this example would be a benefit. This could then transfer to STGs, which do not support optional arcs either.

In some cases, particularly where a component concept features a signal and its



(a) Toggle circuit STG with CSC resolved



(b) Toggle circuit state graph with CSC resolved

Figure 8.3: STG and state graph of a toggle circuit with CSC resolved

inversion together, a future development would be to add a concept which links the initial state of these signals. Again, referring to the set-reset latch example, we force that the `q` output of this signal is initially 0, and its inversion, `nq` is initially 1. However, it may be the case that these are the opposite in practice. Thus, a concept which indicates that when one of these signals is initially low, the other must be initially high would stop the need of forced initial states, which currently will block concepts being compiled if the initial states are be reversed when using this concept.

Plato currently can generate concepts from given Boolean functions, useful for automatically finding a concept specification for an existing circuit, whether these Boolean functions are known (Section 5.1.3) or not (Section 5.2.2). However, the specification produced uses signal-level concepts. For larger circuits, featuring more than a handful of signals, this can make for a particularly large generated concept. Optimization of these concepts could be performed with some future development, which looks at higher protocol- and gate-level concepts, and tries to cover as many of the signal-level concepts with higher-level concepts as possible, resulting in a smaller specification. However, this may prove to be a difficult problem to solve. There may be many possible combinations of concepts which cover the given set of signal-level concepts. A prototype tool which attempts this is *Copter* [77].

In Section 4.5, it is explained that Asynchronous Concepts are automatically translated to STGs in order to take advantage of their theory, and verification and synthesis tools with a long history of development. In the future it may prove to be beneficial to develop concept verification and synthesis tools. These may prove to be more efficient than verification and synthesis via translation to STGs, however the translation process to STGs and state graphs will still be beneficial for the purposes of simulating specifications, to ensure it works as expected.

Currently, combination of concept specifications is performed on translated STGs (see Section 4.6). This is due to some limitations of the concepts language, and Plato, in that there is no method to explicitly specify a place, or identify the exit state of a specification. Ideally, if a collection of specifications have been determined to be sound, then given a template, a tool would automatically determine what can be considered the exit state of a specification, and then connect this to a place, and connect this to the entry state of other specifications. The addition of places would not pose a problem as an additional feature. However, detection of the exit state of a given specification would be, as there could be multiple possible exits. Research into the automated detection of where scenarios switch is being done as part of *Process windows* [40]. This work may be key to improving the combination of concepts in the future.

To improve the clarity of translated STGs of larger specifications, it would be useful to reduce the number of internal signals. As an example, the translated STGs from the multiphase buck controller case study (Figures A.25, A.26 and A.27) are large, and feature many internal signals. Most of these are used as intermediate signals, connecting the output of one logic gate to an input of another. Some are unused output signals, and simply set as internal to not be confused with an important output (e.g. the mutex g1 used in the WAIT element). It would improve clarity to hide these internal signals so that only important internal signals, such as those used to resolve CSC conflicts, are included.

To fix this, a feature to add would be to allow the output of a logic gate be used as the input to a logic gate, or be able to hide a signal which is used neither as an internal signal nor an output. This signal may be given a name for ease of reference in a specification, but it could be identified as one which need not be included in translations.

Instead, the behaviours this signal is involved can be passed into other signals which are related to this.

Finally, an important area of future development would be to provide a language and compiler for Asynchronous Concepts. Currently, concepts are a domain-specific language embedded within Haskell, meaning that many of the Haskell features and syntax carry over. Therefore, a user needs to be reasonably familiar with Haskell itself in order to effectively use the language. Providing a compiler specific to concepts allows us to shape the syntax, and enable useful constructs, such as postfix notation, so a transition can be stated as `x+` instead of `rise x`. This would improve the usability of asynchronous concepts further, while allowing for improvements to the language and the design flow.

Appendix A

Multiphase buck controller components

This appendix section contains the concept specifications, and STGs for the components used in the asynchronous multiphase buck controller. Some specifications are included within Chapter 6 however, the chapter aims to show how a full system specification can be built using component specifications, and as such, the operation of each individual component is not included in this chapter. The concept specifications provided here are imported and used in the specification for the full system in Section 6.2.3. The block diagram for how the components interconnect can be found in Figure 6.14.

The operation of a multiphase buck controller, as well as each component included in the system are briefly discussed in chapter 6, and more information on them can be found in [1].

A.1 High-load handler

The High-load handler is used to set all phases of the multiphase buck into charging mode, when the `hl` signal is high. `hl` is sanitized by a WAIT element, namely HL_WAIT. Regardless of where the token in the system is, the PMOS transistor is switched ON, and the NMOS transistor is switched OFF in all phases.

Figure A.1 contains the concept for the high-load handler. Figure A.2 is the STG when translated from the concept. Figure A.3 is the resynthesized version of this STG.

```
module HLH where

import CircuitConcepts

hlh hl whl ro ao = behaviour <> interface <> initState

  where

    behaviour    = hlHandshake <> reqAckHS

                 <> hlSignalled <> chargeAck

    hlHandshake = handshake whl hl

    reqAckHS     = handshake ro ao

    -- When hl signals, request charge

    hlSignalled = rise hl ~> rise ro

    -- When acknowledged, stop waiting for hl

    chargeAck    = inverter ao whl

    interface    = inputs [hl, ao] <> outputs [whl, ro]

    initState    = initialise0 [hl, ao, whl, ro]
```

Figure A.1: High-load handler component concept specification

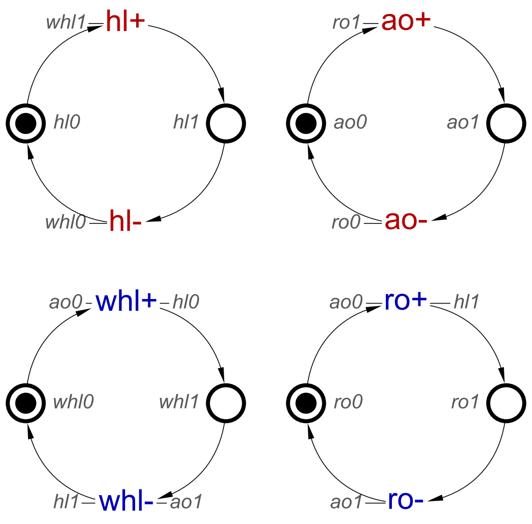


Figure A.2: HLH STG translated from concepts.

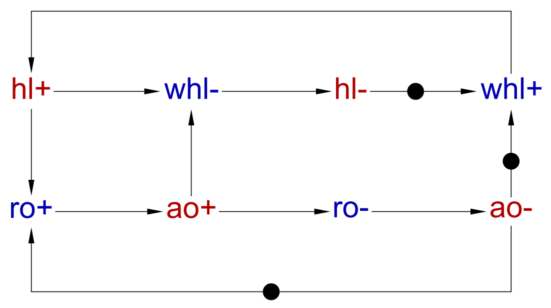


Figure A.3: HLH STG resynthesized.

A.2 Token-control

Token control is used to control the movement of the token between the phases. `get` takes the token from the previous phase, and this is used to activate the current phase. The token is held for a minimum amount of time, which is determined by a delay component, named TOKEN_CONTROL. When the charging section is active, then the token is passed through `pass` to the next phase.

Figure A.4 contains the concept for the Token-control component. Figure A.5 is the STG translated from this concept. Figure A.6 is a resynthesized version of this STG.

```
module TC where
import CircuitConcepts
tc ri ai rd ad ro ao = behaviour <> interface <> initState
  where
    behaviour   = reqAckHS <> sendReq <> startTimer <> ackFunc
    reqAckHS    = handshake ri ai <> handshake rd ad
               <> handshake ro ao
    sendReq     = rise ri ~> rise ro
    startTimer  = rise ri ~> rise rd
    -- Wait for timer and charge acks before passing token
    ackFunc     = cElement ad ao ai
    interface   = inputs [ri, ad, ao] <> outputs [ai, rd, ro]
    initState   = initialise0 [ri, ai, rd, ad, ro, ao]
```

Figure A.4: Token control component concept specification.

