Automated Synthesis of Speed-Independent Circuits using Distributed Finite State Machines

Alex Kai Yin Chan (140345458)

A thesis presented for the degree of

Doctor of Philosophy

μSystems Research Group, School of Engineering

Newcastle University

United Kingdom

2022/23

Contents

List of Figures

List of Tables

Abbreviations

AMS	Analogue and Mixed Signal[1]
BA	Burst Automaton[2]
BAs	Burst Automatons (or Burst Automata)[3]
BM	Burst-Mode[4]
CAD	Computer-Aided Design[5]
CHASM	Coding for Hazard-free Asynchronous State Machines[6]
CSC	Complete State Coding[7]
DC	Direct Current[8]
DFSM	Distributed Finite State Machine[9]
DMA	Direct Memory Access[10]
DME	Distributed Mutual Exclusion[11]
DI	Delay Insensitive[12]
FIFO	First-In First-Out[13]
FSM	Finite State Machine[14]
OC	Over Current[15]
MIC	Multiple Input Change[16]
QDI	Quasi-Delay Insensitive[17]
SCSI	Small Computer System Interface[18]
SI	Speed-Independent[19]
SIC	Single Input Change[20]

SoCs	Systems-on-Chip[21]
STG	Signal Transition Graph[22]
UV	Under Voltage[23]
VME	Versa Module Europa[24]
WTG	Waveform Transition Graph[25]
XBM	Extended Burst-Mode[26]
ZC	Zero Crossing[27]

[1] AMS, Analogue and Mixed Signal, first appearance on page 1.

[2] BA, Burst Automaton, first appearance on page xiii.

[3] BAs, Burst Automatons (or Burst Automata), first appearance on page xiii.

[4] BM, Burst Mode, first appearance on page xiii.

[5] CAD, Computer-Aided Design, first appearance on page 6.

[6] CHASM, Coding for Hazard-free Asynchronous State Machines, first appearance on page 31.

[7] CSC, Complete State Coding, first appearance on page 6.

[8] DC, Direct Current, first appearance on page 126.

[9] DFSM, Distributed Finite State Machine, first appearance on page 95.

[10] DMA, Direct Memory Access, first appearance on page 20.

[11] DME, Distributed Mutual Exclusion, first appearance on page 36.

[12] DI, Delay Insensitive, first appearance on page 10.

[13] FIFO, First-in First-out, first appearance on page 20.

[14] FSM, Finite State Machine, first appearance on page xiii.

[15] OC, Over Current, first appearance on page 126.

[16] MIC, Multiple Input Change, first appearance on page 9.

[17] QDI, Quasi-Delay Insensitive, first appearance on page 6.

[18] SCSI, Small Computer System Interface, first appearance on page 20.

[19] SI, Speed-Independent, first appearance on page 6.

[20] SIC, Single Input Change, first appearance on page 9.

[21] SoCs, Systems-on-Chip, first appearance on page 1.

[22] STG, Signal Transition Graph, first appearance on page xiii.

[23] UV, Under Voltage, first appearance on page 126.

[24] VME, Versa Module Europa, first appearance on page 7.

[25] WTG, Waveform Transition Graph, first appearance on page 5.

[26] XBM, Extended Burst-Mode, first appearance on page 5.

[27] ZC, Zero Crossing, first appearance on page 126.

Acknowledgements

Firstly, I would like to thank my supervisor, Alex Yakovlev, who has been incredibly supportive and motivating throughout my studies, while also teaching me a lot about asynchronous circuits and how to become a better researcher. I would also like to thank Danil Sokolov, who helped me improve my programming skills and had always taken time to chat with me and help with my research worries, and Victor Khomenko, who introduced me to the world of asynchronous circuits and helped me develop as both an electrical engineer and a computer scientist. I greatly appreciate all of their support and their patience with me throughout the years.

Next, I would like to give a special thanks to David Lloyd for always giving a helping hand with my research and project approvals.

I also would like to thank my friends and colleagues at the μSystems group for an enjoyable PhD experience. Especially Ibrahim Haddadi, Yu Liu and Sergey Mileiko for always being there to have a chat with me. Also another thanks to my friends at the SAgE IT team, who were always happy to support me even after my placement.

Penultimately, I would like to thank all of my friends who have been supportive of me throughout my studies, especially Joseph Birks who always taken the time to checkup on me and has been there to have fruitful chats.

Finally, I would like to thank my parents and my sister, for all of their love and support throughout my lifetime and throughout this PhD, where, even at the toughest times, they have always been there with me at every step of my journey.

This research and my PhD studentship was funded by Dialog Semiconductor.

Abstract

Asynchronous circuits are a promising, yet intricate, type of digital circuit that offers higher performance and lower power consumption than their synchronous counterpart. However, asynchronous circuits still see limited usage in today's commercial products, which often is linked to the adaptation challenges that are posed by industry, e.g. the time required for developing new tools and training circuit designers versus existing synchronous-based tools for a faster production line.

Several formal models were introduced to aid with asynchronous circuit design. In particular, the 'legacy' approach of Burst-Mode (BM) Specifications and the 'disruptive' approach of Signal Transition Graphs (STGs). On one hand, BM specifications resemble synchronous Finite State Machines (FSMs) allowing circuit designers to easily adapt and use them, but there is no longer support provided for their tools. On the other hand, STGs have access to state-of-the-art tools that produce well-optimised circuits, yet they are seen as too complicated compared to FSMs.

In this thesis, a new model called Burst Automaton (BA) is proposed. BA is a generic FSM-based model that acts as a framework for enabling interoperability between many models including BM specifications and STGs. BA offers a new design path that bridges the gap between 'legacy' and 'disruptive' approaches, granting circuit designers access to state-of-the-art tools for higher quality implementations without costing their familiarity with FSMs. Thus, removing any adaptation requirements. This design path is implemented as a new Workcraft plugin that supports the design automation of BAs, and is evaluated on some case studies.

List of Publications and Demos

Journal article:

A. Chan, D. Sokolov, V. Khomenko, D. Lloyd and A. Yakovlev, "Burst Automaton: Framework for Speed-Independent Synthesis Using Burst-Mode Specifications," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 42, no. 5, pp. 1560-1573, May 2023, doi: 10.1109/TCAD.2022.3206732.

Conference papers:

A. Chan, D. Sokolov, V. Khomenko, D. Lloyd and A. Yakovlev, "Synthesis of SI Circuits from Burst-Mode Specifications", Design, Automation & Test in Europe (DATE) Conference & Exhibition, 2021, pp. 366-369, doi: 10.23919/DATE51398.2021.9474117.

A. Chan, D. Sokolov, V. Khomenko and A. Yakovlev, "Formal Modelling of Burst-Mode Specifications in a Distributed Environment", Forum on Specification & Design Languages (FDL), 2022, pp. 1-8, doi: 10.1109/FDL56239.2022.9925657.

A. Chan, D. Sokolov, V. Khomenko, and A. Yakovlev, "Bridging the Design Methodologies of Burst-Mode Specifications and Signal Transition Graphs," Accepted for Asia and South Pacific Design Automation Conference (ASP-DAC) 2024.

Technical Demos:

A. Chan, D. Sokolov, V. Khomenko and A. Yakovlev, "Design Automation for Extended Burst-Mode Automata in Workcraft", Design, Automation & Test in Europe (DATE) Conference & Exhibition, 2020 and 2021, University Booth.

Chapter 1

Introduction

The role of digital systems has seen a significant change over the past century, where they were once large sophisticated integrated circuits used in industrial-sized computer systems and are now part of today's smaller, yet efficient, systems-on-chip (SoCs) used in everyday smart phones.

Typically, digital systems are designed using synchronous circuits, where the circuits rely on the use of a global clock signal to synchronise their components at fixed time intervals. However, synchronous circuits also come with several drawbacks, as their operations are forced to wait for one another, even if most of them may already be completed, and that they must always be active to receive any input changes, which leads to more power consumption.

An alternative to design digital circuits are asynchronous circuits, where the circuits remove the use of this global clock signal and instead rely on the local synchronisation between their components [63]. This results in a higher performance as their operations may now be performed as soon as input changes are received, and lower power consumption as they do not have to always be active [52].

Additionally, they also avoid clock-related issues like clock skew [13] as they do not have a clock signal, and their design methodology has even been used in other areas of applications, such as analogue and mixed signal (AMS) systems [61], image sensors [8], machine learning [40] and memory compilers [10].

However, despite the promising aspects about asynchronous circuits, they are still not widely adopted in industry. This is often linked to circuit designers, who are already familiar with the synchronous design methodology, where there is a perceived convention of using state-based methods like Finite State Machines (FSMs) to design digital circuits within the synchronous community.

In particular, there are some asynchronous approaches that even cater towards the circuit designer's familiarity of FSMs. For example, Burst-Mode (BM) specifications [55] are a type of asynchronous FSM that was introduced to (synchronous) circuit designers as a simple entry into asynchronous circuit design, due to the BM specifications' resemblance to synchronous FSMs.

Oppositely, the asynchronous design methodology uses event-based methods to design digital circuits graphically using vertices and arcs, where these connections are used to express many types of behaviours and the circuit designers are required to understand the effects of these causal events, e.g. how firing one signal transition may enable multiple other signal transitions for firing.

One prominent event-based method is Signal Transition Graphs (STGs) [58, 18], which are a type of labelled Petri net [57] that specifies asynchronous circuits and determines the causality of systems, based on its flow of tokens from places to transitions and from transitions to places.

Nevertheless, these event-based methods are seen as too different by the industry when compared to their conventional state-based methods. This suggests that many circuit designers are likely too unfamiliar with STGs and would require training, which can be costly and time consuming. Instead, the industry would much rather opt for a more familiarised approach like BM specifications, which reduces the need for training and helps them quickly meet market demands.

To address the issues that are described above, this thesis introduces a new model

called Burst Automaton (BA) that provides the necessary framework to enable interoperability between many models including BM specifications and STGs. Here, a new 'co-design' route can be established, where circuit designers can specify their BM specification and translate it into a BA to gain access to the STG's well-established tools for subsequent composition, verification, and synthesis.

Notably, this new design route allows circuit designers, who are already familiar with synchronous FSMs and BM specifications, to transition into the STG's world of asynchronous circuit design, where all the circuit designer's unfamiliarities with STGs are 'stripped' away, and hidden behind the BA's design flow that is automated and implemented as a WORKCRAFT [2] plugin.

Thus, in this chapter, we will cover the current design routes for asynchronous circuit design, which involve the 'legacy' route of BM specifications and the 'disruptive' route of STGs, as well as the motivation of this thesis that briefly highlights the need and benefit of this new 'co-design' route of BAs. We will then discuss the contributions of this thesis and how they will aid (unfamiliar) circuit designers with the design of asynchronous circuits, before we show the structure of this thesis, where we will discuss the main topics of each chapter.

1.1 Introductory Motivation

To give us an insight into the motivation of this thesis, let us consider a simplified view of the current design routes shown in Figure 1.1, where we have three design routes that are the 'legacy' route, the 'disruptive' route, and the 'co-design' route.

In the 'legacy' route, BM specifications are used. Here, this design route can be seen as the familiarised route for many circuit designers, as BM specifications are a state-based method that the industry prefers and can be synthesised with a tool like

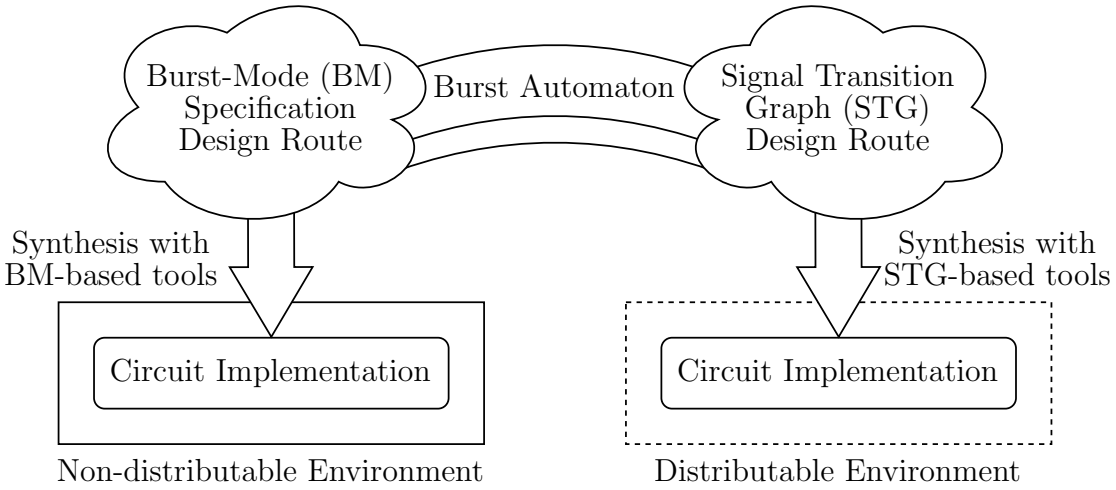


Figure 1.1: Abstracted view of design routes

MINIMALIST [54] to produce some asynchronous BM circuit implementation.

However, support for these BM-based tools has become limited and their circuit implementations may no longer be well-optimised, as these tools are slowly becoming outdated. Additionally, the circuit implementations produced by the BM synthesis tools also do not work with a distributed environment, as the environment is assumed to be 'centralised' and cannot be easily split, due to the BM specification's inherent timing assumption. Moreover, the design of these BM specifications are also not modular, as there are no tools that support the composition of BM specifications, meaning circuit designers must rely on the monolithic approach to specify their systems.

In the 'disruptive' route, STGs are used. Here, this design route can be considered the optimised route for asynchronous circuits, where STGs are supported by well-established tools like PETRIFY [20] and MPSAT [34], which can produce well-optimised circuit implementations that work well with a distributed environment, due to the expressiveness of STGs. Furthermore, STGs are also easily composable, allowing circuit designers to follow a modular approach for specifying their systems.

But, as mentioned above, circuit designers are unfamiliar with STGs as they are

an event-based method, meaning circuit designers will instead opt for the state-based methods like BM specifications to specify their systems.

Thus, we introduce the 'co-design' route that is established by using the BA model, where a 'bridge' is formed between the gap of the 'legacy' route and the 'disruptive' route. In this 'co-design' route, BA allows circuit designers to specify their systems using BM specifications, where they can then gain access to the STG's well-established tool by translating their BM specifications to STGs via BA.

A more technical view of the motivation is covered in Chapter 3, where an example that highlights the issues described for the 'legacy' route and the 'disruptive' route is shown, as well as a new design workflow that involves our 'co-design' route.

1.2 Thesis Contributions

The major contributions of this thesis are as follows:

Burst Automaton model [16]: A new formal model that provides the necessary framework for enabling interoperability between many models including BM specifications, Extended BM (XBM) specifications [68], STGs, FSMs, or generally any formalism that can be automatically translated into any of these models, e.g. Waveform Transition Graphs (WTGs) [50] which have STG-based semantics.

Translation methods from BAs to STGs: Three separate translations that each preserve the language, weak bisimulation, and strong bisimulation, to the original BA, as well as another method for translating the XBM specification's components (e.g. conditionals and "don't cares") to their STG counterparts.

Automated design flow based on the Burst Automaton's 'co-design' route: A new design route that is established between BM specifications and STGs, where (X)BM specifications can be granted access to the STG's well-established tools

for subsequent composition of BM specifications, verification, and synthesis of speed-independent (SI) (quasi-delay insensitive (QDI)) asynchronous circuits.

Implementation of the automated design flow as a Workcraft plugin: A new plugin that supports the design automation of BAs and (X)BM specifications featuring a graphical-based design, automated translation to STGs, composition, simulation, verification, and synthesis with Petrify and Mpsat backends.

1.3 Thesis Structure

This thesis is organised as follows:

Chapter 1 Introduction. In this chapter, we briefly discuss the motivation of this thesis and summarise the contributions, which include the BA model, the translation methods from BAs to STGs, the design flow based on the BA's design route that is established between BM specifications and STGs, and the Workcraft plugin that supports the design automation of BAs and (X)BM specifications.

Chapter 2 Background. This chapter discusses the background of asynchronous circuits, where we cover their delay models, circuit operation modes, and classes. Next, we cover the existing formal models that are used to design asynchronous circuits in greater detail, including FSMs, BM specifications, XBM specifications, Petri nets and STGs, as well as the model equivalences that are used to check the equivalence relation between two given models. Finally, we introduce the existing Computer-Aided Design (CAD) tools that are available for the design, verification and synthesis of (X)BM specifications and STGs.

Chapter 3 Technical Motivation. This chapter provides the technical motivation of this thesis, where we cover the motivation in greater depth by providing an example that is based on a handshake decoupling system, and showing the designs

that are created with the 'legacy' route and the 'disruptive' route. We then cover the new design route that is proposed by BA, where we show how BAs are used to establish the connection between BM specifications and STGs.

Chapter 4 Burst Automata. This chapter introduces the BA model, where we cover its formal definition and its defined asynchronous state graph. Next, we cover the three translation methods from BAs to STGs, where each translation is shown to preserve the language, weak bisimulation, and strong bisimulation between the BA and STG, before we then cover the translation method of the XBM specification's components to their STG counterparts. Lastly, we show how BAs can be composed by translating them to STGs, using one of the three aforementioned methods, via parallel composition before it is synthesised into an asynchronous circuit.

Chapter 5 Design Automation. This chapter discusses the implemented Workcraft plugin that supports the design automation of BAs, BM specifications, and XBM specifications, where we will show the automated design flow and the features of the plugin. We will then cover the experimental results of this Workcraft plugin, where we analyse the size growth of the BA to STG translation, and compare the literal counts between the BM synthesis tools and the STG synthesis tools.

Chapter 6 Case Studies. This chapter covers two case studies involving the design of the buck converter [61] and the design of the Versa Module Europa (VME) bus controller [1] where, in each case study, we will explain their operations and show how they can be specified using STGs. We will then identify the issues posed for BM specifications when specifying these systems, before we show how BAs can be used as an alternative to specify these systems to achieve the same results as STGs.

Chapter 7 Conclusion. This chapter provides a summary of the contributions in this thesis, and discuss future research and development to be considered for BAs and the implemented Workcraft plugin.

Chapter 2

Background

In this chapter, we will cover the background of this thesis, where we will first explore asynchronous circuits in further detail including their delay models, circuit operation modes and classes in Section 2.1.

Next, we will study several formal models with a particular focus on Finite State Machines (FSMs), Burst-Mode (BM) specifications, Extended BM (XBM) specifications, Petri nets and Signal Transition Graphs (STGs) in Section 2.2, before we investigate the model equivalences including strong bisimulation, weak bisimulation, language equivalence and output determinacy in Section 2.3.

Finally, we will analyse some existing tools that are available for (X)BM specifications and STGs in Section 2.4, where we will review the BM-based tools Minimalist, 3D and BM Decomp, and the STG-based tools Petrify, Mpsat, Pcomp and Workcraft.

2.1 Asynchronous Circuits

Asynchronous circuits are a promising type of digital circuit that removes the use of a global clock signal in favour of local synchronisation between its components [63] offering benefits like higher performance, robustness to variability conditions and lower power consumption [52], while resolving clock-related issues like clock skew [13].

In particular, asynchronous circuits and their circuit operation mode can be classified based on the timing assumptions, which are made about the delays between their components and their interaction with the environment.

2.1.1 Delay Models

At the hardware level, asynchronous circuits can be seen as a connection of *wires* between *gates* and *delay elements*, where each wire is connected from the output of a gate (or delay element) to the input of one or more gates (or delay elements). The gates compute a set of outputs using their input, while the delay elements produce a single output delayed by their input. In particular, the delay elements can be categorised based on the timing model that they use. These include:

- A *fixed delay* model where the delay has a fixed value.
- A *bounded delay* model where the delay has a value within a given time interval.
- An *unbounded delay* model where the delay has an arbitrary finite value.

2.1.2 Circuit Operation Modes

The *circuit operation mode* can be described as the interaction between a *device* and an *environment*. The most common circuit operation mode is the *generalised fundamental mode* [39, 21], where the environment is assumed to wait for the circuit to stabilise before it can produce new inputs.

Notably, fundamental mode can be divided into two separate sub-classes called *single input change* (SIC) mode and *multiple input change* (MIC) mode. The former mode forces the inputs to be sequential but restricts the circuit's operation speed,

while the latter mode allows one or more inputs to change after the circuit stabilises but makes the circuit more difficult to implement.

As a result, the *burst-mode timing assumption* [55] is introduced as a trade-off between the two aforementioned modes. In this timing assumption, signals are allowed to change in groups called *bursts*, where these signals may arrive in any order and time. Each burst consists of a non-empty set of inputs that precedes a set of outputs, such that the circuit waits until a complete input burst has arrived from the environment before producing any output bursts and transitioning to a different state. In turn, the environment waits until the circuit has produced its output burst, before it can send any new input bursts. This means that bursts cannot easily capture the concurrency between inputs and outputs, as the firing of input bursts and firing of output bursts are mutually exclusive.

Another well-known circuit operation mode is *input-output mode* [39, 21], where the environment is allowed to respond to a device's outputs without any timing constraints. Unlike fundamental mode and the 'burst-mode' timing assumption, input-output mode enables concurrency between inputs and outputs, allowing many types of concurrent asynchronous circuits like distributed systems to be produced.

2.1.3 Classes of Asynchronous Circuits

Asynchronous circuits follow the same design principles as *huffman* circuits [30], where huffman circuits are designed to work correctly in fundamental mode and a bounded delay is assumed for both gates and wires. In particular, several classes of asynchronous circuits are created and can be identified as the following:

- *Delay insensitive* (DI) circuits [19, 66], which are a set of asynchronous circuits that have been designed to work correctly in input-output mode with an un-

bounded gate and wire delay. In [42], it is shown that only a few DI circuits can be built if the user can only use simple gates. However, many practical DI circuits can also be built using *complex gates* [26], which are constructions of several simple gates that operate in a DI manner and rely on some timing assumptions.

- *Speed-independent* (SI) circuits [51], which are another set of asynchronous circuits that have been designed to work correctly in input-output mode regardless of gate delays and all wire delays are assumed to be negligible (or shorter than any gate delay). In *Quasi-delay-insensitive* (QDI) circuits [41], the assumption about wire delays is relaxed by requiring that (some) wire forks are isochronic, meaning the maximal difference in delays from the root of the fork to the ends of its branches is negligible (or shorter than any gate delay). Intuitively, the isochronic fork assumption means that the fork can be characterised by a single delay, which conceptually can be appended to the driving gate's delay. Thus, this makes QDI circuits very similar to SI circuits.

- *Self-timed* circuits [60], which are a set of asynchronous circuits that have been built using a group of elements, where each element may be an SI circuit or a circuit that relies its operation on local timing assumptions. However, there were no timing assumptions made on the communication between the elements and the circuit operating in input-output mode. If both internal and external timing assumptions were used to optimise the design of these self-timed circuits, then the circuit would instead be classified as a timed circuit [53].

2.2 Formal Models

2.2.1 Finite State Machines

Finite State Machines (FSMs) [28], also known as Finite State Automata (FSA), are abstract mathematical models used to specify and analyse sequential systems. FSMs can be classified as either *Moore machines* [48] or *Mealy machines* [44], where the former produces outputs on states and is determined by the current state rather than by inputs, while the latter produces outputs on transitions rather than on states.

FSMs can be in one of many possible finite states at any given time, and can transition to another state in response to some given *word* from the *alphabet*. The alphabet is a finite set of atomic symbols and a word from the alphabet Σ is a finite sequence of symbols from Σ, where ε denotes the *empty set* (i.e. $\varnothing$) and Σ^* denotes the set of all words over Σ (including ε).

In order to specify concurrent systems, FSMs must allow concurrency between atomic actions to happen in any order by using *interleaving*, which allows all possible actions to take place in any order and forms a 'diamond' shape in the state graph.

FSMs consist of a finite set of states, a finite alphabet, a set of transitions, and an initial state, such that each transition determines the flow of a system by its connection between a source state and a target state.

Formally, FSMs can be defined as a tuple $M = (Q, \Sigma, A, q_0)$ where:

- Q is the finite set of states.
- Σ is the finite alphabet.
- $A \subseteq Q \times (\Sigma \cup \varepsilon) \times Q$ is the set of transitions.
- $q_0 \in Q$ is the initial state.

Notably, the behaviour of FSMs can be also captured by their language and reachable states, where:

- A word w over Σ is accepted by an FSM if $q_0 \xrightarrow{w} q$ for some state $q \in Q$.
- The language (accepted or generated) by an FSM is the set $\mathcal{L}$(FSM) of all words accepted by it, and $\mathcal{L}$(FSM) is prefix-closed.
- A state $q \in Q$ is *reachable* if $q_0 \xrightarrow{w} q$ for some string w.
- A state $q \in Q$ is called a *deadlock* if it has no exit arcs.
- An FSM is said to be *deadlock-free* if none of its reachable states is a deadlock i.e. a reachable state that contains no outgoing arcs.
- An FSM is said to be *reversible* if q_0 can be reached from any reachable state q i.e. the FSM can eventually return to initial state.
- An FSM is said to be *deterministic*, if there are no transitions that labels an empty action (i.e. ε) and that there are no two transitions, which reach different states using the same action (i.e. transitions (q, a, q') and (q, a, q'') such that $q' \neq q''$). Otherwise, the FSM is *non-deterministic*.
- If an FSM has no unreachable states then it is reversible if, and only if, its graph is *strongly connected* i.e. there is a directed path between any pair of its states.

In the graphical model of FSMs, states are drawn as circles and transitions are drawn as arcs (i.e. arrows) such that the initial state is expressed with an arrow pointing at it. For example, Figure 2.1 shows an FSM specifying a vending machine that sells cokes and chocolates [5].

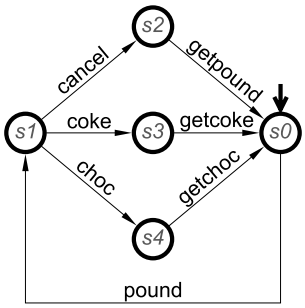


Figure 2.1: FSM of a vending machine selling cokes and chocolates

2.2.2 Burst-Mode and Extended Burst-Mode Specifications

In circuit design, there are *synchronous FSMs* and *asynchronous FSMs*. The former model uses a global clock signal to synchronise its components, while the latter model relies on the local synchronisation between its components in the form of *handshakes*.

Traditionally, many circuit designers are familiar with synchronous circuits and use synchronous FSMs to design their specifications, due to the ease of using FSMs and the industry's perceived convention of using the synchronous design methodology. Though, this design approach requires some assumption that every component works well through a common clock and that clock-related issues do not occur, e.g. clock skew as previously described in Section 2.1.

Alternatively, there have been early works to help adopt the use of asynchronous FSMs in industry, where these FSMs operate in either SIC mode or MIC mode. However, as previously described in Section 2.1.2, both aforementioned modes have a drawback, i.e. SIC mode restricted the speed of the circuit's operation due to its sequential inputs and MIC mode makes a circuit more difficult to implement due to its enabling of multiple input changes. Thus, this leads to the introduction of *Burst-Mode* (BM) specifications [55].

BM specification is a variant of an asynchronous FSM that operates in the 'burst-

mode' timing assumption, where it offers (synchronous) circuit designers a simple entry into asynchronous design due to its similarity with synchronous FSMs.

Like traditional FSMs, BM specifications can be in one of many possible finite states, and can transition to another state after receiving an input burst followed by producing an output burst.

Additionally, BM specifications consist of a finite set of states, a set of inputs, a set of outputs, a set of transitions, an initial state, and two labelling functions that defines the values of its inputs and outputs respectively.

Formally, BM specifications can be defined as a directed graph $G = (V, E, I, O, v_0, in, out)$ where:

- V is a finite set of states.
- $E \subseteq V \times V$ is the set of transitions.
- $I = \{x_1, ..., x_m\}$ is the set of inputs.
- $O = \{z_1, ..., z_n\}$ is the set of outputs.
- $v_0 \in V$ is the initial state.
- $in : V \rightarrow \{0, 1\}^m$ is the labelling function that defines the values of m inputs at the unique entry point of each state, such that $in_i(v)$ denotes the value of x_i entering state v.
- $out : V \rightarrow \{0, 1\}^n$ is the labelling function that defines the values of n outputs at the unique entry point of each state, such that $out_j(v)$ denotes the value of z_j entering state v.

In particular, the labelling functions $trans_i$ and $trans_o$ can also be derived from the BM specification G, where $trans_i : E \rightarrow \mathcal{P}(I)$ defines the set of input changes

(i.e. the input burst) and $trans_o : E \rightarrow \mathcal{P}(O)$ defines the set of output changes (i.e. the output burst), such that $\mathcal{P}(I)$ and $\mathcal{P}(O)$ defines the set of all possible subsets from I and O respectively.

Furthermore, a BM specification must also be well-formed by satisfying the following requirements that include the:

- *Unique entry condition*, where each state must always be entered with the same set of input and output values.

  Formal Definition: For each state v, it is always entered with the input value $in(v)$ and output value $out(v)$. Note that the formal definition of graph G automatically satisfies this requirement, as stated in [55].

- *Non-empty input burst property*, where a burst must contain at least one input in its set of inputs.

  Formal Definition: For every transition $e \in E$, $trans_i(e) \neq \varnothing$. Note that this is enforced by the labelling functions $trans_i$ and $trans_o$.

- *Maximal set property*, where the set of inputs from one burst are not a subset of the set of inputs from another burst, which originate from the same state.

  Formal Definition: For each pair of transitions $(u, v), (u, w) \in E$, $trans_i(u, v) \subseteq trans_i(u, w) \rightarrow v = w$. Note that this is also enforced by the labelling functions $trans_i$ and $trans_o$.

In the graphical model of BM specifications, they are drawn exactly the same as FSMs with the exception that states are labelled with its defined encoding values of both inputs and outputs, and the word w contains both the input burst and output burst that are separated by a slash symbol ('/'). For example, Figure 2.2 shows a BM specification specifying the C-element gate.

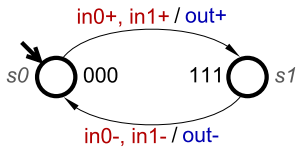


Figure 2.2: BM specification of C-element

However, as previously described in Section 2.1.2, BM specifications cannot express concurrency between inputs and outputs due to their inherent 'burst-mode' timing assumption, where the circuit waits until a complete input burst arrives before producing any outputs and the environment waits until a complete output burst completes before producing any inputs.

As a result, the *Extended Burst-Mode* (XBM) [68] specification was proposed to address BM specification's inability to express input-output concurrency by introducing *conditionals* and *directed "don't cares"*. Note that for the remainder of this thesis, we will call directed "don't cares" as simply "don't cares" unless we need to distinguish them from terminating "don't cares".

Conditionals are level-sensitive inputs that determine the system's control flow based on their sampled value, while "don't cares" are *monotonic signals* (i.e. signals that may only change once) where inputs can change concurrently with outputs.

When a conditional is sampled, they must be stabilised once a *compulsory edged input* (i.e. inputs that are not "don't cares" in the previous burst) appears and hold their value until all subsequent outputs are produced. The minimum delay between the conditional stabilising and the first compulsory input is called the *setup time*, while the minimum delay between the last terminating output and the conditional change is called the *hold time*. Thus, the period between the setup time and hold time is called the *sampling period*.

Formally, XBM specifications can be defined as a directed graph $G' = (G, C, cond)$ where:

- $G = (V, E, I, O, v_0, in, out)$ is a BM specification.
- $C = \{c_1, ..., c_l\}$ is the set of conditional inputs, such that $C \cap I = \varnothing$.
- $cond : E \rightarrow \{0, 1, *\}^l$ is the labelling function that labels each state transition with a set of conditionals, defining the sampled value of each conditional.

Additionally, the labelling functions $trans_{IN} : E \rightarrow \mathcal{P}(I)$ and $trans_{OUT} : E \rightarrow \mathcal{P}(O)$ are implicitly derived for the XBM specification G' as it is an extension to G. Thus, given some state transition $(u, v) \in E$:

- If $in_i(u) \neq in_i(v) \vee in_i(v) = *$ then input $x_i \in trans_{IN}(u, v)$ such that:
  - x_i+ is in input burst, if $in_i(v) = 1 \wedge in_i(u) \neq 1$.
  - x_i- is in input burst, if $in_i(v) = 0 \wedge in_i(u) \neq 0$.
  - x_i* is in input burst, if $in_i(v) = *$.
- If $out_j(u) \neq out_j(v)$ then output $z_j \in trans_{OUT}(u, v)$ such that:
  - z_j+ is in output burst, if $out_j(v) = 1 \wedge out_j(u) = 0$.
  - z_j- is in output burst, if $out_j(v) = 0 \wedge out_j(u) = 1$.

Moreover, the labelling function $ctrans_{IN}(u, v)$ can also be derived from the XBM specification G', where $ctrans_{IN}(u, v) = \{x_i \in trans_{in}(u, v) \mid in_i(u) \neq * \wedge in_i(v) \neq *\}$ defines the set of compulsory input changes, such that x_i does not start or end as a "don't care" for each input $x_i \in trans_{in}(u, v)$.

Like its BM specification predecessor, the XBM specification must also be well-formed by satisfying the BM well-formed requirements (excluding the maximal set property) and the additional XBM well-formed requirements, which include the:

- *Distinguishability constraint*, which extends BM's maximal set property, where bursts that originate from the same state must either have unique conditional values or their set of inputs are not a subset of another burst with consideration of "don't care" transitions.

  Formal Definition: For every pair $(u, v), (u, w) \in E$, function $ctrans_{IN}(u, v) \subseteq trans_{IN}(u, w)$ implies that either $v = w$, or $cond(u, v)$ and $cond(u, w)$ are mutually exclusive, such that there exists a k where $cond_k(u, v) \neq cond_k(u, w) \wedge cond_k(u, v) \neq * \wedge cond_k(u, w) \neq *$.

- *Compulsory input requirement*, where all input bursts must contain at least one compulsory input (i.e. an input that does not appear as a directed "don't care" nor a terminating "don't care").

  Formal Definition: For every state transition (u, v), there exists an input $x_i \in trans_{IN}(u, v)$ such that $in_i(u) \neq * \wedge in_i(v) \neq *$.

- *Toggled termination of "don't cares"*, where all directed "don't cares" must terminate such that the signal toggles. For example, if a signal's state was originally 0 (1) before appearing as a "don't care" then it must go 1 (0) at its next termination.

  Formal Definition: For every sequence of state transitions $u \rightarrow v_1 \rightarrow ... \rightarrow v_n \rightarrow w$ with $n \geqslant 1$ and $in_i(u) = in_i(w) \neq *$, there exists $k \in 1, ..., n$ where $in_i(v_k) \neq *$.

In the graphical model of XBM specifications, they are drawn exactly the same as BM specifications with the addition of $*$ directions for inputs that appear as "don't cares", and the word w is expanded to include conditionals that also appear in the input burst. Note that conditionals are enclosed with angle brackets, i.e. $\langle c \rangle$ where

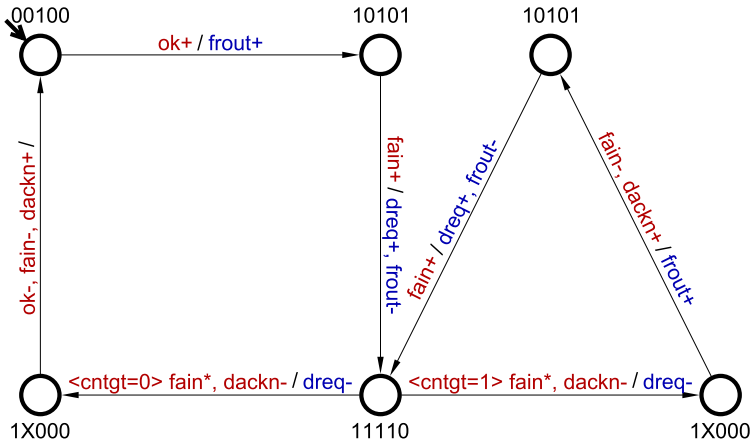


Figure 2.3: XBM specification of the bus interface unit's FIFO to DMA module

conditional $c \in C$. For example, Figure 2.3 shows an XBM specification specifying the first-in first-out (FIFO) to direct memory access (DMA) module of a bus interface unit found in the Small Computer System Interface (SCSI) controller [69].

2.2.3 Petri Nets and Signal Transition Graphs

Petri nets [57] are a simple, yet powerful, mathematical model used to specify and analyse many systems that are, but not limited to, concurrent, asynchronous, distributed, parallel, non-deterministic, and stochastic.

Petri nets can be in one of many possible finite *markings* (i.e. a multiset of *places*) at any given time, and the marking of Petri nets may change in response to *firing* an enabled *transition* when all of its preceding places contain a *token*.

Petri nets consist of a finite set of places, a finite set of transitions, a finite set of arcs, and a finite set of markings, which include the initial marking (i.e. the set of all places that are initially marked with a token) used to identify the start of the Petri net. Each arc determines the flow of tokens and is connected between a place and a transition, or between a transition and a place.

Formally, Petri nets can be defined as a tuple $PN = (P, T, F, M, l)$ where:

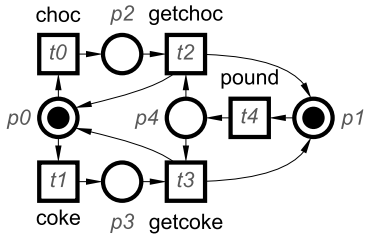


Figure 2.4: Petri net of a vending machine selling cokes and chocolates

- P is the finite set of places.
- T is the finite set of transitions, such that $T \cap P \neq \varnothing$.
- $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions and transitions to places.
- M is a marking of PN where $M : P \rightarrow \{0, 1, 2, ...\}$ such that M_0 is the initial marking.
- $l : T \rightarrow \Sigma$ is a labelling function of transitions, where Σ is the alphabet.

But unlike FSMs, Petri nets can express concurrency without using interleaving meaning they can easily specify input-output concurrency and subsequently, asynchronous circuits.

Notably, Petri nets can also be seen as an extension to FSMs, where a Petri net can model multiple FSMs as one entity especially if the Petri net is *1-safe* (i.e. every place in all markings only ever contain at most one token). Thus, one can see and interpret (1-safe) Petri nets as FSMs.

In the graphical model of Petri nets, places are drawn as circles, transitions are drawn as squares, arcs are drawn as arrows, and tokens are drawn as black dots inside places. For example, Figure 2.4 shows a Petri net specifying the same vending machine that sells cokes and chocolates [6] from Figure 2.1.

Signal Transition Graphs (STG) [58, 18] are labelled Petri Nets that focus on specifying asynchronous systems, where its transitions label the rising and falling edges of signals, such that these signals can be subdivided into inputs, outputs, internals, and dummies with the latter behaving like the empty word ε in language theory.

STGs can be synthesised into SI (QDI) circuits, and are known to be flexible due to their inherent Petri net model, which allows them to easily express input-output concurrency, choices between any type of signals, and both deterministic and non-deterministic specifications.

STGs are also easy to compose and decompose, especially when using high-level asynchronous concepts [11], and enjoy good tool support for verification and synthesis by well-established tools that include Petrify [20] and Mpsat [34, 33], which are both integrated in the visual framework Workcraft [2]. Note that details of Petrify, Mpsat and Workcraft are covered in Section 2.4.2.

Like Petri nets, STGs consist of a finite set of places, a finite set of transitions, a finite set of arcs, and a finite set of markings (including the initial marking) with an additional set of inputs and an additional set of outputs. Places may also contain a token, which depicts the current marking, and arcs determines the flow of tokens between a place and a transition, or between a transition and a place.

Formally, STGs are defined as a tuple $N = (PN, In, Out)$ where:

- $PN = (P, T, F, M, l)$ is a Petri net.
- $In = \{i_0, ..., i_x\}$ is the set of inputs.
- $Out = \{o_0, ..., o_y\}$ is the set of outputs.

Notably, l can be redefined for STGs as $l : T \rightarrow \Sigma$ where $Sig = In \cup Out$ is the set of all signals, where In and Out are disjoint from each other, and $\Sigma := (Sig \times \{+, -\}) \cup \{\lambda\}$ is the alphabet such that λ is the label of dummy transitions.

For an STG to be synthesised into an SI (QDI) circuit, it must ideally satisfy the following implementability properties [4] that include:

- *Consistency* – For every signal $z \in Sig$, where $Sig = In \cup Out$ is the set of all signals and In and Out are disjoint from each other, $z+$ and $z-$ must alternate in each possible trace such that z always starts with the same sign.
- *Deadlock freeness* – Each reachable marking must enable at least one transition.
- *Input properness* – An input must not be triggered by an internal signal, as the environment must be oblivious to internal signals, and must not be disabled by a local (i.e. an output or internal) signal.
- *Output persistency* – A local (i.e. an output or internal) signal must not be disabled by any other signal.
- *Output determinacy* [37] – The STG is said to be not self-contradictory, where this property will trivially hold for determistic STGs. However, for non-deterministic STGs, it may be possible to execute the same trace (i.e. the sequence of signal edges) w in two different ways, such that they reach two different markings. So, if one of these markings enables some output o and the other does not, then there is a contradiction as the STG simultaneously requires the circuit to produce o after w, and forbids to do so.

Additionally, given a constant $k \in \mathbb{N}$, an STG N is called *k-bounded* if for every reachable marking M and every place p where $M(p) \leqslant k$, *bounded* if it is k-bounded for some k, or *safe* if it is 1-bounded such that $k = 1$. Note that N is bounded if, and only if, the set of its reachable markings is finite.

In the definition of $Sig \times \{+, -\}$, each signal $s \in Sig$ is assigned a $+$ or $-$ event where $\{(s, +), (s, -)\} \subseteq Sig \times \{+, -\}$. The event $+$ defines a value change from 0 (i.e.

logical low) to 1 (i.e. logical high), while the event $-$ defines a value change from 1 to 0. For simplicity, we can also define $(s, +)$ and $(s, -)$ as $s+$ and $s-$ respectively.

In particular, the preset of x can be defined as ${}^\bullet x$ and the postset of x can be defined as $x^\bullet$, where $x \in P \cup T$ can be either a place or a transition.

The transition t is said to be *enabled* at a marking M, which can be denoted as $M[t\rangle$, if for each $p \in {}^\bullet t$, $M(p) \geqslant 1$. Such an enabled transition can *fire*, changing M to $M' = M - M(p) + M(q)$ for all $p \in {}^\bullet t$ and all $q \in t^\bullet$, denoted by $M[t\rangle M'$.

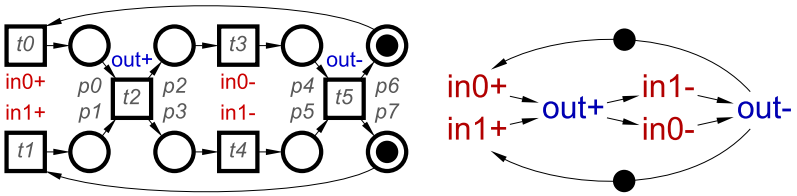
For a transition sequence $v = t_1..t_n$ and markings M and M', $M[v\rangle M'$ can be written if there are markings $M_1, \ldots M_{n-1}$ such that $M[t_1\rangle M_1[t_2\rangle ...M_{n-1}[t_n\rangle M'$.

Moreover, M is said to be reachable if there exists a transition sequence v such that $M_0[v\rangle M$, and an empty transition λ is said to be enabled under every marking where λ does not correspond to any signal change. Note that hiding a signal s means all $s\pm$ labelled transitions are changed to λ, while unhiding a signal s means the transition's labels are changed back to their initial values.

In the graphical model of STGs, they can be drawn out the same as Petri nets with the exception that transitions labelling inputs have their text coloured in red, and transitions labelling outputs have their text coloured in blue. For example, Figure 2.5a shows an STG specifies the C-element gate.

In WORKCRAFT, we can simplify the drawing of STGs, where transitions labelling inputs or outputs are replaced with a red text block or a blue text block respectively, such that they act as *signal transitions*. If a place has exactly one incoming transition and one outgoing transition, then this place and its connecting arcs can be replaced with an arc between the original transitions, where this arc contains an 'implicit' place. For example, Figure 2.5b shows the simplified STG drawing of Figure 2.5a.

Lastly, we can also obtain the *reachability graph* of an STG by decoupling the markings after each transition. As such, the reachability graph of an STG N can be



(a) Petri net drawing (b) WORKCRAFT drawing

Figure 2.5: STG specification of C-element gate

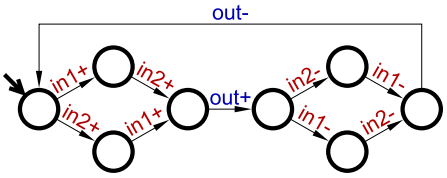


Figure 2.6: Reachability graph for the STG specification of C-element

described as an arc-labelled directed graph $RG_N = (Q, A, q_0)$ where:

- Σ is the alphabet of N.
- Q is the set of reachable markings of N.
- $A \subseteq Q \times \Sigma \times Q$ is the set of labelled arcs and $(q, \gamma, q') \in A$ iff there exists a γ-labelled transition t of N such that $q[t\rangle q'$.
- $q_0 = M_0$ is the initial marking of N.

If an STG is said to be safe then RG_N is finite and can be interpreted as a finite automaton in the language theory, or as an FSM. In particular, the *language* $L(N)$ can be defined as the language of RG_N, where all states of RG_N are considered accepting, and N is said to be deterministic if RG_N is a deterministic automaton as in the language theory, i.e. it contains no ε-transitions and no distinct arcs originating from the same state and having the same label.

For an example of an STG's reachability graph, Figure 2.6 shows the reachability graph for the STG specification specifying the C-element gate from Figure 2.5.

2.3 Model Equivalences

2.3.1 Bisimulation

Bisimulation [29, 46] is the equivalence relation between two associating systems that behave in a similar manner, where one system can simulate the other system's behaviour and vice versa. So, given an FSM $M = (Q, A, q_0)$ and another FSM $M' = (Q', A', q_0')$, M and M' are said to be *(strongly) bisimilar* if there exists a relation between their states called bisimulation (denoted as $\sim$), such that:

- The initial states of M and M' are related, i.e. $q_0 \sim q_0'$.
- For every pair of related states $q \sim q'$:
  - For each step $q \xrightarrow{a} r$ in M, there exists a matching step $q' \xrightarrow{a} r'$ in M' such that $r \sim r'$.
  - Similarly, for each step $q' \xrightarrow{a} r'$ in M', there exists a matching step $q \xrightarrow{a} r$ in M such that $r \sim r'$.

If both M and M' are bisimilar then $\mathcal{L}(M) = \mathcal{L}(M')$ the language of M is equal to the language of M', but not vice versa in general. Note that bisimilarity coincides with language equivalence for deterministic FSMs.

For some examples of strong bisimulation, Figure 2.7 shows several scenarios of two FSMs that are strongly bisimilar to each other.

In Figure 2.7a, the initial states must first be related and the transition $s_0 \xrightarrow{a} s_0$ on the left FSM can also be matched by the transition $q_0 \xrightarrow{a} q_0$ on the right FSM and vice versa, meaning $s_0 \sim q_0$. Also, the transition $s_0 \xrightarrow{a} s_1$ on the left FSM can be matched by the transition $q_0 \xrightarrow{a} q_0$ on the right FSM, meaning $s_1 \sim q_0$. Thus, these two FSMs are strongly bisimilar to each other.

In Figure 2.7b, the initial states must first be related and the transition $s_0 \xrightarrow{a} s_1$ on the left FSM can also be matched by the transition $q_0 \xrightarrow{a} q_0$ on the right FSM and vice versa, meaning $s_0 \sim q_0$. Also, the transition $s_1 \xrightarrow{a} s_1$ on the left FSM can be matched by the transition $q_0 \xrightarrow{a} q_0$ on the right FSM, meaning $s_1 \sim q_0$. Thus, these two FSMs are strongly bisimilar to each other.

In Figure 2.7c, the initial states must first be related, meaning $s_0 \sim q_0$. Next, the transitions $s_0 \xrightarrow{a} s_1$ and $s_0 \xrightarrow{a} s_2$ on the left FSM can be matched by the transition $q_0 \xrightarrow{a} q_1$ on the right FSM and vice versa. So, $s_1 \sim q_1$ and $s_2 \sim q_1$. Lastly, the transition $s_1 \xrightarrow{b} s_3$ on the left FSM can be matched by the transition $q_1 \xrightarrow{b} q_2$ on the right FSM and vice versa, while the transition $s_2 \xrightarrow{c} s_4$ on the left FSM can be matched by the transition $q_1 \xrightarrow{c} q_3$ on the right FSM and vice versa. So, $s_3 \sim q_2$ and $s_4 \sim q_3$. Thus, these two FSMs are strongly bisimilar to each other.

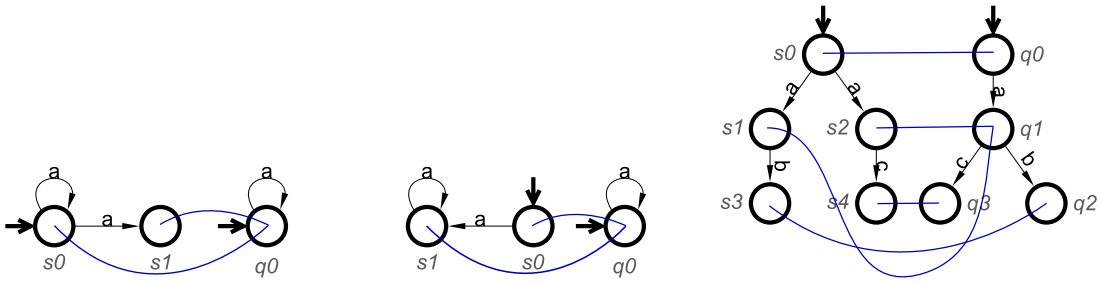


Figure 2.7: Strong bisimulation examples

For two systems to be strongly bisimilar to each other, they are required to match each other's every action, where one can perform the action of the other system's action and vice versa. However, this requirement can be considered too sharp of an equivalence to check, if a system's internal behaviour, and sometimes its external behaviour, needs to be observed and distinguished.

Hence, the above definition of bisimulation can be redefined, where two systems can still be seen as equivalent if they exhibit the same external behaviour, regardless of

any intermediate internal behaviour that may occur, as they are invisible to external observers and are not required for the systems to be matched.

So, given an FSM $M = (Q, A, q_0)$ and another FSM $M' = (Q', A', q_0')$, the notion of bisimulation above can be relaxed by replacing all $\xrightarrow{a}$ by $\xrightarrow{w}$, where w is either ε or $a \in \Sigma$. In particular, M and M' are said to be *weakly bisimilar* if:

- Each ε-labelled step of FSM is matched by 0 or more ε-labelled steps of FSM'.
- Each a-step of FSM is matched by 0 or more ε-labelled steps followed by an a-step followed by 0 or more ε-labelled steps of FSM'.

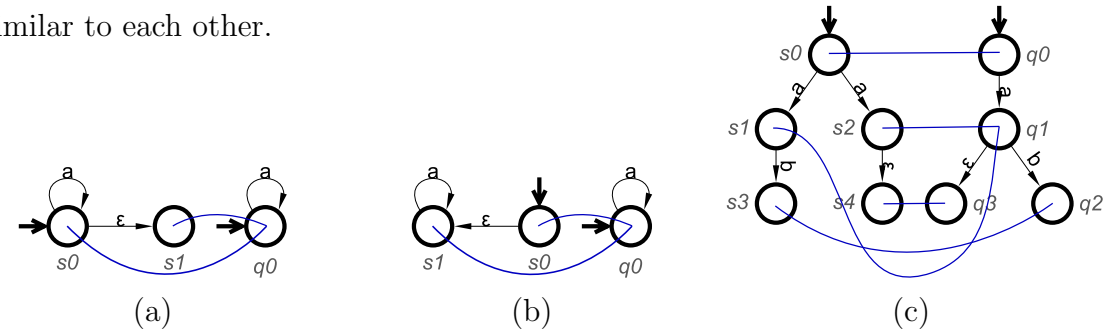
For some examples of weak bisimulation between two FSMs, Figure 2.8 shows several scenarios of two FSMs that are weakly bisimilar to each other.