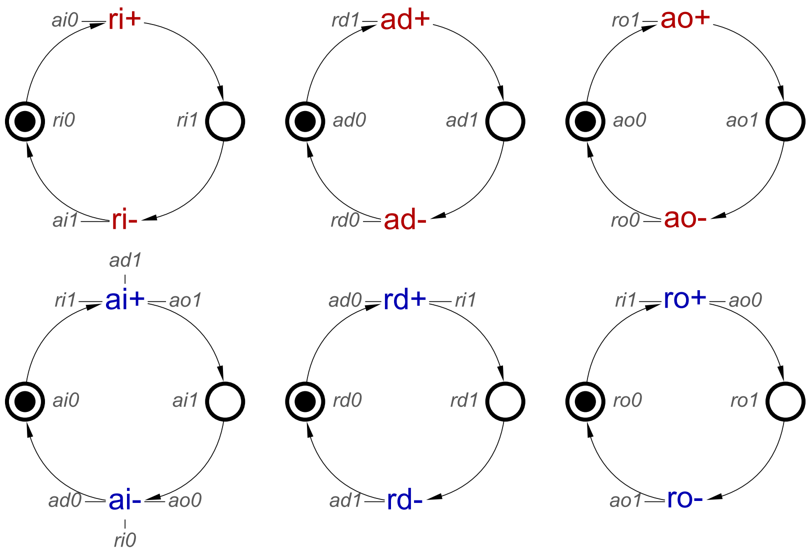


Figure A.5: TC STG translated from concepts.

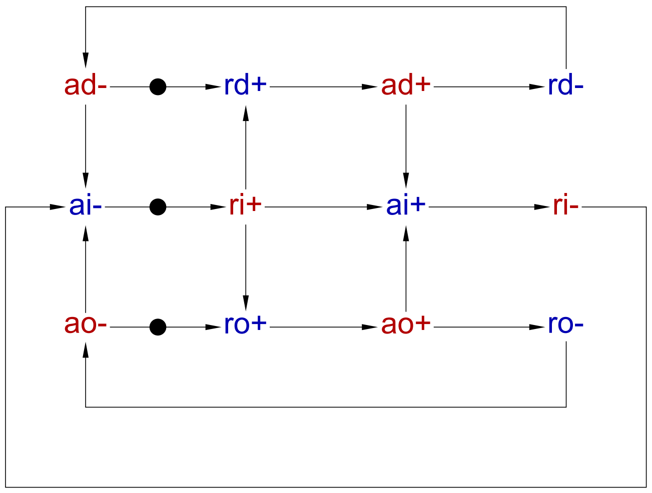


Figure A.6: TC STG resynthesized.

A.3 Delay

A delay is used as a timing element. When the input request, `ri` goes high, a second handshake is started, which takes a set period of time to complete. This will then allow the input acknowledge, `ai` to go high, indicating the minimum length of time has passed. This component is used to ensure that the token is held in a phase for a minimum length of time, and that the PMOS and NMOS transistors are switched ON for a minimum length of time.

Figure A.7 contains the concept for this element. It is used multiple times throughout the specification. Figure A.8 is the translated STG for this concept. Figure A.9 is the resynthesized version of this STG.

```
module D where
import CircuitConcepts
delay ri ai ro ao csc1 = behaviour <> interface <> initState
  where
    behaviour = reqAckHS <> riFunc <> aiReact <> cscRes
    reqAckHS  = handshake ri ai <> handshake ro ao
    riFunc    = rise ri ~> rise ro
    aiReact   = fall ao ~> rise ai
    cscRes    = rise ao ~> rise csc1
             <> [fall ai, fall ri] ~&~> fall csc1
             <> rise csc1 ~> fall ro <> rise csc1 ~> rise ai
             <> fall csc1 ~> rise ri <> fall csc1 ~> rose ro
    interface = inputs [ri, ao] <> outputs [ai, ro]
             <> internals [csc1]
    initState = initialise0 [ri, ai, ro, ao, csc1]
```

Figure A.7: Delay component concept specification.

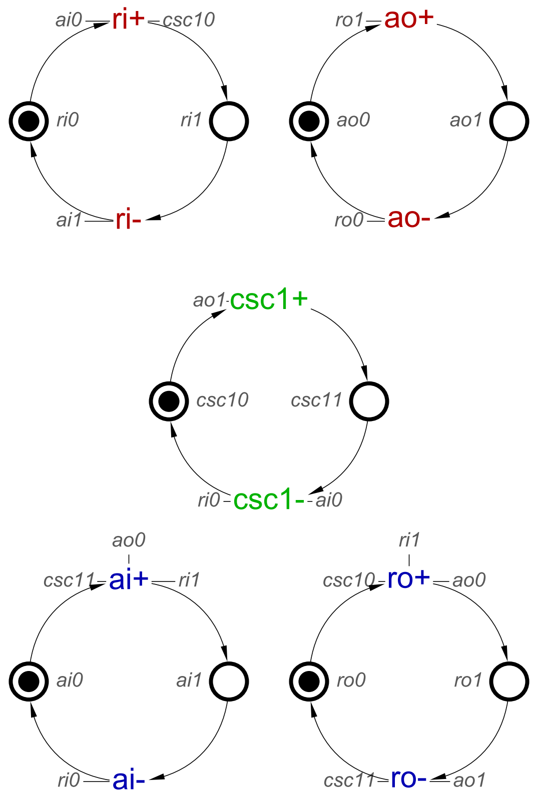


Figure A.8: Delay STG translated from concepts.

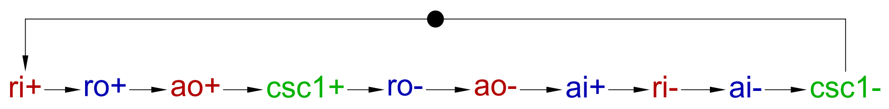


Figure A.9: Delay STG resynthesized.

A.4 MERGE element

The MERGE element is used to combine request-acknowledge handshakes from two sources. If either of the requests, `ri1` or `ri2`, signals then the output request will signal. The MERGE element also serves to arbitrate between the requests of these two sources when both requests arrive at almost the same time, which can lead to an issue of metastability.

Figure A.10 contains the concept for the MERGE element. Figure A.11 is the translated STG. For the MERGE element we also provide a simplified state graph which removes several states, to provide a clearer view of the operation of a MERGE element. The states removed include the output request and acknowledge transitions, and states which lead back to the device resetting. For this reason, the state labels have been removed.

```
module MERGE where
import CircuitConcepts
merge r1 r2 r a1 a2 a i1 i2 = behaviour <> interface <> initState
  where
    behaviour     = reqAckHS <> reqHandshake <> singleAck
                 <> outRequest <> twoWayHandshake a a1 a2
    reqAckHS      = handshake r1 a1 <> handshake r2 a2
                 <> handshake r a
    reqHandshake  = inverter r r1 <> inverter r r2
                 <> [rise r1, rise r2] ~|~> rise r
    singleAck     = mutex a1 a2
    -- Ensure that the request is only sent when r1 OR r2 is high
    outRequest    = bubble r1 (andGate r1 a1 i1)
                 <> bubble r2 (andGate r2 a2 i2)
                 <> bubble r (orGate i1 i2 r)
    -- i1 and i2 are intermediates between and gates and or gate
    interface     = inputs [r1, r2, a] <> outputs [r, a1, a2]
                 <> internals [i1, i2]

-- A handshake with one input causing only one of two outputs
twoWayHandshake a b c = rise a ~> rise b <> rise a ~> rise c
                     <> [rise b, rise c] ~|~> fall a
                     <> fall a ~> fall b <> fall a ~> fall c
                     <> [fall b, fall c] ~&~> rise a
```

Figure A.10: MERGE component concept specification.

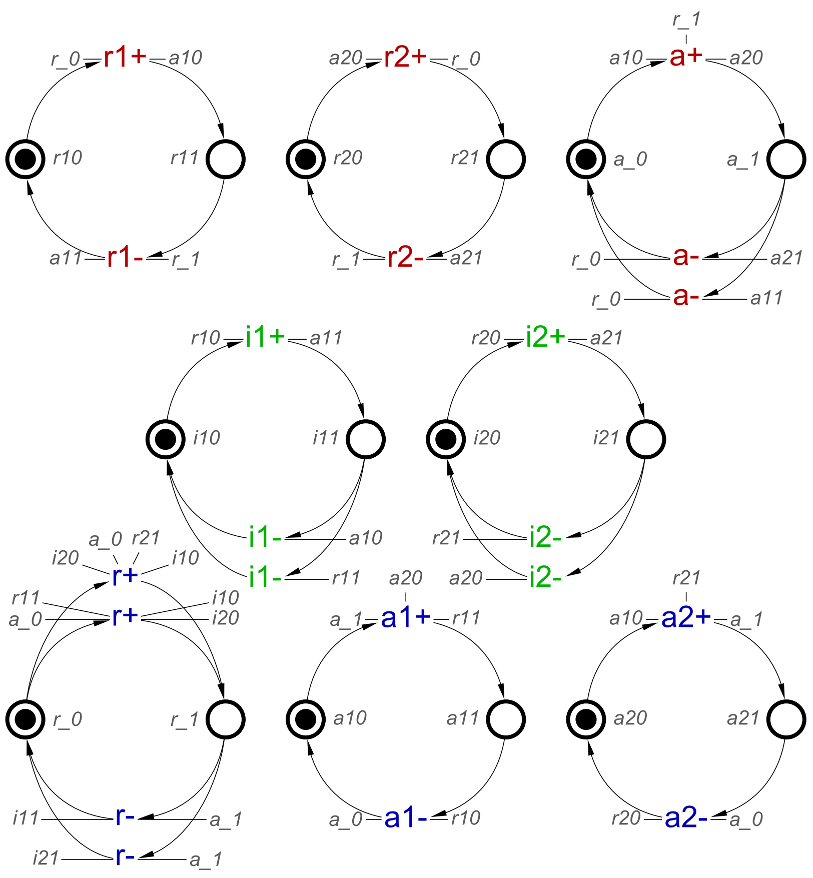


Figure A.11: MERGE STG translated from concepts.

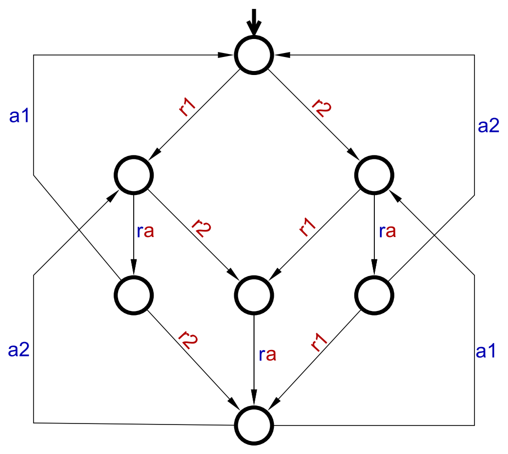


Figure A.12: Simplified state graph of a MERGE element

A.5 Under-voltage handler

The under-voltage handler is a component which, when the charging section is activated by a request coming from the activation section, will check the `uv` signal. This signal is sanitized by a WAIT element, UV_WAIT. When uv signals, indicating the under-voltage condition, an acknowledgement is passed back to the activation section through ai, which then allows the token to be passed to the next phase, and the request is set high, ro, which is passed to the ZCH module, to ultimately switch the PMOS transistor ON, and the NMOS transistor OFF.

Figure A.13 contains the concept for the under-voltage handler. Figure A.14 is the STG translated from these concepts. Figure A.15 is a clearer, resynthesized version of the STG.

```
module UVH where
import CircuitConcepts
uvh wuv uv ri ai ro ao i1 = behaviour <> interface <> initState
  where
    behaviour   = reqAckHS <> sendReq <> activateAck <> wuvReact
    reqAckHS    = handshake wuv uv <> handshake ri ai
               <> handshake ro ao
    sendReq     = rise uv ~> rise ro
    activateAck = rise uv ~> rise ai
    wuvReact    = buffer ri wuv <> fall ro ~> fall wuv
               <> fall ao ~> fall wuv <> rise i1 ~> rise wuv
               <> bubbles [wuv, ao] (cElement ao wuv i1)
    interface   = inputs [ri, ao, uv] <> outputs [wuv, ai, ro]
               <> internals [i1]
    initState   = initialise0 [wuv, uv, ri, ai, ro, ao]
               <> initialise1 [i1]
```

Figure A.13: UVH component concept specification.

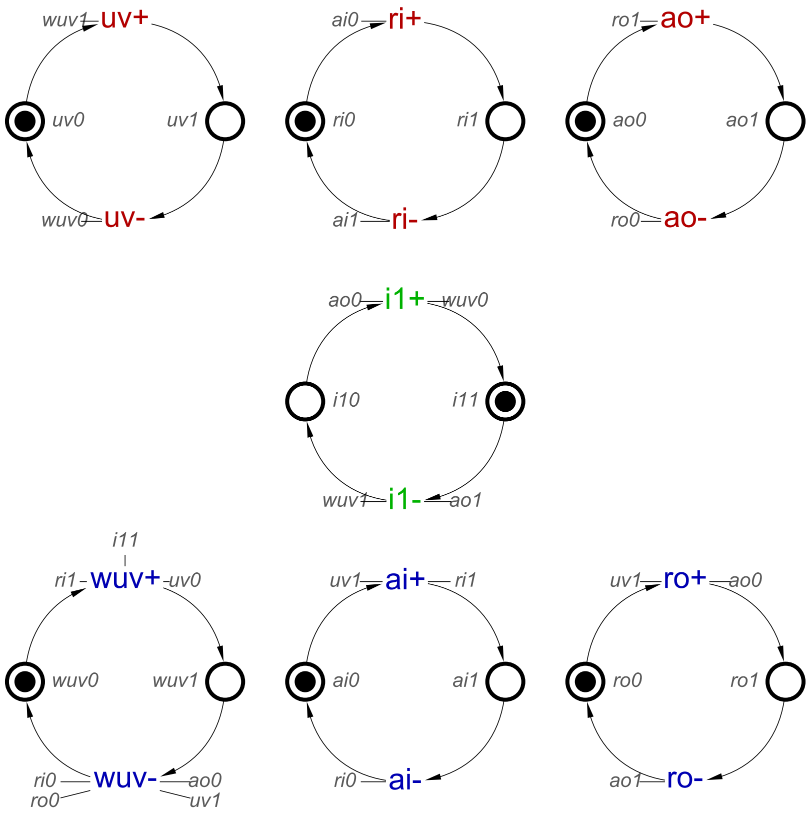


Figure A.14: UVH STG translated from concepts.

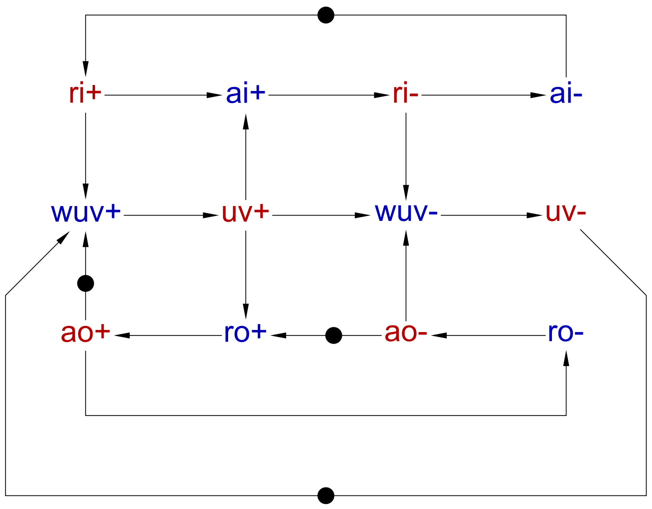


Figure A.15: UVH STG resynthesized

A.6 Zero-crossing handler

The zero-crossing handler deals with the zero-crossing condition, indicated by the `zc` signal. A request comes from the UVH module, `ri`, which may or may not signal before zc, or zc may not signal at all. If it does not, or signals after the request from UVH, then the zero-crossing condition is ignored, and the output request, ro is set high, which will switch both transistors OFF, and then high, which will switch the PMOS transistor ON, to charge which solves the under-voltage condition. If zc signals before the request from the UVH component, then the output request, ro, will be set high to switch both transistors OFF until uv signals.