In Figure 2.8a, the initial states must first be related and the transition $s_0 \xrightarrow{a} s_0$ on the left FSM can be matched by the transition $q_0 \xrightarrow{a} q_0$ on the right FSM and vice versa, meaning $s_0 \sim q_0$. Also, the left FSM can do the transition $s_0 \xrightarrow{\varepsilon} s_1$, which can only be matched by the right FSM doing nothing at q_0, so $s_1 \sim q_0$. Thus, these two FSMs are weakly bisimilar to each other.

In Figure 2.8b, the initial states must first be related and the left FSM can do the transition $s_0 \xrightarrow{\varepsilon} s_1$, which can only be matched by the right FSM doing nothing at q_0. So, $s_0 \sim q_0$. Also, the transition $s_1 \xrightarrow{a} s_1$ on the left FSM can be matched by the transition $q_0 \xrightarrow{a} q_0$ on the right FSM and vice versa, meaning $s_1 \sim q_0$. Thus, these two FSMs are weakly bisimilar to each other.

In Figure 2.8c, the initial states must first be related, meaning $s_0 \sim q_0$. Next, the transitions $s_0 \xrightarrow{a} s_1$ and $s_0 \xrightarrow{a} s_2$ on the left FSM can be matched by the transition $q_0 \xrightarrow{a} q_1$ on the right FSM and vice versa. So, $s_1 \sim q_1$ and $s_2 \sim q_1$. Also, the transition $s_1 \xrightarrow{b} s_3$ on the left FSM can be matched by the transition $q_1 \xrightarrow{b} q_2$ on the right FSM and vice versa, meaning $s_3 \sim q_2$. Lastly, the transition $q_1 \xrightarrow{\varepsilon} q_3$ on the right FSM

can be matched by the transition $s_2 \xrightarrow{\varepsilon} s_4$ on the left FSM, or by the left FSM doing nothing at s_1 or s_2. So, $s_1 \sim q_3$ and $s_2 \sim q_3$. Similarly, the transition $s_2 \xrightarrow{\varepsilon} s_4$ on the left FSM can be matched by the transition $q_1 \xrightarrow{\varepsilon} q_3$ on the right FSM or the right FSM doing nothing at q_1. So, $s_4 \sim q_1$ and $s_4 \sim q_3$. Thus, these two FSMs are weakly bisimilar to each other.



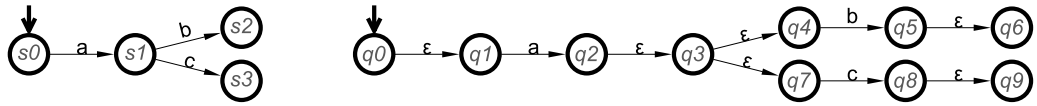
(a) (b) (c)

Figure 2.8: Weak bisimulation examples

2.3.2 Language Equivalence

Language equivalence is the relation between two associating systems whose languages match each other, where one system can capture the other's sets of traces through its own sets of traces and vice versa. So, given an FSM M and an FSM M', if the language of M can be captured by M' and vice versa then M and M' are said to be language equivalent such that $\mathcal{L}(M) = \mathcal{L}(M')$.

For an example of language equivalence between two FSMs, Figure 2.9 shows two FSMs that are language equivalent to each other. Here, the language of the left FSM is $\mathcal{L}_{left} = \{\varepsilon, a, ab, ac\}$ and the language of the right FSM is $\mathcal{L}_{right} = \{\varepsilon, a, ab, ac\}$, where all ε-transitions are represented by the ε in the set. Thus, $\mathcal{L}_{left} = \mathcal{L}_{right}$.



(a) (b)

Figure 2.9: Language equivalence example

2.3.3 Output determinacy

While equivalences like bisimulation may be more desirable in situations where both systems must behave similarly, there are still some instances where it is enough for both systems to be language equivalent, e.g. when using *output-determinate* STGs.

In [37], the semantics of non-deterministic STGs are formally defined and a property called *output-determinacy* is introduced. Output determinacy is defined as a relaxation of determinism, where non-deterministic STGs are said to be output-determinate if they can perform the same trace in two different ways and reach different states M_1 and M_2, such that the outputs enabled by M_1 and the outputs enabled by M_2 are the same. If an STG is not output-determinate then it is said to be ill-formed and cannot be correctly implemented by a circuit, as it shows that the language is not a satisfactory semantic of non-deterministic STGs in general.

This means that there is no need to preserve stronger equivalences like bisimulation, as models preserving the language are adequate enough in practice. Note that any attempt to synthesise a non-output-determinate STG will fail or result in an incorrect circuit. So, an STG N is called output-determinate if $M_N[w\rangle\rangle M_1$ and $M_N[w\rangle\rangle M_2$ is implied for every $x \in Out_N$ where $M_1[x^{\pm}\rangle\rangle$ if, and only if, $M_2[x^{\pm}\rangle\rangle$.

For an example of output-determinate STGs, Figures 2.10a and 2.10b shows two output-determinate STGs as neither STG produces a contradictory output after firing a+, i.e. firing a+ only enables x+, and Figure 2.10c shows a non-output-determinate STG, as firing a+ on the right enables x+ but firing a+ on the left enables y+.

(a) (b) (c)

Figure 2.10: Output determinacy examples

2.4 Computer-Aided Design Tools

Computer-aided design (CAD) is the concept, where computers are used to aid designers with the creation, modification, analysis, and optimisation of their design. Notably, there are several CAD tools that aid with the design and synthesis of asynchronous circuits including MINIMALIST, 3D and BM DECOMP for BM specifications, and PETRIFY, MPSAT, PCOMP and WORKCRAFT for STGs.

2.4.1 Tools for Burst-Mode Specifications

MINIMALIST [54] is a framework that was developed to support the synthesis and verification of optimised BM specifications, and consists of a complete technology-independent synthesis path using exact and heuristic algorithms for the synthesis of asynchronous circuits that include:

- Coding for Hazard-free Asynchronous State Machines (CHASM) [27], an exact-based method that focuses on the optimisation of state encodings.
- HFMIN [27], ESPRESSO-HF [65], and IMPYMIN [64], exact-based and heuristic-based tools that focus on two-level hazard-free minimisation of logic.
- Synthesis for testability [56], a method that focuses on targetting multi-level logic, and producing circuits that are hazard-free and testable under either stuck-at or robust path delay fault models with little to no overhead.

MINIMALIST features a command-line interface and graphical interface, where the former interface is used to synthesise and optimise BM specifications, while the latter interface is used to display the BM specifications and the produced circuit implementations from MINIMALIST's synthesis results.

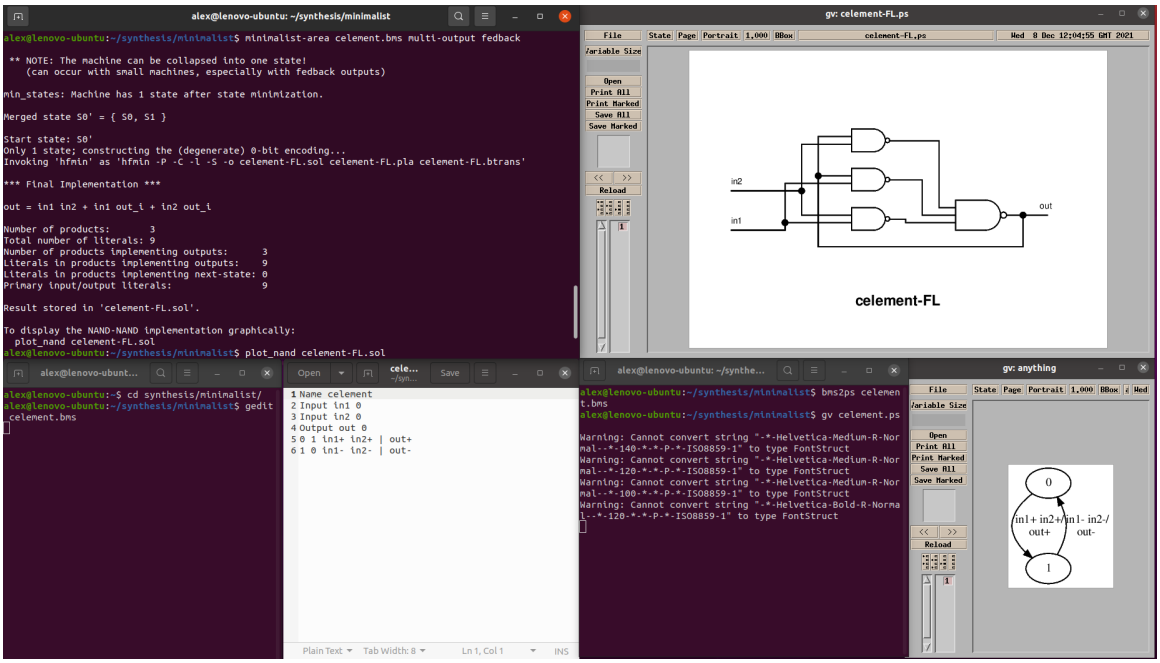


Figure 2.11: Synthesis of the C-element specification using MINIMALIST

MINIMALIST also supports an extension that allows new tools to be added through its defined interfaces, and allows the designer to explore different trade-offs between the available synthesis options, which include a focus on the optimisation of literals, optimisation of product, minimisation of single-output logic, and minimisation of multi-output logic, with the option to include feedback outputs as state variables.

For an example of the design and synthesis procedure in MINIMALIST, Figure 2.11 shows each step of the synthesis approach, where the BM specification of a C-element gate is first specified in text, displayed, and then synthesised with the options set to optimise literals, minimise multi-output logic and to include feedback outputs, before the produced circuit implementation is displayed.

Also, for an example of the verification procedure in MINIMALIST, Figure 2.12 shows MINIMALIST detecting a violated well-formed requirement from some BM specification (i.e. non-empty input burst property), before it stops the synthesis process and displays an error message on the command-line interface window.

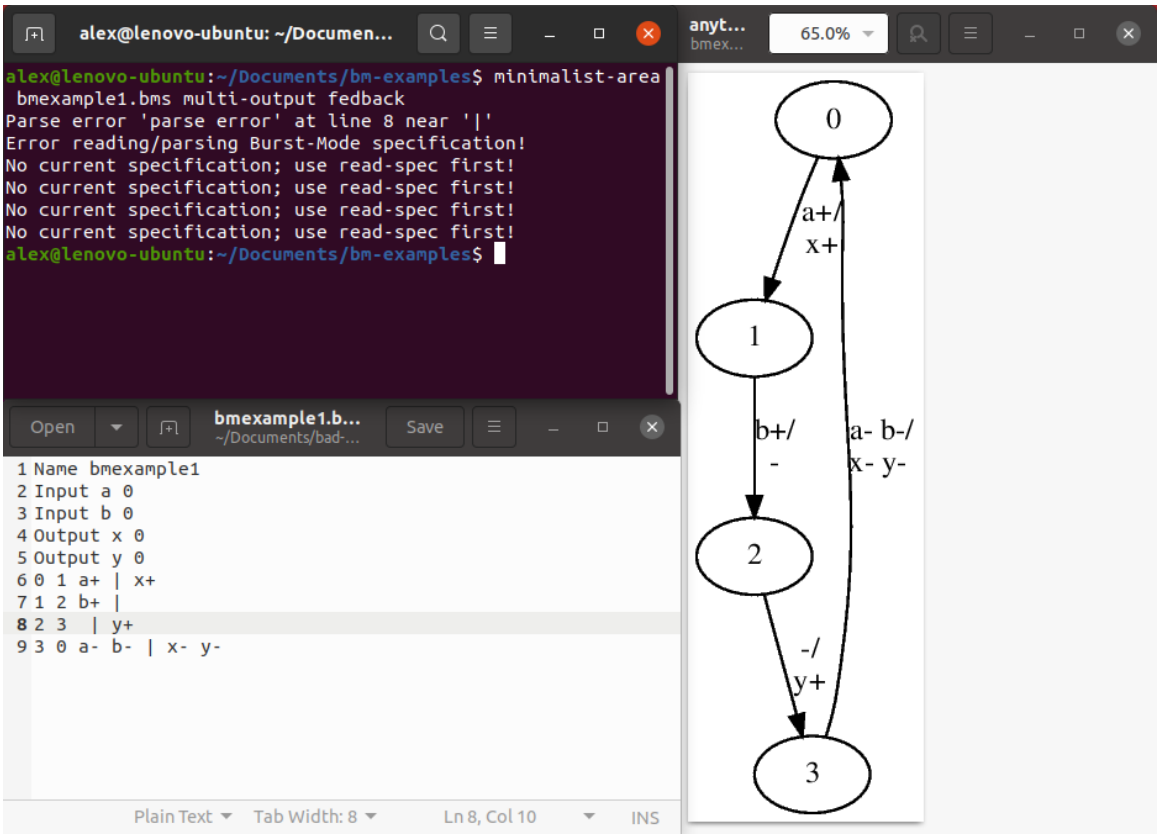


Figure 2.12: Verification of MINIMALIST framework

While MINIMALIST helps produce BM controllers, there are some notable caveats:

1. MINIMALIST does not support XBM specifications meaning it cannot read the XBM syntax, e.g. conditionals and "don't cares". In [54], it was planned that MINIMALIST will support XBM specifications, though this was never completed.

2. MINIMALIST only verifies that specifications satisfy the BM's well-formed requirements during its synthesis step. This can be problematic as:

   (a) This verification is based on properties that are only applicable for BM specifications. While it is useful to ensure that BM specifications are well-formed by design, it cannot be easily reused for other formal models similar to BM specifications, e.g. other types of asynchronous FSMs.

(b) It may be more convenient to separate the verification and synthesis processes, as a designer may not necessarily want to synthesise their specification after verification, e.g. the designer may wish to verify the specification and then synthesise it with other available tools.

3. While MINIMALIST provides some graphical interface to display the BM specifications and the circuit implementation from its synthesis results, there is no dedicated graphical interface to aid the designer with creating their BM specification. Instead, the designers are required to create their BM specification using a text editor, where they must type out the whole specification and ensure that it correctly follows BM's syntax. This process can be quite tedious, especially if the specification is large, and it requires the designer to understand the BM's syntax in file format, which can be non-trivial for new circuit designers. Also, this process can be extremely error-prone, as a designer can make a mistake like typing out the name of a non-existing signal.

3D [68] is a tool that was developed to support the synthesis and verification of BM specifications and XBM specifications using a complete set of automated sequential synthesis algorithms, which include hazard-free state assignment, hazard-free state minimisation, and critical-race-free state encoding. Unlike MINIMALIST, 3D uses heuristic greedy state minimisation and encoding algorithms, and only performs single output logic minimisation using HFMIN to produce high-performing circuit implementations. However, these methods may also not guarantee that the produced results are the most optimum one, as shown in the benchmarks in [54].

Nevertheless, because support for XBM specifications was never implemented in MINIMALIST, this means that 3D remains to be the only tool that supports the synthesis and verification of XBM specifications.

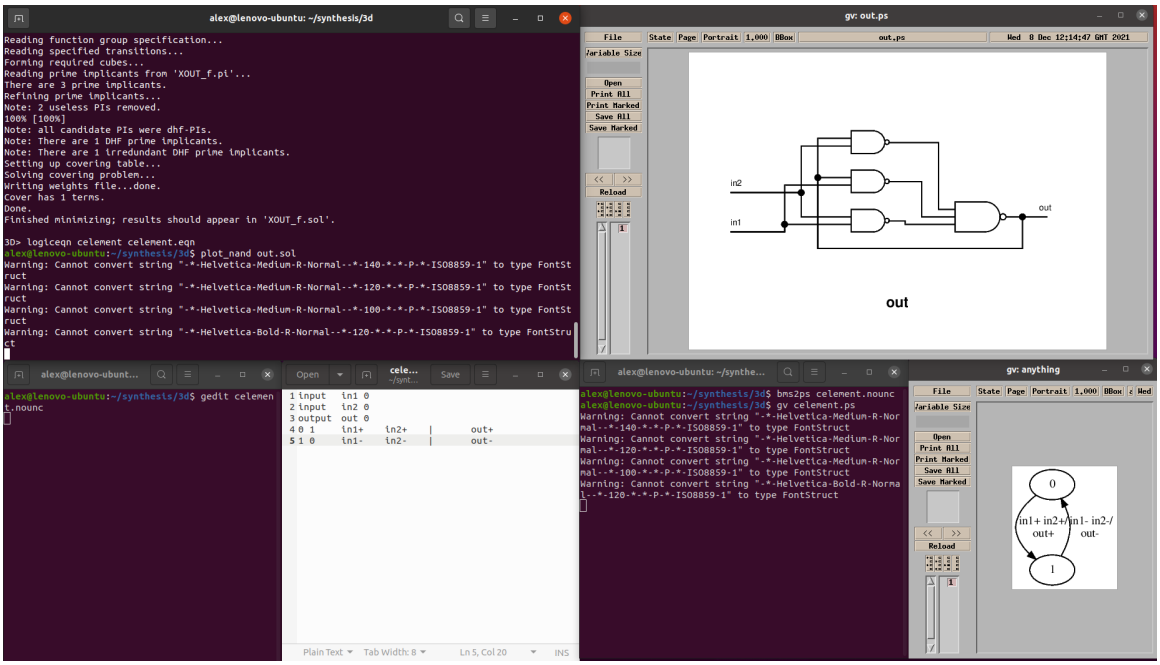


Figure 2.13: Synthesis of the C-element specification using 3D

For an example of the design and synthesis procedure in 3D, Figure 2.13 shows each step of the synthesis approach, where the XBM specification of a C-element gate is first specified in text, displayed (using MINIMALIST's binaries), and synthesised into an XBM controller, before the produced circuit implementation is displayed.

Similarly, for an example of the verification procedure in 3D, Figure 2.14 shows 3D detecting a violated well-formed requirement from some XBM specification (i.e. distinguishability constraint), before it stops the synthesis process and displays an error message on the command-line interface window.

BM DECOMP [3] is a command-line tool that decomposes (X)BM specifications into smaller groups of (X)BM specifications. BM DECOMP uses the cycle-based decomposition method proposed in [7], where it searches through a given (X)BM specification and determines how many cycles are contained in the specification.

If only one cycle is found then the decomposition does not take place. Otherwise,

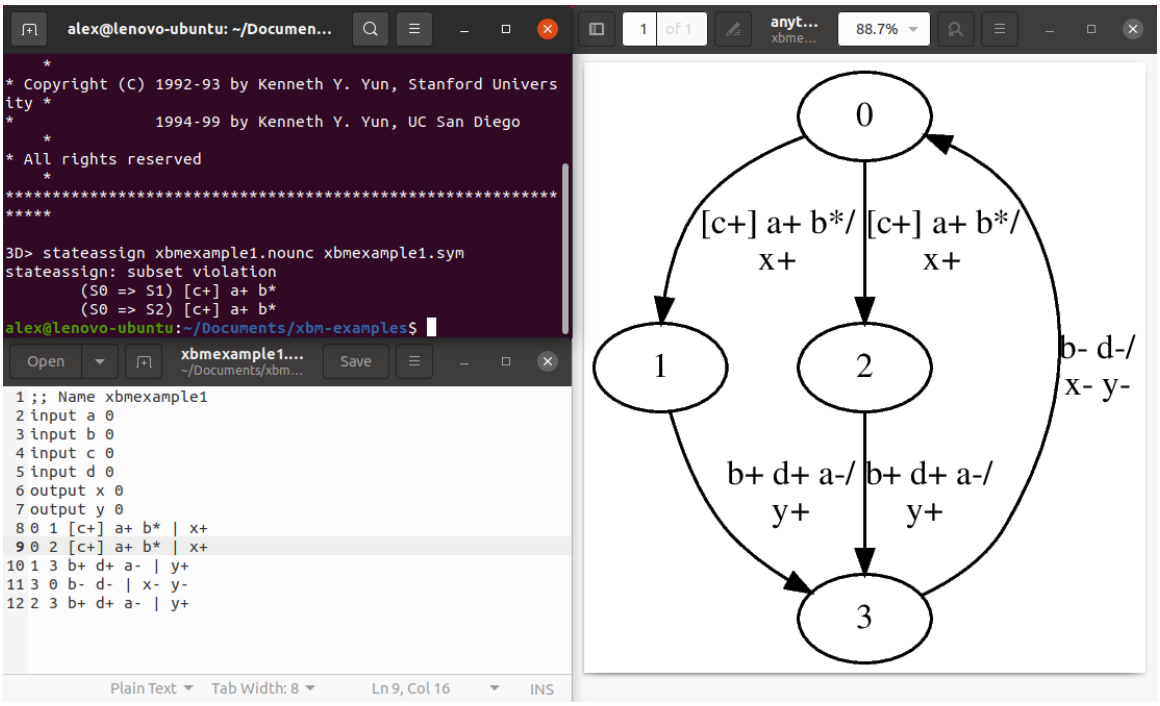
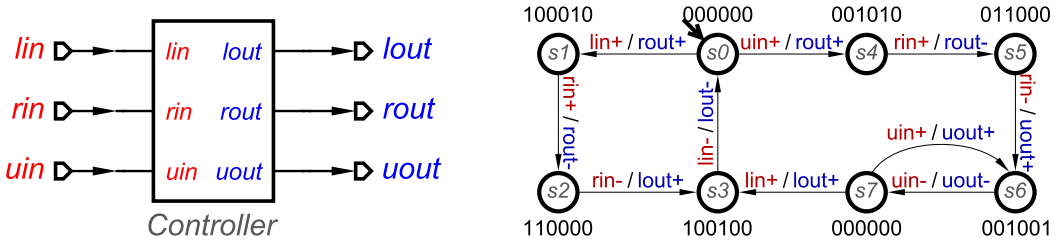


Figure 2.14: Verification of 3D tool

if n-number of cycles are found, where $n > 1$, then the (X)BM specification is decomposed $n + 1$ times to create one "master" and n-number of "machine" (X)BM specifications. These (X)BM specifications may then be synthesised using MINIMALIST to produce a BM controller or 3D to produce an (X)BM controller.

For an example of the decomposition procedure in BM DECOMP, let us consider the circuit block diagram of a distributed mutual-exclusion (DME) controller shown in Figure 2.15a with its equivalent BM specification shown in Figure 2.15b.



(a) Circuit block diagram (b) BM specification

Figure 2.15: DME controller

Here, we can see that there are three cycles, where the first cycle can be reached by firing lin+ / rout+ → rin+ / rout- → rin- / lout+ → lin- / lout- from the initial state $s0$, the second cycle can be reached by firing uin+ / rout+ → rin+ / rout- → rin- / uout+ → uin- / uout- → → lin+ / lout+ → lin- / lout- also from state $s0$, and the third cycle can be reached by firing uin- / uout- → uin+ / uout+ from state $s6$.

Now, if we were decompose this BM specification using BM Decomp then BM Decomp will first detect these three cycles, and generate a master "master" BM specification and three "machine" BM specifications as shown in Figure 2.16, where the latter corresponds to each cycle that was described above.

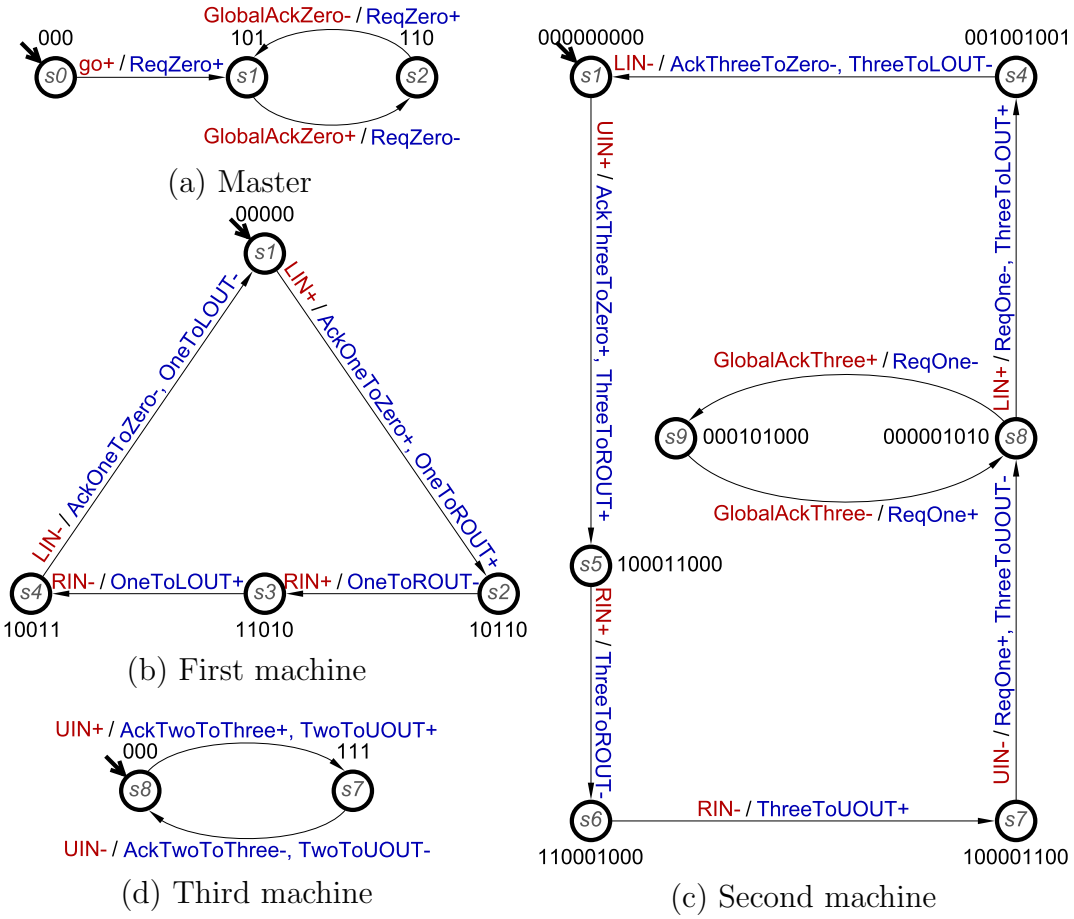


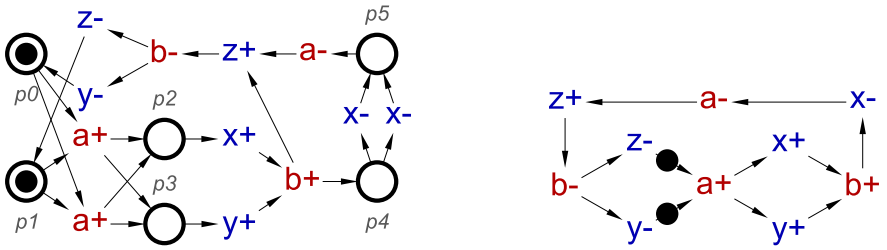
Figure 2.16: Decomposed BM specifications of the DME Controller

Note that the communication between these (X)BM specifications happen through a globalised set of request and acknowledgement signals, where the 'control' is given to a (X)BM specification via a request from the active (X)BM specification, such that there can only be at most one active BM specification and all other BM specifications are either inactive (i.e. waiting to receive control) or suspended (i.e. waiting for the control to return). This, in particular, means that this decomposition method [7] is purely "syntactic" and does not allow concurrency between different components and their environments.