Figure A.16 is the concept for the zero-crossing handler component. Figure A.17 is the STG which is translated from the concept, and Figure A.18 is a resynthesized version of this.

```
module ZCH where
import CircuitConcepts
zch zc ri ai ro ao no_zc i1 = behaviour <> interface <> initState
  where
    behaviour   = reqAckHS <> choice <> zcDecided <> zcComplete
                <> merge <> late_noZC <> earlyZC
    reqAckHS    = handshake ro ao <> handshake ro ao
    choice      = fall ai ~> rise ri <> fall ai ~> rise zc
    zcDecided   = orGate zc no_zc i1 <> latchHelp
    latchHelp   = bubble ri (srHalfLatch i1 ri ai)
    zcComplete  = fall zc ~> fall ai
                <> [fall ao, fall zc] ~&~> rise ai
    merge       = [rise ri, rise zc] ~&~> fall ro
    late_noZC   = rise ri ~> rise no_zc
                <> [rise ri, rise zc] ~|~> rise ro
    earlyZC     = rise ro ~> fall zc
    -- i1 is an intermediate. no_zc is instead of a dummy signal
    interface   = inputs [ri, ao, zc] <> outputs [ai, ro]
                <> internals [no_zc, i1]
    initState   = initialise0 [zc, ri, ai, ro, ao, no_zc, i1]
```

Figure A.16: ZCH component concept specification.

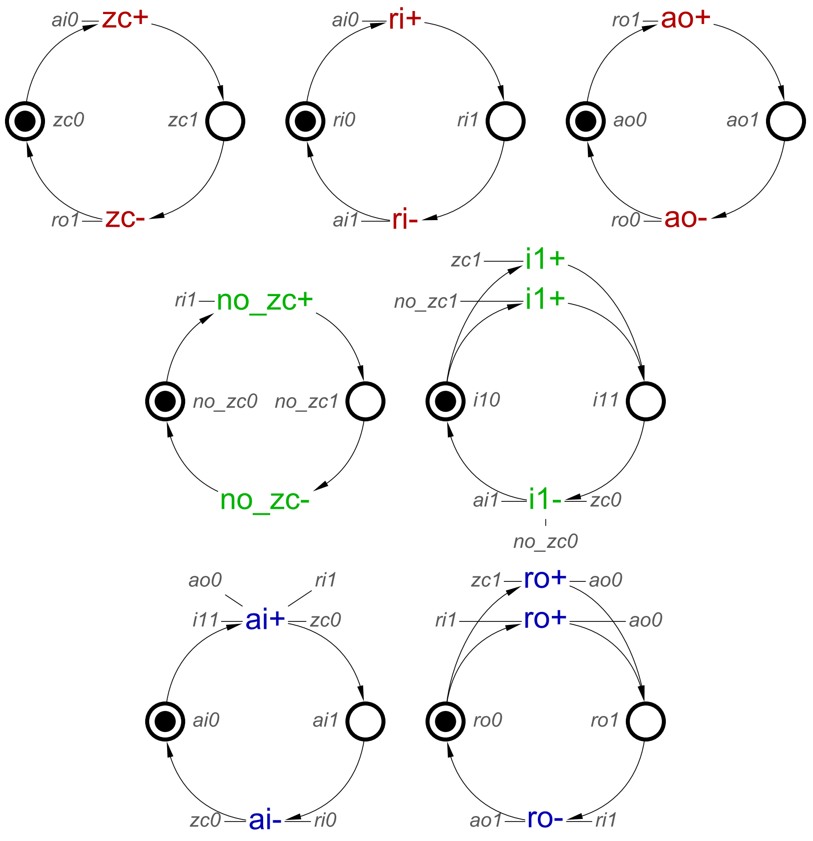


Figure A.17: ZCH STG translated from concepts.

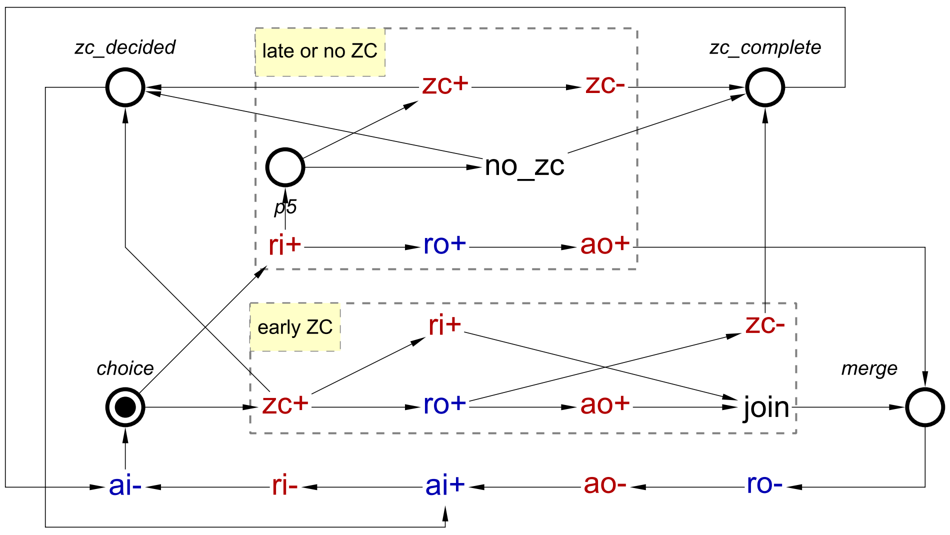


Figure A.18: ZCH STG resynthesized, with differing ZC signalling timings

A.7 Over-current handler

The over-current handler deals with the over-current condition, indicated by the oc signal. It also handles the switching of the transistors. When a request comes from ZCH via ri, this will switch the NMOS transistor OFF, setting rn low, and both transistors will be OFF. When ri goes low, the PMOS transistor will be switched ON by setting rp high, which will correct the under-voltage condition. When oc signals, this will switch the PMOS transistor OFF, and then the NMOS transistor ON, to correct the over-current condition.

Figure A.19 is the over-current handler concept. Figure A.20 contains the translated STG from this concept. Figure A.21 is a clearer STG, resynthesized from the translated STG.

```
module ZCH where
import CircuitConcepts
zch zc ri ai ro ao no_zc i1 = behaviour <> interface <> initState
  where
    behaviour  = reqAckHS <> pmosON <> nmosOFF <> noShort
               <> ocReact <> ocFunc
    reqAckHS   = handshake rp ap <> handshake ri ai
               <> handshake rn an
    pmosON     = fall ri ~> rise rp <> rise ap ~> fall ai
    nmosOFF    = rise ri ~> fall rn <> fall an ~> rise ai
               <> fall rn ~> rise ai <> fall ai ~> fall ap
    noShort    = mutex rp rn <> fall an ~> rise rp
               <> fall ap ~> rise rn
    ocReact    = rise rp ~> rise oc <> rise rn ~> fall oc
    ocFunc     = rise oc ~> fall rp <> rise oc ~> rise rn
               <> fall oc ~> fall rn <> fall oc ~> rise rp
               <> fall oc ~> rise ri <> fall oc ~> rise ai
    interface  = inputs [oc, ri, ap, an] <> outputs [rp, rn, ai]
    initState  = initialise0 [oc, ap, an, rp, rn, ri, ai]
```

Figure A.19: OCH component concept specification.

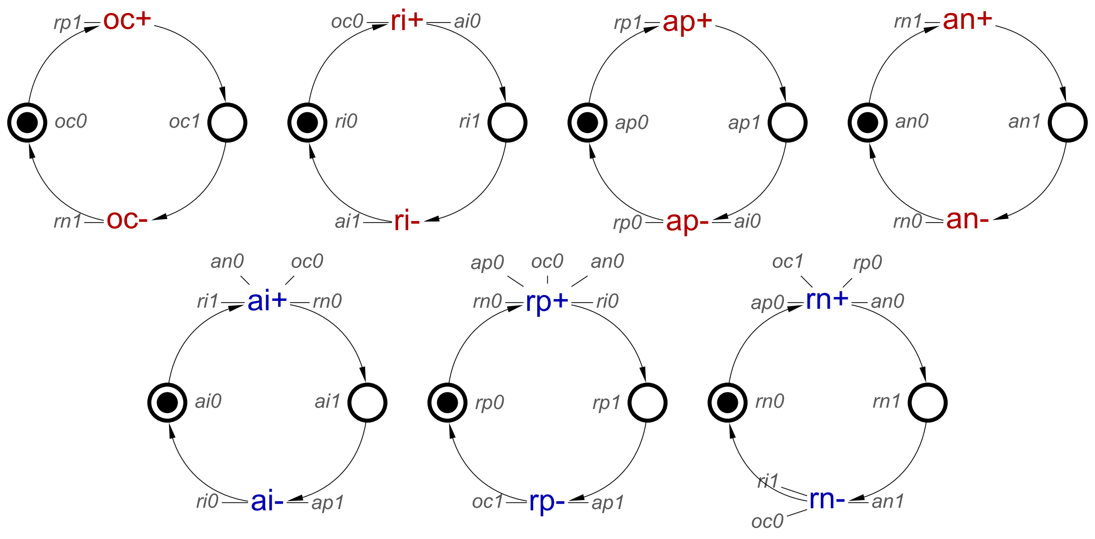


Figure A.20: OCH STG translated from concepts.

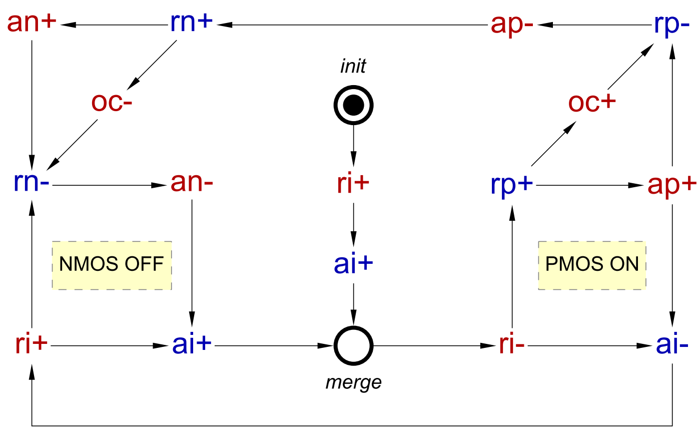


Figure A.21: OCH STG resynthesized

A.8 Transistor Delay Controller

The transistor delay controller is used to switch a transistor ON or OFF, based on the requests incoming from the OCH module. Two of these components are used, one for the PMOS transistor, and one for the NMOS transistor. When a request comes in via `ri`, it will set `ro` high to switch the transistor on. It will also set `rd` high, which starts the delay timer, as discussed in Section A.3. This will ensure that the transistor is switched on for a minimum amount of time. The transistor being switched on is acknowledged by the `ao` signal, and when the delay acknowledges via `ad`, then the transistor can be switched OFF when the input request, `ri`, is set low.

Figure A.22 contains the concept for the transistor delay controller. This is used twice in the full specification, one for each transistor in a single phase. The translated STG from this concept is found in Figure A.23, and a resynthesized version is seen in Figure A.24.

```
module DC where
import CircuitConcepts
dc ri ai rd ad ro ao = behaviour <> interface <> initState
  where
    behaviour   = reqAckHS <> ioInteract <> delay
    reqAckHS    = handshake ri ai <> handshake ro ao
                <> handshake rd ad
    ioInteract  = buffer ri ro <> buffer ao ai
    delay       = rise rd ~> rise ai <> rise ao ~> rise rd
    interface   = inputs [ad, ri, ao] <> outputs [rd, ai, ro]
    initState   = initialise0 [rd, ad, ri, ai, ro, ao]
```

Figure A.22: DC component concept specification.

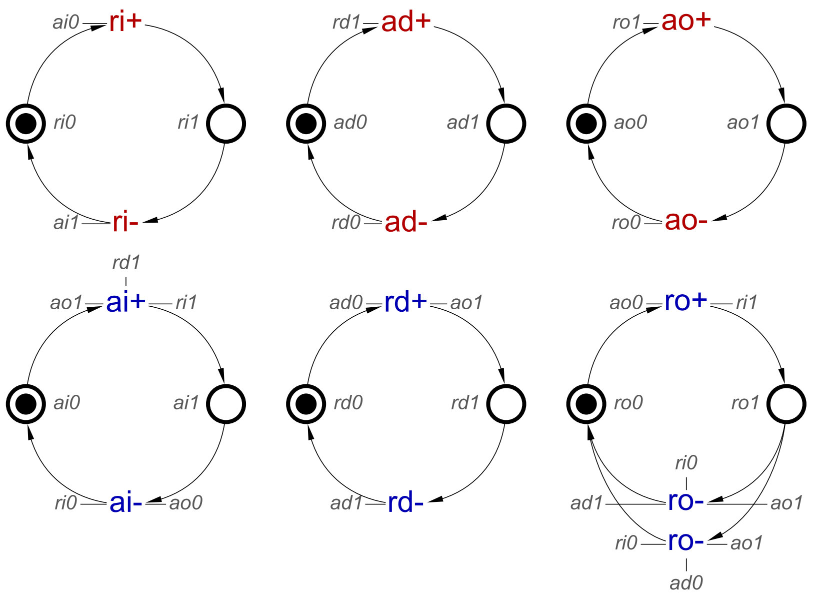


Figure A.23: DC STG translated from concepts.

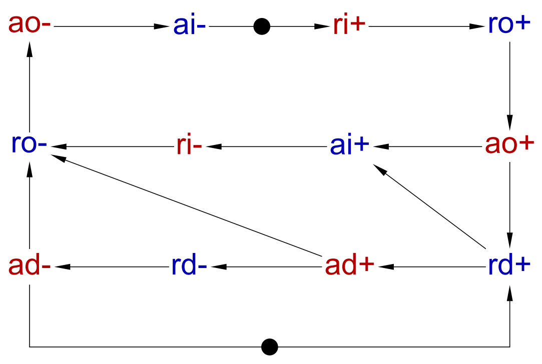


Figure A.24: DC STG resynthesized

A.9 Full system translated STGs

Figures A.25, A.26 and A.27 are STGs translated from specifications for the activation section, charging section and full multiphase buck specification from Section 6.2.