2.4.2 Tools for Signal Transition Graphs

PETRIFY [20] is a logic-based synthesis tool used to manipulate concurrent specifications such as Petri nets and STGs, and to synthesise and optimise asynchronous circuits. When a Petri net or an STG is given, PETRIFY can 'net synthesis' this Petri net or STG, where it will optimise and produce a new corresponding Petri net or STG that is simpler than the original one.

For example, let us consider an STG that contains some redundancies (e.g. doubled a+ transitions, doubled x+ transitions and an ineffective arc from b+ to z+) in Figure 2.17a. Here, if we apply 'net synthesis' to this STG, PETRIFY will then optimise and remove the redundances, as shown in Figure 2.17b.

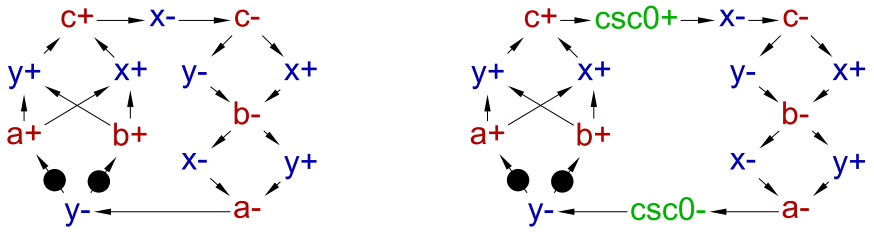


(a) Redundant STG (b) Optimised STG

Figure 2.17: PETRIFY optimisation of an STG using 'net synthesis'

PETRIFY can also produce an optimised net-list of asynchronous controllers in a target gate library, while preserving any specified input-output behaviour. When a Petri net or STG is synthesised into an asynchronous circuit, PETRIFY will first perform state assignment, which consists of logic minimisation and speed-independent technology mapping to the target library, by resolving any complete state coding (CSC) problems. Thus, resulting in a net-list that is speed-independent.

For example, let us consider an STG that contains a CSC conflict in Figure 2.18a [20]. Here, if we use PETRIFY to resolve the CSC conflict in this STG, then PETRIFY will insert a new state signal csc0 to resolve the conflict, such that csc0+ and csc0- were inserted to ensure the resulting logic is optimised, as shown in Figure 2.18b.



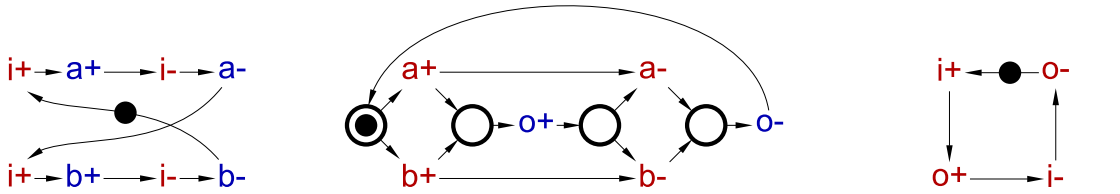
(a) STG with CSC conflicts (b) STG with resolved CSC conflicts

Figure 2.18: PETRIFY resolving some CSC conflicts found in an STG

The UNFOLDING TOOLS [32] are a collection of command-line based tools that are used to provide various verification and synthesis flows, based on Petri net unfoldings.

MPSAT [34, 33] is the unfolding tool used to verify and synthesis STGs using an incremental Boolean satisfiability approach, where MPSAT addresses the issue of deriving equations for logic gates that implement each other signal, e.g. when the reachability graph of an STG is used to be synthesised into an asynchronous circuit, by not constructing the reachability graph of the STG. Instead, MPSAT performs the synthesis based on the STG's causality, avoiding the STG's structural conflicts between its events that are involved in a finite and complete prefix of its unfolding.

Additionally, MPSAT also provides a host of verification including the STG's implementability properties provided in Section 2.2.3, CSC checks, and N-way conformation. For example in Figure 2.19, if we verified N-way conformation between the three STGs [9] that are specifying a toggle (i.e. Figure 2.19a), a call (i.e. Figure 2.19b) and the environment (i.e. Figure 2.19c), then we will find that these STGs are indeed conformant with each other.



(a) Toggle (b) Call (c) Environment

Figure 2.19: N-way conformant STGs

Another tool in the UNFOLDING TOOLS is PCOMP [9] , which is used to combine several STGs into one singular STG via parallel composition, where the designer follows the modular design approach by designing the models of subsystems and then combining them to create the model of the whole system, rather than the monolithic design approach of designing the model of the whole system.

To show how PCOMP works, let us re-consider the three STGs in Figure 2.19. If we apply parallel composition to these three STGs using PCOMP, then we can achieve the composed STG shown in Figure 2.20.

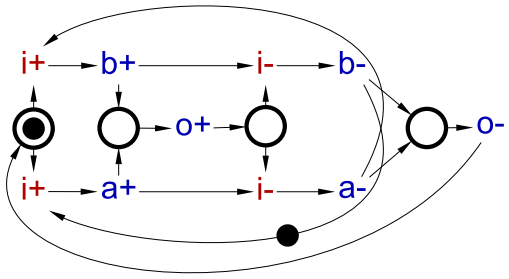


Figure 2.20: Composed STG of the three STGs from Figure 2.19 using PCOMP

WORKCRAFT [2] is a visual framework that provides rich support for interpreted graph models including FSMs, Petri nets, and in particular STGs.

WORKCRAFT has a graphical front-end that supports the editing and simulation of specifications, as well as an established back-end of tools that include PETRIFY and UNFOLDING TOOLS like MPSAT for the verification and synthesis of specifications, as well as PCOMP for the composition of STGs, which are automated. Moreover, WORKCRAFT uses a plugin-based architecture that allows new plugins and new back-end tools to be easily integrated.

For an example of WORKCRAFT and its features, Figure 2.21 shows the design, simulation, verification, and synthesis of an STG specifying the C-element gate.

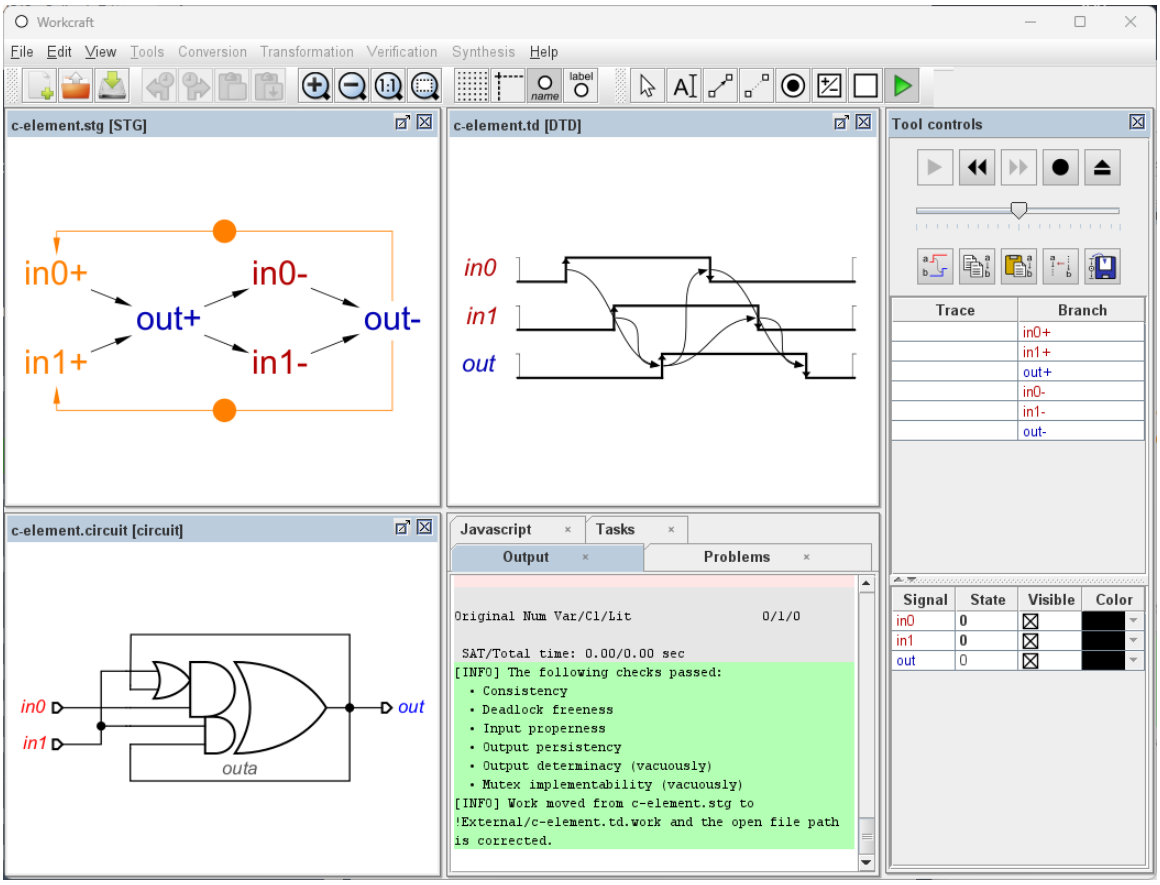


Figure 2.21: Design, simulation, verification, and synthesis of an STG specifying a C-element gate in WORKCRAFT

Chapter 3

Technical Motivation

In this chapter, we will cover the motivation of this thesis in greater detail, where we also include the technical aspects that are not yet covered in Section 1.1.

Firstly, we will revisit the current design philosophy, where circuit designers must choose between the 'legacy' design route of Finite State Machines (FSMs) involving Burst-Mode (BM) specifications and the 'disruptive' design route of Petri nets involving Signal Transition Graphs (STGs).

Next, we will provide a motivating example that involves the design of a handshake decoupling system. In this example, we have a three-part asynchronous controller that interacts with a distributed environment that consists of a two-part generator and two handshake handlers, where there are two concurrent handshakes that can be decoupled as far as possible depending on the environment conditions. We will then attempt to model this example using both BM specifications and STGs, before we evaluate each model according to the design philosophy.

Finally, we will establish the connection between BM specifications and STGs by proposing a new model called the Burst Automaton (BA), which acts as a framework for enabling interoperability between many models including BM specifications and STGs. Here, we also cover a new design route that is provided by our model, where we can transition from the design of BM specifications to the circuit synthesis of STGs, before we show how BAs are used to model the motivational example.

3.1 Design Philosophy of Asynchronous Circuits

In this section, we will cover the two design routes of asynchronous circuits that were briefly covered in Section 1.1, where the first is the 'legacy' design route based on FSMs and the second is the 'disruptive' design route based on Petri nets.

To begin, let us revisit the two aforementioned design routes shown in Figure 3.1 that is expanded to include the synthesis tools used in their respective route.

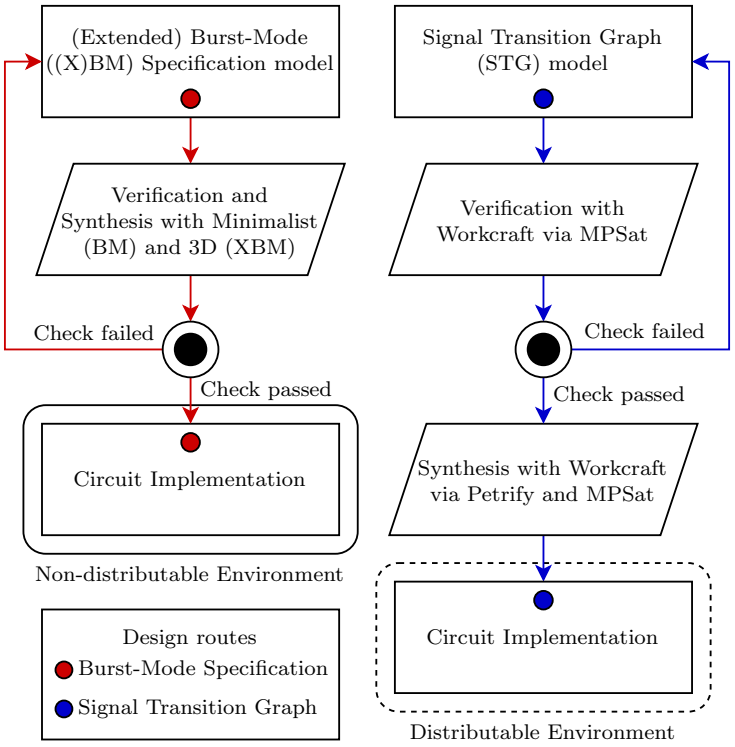


Figure 3.1: Existing Design Routes with BM specifications and STGs

In the first design route, BM specifications are used to model and synthesise BM controllers. In Section 2.2.2, BM specifications are described to be a simple entry into asynchronous circuit design for (synchronous) circuit designers, as they resemble synchronous FSMs [55]. BM specifications favour the circuit designer's familiarity of state-based methods, meaning they can easily adapt and use BM specifications.

Additionally, BM specifications are supported by the synthesis tool MINIMALIST to produce optimal BM controllers [54]. Note that BM specifications cannot express input-output concurrency, which led to the proposal of its extension, the Extended BM (XBM) specification, where it enables input-output concurrency by introducing conditionals and "don't cares" [68], and is supported by the synthesis tool 3D.

However, while BM specifications prove to be quite effective and have seen use by both academia and industry to design and implement a number of significant circuits [54], the circuit implementations produced by both MINIMALIST and 3D may no longer be well-optimised, as these tools have limited support and are no longer regularly updated. Moreover, the produced circuit implementations are limited to a non-distributable environment, due to BM's inherent 'burst-mode' timing assumption and BM's well-formed requirements, which can be limiting if a particular behaviour is required, e.g. decoupled signals and output choices as later shown in Section 3.2.

In the second design route, STGs are used to model and synthesise speed-independent (SI) (or equivalently quasi-delay-insensitive (QDI)) asynchronous circuits. In Section 2.2.3, STGs are described to be flexible as they can express many behaviours including input-output concurrency, output choices, non-determinism and other inherent behaviours from distributed systems. STGs favour producing well-optimised circuits using their well-established tools like PETRIFY and MPSAT, which aims for circuit implementations that are small and yield a high performance.

Nevertheless, while STGs are desirable for asynchronous circuit design, STGs are also seen as too different by the industry, due to the industry's unfamiliarity with event-based methods. This suggests that circuit designers may opt for more familiarised approaches that are state-based methods like BM specifications.

In the following sections, we will show how our BA model bridges the gap between the 'legacy' design route and the 'disruptive' design route.

3.2 Motivating Example of a Handshake Decoupler

In this section, we will use a distributed control system that decouples handshakes as an example to highlight some of the issues described for BM specifications and STGs. The circuit block diagram of our handshake decoupler is shown in Figure 3.2, where an asynchronous controller interacts with a distributed environment that consists of a two-part generator and two other environment parts called 'left' and 'right' respectively. For each side of the controller, there is a pair of handshakes that can be decoupled as far as possible, depending on the environment conditions.

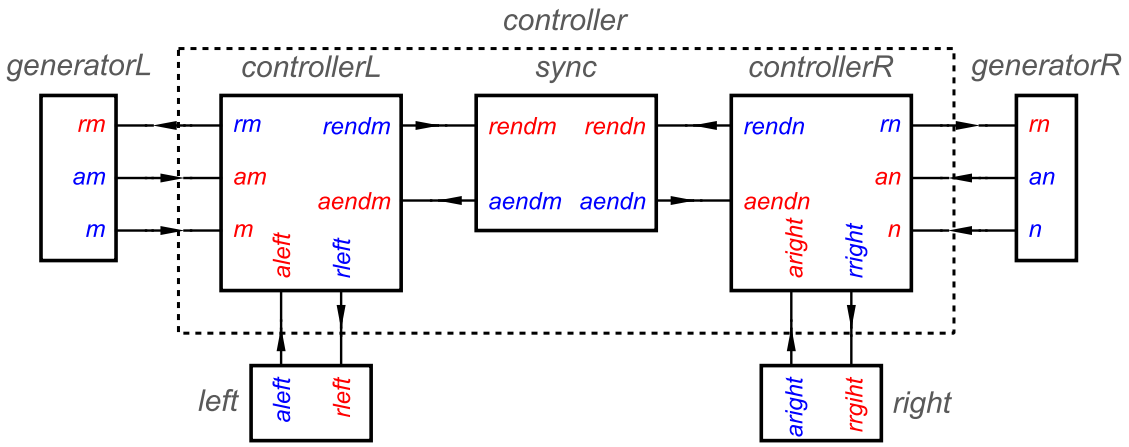


Figure 3.2: Block Diagram of the Handshake Decoupler

As shown in the circuit block diagram, there are three components in our asynchronous controller: the left side of the controller, the right side of the controller, and a synchroniser. The left side of the controller interacts with the left generator, left environment and synchroniser via handshakes rm/am, rleft/aleft and rendm/aendm, while the right side of the controller interacts with the right generator, right environment and synchroniser via handshakes rn/an, rright/aright and rendn/aendn.

For each iteration of the handshake rm/am (rn/an), the left (right) generator provides a reading of m (n) to the controller's left (right) side, where:

- If m=1 (n=1) then the controller's left (right) side will initiate the handshake rleft/aleft (rright/aright) with the left (right) environment, such that when the handshakes rm/am (rn/an) and rleft/aleft (rright/aright) are completed, the controller is reset for the next reading of m (n) and the left (right) generator's counter is incremented. Note that this transition sequence can be repeated up to i (j) times where $i \in \mathbb{N}$ ($j \in \mathbb{N}$).

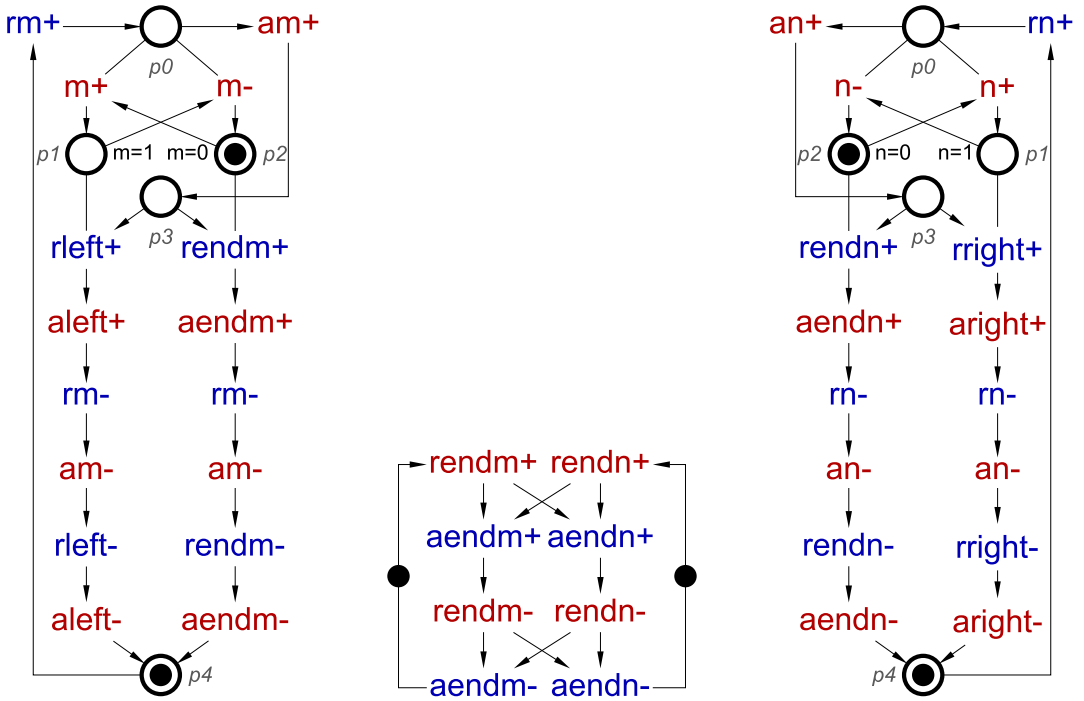
- If m=0 (n=0) then the controller's left (right) side will instead initiate the handshake rendm/aendm (rendn/aendn) with the synchroniser, where the controller's left (right) side is prevented from initiating further handshakes of rm/am (rn/an) with the left (right) generator. These handshakes may only resume once the controller's right (left) side also initiates its handshake rendn/aendn (rendm/aendm) with the synchroniser, which resets both sides of the controller to their initial state and resets the counters of both the left generator and right generator to zero. Note that only the rendm input events should follow the aendm output events, and only the rendn input events should follow the aendn output events.

Now, suppose that we want to design all three parts of the asynchronous controller in our handshake decoupler by following either design route and using its formalism.

Firstly, let us consider the 'disruptive' design route. Here, we can easily design and capture the specified behaviour of the controller using STGs, as shown in Figure 3.3.

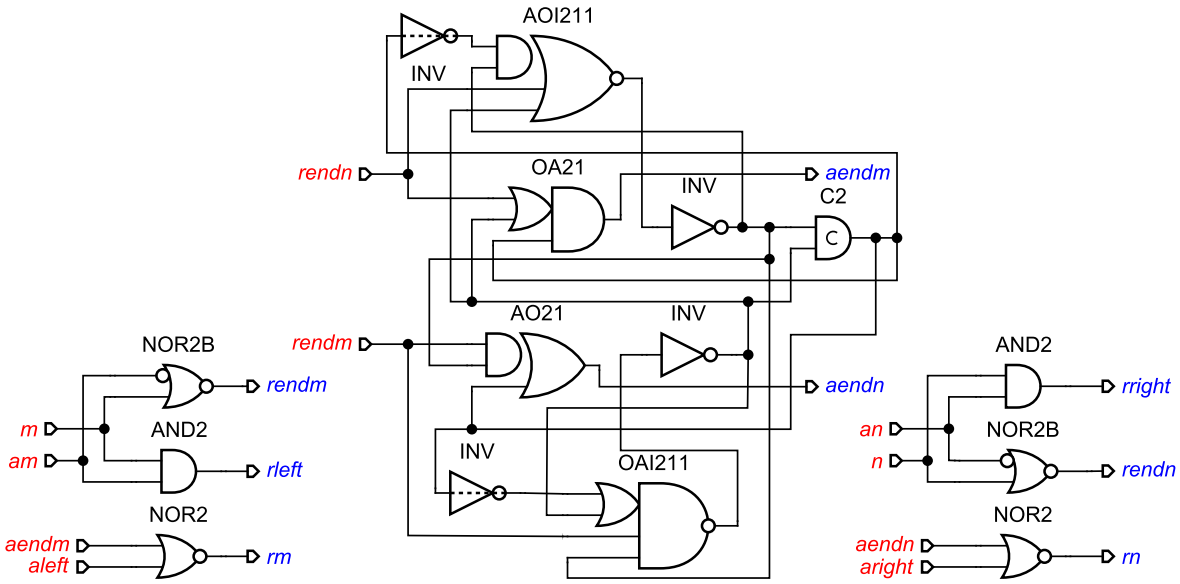
Next, we can verify in Workcraft that each STG satisfies the standard STG implementability properties that include consistency, deadlock-freeness, input properness, output-persistency and output determinacy.

Lastly, once verification is complete, we can synthesise the three STGs using either Petrify or Mpsat backends to produce some possible circuit implementations like the SI (QDI) circuits shown in Figure 3.4.



(a) Left Side (b) Sync Component (c) Right Side

Figure 3.3: STG of each controller part



(a) Left Side (b) Sync Component (c) Right Side

Figure 3.4: Possible Circuit Implementation of the Controller Parts

Alternatively, if we want to compose the STGs together, then we can easily use WORKCRAFT to perform composition of our three STGs using the PCOMP backend, as shown in Figure 3.5 Note that all shared signals are converted into outputs and the composed STG is set to guarantee N-way conformation between the three STGs.

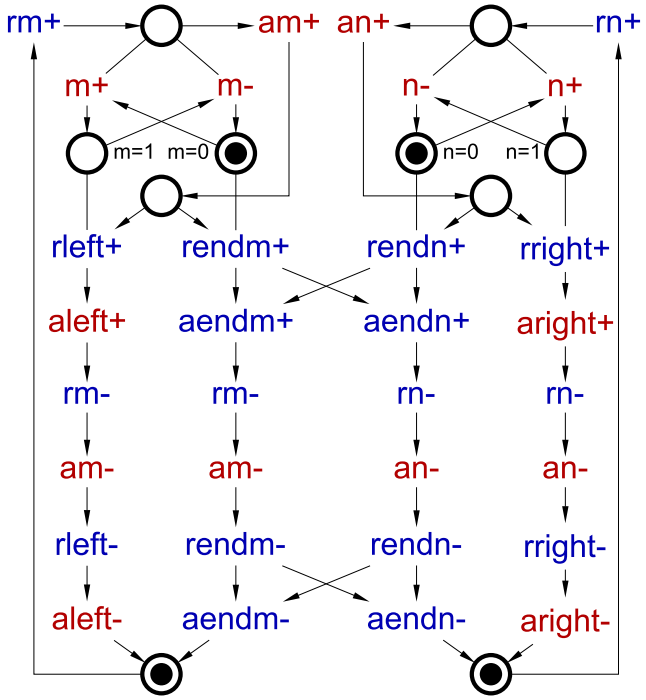


Figure 3.5: Composed STG from the Decoupling Handshake Controller Parts

Again, we can verify in WORKCRAFT that this composed STG also satisfies the aforementioned standard STG implementability properties, and then synthesise it using either PETRIFY or MPSAT backends to produce a possible circuit implementation like the SI (QDI) circuit shown in Figure 3.6.

Nevertheless, while STGs are shown to be flexible, i.e. they are concurrent, can express behaviours not available in (X)BM specifications, can be easily composed and have access to well-established tools like PETRIFY and MPSAT, the problem with the 'disruptive' design route is that circuit designers are not generally familiar with

event-based methods. Instead, they rather opt for the 'legacy' design route as they are more familiarised with state-based methods, despite the disadvantages that this route may pose, e.g. not well-optimised circuit implementations.

In fact, this need for the circuit designer's familiarity of FSMs can even be seen in some STG-based approaches, e.g. Extended STGs (XSTGs) that has some elements of XBM specifications [45, 24], several design flows that partition STGs into several XBM specifications [38, 12, 43], and an STG-based flow with some FSM entry [25].

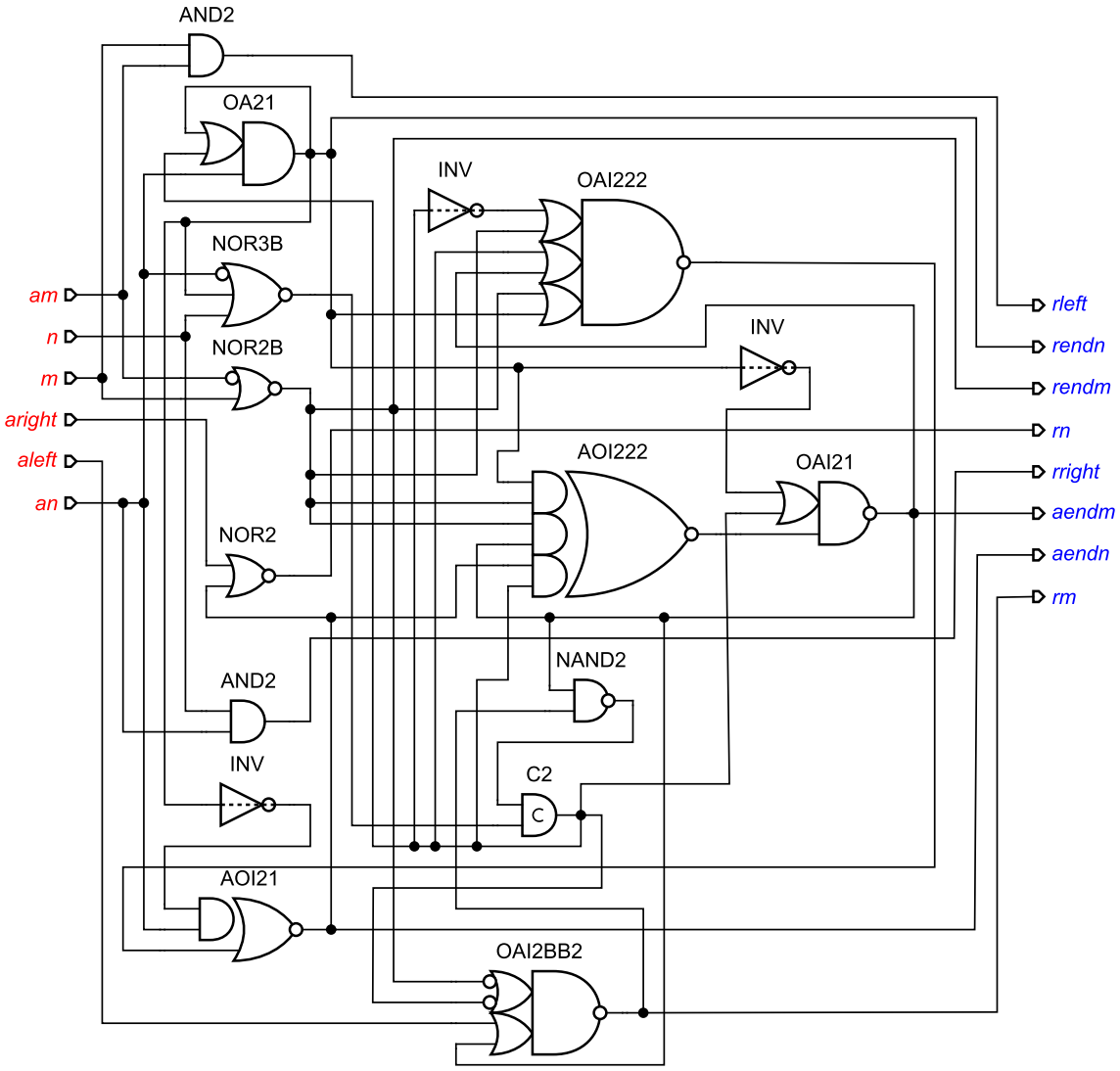
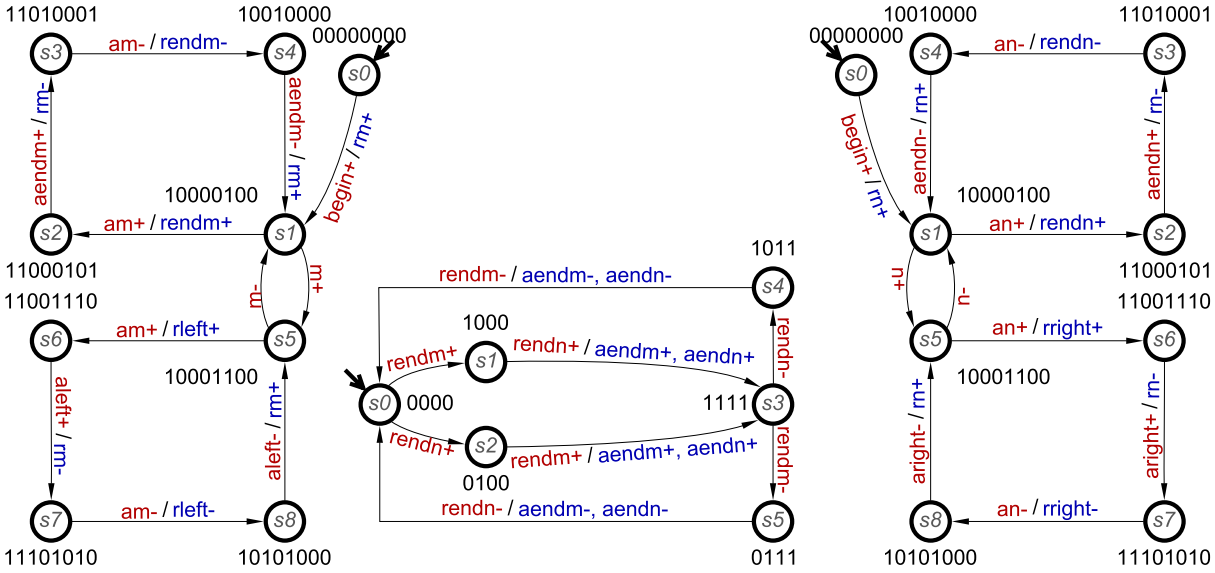


Figure 3.6: Possible Circuit Implementation of the Composed Controller

Now, let us consider using the 'legacy' design route. Here, we try to design and capture the specified behaviour of the controller using BM specifications, while following the STG's transition sequence in Figure 3.3, as shown in Figure 3.7.



(a) Left Side (b) Sync Component (c) Right Side

Figure 3.7: BM Specification of each controller part

Note that if the rising and falling of the environment conditions m and n is undetermined, then it is more convenient to use XBM specifications to capture m and n with conditionals, as this would otherwise require the whole model of the BM specification to be replicated at least once. But, as shown in the left and right STGs in Figure 3.3, we can interpret m and n to only rise or fall when there is a token at the place between rm and am, and the place between rn and an respectively. Thus, we can capture this behaviour using BM specifications.

When we verify the three BM specifications, they satisfy BM's well-formed requirements that include the maximal set property (as we have no non-deterministic choices), non-empty input burst property (as all outputs appear with an input), and the unique state entry condition (as encodings are not changed by signal changes).

However, while the three BM specifications are all well-formed, there are two compromises that had to be made. In particular:

1. A new input called begin is added at the start of the controller's left side and the controller's right side, as the controller's left side initiates its handshake with the left generator by firing output event rm+ and the controller's right side initiates its handshake with the right generator by firing output event rn+. If the input events begin+ are not included in either BM specification then the non-empty input burst property is violated. So, we added the begin+ input events and assume the environment sending these events is well-behaved.

2. Due to the structure of bursts in the 'burst-mode' timing assumption, the output events aendm and aendn in the controller's synchroniser must be grouped up. This means that both input events rendm and rendn will always follow both output events aendm and aendn despite the environment is distributed, where only rendm should follow aendm and only rendn should follow aendn as shown in the synchroniser STG in Figure 3.3b. Note that grouping the output events aendm and aendn also causes a conformation violation between both sides of the controller and the synchroniser, as the controller's left side is forced to wait for aendn and the controller's right side is forced to wait for aendm, despite neither side should wait for the other to receive their respective outputs (e.g. the controller's left side should be able to send rendm after receiving aendm from the synchroniser without waiting for the controller's right side to receive aendn). Typically, we resolve this by interleaving the output events aendm and aendn to force some order. But, this is not possible in BM specifications without violating their non-empty input burst property.

Furthermore, if we wish to compose these BM specifications together, this is unfortunately impossible (or at least extremely difficult), as there are no automated methods that are available to conveniently compose (X)BM specifications or compose their circuit implementations. Also, the 'burst-mode' timing assumption makes it non-trivial to compose (X)BM specifications due to the coupling of bursts [17], and BM's well-formed requirements can limit how the (X)BM specifications are composed, e.g. composing two BM specifications may result in a non-deterministic choice, which violates the maximal set property (or the distinguishability constraint in the case of XBM specifications), or bursts that only contain outputs after the shared signals are turned into outputs, which violates the non-empty input burst property.

3.3 Proposal of Burst Automaton Route

To address the issues that are described for the 'legacy' design route of BM specifications and the 'disruptive' design route of STGs, we propose a new model called Burst Automaton (BA) [16]. BA is a generic FSM-like model that labels its arcs with sets of actions, where it can specify many types of systems including concurrent systems, asynchronous circuits, and distributed systems.

BA also provides the necessary framework to enable interoperability between many types of models including FSMs, BM specifications, XBM specifications, Petri nets, STGs, and generally any model that can be automatically translated into any of these, e.g. Waveform Transition Graphs (WTGs) [50] which have STG-based semantics.

Note that Chapter 4 covers BAs in greater detail, where it discusses BA's model description, mathematical definitions (i.e. formal definition and asynchronous reachability graph), translation methods to STG including XBM components (e.g. conditionals and "don't cares"), and distribution method to compose several BAs together.

By proposing BAs, we create a new 'co-design' route that establishes a connection between the 'legacy' and 'disruptive' design routes as shown in Figure 3.8, where the 'legacy' design route's flow is highlighted in red, the 'disruptive' design route's flow is highlighted in blue, the 'co-design' route's flow is highlighted in green, and all common paths are not highlighted with a particular colour.

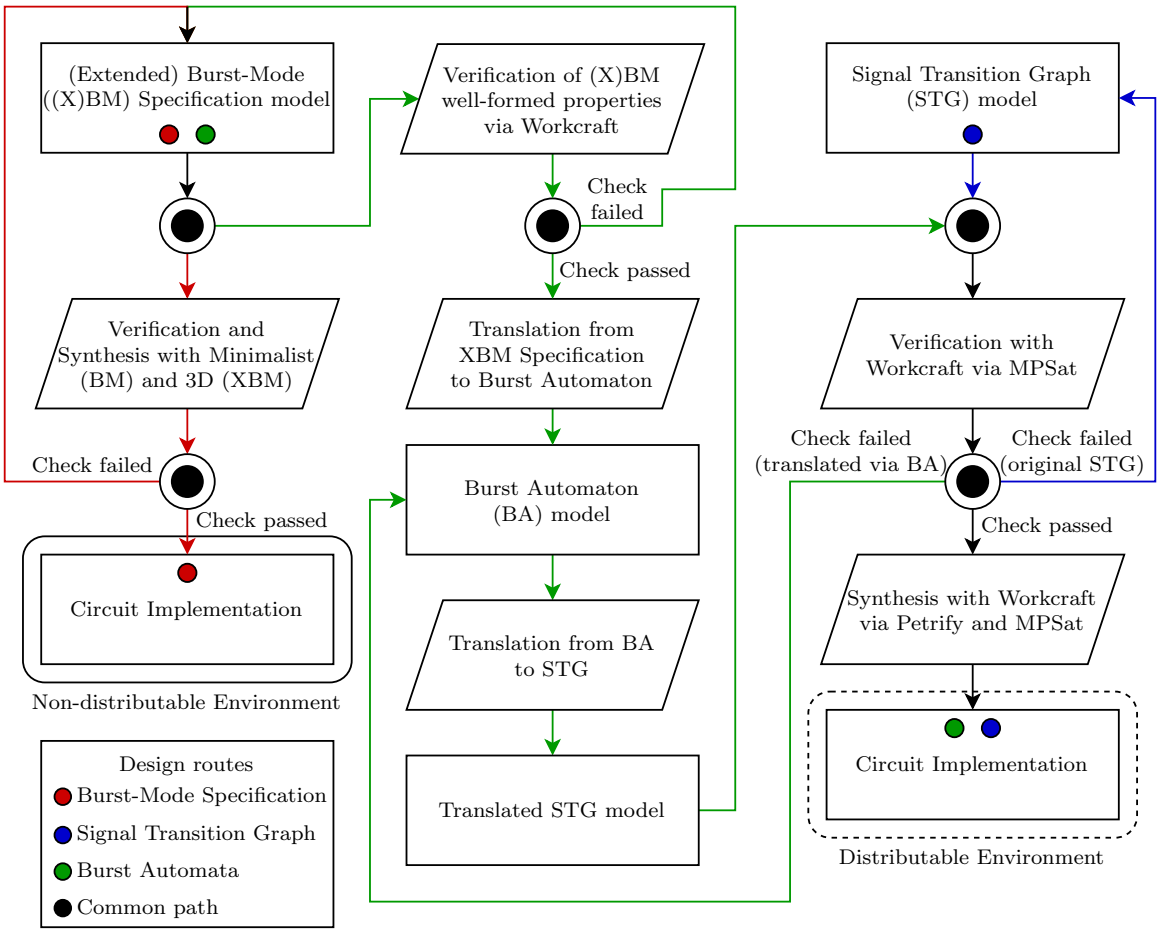


Figure 3.8: Proposal of new 'co-design' route established by Burst Automata

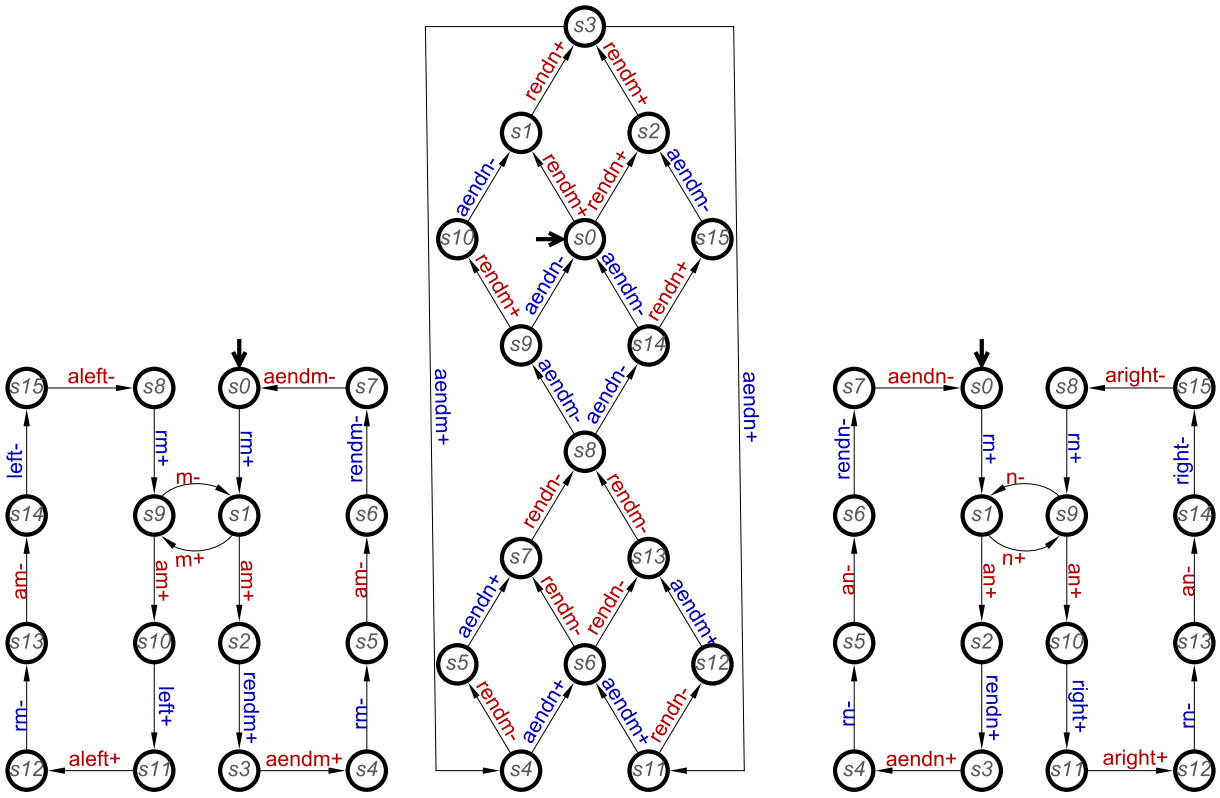
In this 'co-design' route, circuit designers can create their (X)BM specifications, automatically translate them to BAs for subsequent translation to STGs, where they can access the STG's tools to compose their (X)BM specifications using PCOMP and synthesise them into SI (QDI) circuits using PETRIFY and MPSAT.

This approach essentially allows circuit designers to produce well-optimised circuit implementations using (X)BM specifications, where it does not compromise the circuit designer's familiarity of FSMs and the circuit designer does not need to re-specify their systems using STGs, as this design route automates the translation process from (X)BM specifications to STGs and hides it from the circuit designer's view. Thus, this grants circuit designers convenient access to the STG's tools for their (X)BM specification without the need to train and understand the STG's semantics.

To understand the design, verification and synthesis processes in this 'co-design' route, let us revisit the handshake decoupler and design all three parts of its asynchronous controller. Here, we can easily design and capture the specified behaviour of the handshake decoupler's controller using BAs, while also following the STG's transition sequence in Figure 3.3, as shown in Figure 3.9. Note that in some cases, we can also translate (X)BM specifications to BAs by relaxing their inherent 'burst-mode' timing assumption and their well-formed requirements, as later shown in Section 6.1.

Although the BAs in Figure 3.9 are larger than the BM specifications in Figure 3.7, BA's fine-grain style of specifying behaviour allows us to easily capture the decoupling of handshakes rendm/aendm and rendn/aendn, whereas BM specification's coarse-grain style of specifying behaviour cannot do so without violating one of its well-formed requirements or the 'burst-mode' timing assumption.

Once the three BAs are designed, we can translate them to the STGs shown in Figure 3.10, where we can even simplify these STGs by 'net synthesis' using Petrify to obtain similar or the same STGs in Figure 3.3. We can then verify in Workcraft that the three translated STGs satisfy the standard STG implementability properties (i.e. consistency, deadlock-freeness, input properness, output-persistency and output determinacy), and synthesise them using Petrify or Mpsat backends to produce the same possible circuit implementations like the SI (QDI) circuits shown in Figure 3.4.



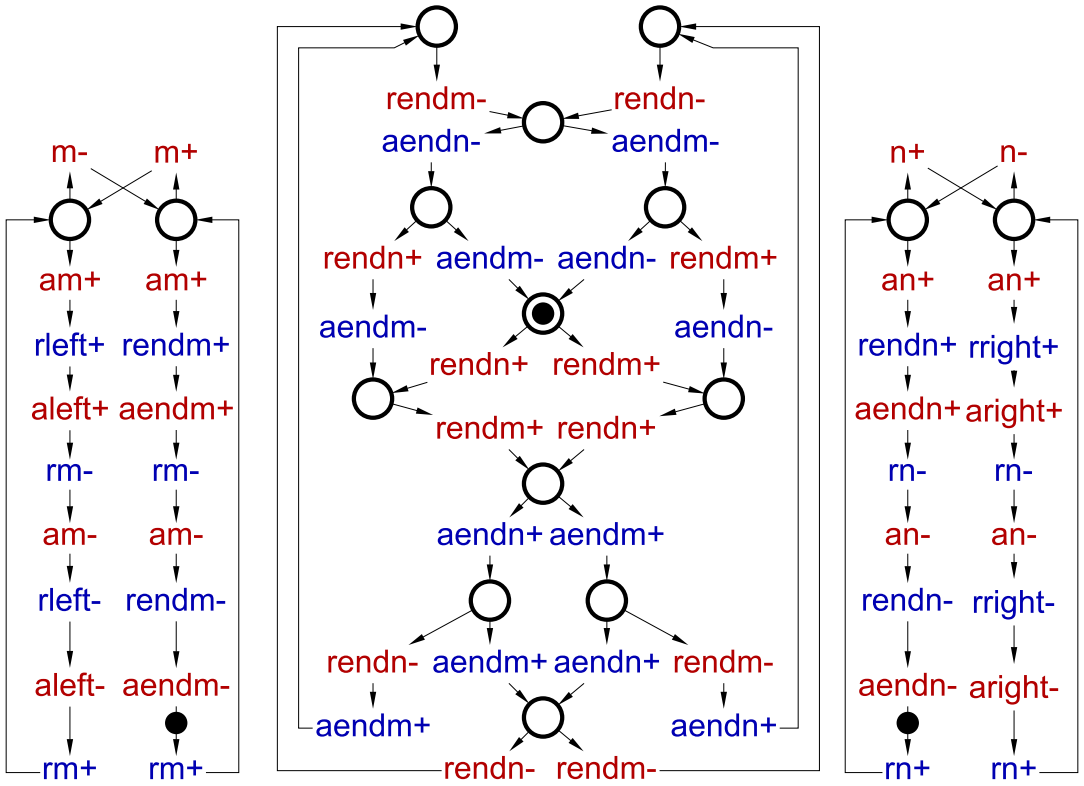
(a) Left Side (b) Sync Component (c) Right Side

Figure 3.9: BA of each controller part

Alternatively, we can also compose the three translated STGs together using PCOMP to obtain the composed STG shown in Figure 3.11.

Again, we can verify in WORKCRAFT that this composed STG also satisfies the aforementioned standard STG implementability properties, and then either synthesise it using PETRIFY or MPSAT backends to produce the same possible circuit implementation like the SI (QDI) circuit shown in Figure 3.6, or build its reachability graph that can be interpreted as the composed BA.

Note that the reachability graph of the composed STG is not included in this section and can instead be found in Appendix A via Figure A.1, as the model cannot reasonably fit due to its monolithic size after translating many concurrent transitions.



(a) Left Side (b) Sync Component (c) Right Side

Figure 3.10: Translated STG of each controller part

However, one behaviour that cannot be easily implemented by BAs is *mutual exclusion (mutex) elements* [59], which are used to resolve *metastability* issues.

In circuit design, systems are required to make an arbitrary decision, where metastability can persist for a long period of time. Although the probability of a long delay is small, if this indecision is repeated enough times, then this can cause a malfunction in some systems like synchronous circuits, e.g. when a delay becomes longer than a clock cycle. Additionally, metastability is analogue by nature and should therefore not propagate into the digital part of the system.

For asynchronous circuits, mutex elements are introduced to resolve this metastability issue. Although mutex elements do not fail, it can take an indefinitely long time to resolve. Thus, it is important that they are removed from critical paths.