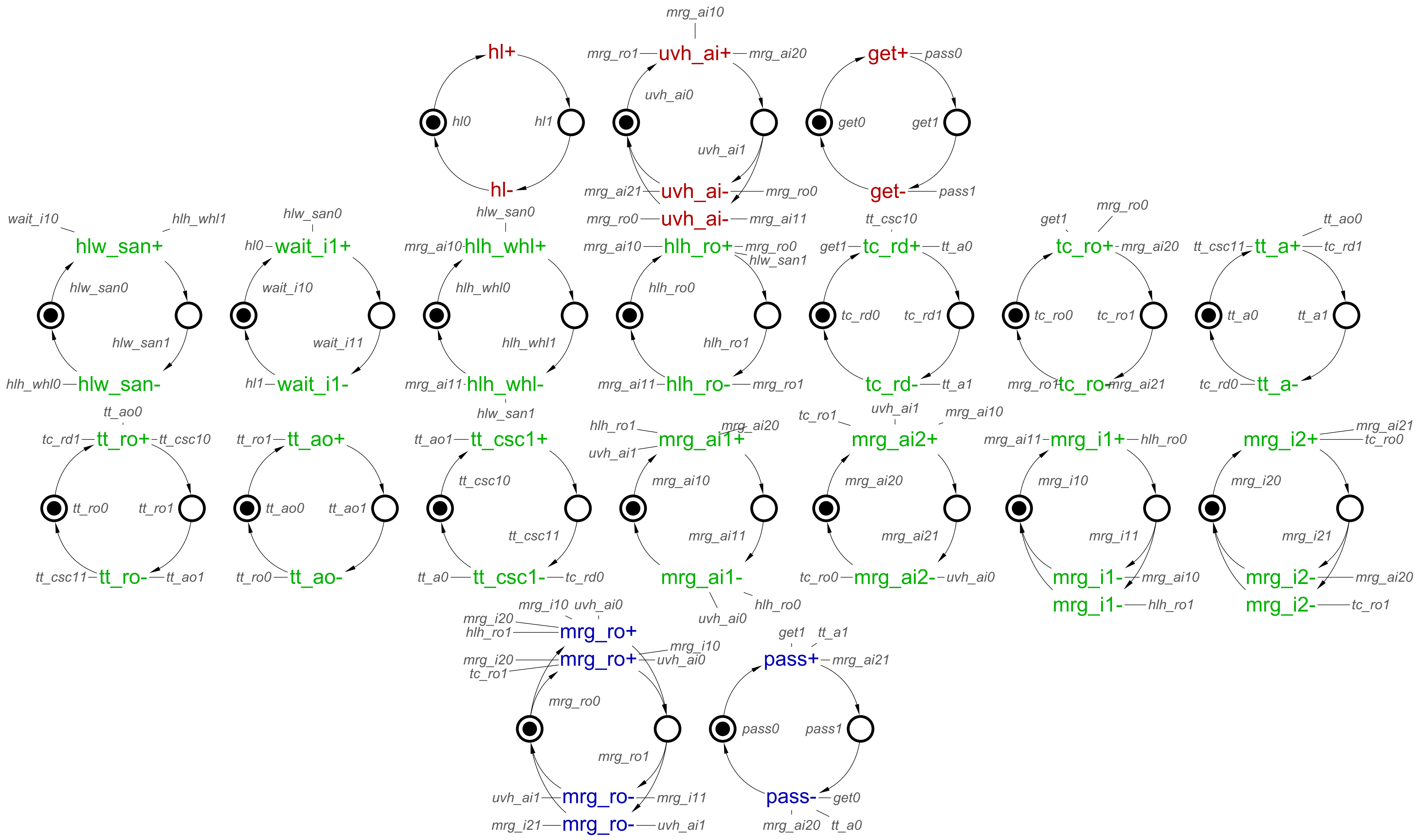


Figure A.25: Translated STG of the activation section.

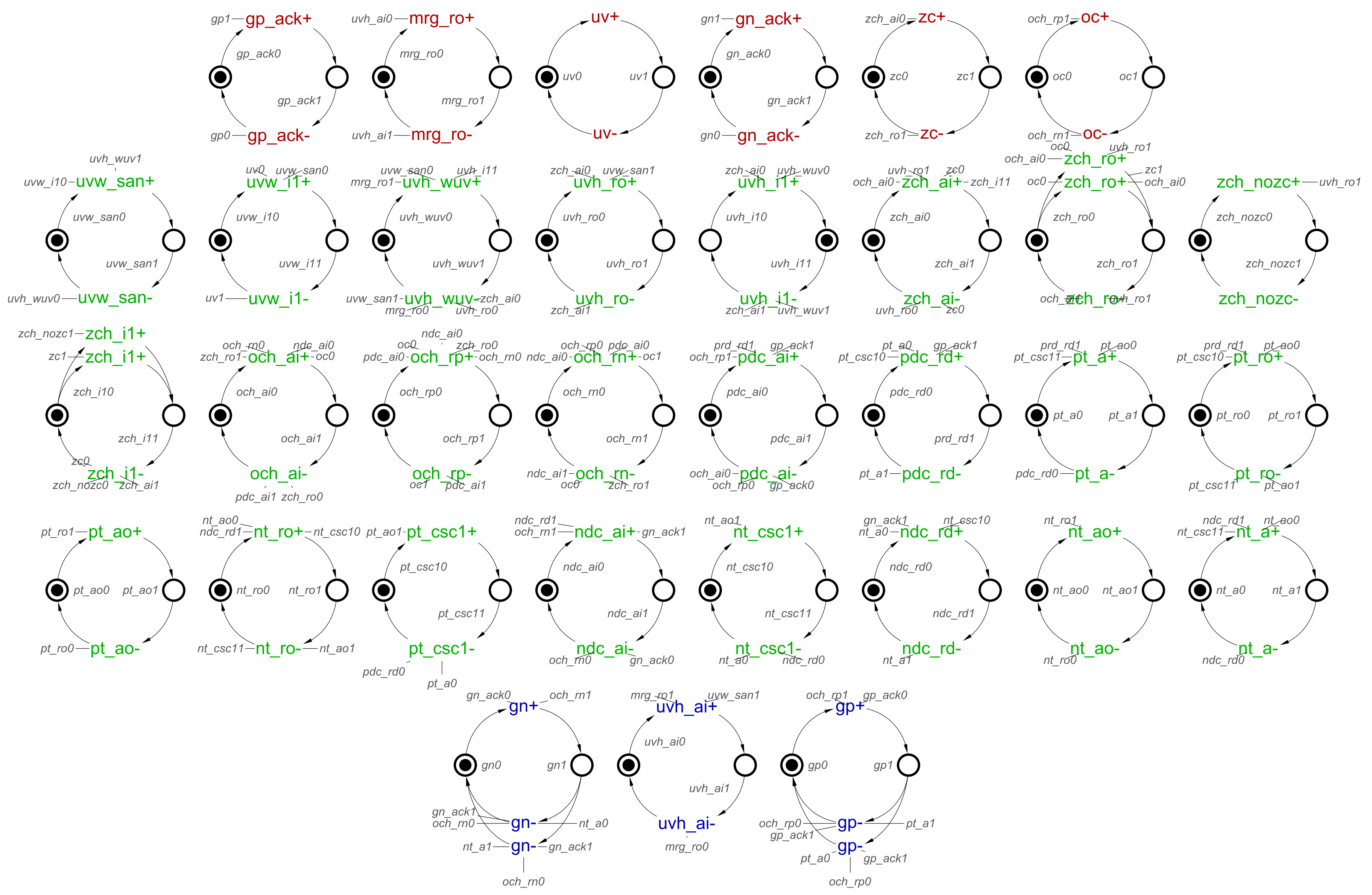


Figure A.26: Translated STG of the charging section.

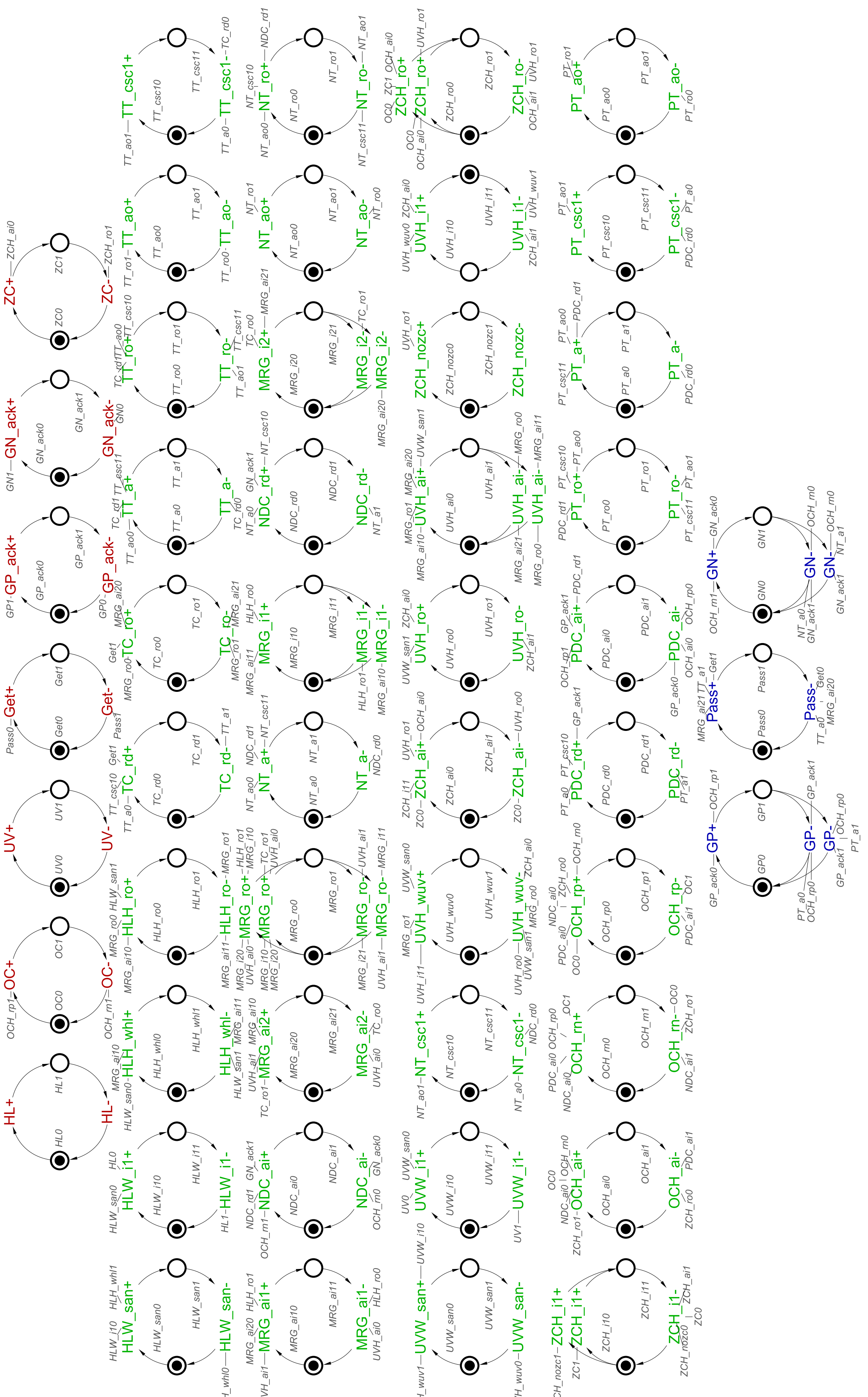


Figure A.27: Translated STG of the full multiphase buck specification.

Bibliography

[1] D. Sokolov, V. Khomenko, A. Mokhov, A. Yakovlev, and D. Lloyd, "Design and verification of speed-independent multiphase buck controller," in *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*. IEEE, 2015, pp. 29–36.

[2] J. A. Brzozowski and C.-J. H. Seger, *Asynchronous circuits*. Springer Science & Business Media, 2012.

[3] (2011) Intel to Acquire Fulcrum Microsystems. [Online]. Available: https://newsroom.intel.com/news-releases/intel-to-acquire-fulcrum-microsystems/

[4] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems." STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1984.

[5] J. Myers, A. Savanth, R. Gaddh, D. Howard, P. Prabhat, and D. Flynn, "A subthreshold arm cortex-m0+ subsystem in 65 nm cmos for wsn applications with 14 power domains, 10t sram, and integrated voltage regulator," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 31–44, Jan 2016.

[6] A. Talbot, "Holistic mixed signal design in ultra deep sub-micron technologies," *NMI R&D Workshop: Analog and Mixed-Signal Design*, 2016.

[7] Y. Lee, Y. Kim, D. Yoon, D. Blaauw, and D. Sylvester, "Circuit and system design guidelines for ultra-low power sensor nodes," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 1037–1042.

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

[9] C. H. V. Berkel, M. B. Josephs, and S. M. Nowick, "Applications of asynchronous circuits," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 223–233, Feb 1999.

[10] T. Chu, C. K. Leung, and T. S. Wanuga, "A design methodology for concurrent vlsi systems," in *Proc. of*, vol. 901, 1985, pp. 407–410.

[11] T.-A. Chu, "Synthesis of self-timed vlsi circuits from graph-theoretic specifications," Ph.D. dissertation, Massachusetts Institute of Technology, 1987.

[12] L. Rosenblum and A. Yakovlev, "Signal graphs: from self-timed to timed ones," *International Workshop on Timed Petri Nets*, pp. 199–206.

[13] J. Beaumont, A. Mokhov, D. Sokolov, and A. Yakovlev, "Compositional design of asynchronous circuits from behavioural concepts," in *ACM-IEEE International Conference on Formal Methods and Models for System Design MEMOCODE15*, June 2015.

[14] J. Beaumont, A. Mokhov, D. Sokolov, and A. Yakovlev, "High-level asynchronous concepts at the interface between analogue and digital worlds," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[15] J. Beaumont, "Plato: a tool for behavioural specification of asynchronous circuits," in *International Conference on Application of Concurrency to System Design (ACSD 2017)*, 2017.

[16] Plato tool repository. [Online]. Available: https://github.com/tuura/plato

[17] 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*. Newcastle University, 2016. [Online]. Available: http://async.org.uk/ay-festschrift/paper25-Alex-Festschrift.pdf

[18] Workcraft framework webpage. [Online]. Available: www.workcraft.org

[19] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic Synthesis for Asynchronous Controllers and Interfaces*. Springer, 2002.

[20] V. Khomenko, "Model checking based on prefixes of petri net unfoldings," 2003.

[21] V. Khomenko and A. Mokhov, "An algorithm for direct construction of complete merged processes," in *International Conference on Application and Theory of Petri Nets and Concurrency*. Springer, 2011, pp. 89–108.

[22] A. Mokhov, J. Carmona, and J. Beaumont, "Mining Conditional Partial Order Graphs from Event Logs," in *Transactions on Petri Nets and Other Models of Concurrency XI*. Springer Berlin Heidelberg, 2016, pp. 114–136.

[23] Pgminer tool repository. [Online]. Available: https://github.com/tuura/process-mining

[24] V. Taraate, *Finite State Machines*. New Delhi: Springer India, 2016, pp. 197–217. [Online]. Available: http://dx.doi.org/10.1007/978-81-322-2791-5_8

[25] A. Gill *et al.*, "Introduction to the theory of finite-state machines," 1962.

[26] A. Valmari, "The state explosion problem," *Lectures on Petri nets I: Basic models*, pp. 429–528, 1998.

[27] C. Petri, "Kommunikation mit automaten (communicating with automata)," Ph.D. dissertation, University of Bonn, 1962.

[28] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr 1989.

[29] M. Kilp, U. Knauer, and A. V. Mikhalev, *Monoids, Acts and Categories: With Applications to Wreath Products and Graphs. A Handbook for Students and Researchers*. Walter de Gruyter, 2000, vol. 29.

[30] B. Pfahringer, *Conjunctive Normal Form*. Boston, MA: Springer US, 2010, pp. 209–210. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_158

[31] B. Pfahringer, *Disjunctive Normal Form*. Boston, MA: Springer US, 2017, pp. 371–372. [Online]. Available: https://doi.org/10.1007/978-1-4899-7687-1_223

[32] A. Alekseyev, I. Poliakov, V. Khomenko, and A. Yakovlev, "Optimisation of Balsa control path using STG resynthesis," in *UK Asynchronous Forum*, 2009.