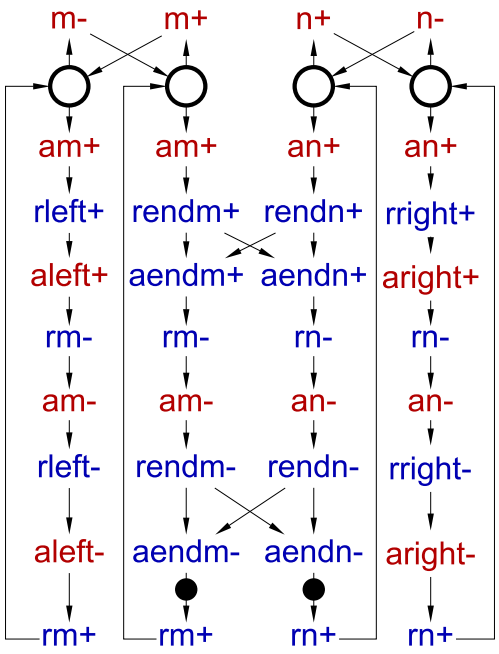
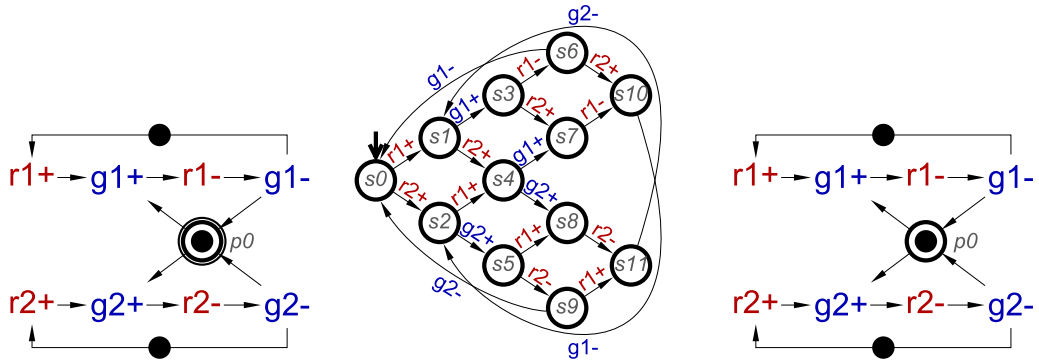


Figure 3.11: Composed Translated STG of the Handshake Decoupler's Controller



(a) Output persistent STG (b) BA (c) Non-output persistent STG

Figure 3.12: Modelling a mutex element

In STGs, we can simply tag a place as a 'mutex' using WORKCRAFT as shown in Figure 3.12a. But, this is not possible at the level of BAs. If we design a system that contains some arbitration, e.g. an N-way arbiter, as the BA in Figure 3.12b and translate it to the STG in Figure 3.12c, then this can violate output persistency unless we tag the appropriate place as a 'mutex' in the translated STG.

3.4 Benefits from Burst Automaton Route

From this modelling exercise, we can identify several potential areas of improvements for the 'legacy' design route of (X)BM specifications and the 'disruptive' design route of STGs, which our 'co-design' route of BAs achieves. These include:

- **Access to the STG's well-established tools for enabling composition, verification and synthesis of SI (QDI) circuits for BM specifications**: Using BAs, we grant BM specifications access to the STG's well-established tools after subsequent translations to STGs. This, in particular, enables the necessary step for the verification and synthesis of BM specifications into well-optimised SI (QDI) circuit implementations using PETRIFY and MPSAT backends, and the composition of BM specifications (at the level of BAs) using the PCOMP backend. With the support of WORKCRAFT, we are also able to design, simulate and translate our BM specifications and BAs seamlessly.

- **Familiarity for state-based circuit designers**: Because BM specifications can be implemented by BAs, this allows state-based circuit designers to access the STG's well-established tools by simply translating their BM specifications into BAs and then subsequently into STGs. This means that state-based circuit designers are not required to fully understand the design and semantics of STGs or re-specify their systems using STGs, as BAs automate the process of building an STG from a (X)BM specification in the background through the implemented tool support in WORKCRAFT, which is covered in Chapter 5

- **Simpler model definition for (X)BM specifications**: By generalising some of the (X)BM specification's well-formed properties, e.g. the maximal set property and non-empty input burst property, BAs can express many behaviours in-

cluding input-output concurrency, output choices and non-deterministic choices, while enabling an easy way to compose (X)BM specifications. In fact, BAs are easier to define than (X)BM specifications as there are no constraints (i.e. well-formed requirements) that determine how systems are specified, and we even ensure that the state-based circuit designer's familiarity of FSMs is retained through the BA's design. Moreover, BAs are more symbolic and are more of a translation-oriented model, meaning it has less "physicalities" than (X)BM specifications, e.g. the states in BAs are not physically implemented as states variables and do not store any information, whereas the states in (X)BM specifications store the encoded value of every signal.

- **Model interoperability**: There are great advantages of modelling formalisms that have compatible semantics, as this allows a fluid transition (via an automatic translation process) from one formalism to another, and allows the design of large systems using pre-existing blocks that are expressed in different formalisms. With BAs, we achieve interoperability between BM specifications, XBM specifications, STGs, FSMs, Petri nets, and generally any formalism that can be automatically translated into any of these formalisms, e.g. WTGs [50] which have STG-based semantics.

3.5 Summary

In this chapter, we cover the current design routes that are available for circuit designers, where the first, i.e. 'legacy', design route involves BM specifications and the second, i.e. 'disruptive', design route involves STGs.

For our motivating example, we use a handshake decoupler to highlight and analyse the processes involved in both the 'legacy' and 'disruptive' design routes.

In the 'legacy' design route, (X)BM specifications are able to model each part of the handshake decoupler's asynchronous controller sufficiently without violating any of the (X)BM well-formed requirements, and can be synthesised into a BM controller. However, several compromises are also made to each model to ensure that the (X)BM well-formed requirements and the 'burst-mode' timing assumption are not violated (e.g. adding begin events to prevent violating the non-empty input burst property, and coupling the handshakes rendm/aendm and rendn/aendn to prevent violating the 'burst-mode' timing assumption). Additionally, there are also no methods that can conveniently compose (X)BM specifications together, meaning manual composition is required, which can be a non-trivial and very time consuming task.

In the 'disruptive' design route, STGs are able to model each part of the handshake decoupler's asynchronous controller without any compromises, as they are flexible, can be easily composed, and can be easily verified and synthesised into an SI (QDI) circuit using their well-established tools. But, despite these advantages, the industry is not familiar with event-based methods like STGs and rather opt for more familiarised state-based approaches like (X)BM specifications, as using STGs require training the circuit designers, which can be expensive and time-consuming.

Thus, we propose a new model called BA, which is generic-FSM like model that labels its arcs using sets of actions and acts as a framework that enables interoperability between many models including (X)BM specifications and STGs. By proposing BAs, we also create a new 'co-design' route that establishes a connection between the 'legacy' design route and the 'disruptive' design route, and enable the co-design of (X)BM specifications and STGs, where we can verify and synthesise the (X)BM specifications into SI (QDI) circuits and compose a group of (X)BM specifications together through an automated method, which translates BM specifications to BAs and subsequently STGs.

Chapter 4

Burst Automata

In this chapter, we will study the Burst Automaton (BA) model, which is the main contribution of this thesis.

In Section 4.1, we will cover the model description of BAs, where we provide a textbook definition of BAs and a basic view of the BA's design.

In Section 4.2, we will cover the mathematical definitions for BAs that include their formal definition and their asynchronous reachability graph, where the latter can be used to check for model equivalences with the reachability graph of another model, e.g. Signal Transition Graphs (STGs).

In Section 4.3, we explore the three translation methods from BAs to STGs, where each translation method preserves differing model equivalences (i.e. the language, weak bisimulation and strong bisimulation), and the translation method for Extended Burst-Mode (XBM) specification's components to their STG counterparts.

Finally, in Section 4.4, we investigate the distribution methodology for BAs, where we provide a textbook definition of a distributed Finite State Machine (FSM) and their distribution criteria. We will then cover the composition method of BAs, which include the verification and synthesis of speed-independent (SI) (or equivalently, quasi-delay-insensitive (QDI)) circuits via STGs.

4.1 Model Description

In this section, we cover the model concept of BAs. Here, we discuss the design of BAs and provide some basic examples, where we will describe the properties of BAs and highlight the differences between BAs and FSMs. Next, we will highlight the purpose and benefits of BAs, before we show the design workflow of BAs and explain how it achieves interoperability between many different models.

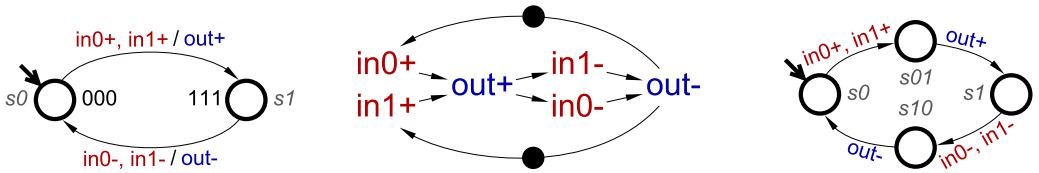
Note that for the comparison between BAs and FSMs, we will focus on Burst-Mode (BM) specifications, as they can be seen as a sub-class of fundamental mode asynchronous FSMs that operate in the 'burst-mode' timing assumption, where the fundamental mode is essentially 'extended' to work with bursts.

4.1.1 Definition of Burst Automaton

BA [16] is a generic FSM-like model that acts as a framework for enabling interoperability between many types of models, and is akin to Mealy machines [44].

The BA's framework allows us to dynamically change between many models including FSMs, BM specifications, XBM specifications, Petri nets and Signal Transition Graphs (STGs), which can be particularly useful when we wish to change our specification from one model (e.g. BM specification) to another model (e.g. STG), or even combine specifications that may be expressed using different models.

BA also allows us to create a new design route for circuit designers, where we establish a connection between the 'legacy' design route (X)BM specifications and the 'disruptive' route of STGs. This essentially grants BM specification access to the well-established tools that are available for STGs, which enables subsequent composition, verification and synthesis of SI (QDI) circuits via translation to STGs [15].



(a) BM Specification (b) STG (c) BA

Figure 4.1: Design of C-element Gate

4.1.2 Summarised Design of Burst Automaton

To understand the general design of BAs, let us first consider the C-element gate that is specified as a BM specification in Figure 4.1a and as an STG in Figure 4.1b, before we view the C-element gate that is specified as a BA in Figure 4.1c.

For the purpose of this example, we will only focus on the design aspect of the C-element gate. By observing the BM specification, the STG and the BA, there are two key points that can be made about these models:

1. We can see that the design of these three models are very similar. Indeed, the firing sequence of the signals are the same for all three models, where the key difference is that the BM specification fires its burst in the order of inputs then outputs, the STG fires its concurrent transitions that comprise the burst, and the BA fires its set of signals using arcs that require a step for inputs and another step for outputs.

2. The states in the BM specification and the BA are akin to the places in the STG, as the STG is 1-safe. The BA also models the intermediate state between the BM specification's input burst and output burst, where this intermediate state represents when the inputs have arrived and the outputs are not yet produced.

Due to the similiarity between the designs of these three models, we can use BAs as the intermediate model to translate between BM specifications and STGs.

Thus, this allows BAs to act as a framework that enables interoperability between BM specifications, STGs, FSMs, or any other models that share their semantics, e.g. Waveform Transition Graphs (WTGs) [50] which uses STG semantics.

Note that while WTGs are also introduced as a new formal model to bridge the gap between designers and formal models like BAs, the design of WTGs are also based on waveforms that are generally understood by analogue circuit designers but not digital circuit designers, whereas BAs resemble FSMs that are generally understood by both analogue circuit designers and digital circuit designers. Thus, this allows BAs to be easily incorporated in other design flows of asynchronous circuit design, e.g. the analogue-asynchronous design flow [61].

4.2 Mathematical Definitions

In this section, we will cover the formal definition of BAs and the asynchronous reachability graph of BAs, where the latter can be used to check for model equivalence with the reachability graph of another model, e.g. STGs.

4.2.1 Formal Definition

Generally, a BA can be interpreted as an FSM with its arc labels being sets of actions, where it is defined as a tuple $B = (\Sigma, S, A, s_0)$ such that:

- Σ is an alphabet of atomic actions.
- S is a finite set of states.
- $A \subseteq S \times 2^{\Sigma} \times S$ is the set of arcs determining the flow relation.
- $s_0 \in S$ is the initial state.

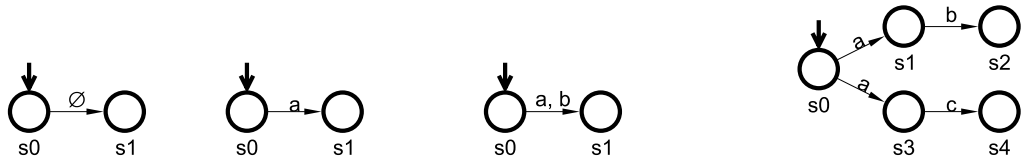
In this definition, we do not partition the BA's alphabet into inputs and outputs nor do we assign any directions (i.e. + or −) to its actions, as these can instead be viewed as refinements of the BA since it is a generic model.

There are no requirements that are similar to the BM specification's maximal set property nor the XBM specification's distinguishability constraint, as BAs allow arbitrary non-determinism, e.g. it is possible for two distinct arcs to originate from the same state and be labelled by sets of actions that are a subset of or equal to each other. It is also possible to have an empty set of actions as an arc label, which can be interpreted as an ε transition. In fact, we can see that BAs naturally extend FSMs, where the latter are essentially BAs that have its arcs labelled by singletons or by $\varnothing$.

In (X)BM specifications, bursts typically comprise a set of inputs followed by a set of outputs. In BAs, we can model these bursts by explicitating the intermediate state where all inputs have arrived and no outputs have been produced, such that a BA requires two steps to fire an (X)BM-like burst. Note that it is possible for the graphical notation to hide this intermediate state and be similar to the (X)BM notation. Alternatively, it is also possible to mix inputs and outputs in the BA's arc labels, which in particular enables input-output concurrency.

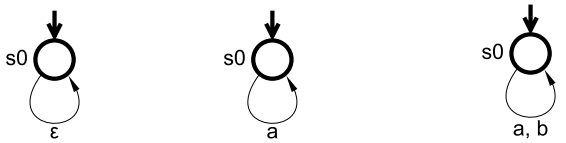
Furthermore, we can consider the XBM specification's conditionals and "don't cares" as auxiliary components that can be attached to the BA to create an XBM-like model. Note that the translation from XBM specification's conditionals and "don't cares" to their STG equivalent is covered in Section 4.3.4.

To help us understand the design of our BA formally, Figure 4.2 shows several BAs with arcs labelling $\varnothing$, a set containing a singleton, a set containing multiple actions, and a non-deterministic choice between two sub-related sets, while Figure 4.3 shows several BAs with self-loop arcs labelling $\varnothing$, a set containing a singleton, and a set containing multiple actions.



(a) Empty set (b) Singleton (c) Multiple actions (d) Non-deterministic choice

Figure 4.2: Examples of BA's labelled arcs



(a) Empty set (b) Singleton (c) Multiple actions

Figure 4.3: Self loop examples of BA's labelled arcs

4.2.2 Reachability Graph

In addition to the formal definition of BAs, we can also assign interleaving semantics to the BAs by defining their asynchronous reachability graph, where the actions in the labelled arc's set can fire in any order. These actions are atomic, as BAs are a natural extension to FSMs, where the reachability graph of a BA can be interpreted as a BA with all of its arcs labelled by singletons or by $\varnothing$.

Note that it is not possible to interpret the reachability graphs of (X)BM specifications as (X)BM specifications, because they are not a proper extension of FSMs with arcs labelling single events. In particular, the reachability graphs of (X)BM specifications cannot be well-formed (X)BM specifications, as the 'burst-mode' timing assumption requires the input bursts and output bursts to alternate, and the singular events expressing outputs violate the non-empty input burst property.

By defining the reachability graph of BAs, we can directly compare the BA with another formalism, e.g. STG, through their reachability graphs and check their bisimulation relation to each other.

To help us understand the formal definition of the BA's asynchronous reachability graph, Figure 4.4 shows a simple example of some BA with the labelled arcs 'a', 'b, c' and 'x, y, z' in Figure 4.4a and its reachability graph in Figure 4.4b. In this example, we will categorise each component found in the BA's reachability graph and provide a description of its role within the model. These components include:

- *Original states* that appear in the original BA. In this example, we have the original states s'_0, s'_1, s'_2 and s'_3 where s'_0 is the initial state.

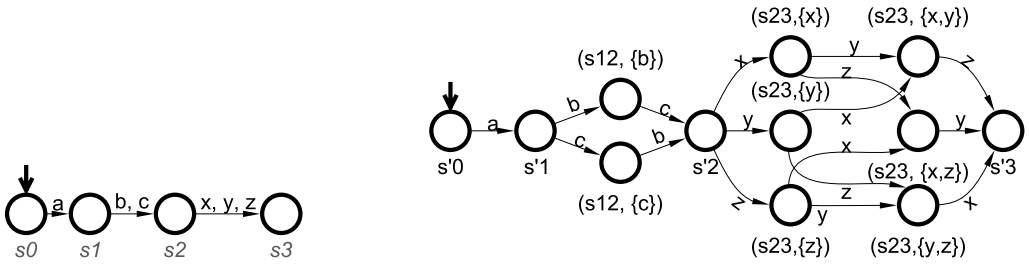
- *Intermediate states* that appear after firing some action that does not reach an original state. In this example, we have the intermediate states $(s12, \{b\})$, $(s12, \{c\})$, $(s23, \{x\})$, $(s23, \{y\})$, $(s23, \{z\})$, $(s23, \{x, y\})$, $(s23, \{x, z\})$ and $(s23, \{y, z\})$.

- *Original to original connections*, which are arcs between the BA's original states. In this example, we only have one original to original arc that is $(s0, a, s1)$.

- *Original to intermediate connections*, which are arcs from an original state to an intermediate state. In this example, we have the original to intermediate connections $(s1, b, (s12, \{b\}))$, $(s1, c, (s12, \{c\}))$, $(s2, x, (s23, \{x\}))$, $(s2, y, (s23, \{y\}))$ and $(s2, z, (s23, \{z\}))$.

- *Intermediate to intermediate connections*, which are arcs between intermediate states. In this example, we have the intermediate to intermediate connections $((s23, \{x\}), y, (s23, \{x, y\}))$, $((s23, \{x\}), z, (s23, \{x, z\}))$, $((s23, \{y\}), x, (s23, \{x, y\}))$, $((s23, \{y\}), z, (s23, \{y, z\}))$, $((s23, \{z\}), x, (s23, \{x, z\}))$ and $((s23, \{z\}), y, (s23, \{y, z\}))$.

- *Intermediate to original connections*, which are arcs from an intermediate state to an original state. In this example, we have the intermediate to original connections $((s23, \{x, y\}), z, s3)$, $((s23, \{x, z\}), y, s3)$ and $((s23, \{y, z\}), x, s3)$.



(a) BA (b) Reachability graph

Figure 4.4: Simple example demonstrating the design of a BA's reachability graph

From this simple example, we can formally define the asynchronous reachability graph of a BA B as an arc-labelled directed graph $RG_B = (\Sigma, S', A', s'_0)$ where:

- The alphabet Σ is inherited from B.
- $S' = S \cup (A \times (2^{\Sigma} \setminus \{\varnothing, \Sigma\}))$ is the set of reachable states such that for every pair $(a, D') \in S'$, $a = (s_u, D, s_v) \in A$ is a labelled arc and $\varnothing \neq D' \subset D$ is a subset of a's label.
- $A' \subseteq S' \times (\Sigma \cup \{\varepsilon\}) \times S'$ is the set of labelled arcs, which establishes the following connections between the states:

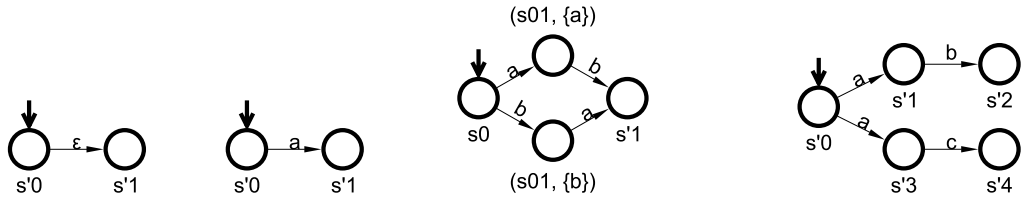
  Original $\rightarrow$ Original: $(s_u, \varepsilon, s_v) \in A'$ if $(s_u, \varnothing, s_v) \in A$ and $(s_u, l, s_v) \in A'$ if $(s_u, \{l\}, s_v) \in A$.

  Original $\rightarrow$ Intermediate: $(s_u, l, (a, \{l\})) \in A'$ if $a = (s_u, D, s_v) \in A$ and $l \in D$.

  Intermediate $\rightarrow$ Intermediate: $((a, D'), l, (a, D' \cup \{l\})) \in A'$ if $a = (s_u, D, s_v) \in A$ and $l \notin D'$ and $D' \cup \{l\} \subset D$.

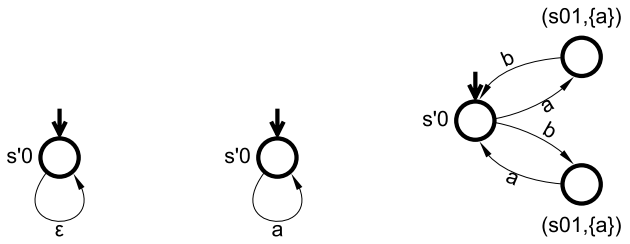
  Intermediate $\rightarrow$ Original: $((a, D'), l, s_v)) \in A'$ if $a = (s_u, D, s_v) \in A$ and $l \notin D'$ and $D' \cup \{l\} = D$.
- $s'_0 = s_0$ is the initial state.

Again, to help us understand the design of our BA's reachability graph formally, Figure 4.5 shows several reachability graphs of BAs with arcs labelling $\varnothing$, a set containing a singleton, a set containing multiple actions, and a non-deterministic choice between two sub-related sets, while Figure 4.6 shows several reachability graphs of BAs with self-loop arcs labelling $\varnothing$, a set containing a singleton, and a set containing multiple actions.



(a) Empty set (b) Singleton (c) Multiple actions (d) Non-deterministic choice

Figure 4.5: Examples of BA's reachability graph's labelled arcs



(a) Empty set (b) Singleton (c) Multiple actions

Figure 4.6: Self loop examples of BA's reachability graph's labelled arcs

When we observe the reachability graph of a BA, we can see that it is just an FSM (or even a BA with all of its arcs labelled by singletons and $\varnothing$) and is very similar to the reachability graph of STGs. Note that the size of the BA's reachability graph is exponential to the maximal cardinality of the BA's bursts, although large bursts are rare in practice.

4.3 Translation to Signal Transition Graphs

In this section, we will cover three translation methods from BAs to STG and a translation method from the XBM specification's components to their STG counterparts.

The first translation method in Section 4.3.1 produces an STG with "fork" and "join" dummy transitions, which preserves the language and is linear to the size of the original BA.

The second translation method in Section 4.3.2 produces an STG with "join" dummy transitions, which preserves weak bisimulation but is potentially exponential to the size of the original BA.

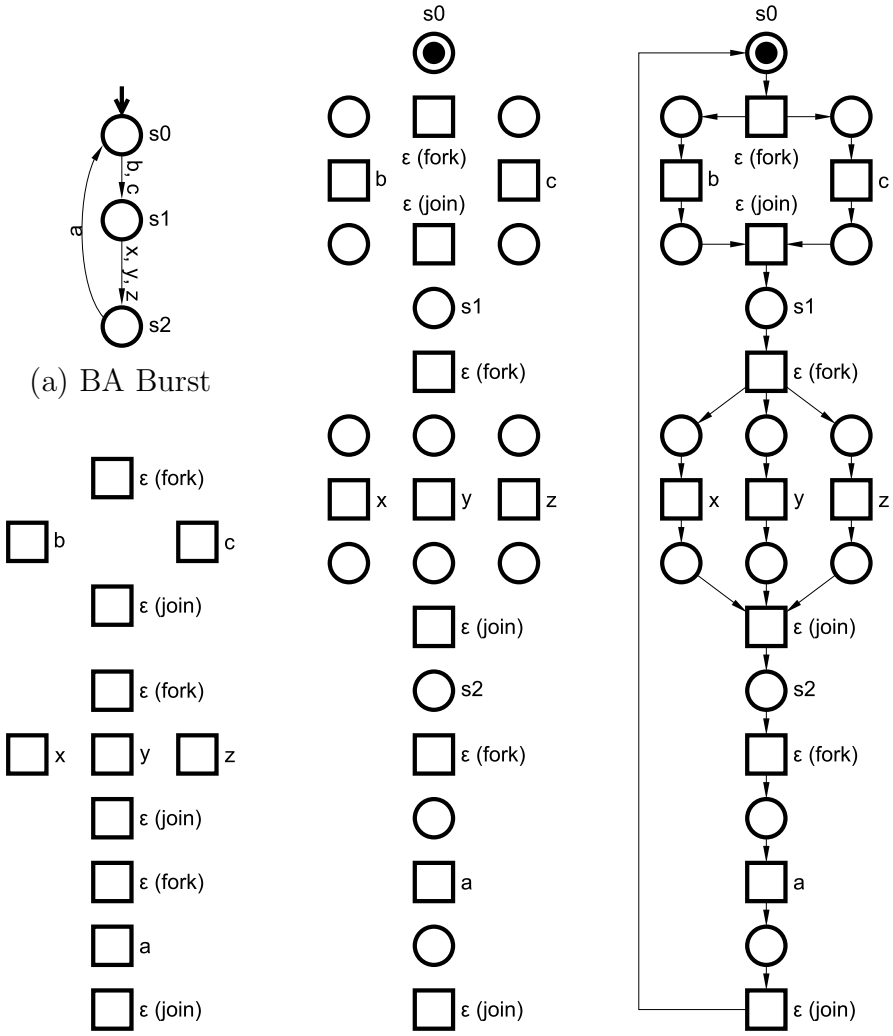
The third translation method in Section 4.3.3 produces an STG with neither "fork" nor "join" dummy transitions, which preserves strong bisimulation but is also potentially exponential to the size of the original BA.

Finally, the translation method from the XBM specification's components to their STG counterpart in Section 4.3.4 explicitates fake outputs and "don't cares", and translates conditionals into elementary cycles with lock places.

4.3.1 STGs with dummy transitions

In this translation, we produce an STG that is language equivalent to the original BA. Intuitively, the BA's states are translated into STG places where the initial state is represented by the place containing a token, and the BA's bursts are translated into concurrent STG transitions where two extra dummy transitions (i.e. ε-labelled transitions) called 'fork' and 'join' are created for each burst.

To help understand how this translation works, Figure 4.7 shows a simple example where we translate a BA with the labelled arcs '`a`', '`b, c`' and '`x, y, z`' to an STG.



(a) BA Burst (b) Transitions (c) Places (d) Arcs

Figure 4.7: Translating a BA to an STG with "fork" and "join" dummy transitions

To begin, we create six transitions labelled a, b, c, x, y and z respectively, where transition a comprises the labelled arc 'a', transitions b and c comprise the labelled arc 'b, c', and transitions x, y, and z comprise the labelled arc 'x, y, z'. Then, we create two additional dummy transitions called 'fork' and 'join' for each labelled arc.

Next, we create a place for each state $s0$, $s1$ and $s2$, before we add a token to the place corresponding to $s0$ as it is the initial state. Then, for each labelled transition $t \in \{a, b, c, x, y, z\}$, we create one incoming place and one outgoing place.

Now, for each state $s \in \{s0, s1, s2\}$, we connect an arc from the place corresponding to s to the 'fork' dummy transition created from the labelled arc where s is the source state, and an arc from the 'join' dummy transition created from the labelled arc where s is the destination state to the place corresponding to s. Then, for each labelled arc $a \in \{(s2, `a', s0), (s0, `b, c', s1), (s1, `x, y, z', s2)\}$, we connect arcs from the 'fork' dummy transition created from $a$ to all incoming places created by $a$'s label, and from all outgoing places created by $a$'s label to the 'join' dummy transition created from $a$. Finally, for each labelled arc $a \in \{(s2, `a', s0), (s0, `b, c', s1), (s1, `x, y, z', s2)\}$, we connect an arc from each incoming place created by a's label to the labelled transitions that comprise a's label, and an arc from the labelled transitions that comprise a's label to each outgoing place created by a's label, such that there is only one incoming place and one outgoing place for each each labelled transition.

By completing this example, we can determine how a BA is translated into an STG with this method, and formalise the above steps for every burst found in the BA as shown below, where each step is categorised by the STG component being created.

Transitions: For each arc in the BA, we create two ε-labelled STG transitions called 'fork' and 'join' and a transition for every burst label element, such that:

- If the arc's label is $\{\sigma_1, \sigma_2, ..., \sigma_k\}$, where $k \geqslant 1$, then we create k STG transitions labelled σ_1, σ_2, ..., σ_k.
- If the burst's label is $\varnothing$, then we create an STG transition labelled ε.

Places: For each state in the BA, we create a place and add a token to this place if it corresponds to the BA's initial state. Then, for each burst in the BA, we create a set of incoming places and a set of outgoing places based on the burst's cardinality, e.g. if the burst's cardinality is $k \geqslant 1$ then we create k incoming places and k outgoing places. For empty bursts, one incoming place and one outgoing place are created.

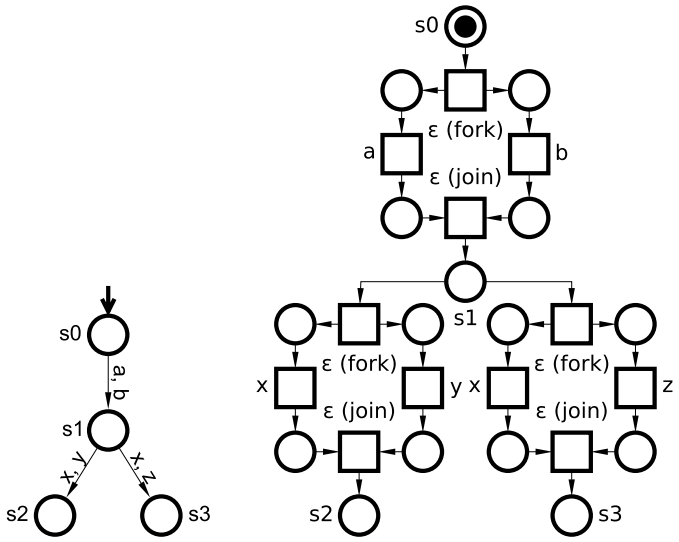
Arcs: For each burst in the BA, we create an arc from:

1. The place corresponding to the source state to the 'fork' dummy transition.
2. The 'fork' dummy transition to every incoming place.
3. The i-th incoming place to the burst transition labelled by σ_i (or ε-labelled transition in case of an empty burst).
4. The burst transition labelled by σ_i (or ε-labelled transition in case of an empty burst) to the i-th outgoing place.
5. Each outgoing place to the 'join' dummy transition.
6. The 'join' dummy transition to the place corresponding to the destination state.

To show that the resulting STG from this translation is linear to the original BA, Figure 4.8 shows an example where we translate a BA that contains the labelled arc 'a, b' followed by a choice between the labelled arcs 'x, y' and 'x, z'.

By studying this example, we can see that the resulting STG is linear to the BA and preserves the BA's language. In particular, there is a one-to-one correspondence between the BA's states and the STG's places (excluding the incoming and outgoing places that are created from the bursts), and the firing of a burst in BAs is modelled by the firing of concurrent transitions in STGs, where the labels of transitions correspond to the actions that comprise the burst.

To illustrate how this translation preserves the language, Figure 4.9 shows the reachability graphs of the BA and the translated STG in Figure 4.8, where two possible traces are shown and highlighted in blue. The first trace in Figure 4.9a shows firing labelled arc 'a, b' followed by labelled arc 'x, y', and the second trace in Figure 4.9b shows firing labelled arc 'a, b' followed by labelled arc 'x, z'.

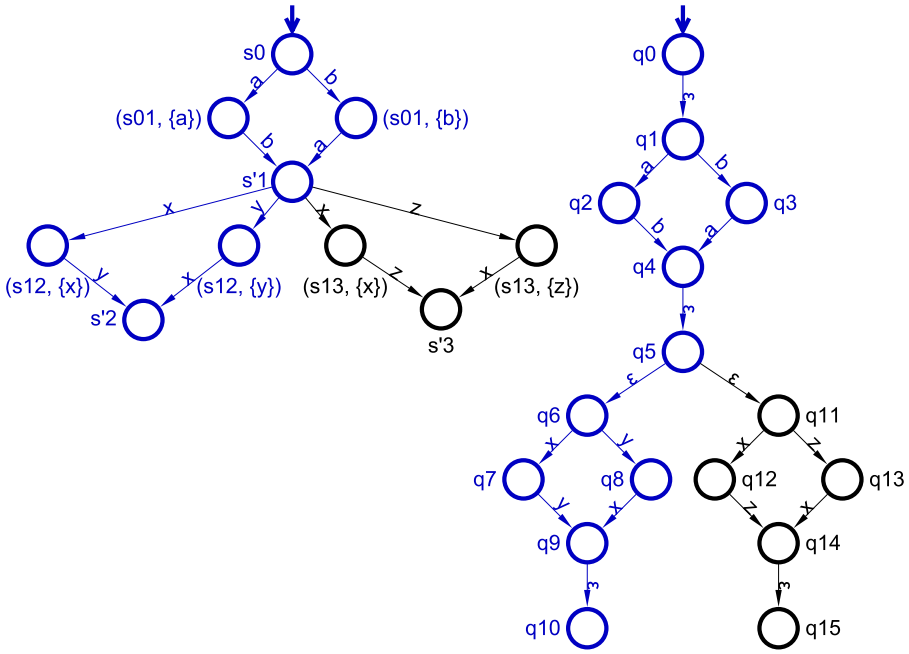


(a) BA (b) Translated STG

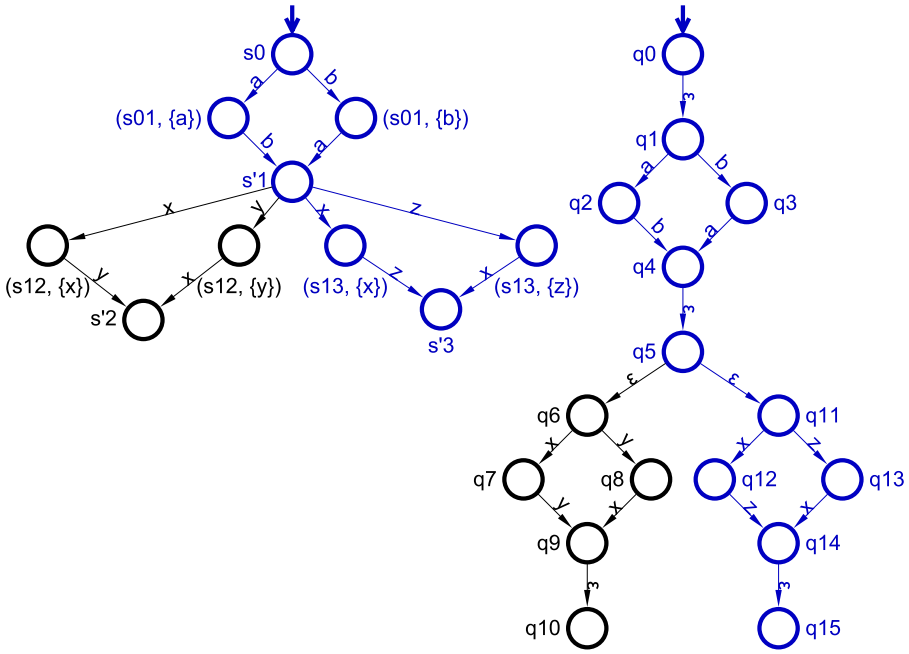
Figure 4.8: Linear translation of simple example

Note that the left and right FSMs in Figure 4.9a produce the language $L_1 = \{\varepsilon, a, b, ab, ba, abx, aby, bax, bay, abxy, baxy\}$, while the left and right FSMs in Figure 4.9b produce the language $L_2 = \{\varepsilon, a, b, ab, ba, abx, abz, bax, baz, abxz, baxz\}$. If we merge the language sets L_1 and L_2, then we can generate the language $\mathcal{L} = L_1 \cup L_2 = \{\varepsilon, a, b, ab, ba, abx, aby, abz, bax, bay, baz, abxy, abxz, baxy, baxz\}$ for both the reachability graph of the BA and the reachability graph of the translated STG. Thus, meaning $\mathcal{L}(BA_{RG}) = \mathcal{L}(STG_{RG})$.

While we can see that this translation preserves the language of the original BA by inspecting the traces of the BA's asynchronous reachability graph and the resulting STG's reachability graph, it however does not preserve weak bisimulation when there are choices between multiple 'fork' dummy transitions. For example, Figure 4.10 shows us trying to establish a bisimulation relation between the reachability graph of the BA and the reachability graph of the translated STG from Figure 4.9.



(a) First trace: $L_1 = \{\varepsilon, a, b, ab, ba, abx, aby, bax, bay, abxy, baxy\}$



(b) Second trace: $L_2 = \{\varepsilon, a, b, ab, ba, abx, abz, bax, baz, abxz, baxz\}$

Figure 4.9: Language preservation of linear translation example

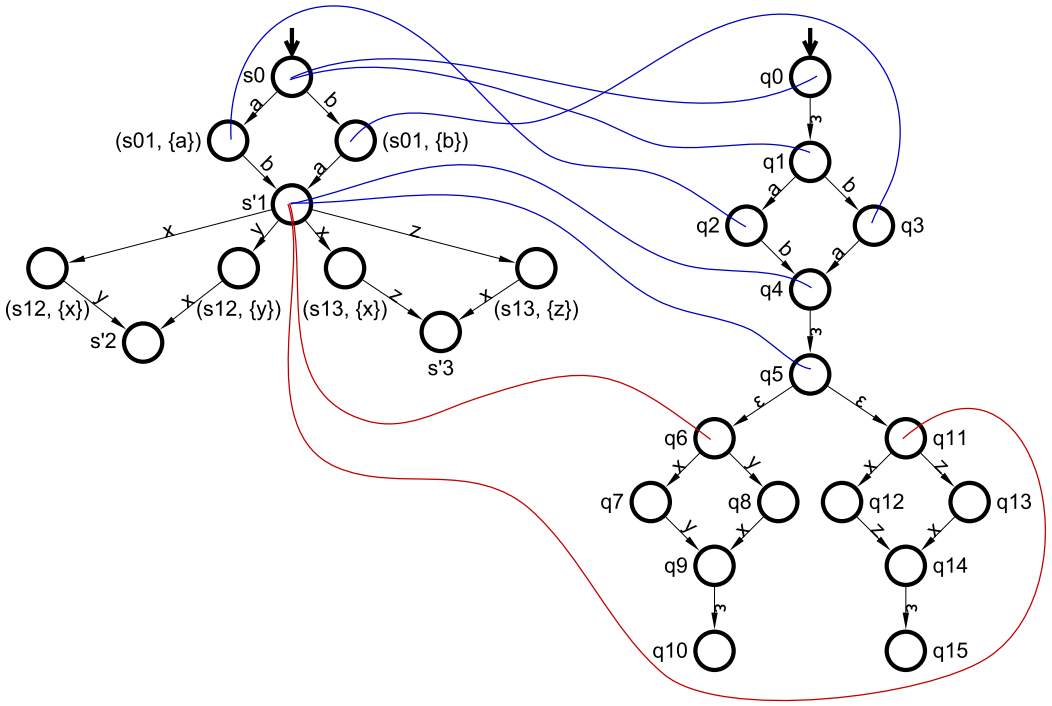


Figure 4.10: Violation of weak bisimulation found in the linear translation example

For the sake of contradiction, suppose that these two reachability graphs (i.e. FSMs) are weakly bisimilar. Then, according to the definition of bisimulation in Section 2.3.1, there is a bisimulation relation $\sim$ between the states of these FSMs and it must relate the initial states, meaning $s'_0 \sim q_0$.

Here, the right FSM can make a transition $q_0 \xrightarrow{\varepsilon} q_1$, and the only way to match this transition in the left FSM is to do nothing and stay in s'_0, meaning $s'_0 \sim q_1$.

Next, the left FSM can make a transition $s'_0 \xrightarrow{a} (s_{01}, \{a\})$ that can be matched by the right FSM making a transition $q_1 \xrightarrow{a} q_2$ and vice versa. Also, the left FSM can make a transition $s'_0 \xrightarrow{b} (s_{01}, \{b\})$ that can be matched by the right FSM making a transition $q_1 \xrightarrow{b} q_3$ and vice versa. Thus, meaning $(s_{01}, \{a\}) \sim q_2$ and $(s_{01}, \{b\}) \sim q_3$.

Afterwards, the left FSM can make the transitions $(s_{01}, \{a\}) \xrightarrow{b} s'_1$ and $(s_{01}, \{b\}) \xrightarrow{a} s'_1$, which can be matched by the right FSM making the transitions $q_2 \xrightarrow{b} q_4$ and $q_3 \xrightarrow{a} q_4$ respectively and vice versa. So, $s'_1 \sim q_4$.

Like the first step, the right FSM can make a transition $q_4 \xrightarrow{\varepsilon} q_5$, and the only way to match this transition in the left FSM is to do nothing and stay in s'_1, meaning $s'_1 \sim q_5$.

Upon reaching state q_5 on the right FSM, there is now a choice between transitions $q_5 \xrightarrow{\varepsilon} q_6$ and $q_5 \xrightarrow{\varepsilon} q_7$. Note that the only way to match either of these transitions in the left FSM is to do nothing and stay in s'_1, meaning $s'_1 \sim q_6$ and $s'_1 \sim q_7$.

However, when the left FSM is at state s'_1, it can make a transition $s'_1 \xrightarrow{z} (s_{13}, \{z\})$, which the right FSM cannot match at state q_6 due to no choice of z. Furthermore, when the left FSM is at state s'_1, it can also make a transition $s'_1 \xrightarrow{y} (s_{12}, \{y\})$, which the right FSM cannot match at state q_7 due to no choice of y. Thus, this contradicts $s'_1 \sim q_6$ and $s'_1 \sim q_7$ meaning these two FSMs are not weakly bisimilar.

Nevertheless, as we covered in Section 2.3.2, the semantics of non-deterministic STGs are formally defined in [37], where it is argued that non-output-determinate STGs are ill-formed and that language equivalence is adequate for output-determinate STGs. This means that there is no requirement to preserve bisimulation, and that the models produced by our translation are adequate enough in practice, since the language is preserved and the models remain output determinate.

4.3.2 STGs with join dummy transitions

In Section 4.3.1, we presented a translation that produces an STG with 'fork' dummy transitions and 'join' dummy transitions, which is linear to the size of the original BA and preserves the language. However, Figure 4.10 shows that the translation violates weak bisimulation when there are choices between multiple 'fork' dummy transitions, as the choice of left x, right x, y and z in the original BA is replaced by a non-deterministic choice of two ε-labelled 'fork' transitions in the translated STG.

In this translation, we produce an STG that is weakly bisimilar to the original BA, where we remove the 'fork' dummy transitions and preserve the 'join' dummy transitions from the first translation. The cost to remove the 'fork' dummy transitions is that places, which correspond to the BA states that contain multiple outgoing arcs, must be replicated. Note that this may cause the resulting STG to become exponential to the size of the original BA in the worst case, though the size of these resulting STGs tend to be small in practice. Also, by removing the 'fork' dummy transitions, it may be easier for verification and synthesis tools to handle as there are fewer ε-transitions, e.g. MPSAT uses STG unfoldings and preserves ε-transitions.

To help understand how this translation works, Figure 4.11 shows a simple example where we translate the same BA from Figure 4.7 to an STG.

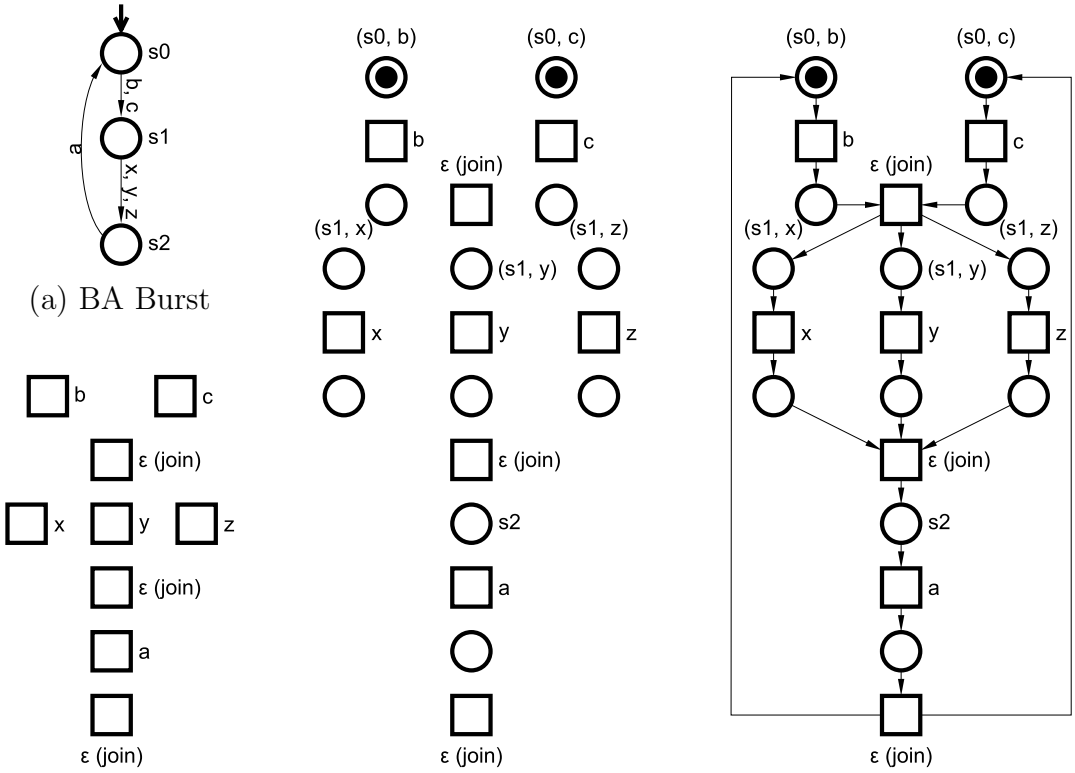


Figure 4.11: Translating a BA to an STG with "join" dummy transitions

Like the previous translation, we again create six transitions labelled a, b, c, x, y and z respectively, where transition a comprises the labelled arc 'a', transitions b and c comprise the labelled arc 'b, c', and transitions x, y, and z comprise the labelled arc 'x, y, z'. Then, we create one additional dummy transition called 'join' for each labelled arc.

Next, for each state $s \in \{s0, s1, s2\}$, we create n places where $n \in \mathbb{N}$ is the number of actions found in the labelled arc that s is the source state, i.e. we create two places for $s0$, three places for $s1$ and one place for $s2$. Note that these places replace the connections from s to the 'fork' dummy transition and from the 'fork' dummy transition to the transitions that comprise the labelled arc's label, where this 'fork' dummy transition corresponds to the labelled arc that s is the source state. Then, we add tokens to the places that correspond to $s0$ as $s0$ is the initial state, and create one outgoing place for each labelled transition $t \in \{a, b, c, x, y, z\}$.

Now, for each state $s \in \{s0, s1, s2\}$, we connect an arc from the i-th place corresponding to s to the labelled transition of the j-th label in the labelled arc that s is the source state, where $i, j \in \mathbb{N}$ and each labelled transition is connected by only one place, i.e. we create an arc from the places corresponding to $s0$ to the labelled transitions b and c individually, an arc from the places corresponding to $s1$ to the labelled transitions x, y and z individually, and an arc from the place corresponding to $s2$ to the labelled transition a. Then, we connect an arc from the 'join' dummy transition created from the labelled arc where s is the destination state to the places corresponding to s. Finally, for each labelled arc $a \in \{(s2, \text{'}a\text{'}, s0), (s0, \text{'}b, c\text{'}, s1), (s1, \text{'}x, y, z\text{'}, s2)\}$, we connect an arc from all outgoing places created by a's label to the 'join' dummy transition created by a, and an arc from the labelled transitions that comprise a's label to each outgoing place created by a's label, such that each labelled transition is connected by only one outgoing place.

By completing this example, we can determine how a BA is translated into an STG with this method, and formalise the above steps for every burst found in the BA as shown below, where each step is categorised by the STG component being created.

Transitions: The transitions are created in the same way as the first translation in Section 4.3.2 except no 'fork' dummy transitions are created.

Places: For every state s in the BA, we create a set of places in the STG as follows. Suppose the bursts labelling the outgoing arcs from s are $Out_1, Out_2, \ldots, Out_k$, $k \geqslant 0$, where empty bursts are encoded as $\{\varepsilon\}$. Then we create a new place for each tuple in the Cartesian product $Out_1 \times Out_2 \times \cdots \times Out_k$. If $k = 0$ then this Cartesian product contains the empty tuple as the only element. Additionally, if s is the initial state of the BA, then these places are initially marked (i.e. a token is added). Moreover, for each burst in the BA, we create a set of outgoing places based on the burst's cardinality. For example, if the burst cardinality is $k \geqslant 1$ then we create k outgoing places. For empty bursts, only one outgoing place is created.

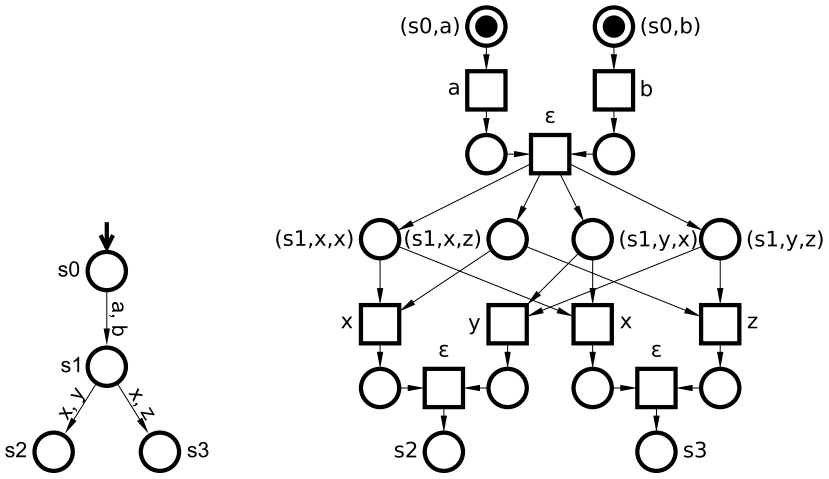
Arcs: For each burst in the BA, we create an arc from:

1. The place corresponding to a BA state s to a tuple $(o_1, \ldots, o_k)$ to the o_i-labelled transition (which may be ε-labelled transition in case of an empty burst) in the i-th burst, for each $i \in \{1, \ldots, k\}$.

2. Each burst's j-th transition to the j-th outgoing place, where j ranges from 1 to the cardinality of the burst. Note that empty bursts are encoded as $\{\varepsilon\}$ and so only one arc is created if the burst is empty.

3. Each outgoing place to the 'join' dummy transition.

4. The 'join' dummy transition to every place corresponding to the destination state.



(a) BA (b) Translated STG

Figure 4.12: Exponential translation of simple example

To show that the resulting STG from this translation does not exponentially explode to the size of the original BA, Figure 4.12 shows an example where we translate the same BA from Figure 4.8 that contains the labelled arc '`a, b`' followed by a choice between the labelled arcs '`x, y`' and '`x, z`'. Note that the first translation's 'join' dummy transitions are not this translation's 'fork' dummy transitions, as the 'fork' dummy transitions are *contracted* (i.e. removed from the STG). However, one may interpret the 'join' dummy transitions as 'join_fork' dummy transitions (i.e. a 'join' dummy transition followed by subsequent a 'fork' dummy transition).

By studying this example, we can see that the resulting STG is not exponential to the size of the original BA, meaning this translation is small for practical BAs. Note that the exponential explosion only occurs when there are many arcs originating from the same state and are labelled with large bursts, where if there are n arcs with non-empty bursts of cardinalities $k_1, k_2, \ldots, k_n$ then $k_1 \cdot k_2 \cdot \ldots \cdot k_n$ STG places are created. Fortunately, the presence of these large bursts are rare in practice, as discussed in Section 4.2.2.