[33] B. H. Arnold, *Logic and Boolean algebra*. Courier Corporation, 2011.

[34] I. David, R. Ginosar, and M. Yoeli, "An efficient implementation of boolean functions as self-timed circuits," *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 2–11, Jan 1992.

[35] A. Mokhov, V. Khomenko, D. Sokolov, and A. Yakovlev, "On dual-rail control logic for enhanced circuit robustness," in *2012 12th International Conference on Application of Concurrency to System Design*, June 2012, pp. 112–121.

[36] F. Bujtor, S. Fendrich, G. LÃijttgen, and W. Vogler, "Nondeterministic modal interfaces," *Theoretical Computer Science*, vol. 642, no. Supplement C, pp. 24 – 53, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S030439751630247X

[37] A. Mokhov and A. Yakovlev, "Conditional partial order graphs: Model, synthesis, and application," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1480–1493, 2010.

[38] A. Alekseyev, V. Khomenko, A. Mokhov, D. Wist, and A. Yakovlev, "Improved parallel composition of labelled petri nets," in *International Conference on Application of Concurrency to System Design (ACSD)*, 2011, pp. 131–140.

[39] J. d. S. Pedro, T. Bourgeat, and J. Cortadella, "Specification mining for asynchronous controllers," in *2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2016, pp. 107–114.

[40] A. Mokhov, J. Cortadella, and A. de Gennaro, "Process windows," in *2017 17th International Conference on Application of Concurrency to System Design (ACSD)*, June 2017, pp. 86–95.

[41] J. Cortadella, A. Moreno, D. Sokolov, A. Yakovlev, and D. Lloyd, "Waveform transition graphs: A designer-friendly formalism for asynchronous behaviours," in *23rd IEEE International Symposium on Asynchronous Circuits and Systems*, 2017.

[42] D. Wist and R. Wollowski, "Avoiding irreducible csc conflicts in component stgs," in *Proceedings of the 19th UK Asynchronous Forum. Imperial College London*, 2007.

[43] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev, "Basic gate implementation of speed-independent circuits," in *Proceedings of the 31st annual Design Automation Conference*. ACM, 1994, pp. 56–62.

[44] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, *Graphviz—Open Source Graph Drawing Tools*. Berlin, Heidelberg: Springer Berlin

Heidelberg, 2002, pp. 483–484. [Online]. Available: https://doi.org/10.1007/3-540-45848-4_57

[45] D. Sokolov, A. Mokhov, A. Yakovlev, and D. Lloyd, "Towards asynchronous power management," in *IEEE Faible Tension Faible Consommation (FTFC)*, May 2014, pp. 1–4.

[46] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley *et al.*, "Best practices for scientific computing," *PLoS biology*, vol. 12, no. 1, p. e1001745, 2014.

[47] A. Mokhov, D. Sokolov, V. Khomenko, and A. Yakovlev, "Asynchronous arbitration primitives for new generation of circuits and systems," in *2017 New Generation of CAS (NGCAS)*, Sept 2017, pp. 81–84.

[48] A. Mokhov and V. Khomenko, "Algebra of parameterised graphs," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4s, 2014.

[49] A. Mokhov, "Algebra of switching networks," *IET Computers & Digital Techniques*, 2015.

[50] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[51] K. Van Berkel, *Handshake circuits: an asynchronous architecture for VLSI programming*. Cambridge University Press, 1993, vol. 5.

[52] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The vlsi-programming language tangram and its translation into handshake circuits," in *Proceedings of the conference on European design automation*. IEEE Computer Society Press, 1991, pp. 384–389.

[53] G. Jin, L. Wang, and Z. Wang, "A new description language for data-driven asynchronous circuits and its design flow," in *Circuits, Communications and Systems, 2009. PACCS '09. Pacific-Asia Conference on*, May 2009, pp. 322–325.

[54] S. Burns and A. Martin, "A synthesis method for self-timed vlsi circuits," in *Proceedings of the International Conference on Computer Design*, 1987.

[55] A. Martin, "Compiling communicating processes into delay-insensitive vlsi circuits," *Distributed Computing, vol. 1(4)*, pp. 226–234, 1986.

[56] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[57] A. J. Martin and C. D. Moore, "Chp and chpsim: A language and simulator for fine-grain distributed computation," Caltech Technical Report CS-TR-1-2011, Tech. Rep., 2011.

[58] J. Kuper and C. Baaij, "Hardware specification with cλash," *DSL 2013*, 2013.

[59] C. Baaij, "Cλash : from Haskell to hardware," December 2009. [Online]. Available: http://essay.utwente.nl/59482/

[60] A. Mokhov, "Conditional partial order graphs," Ph.D. dissertation, Newcastle University, 2009.

[61] 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, no. 6, pp. 1552–1566, 2014.

[62] A. Mokhov, D. Sokolov, and A. Yakovlev, "Adapting asynchronous circuits to operating conditions by logic parametrisation," *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, pp. 17–24, 2012.

[63] M. Josephs and J. Udding, "An overview of d-i algebra," in *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. i, Jan 1993, pp. 329–338 vol.1.

[64] A. V. Yakovlev, A. M. Koelmans, and L. Lavagno, "High-level modeling and design of asynchronous interface logic," *IEEE Design Test of Computers*, vol. 12, no. 1, pp. 32–40, Spring 1995.

[65] P. Lam and H. Li, "Hierarchical design of delay-insensitive systems," *Computers and Digital Techniques, IEE Proceedings E*, vol. 137, no. 1, pp. 41–56, 1990.

[66] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ACM SIGPLAN Notices*, vol. 34, no. 1. ACM, 1998, pp. 174–184.

[67] P. Beerel, G. Dimou, and A. Lines, "Proteus: An asic flow for ghz asynchronous designs," *Design Test of Computers, IEEE*, vol. 28, no. 5, pp. 36–51, Sept 2011.

[68] G. Gill and M. Singh, "Automated microarchitectural exploration for achieving throughput targets in pipelined asynchronous systems," in *Asynchronous Circuits and Systems (ASYNC), 2010 IEEE Symposium on*, May 2010, pp. 117–127.

[69] I. Benko and J. Ebergen, "Composing snippets," in *Concurrency and Hardware Design*, ser. Lecture Notes in Computer Science, J. Cortadella, A. Yakovlev, and G. Rozenberg, Eds. Springer Berlin Heidelberg, 2002, vol. 2549, pp. 1–33. [Online]. Available: http://dx.doi.org/10.1007/3-540-36190-1_1

[70] C. Armenti, "Get to market faster with modular circuit design," *Electronic Engineering Journal*, 2015. [Online]. Available: http://www.eejournal.com/archives/articles/20150122-zuken/

[71] A. Yakovlev, P. Vivet, and M. Renaudin, "Advances in asynchronous logic: From principles to gals amp; noc, recent industry applications, and commercial cad tools," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 1715–1724.

[72] R. B. Reese, S. C. Smith, M. Thornton *et al.*, "Uncle-an rtl approach to asynchronous design," in *Asynchronous Circuits and Systems (ASYNC), 2012 18th IEEE International Symposium on*. IEEE, 2012, pp. 65–72.

[73] 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*, March 2017, pp. 1751–1756.

[74] D. Sokolov, A. de Gennaro, and A. Mokhov, "Reconfigurable asynchronous pipelines: from formal models to silicon," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018.

[75] S. Golubcovs, D. Shang, F. Xia, A. Mokhov, and A. Yakovlev, "Modular approach to multi-resource arbiter design," in *Asynchronous Circuits and Systems, 2009. ASYNC'09. 15th IEEE Symposium on*. IEEE, 2009, pp. 107–116.

[76] A. Mokhov, V. Khomenko, and A. Yakovlev, “Flat arbiters,” *Fundamenta Informaticae*, vol. 108, no. 1-2, pp. 63–90, 2011.

[77] Copter prototype repository. [Online]. Available: https://github.com/gtarawneh/copter