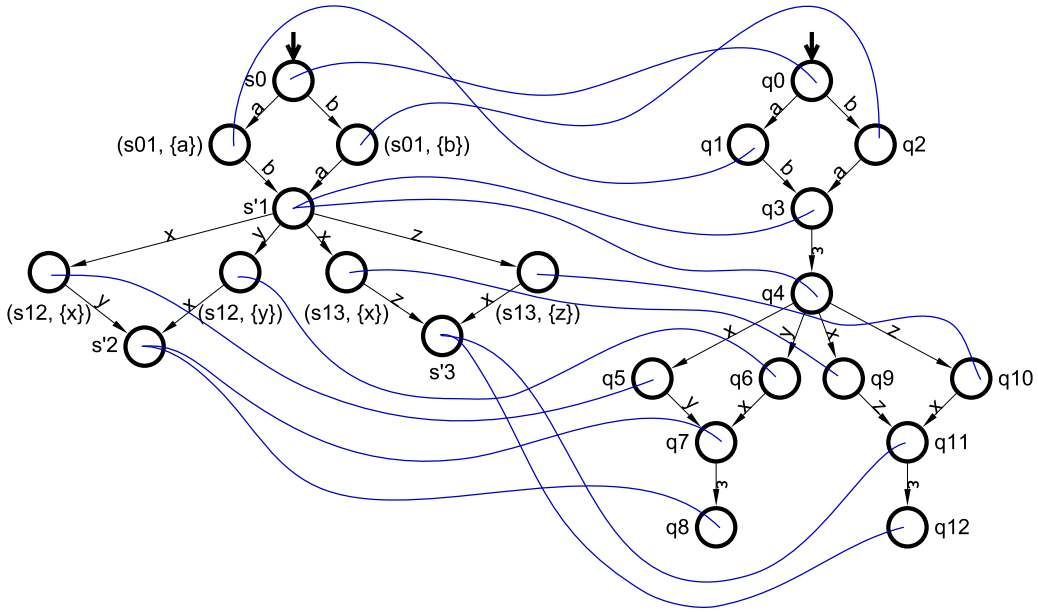


Figure 4.13: Example preserving weak bisimulation

Unlike the previous translation in Section 4.3.1, this translation preserves weak bisimulation between the resulting STG's reachability graph and the BA's reachability graph. In particular, there are no longer any choices between multiple dummy transitions like the first translation's 'fork' dummy transition, which caused the violation. For example, let us consider Figure 4.13 which shows the reachability graph of the BA from Figure 4.9 and the reachability graph of the translated STG in Figure 4.12b.

According to the definition of bisimulation in Section 2.3.1, there is a bisimulation relation $\sim$ between the FSMs' states and it must relate the initial states, so $s'_0 \sim q_0$.

Here, the left FSM can make a transition $s'_0 \xrightarrow{a} (s_{01}, \{a\})$ that can be matched by the right FSM making the transition $q_0 \xrightarrow{a} q_1$ and vice versa. Also, the left FSM can make a transition $s'_0 \xrightarrow{b} (s_{01}, \{b\})$ that can be matched by the right FSM making the transition $q_0 \xrightarrow{b} q_2$ and vice versa. Thus, meaning $(s_{01}, \{a\}) \sim q_1$ and $(s_{01}, \{b\}) \sim q_2$.

Next, the left FSM can make the transitions $(s_{01}, \{a\}) \xrightarrow{b} s'_1$ and $(s_{01}, \{b\}) \xrightarrow{a} s'_1$, which can be matched by the right FSM making the transitions $q_1 \xrightarrow{b} q_3$ and $q_2 \xrightarrow{a} q_3$

respectively and vice versa. So, $s'_1 \sim q_3$. Note that the right FSM can make a transition $q_3 \xrightarrow{\varepsilon} q_4$, and the only way to match this transition in the left FSM is to do nothing and stay in s'_1, meaning $s'_1 \sim q_4$.

Now, at state s'_1 in the left FSM and state q_4 in the right FSM, there is a choice between left x, y, right x and z. Here, the left FSM can make the transitions $s'_1 \xrightarrow{x} (s_{12}, \{x\})$ and $s'_1 \xrightarrow{x} (s_{13}, \{x\})$, which can be matched by the right FSM making the transitions $q_4 \xrightarrow{x} q_5$ and $q_4 \xrightarrow{x} q_9$ respectively and vice versa. So, $(s_{12}, \{x\}) \sim q_5$ and $(s_{13}, \{x\}) \sim q_9$. Also, the left FSM can make the transition $s'_1 \xrightarrow{y} (s_{12}, \{y\})$ that can be matched by the right FSM making the transition $q_4 \xrightarrow{y} q_6$ and vice versa, meaning $(s_{12}, \{y\}) \sim q_6$. Similarly, the left FSM can make the transition $s'_1 \xrightarrow{z} (s_{13}, \{z\})$ that can be matched by the right FSM making the transition $q_4 \xrightarrow{z} q_{10}$ and vice versa, meaning $(s_{13}, \{z\}) \sim q_{10}$. Note that the left FSM's transition $s'_1 \xrightarrow{x} (s_{12}, \{x\})$ can also be matched by the right FSM's transition $q_4 \xrightarrow{x} q_9$, and that the left FSM's transition $s'_1 \xrightarrow{x} (s_{13}, \{x\})$ can also be matched by the right FSM's transition $q_4 \xrightarrow{x} q_5$, despite $(s_{12}, \{x\})$ can only fire y while q_9 can fire z and $(s_{13}, \{x\})$ can only fire z while q_5 can fire y. However, this does not contradict bisimulation, as the states are still matched by at least one state, i.e. $(s_{12}, \{x\})$ with q_5 and $(s_{13}, \{x\})$ with q_9.

Finally, the left FSM can make the transitions $(s_{12}, \{x\}) \xrightarrow{y} s'_2$ and $(s_{12}, \{y\}) \xrightarrow{x} s'_2$, which can be matched by the right FSM making the transitions $q_5 \xrightarrow{y} q_7$ and $q_6 \xrightarrow{x} q_7$ respectively and vice versa. So, $s'_2 \sim q_7$. Also, the left FSM can make the transitions $(s_{13}, \{x\}) \xrightarrow{z} s'_3$ and $(s_{13}, \{z\}) \xrightarrow{x} s'_3$, which can be matched by the right FSM making the transitions $q_9 \xrightarrow{z} q_{11}$ and $q_{10} \xrightarrow{x} q_{11}$ respectively and vice versa. So, $s'_3 \sim q_{11}$.

Again, the right FSM can make the transitions $q_7 \xrightarrow{\varepsilon} q_8$ and $q_{11} \xrightarrow{\varepsilon} q_{12}$, and the only way to match these transitions in the left FSM is to do nothing and stay s'_2 and $s'3$ respectively, meaning $s'_2 \sim q_8$ and $s'_3 \sim q_{12}$.

Thus, the reachability graphs of the resulting STG and BA are weakly bisimilar.

4.3.3 STGs without dummy transitions

In Section 4.3.2, we presented a translation method that produces an STG with 'join' dummy transitions, which is weakly bisimilar to the original BA. While the resulting STG could be exponential to the size of the BA in the worst case, it is shown that this is rare in practice due to the unlikelihood of large bursts.

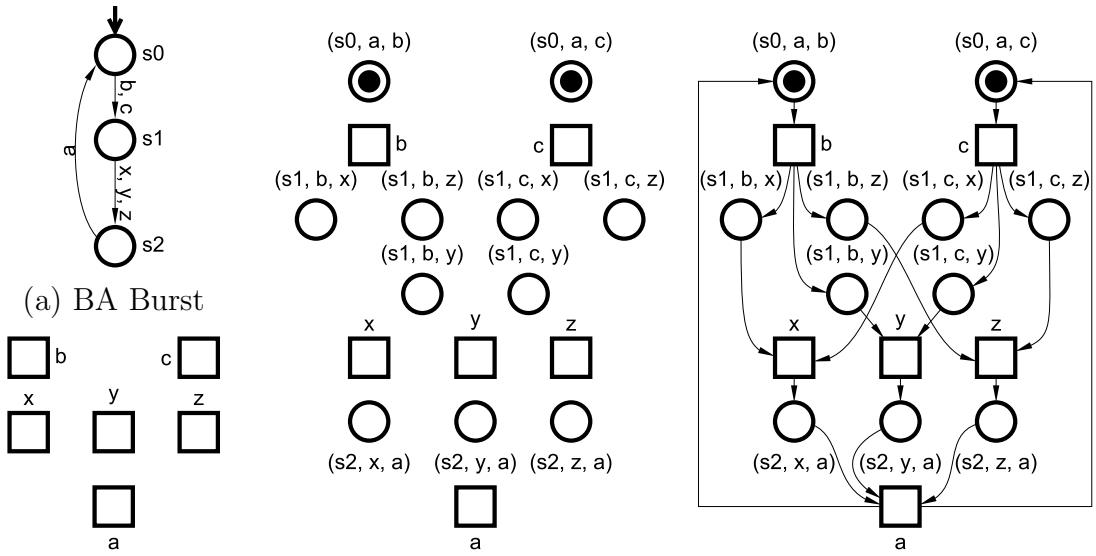
However, despite that the translation preserves weak bisimulation, it does not preserve strong bisimulation due to the remaining 'join' dummy transitions, as these dummy transitions do not correspond to any actions of the original BA.

In this translation, we produce an STG that is strongly bisimilar to the original BA, where we remove both the 'fork' dummy transitions and 'join' dummy transitions from the prior translations. The cost to remove both 'fork' dummy transitions and 'join' dummy transitions is that places, which correspond to the BA states with multiple incoming arcs and multiple outgoing arcs, have to be replicated. Note that this may cause the resulting STG to become exponential to the size of the original BA and potentially larger than the second translation in the worst case. Again, the size of these resulting STGs also tend to be small in practice, and it may even be easier for verification and synthesis tools to handle as there are no ε-transitions.

To help understand how this translation works, Figure 4.14 shows a simple example where we translate the same BA from Figures 4.7 and 4.11 to an STG.

Like the previous translations, we again create six transitions labelled a, b, c, x, y and z respectively, where transition a comprises the labelled arc 'a', transitions b and c comprise the labelled arc 'b, c', and transitions x, y, and z comprise the labelled arc 'x, y, z'. However this time, we do not create any additional dummy transitions.

Next, for each state $s \in \{s0, s1, s2\}$, we create $m * n$ places where $m \in \mathbb{N}$ is the number of actions found in the labelled arc that s is the destination state, and $n \in \mathbb{N}$



(a) BA Burst (b) Transitions (c) Places (d) Arcs

Figure 4.14: Translating a BA to an STG with no "fork" or "join" dummy transitions

is the number of actions found in the labelled arc that s is the source state, i.e. we create two places for $s0$, six places for $s1$ and three places for $s2$. Note that these places replace the connections from the transitions that comprise the previous labelled arc's label to the 'join' dummy transition, from the 'join' dummy transition to the places corresponding to s, from the places corresponding to s to the 'fork' dummy transition and from the 'fork' dummy transition to the transitions that comprise the current labelled arc's label, where the 'fork' dummy transition corresponds to the labelled arc that s is the source state and the 'join' dummy transition corresponds to the labelled arc that s is the destination state. Then, we add tokens to the places that correspond to $s0$ as $s0$ is the initial state.

Now, for each state $s \in \{s0, s1, s2\}$, we connect an arc from the m-th place that corresponds to s to the labelled transition of the i-th label in the labelled arc that s is the source state, where $m, i \in \mathbb{N}$ and each labelled transition is connected by k incoming places, such that $k \in \mathbb{N}$ is the number of actions in the labelled arc where s is the destination state, i.e. we create an arc from the places corresponding to $s0$ to

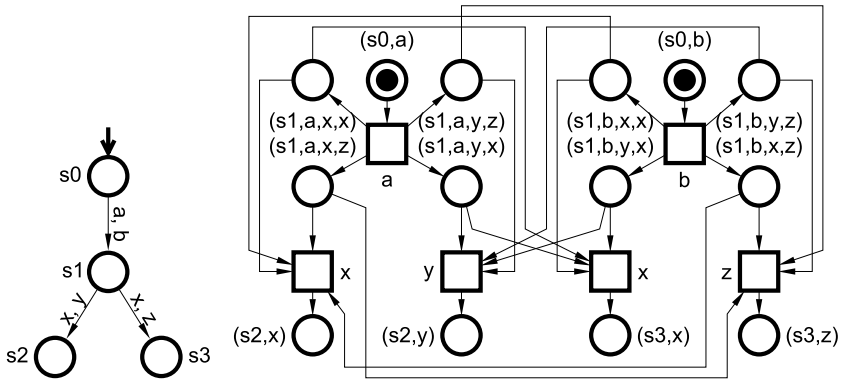
the labelled transitions b and c individually, two arcs from the places corresponding to $s1$ to the labelled transitions x, y and z individually, and three arcs from the place corresponding to $s2$ to the labelled transition a. Similarly, we also connect an arc from the labelled transition of the j-th label in the labelled arc that s is the destination state to the n-th place that corresponds to s, where $n, j \in \mathbb{N}$ and each labelled transition is connected to l outgoing places, such that $l \in \mathbb{N}$ is the number of actions in the labelled arc where s is the source state, i.e. we create two arcs from the labelled transition a to the places corresponding to $s0$ each, three arcs from the labelled transitions b and c to the places corresponding to $s1$ each, an arc from the labelled transitions x, y and z to the place corresponding to $s2$.

Note that transition a can only fire once all places corresponding to $s2$ have a token, i.e. when transitions x, y and z have fired. Similarly, b and c can only fire when all places corresponding to $s0$ receive their token via a, and transitions x, y and z can only fire when all places corresponding to $s1$ receive their tokens via b and c.

By completing this example, we can determine how a BA is translated into an STG with this method, and formalise the above steps for every burst found in the BA as shown below, where each step is categorised by the STG component being created.

Transitions: The transitions are created in the same way as the prior translations in Sections 4.3.1 and 4.3.2, except neither 'fork' nor 'join' transitions are created.

Places: For every state s in the BA, we create a set of places in the STG as follows. Suppose the bursts labelling the arcs incoming to s are $In_1, In_2, \ldots, In_k$, and the bursts labelling the arcs outgoing from s are $Out_1, Out_2, \ldots, Out_{k'}$, $k, k' \geqslant 0$, where empty bursts are encoded as $\{\varepsilon\}$. Then we create a new place for each tuple in the Cartesian product $In_1 \times In_2 \times \cdots \times In_k \times Out_1 \times Out_2 \times \cdots \times Out_{k'}$. If $k = 0$ and $k' = 0$ then this Cartesian product contains the empty tuple as the only element. Additionally, if s is the initial state of the BA, these places are initially marked.



(a) BA (b) Translated STG

Figure 4.15: Exponential translation of simple example

Arcs: For each place p corresponding to a BA state s and a tuple $(i_1, \ldots, i_k, o_1, \ldots, o_{k'})$, we create an arc from:

1. The i_j-labelled transition (which may be ε-labelled transition in case of an empty burst) in jth incoming burst to p, for each $j \in \{1, \ldots, k\}$.

2. p to the o_j-labelled transition (which may be ε-labelled transition in case of an empty burst) in jth outgoing burst, for each $j \in \{1, \ldots, k'\}$.

To show that the resulting STG from this translation also does not exponentially explode to the size of the original BA, Figure 4.15 shows an example where we translate the same BA from Figures 4.8 and 4.12 that contains the labelled arc '`a, b`' followed by a choice between the labelled arcs '`x, y`' and '`x, z`'.

By studying this example, we can see that the resulting STG is not exponential to the size of the original BA, meaning this translation also is small for practical BAs. Again, note that the exponential explosion only occurs when there are many arcs originating from the same state and are labelled with large bursts, where the presence of such large bursts are rare in practice, as discussed in Section 4.2.2.

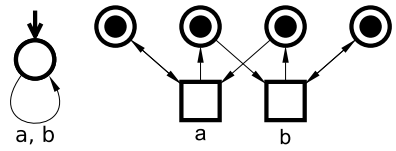


Figure 4.16: Violation of safeness when translating BAs with non-singleton bursts

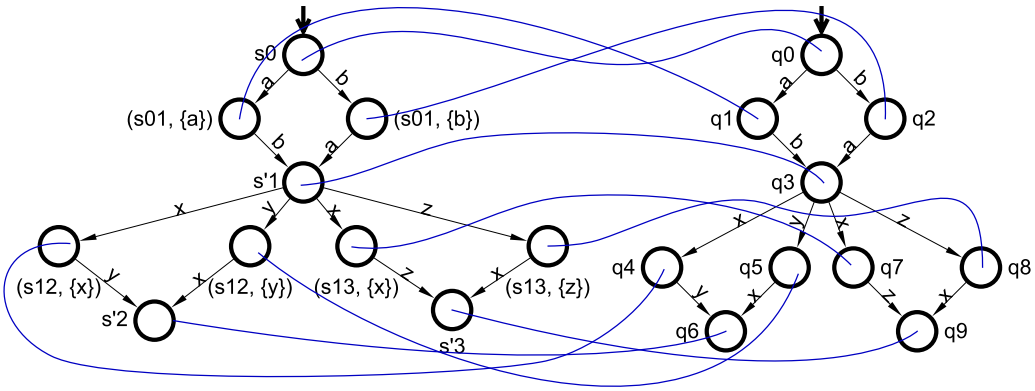


Figure 4.17: Example preserving strong bisimulation

If the original BA contains any self-loops with non-singleton bursts, then this translation may yield an unsafe STG that is 2-bounded. For example, Figure 4.16 shows a BA on the left and its translated STG on the right, where firing the transition labelling a will produce another token on the second place to the left.

In the context of circuit synthesis, labels are usually directed (i.e. they have a $+$ or $-$) and the consistency requirement is imposed for each signal s, where $s+$ and $s-$ must alternate in every trace. If such a directed action occurs in a self-looped labelled arc, then the consistency requirement is clearly violated. Thus, circuit specifications do not use self-loops and will not be translated into unsafe STGs.

By removing both 'fork' and 'join' dummy transitions, this translation preserves strong bisimulation between the reachability graphs of the resulting STG and BA. For example, let us consider Figure 4.17 which shows the reachability graph of the BA from Figure 4.9 and the reachability graph of the translated STG in Figure 4.15b.

According to the definition of bisimulation in Section 2.3.1, there is a bisimulation relation $\sim$ between the FSMs' states and it must relate the initial states, so $s'_0 \sim q_0$.

Here, the left FSM can make a transition $s'_0 \xrightarrow{a} (s_{01}, \{a\})$ that can be matched by the right FSM making the transition $q_0 \xrightarrow{a} q_1$ and vice versa. Also, the left FSM can make a transition $s'_0 \xrightarrow{b} (s_{01}, \{b\})$ that can be matched by the right FSM making the transition $q_0 \xrightarrow{b} q_2$ and vice versa. Thus, meaning $(s_{01}, \{a\}) \sim q_1$ and $(s_{01}, \{b\}) \sim q_2$.

Next, the left FSM can make the transitions $(s_{01}, \{a\}) \xrightarrow{b} s'_1$ and $(s_{01}, \{b\}) \xrightarrow{a} s'_1$, which can be matched by the right FSM making the transitions $q_1 \xrightarrow{b} q_3$ and $q_2 \xrightarrow{a} q_3$ respectively and vice versa. So, $s'_1 \sim q_3$.

Now, at state s'_1 in the left FSM and state q_4 in the right FSM, there is a choice between left x, y, right x and z. Here, the left FSM can make the transitions $s'_1 \xrightarrow{x} (s_{12}, \{x\})$ and $s'_1 \xrightarrow{x} (s_{13}, \{x\})$, which can be matched by the right FSM making the transitions $q_3 \xrightarrow{x} q_4$ and $q_3 \xrightarrow{x} q_7$ respectively and vice versa. So, $(s_{12}, \{x\}) \sim q_4$ and $(s_{13}, \{x\}) \sim q_7$. Also, the left FSM can make the transition $s'_1 \xrightarrow{y} (s_{12}, \{y\})$ that can be matched by the right FSM making the transition $q_3 \xrightarrow{y} q_5$ and vice versa, meaning $(s_{12}, \{y\}) \sim q_5$. Similarly, the left FSM can make the transition $s'_1 \xrightarrow{z} (s_{13}, \{z\})$ that can be matched by the right FSM making the transition $q_3 \xrightarrow{z} q_8$ and vice versa, meaning $(s_{13}, \{z\}) \sim q_8$. Note that the left FSM's transitions $s'_1 \xrightarrow{x} (s_{12}, \{x\})$ and $s'_1 \xrightarrow{x} (s_{13}, \{x\})$ can also be matched by the right FSM's transitions $q_4 \xrightarrow{x} q_9$ and $q_4 \xrightarrow{x} q_5$ respectively, despite $(s_{12}, \{x\})$ can only fire y while q_9 can fire z and $(s_{13}, \{x\})$ can only fire z while q_5 can fire y. Again, this does not contradict bisimulation, as the states are still matched by one state, i.e. $(s_{12}, \{x\})$ with q_3 and $(s_{13}, \{x\})$ with q_7.

Finally, the left FSM can make the transitions $(s_{12}, \{x\}) \xrightarrow{y} s'_2$ and $(s_{12}, \{y\}) \xrightarrow{x} s'_2$, which can be matched by the right FSM making the transitions $q_4 \xrightarrow{y} q_6$ and $q_5 \xrightarrow{x} q_6$ respectively and vice versa. So, $s'_2 \sim q_6$. Also, the left FSM can make the transitions $(s_{13}, \{x\}) \xrightarrow{z} s'_3$ and $(s_{13}, \{z\}) \xrightarrow{x} s'_3$, which can be matched by the right FSM making the transitions $q_7 \xrightarrow{z} q_9$ and $q_8 \xrightarrow{x} q_9$ respectively and vice versa. So, $s'_3 \sim q_{11}$.

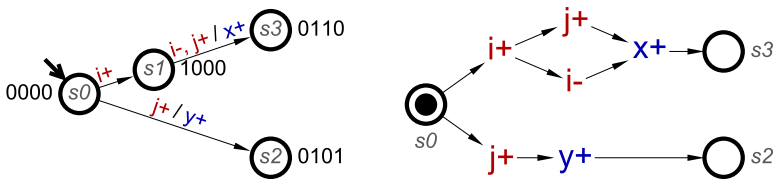
Thus, the reachability graphs of the resulting STG and BA are strongly bisimilar.

4.3.4 Extended Burst-Mode Components

In this translation, we cover how the XBM specification's fake outputs, conditionals, and "don't cares" are translated to their respective STG counterparts [15]. Here, these (X)BM components are considered auxiliary features that can be attached to the BA, where we can interpret it as an (X)BM-like model. Note that the use of this (X)BM-like model may be useful in some scenarios, as shown below.

Fake Outputs

In Section 2.2.2, (X)BM specifications are described to create a "dummy" output (i.e. a fake output) for every burst without a set of outputs to signify an internal change within the system. In BAs, we can model empty bursts as $\varnothing$-labelled arcs that can be interpreted as ε transitions, whether it is originally an input burst or output burst. However, there are still some scenarios where these fake outputs can be useful. For example, Figure 4.18 shows a BM specification with a choice between an input-output burst and an input-only burst, and its translated STG without "fake" outputs, where this BM specification's choice can prevent the synthesis of the resulting STG.



(a) BM Specification (b) Resulting STG

Figure 4.18: Hidden unresolvable CSC conflict

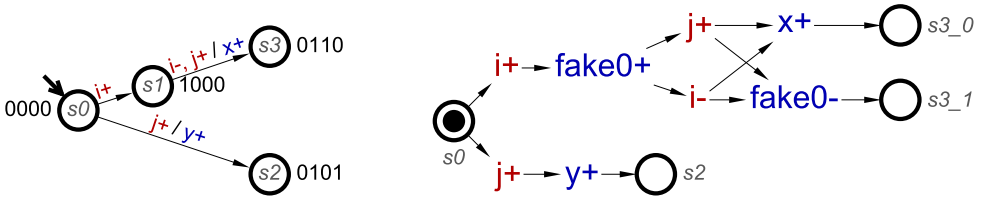
In the BM specification, there is a choice between bursts i+ / and j+ / y+, where i+ / reaches state $s1$ and is followed by burst i-, j+ / x+ to reach state $s3$, while j+ / y+ reaches state $s2$. This is a valid BM specification that can be synthesised by MINIMALIST and 3D, as a fake output is added to i+ / to signify an internal change.

Now, suppose that we translate this BM specification to its equivalent STG without any modification and try to synthesise it with PETRIFY or MPSAT. Unfortunately, this is not possible as there is an unresolvable complete state coding (CSC) conflict. By studying the STG, we can see that firing i+ and then i- will lead to a state, where it is uncertain whether the system will next fire x+ or y+ after j+ is received.

While the above scenario is trivial and can be fixed by adding a transition that explicitates this fake output [15], this may be a difficult and tedious task if these hidden CSC conflicts occur many times in the BM specification. Thus, this necessitates an optional translation to the STG, where these fake outputs are explicitated.

To implement these explicit fake output transitions, we must identify all bursts in the (X)BM-like model (i.e. BA) that have an empty set of outputs, and create an output transition fakeK+ for each k-th burst where $k \in \mathbb{N} > 1$. For the explicit fake output transitions to be consistent, they are mapped to an input transition of the input burst, which the fake output appears in, and are replicated for every occurrence of that mapped input transition such that their edges match.

For example, Figure 4.19 shows the same BM specification from Figure 4.18 and its translated STG with explicit fake output transitions.



(a) BM specification (b) Resulting STG

Figure 4.19: Resolving the hidden CSC conflict during translation

By observing the resulting STG, we can see that fake0 is mapped to i, where every occurrence of i+ (i-) is correctly followed by fake0+ (fake0-). If we try to synthesise this STG, it will now be successful as there are no more unresolvable CSC conflicts.

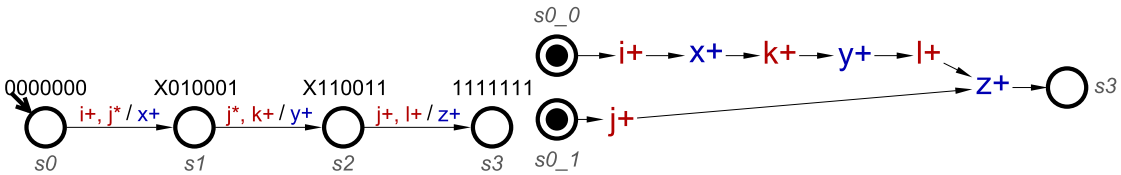
"Don't cares"

In Section 2.2.2, "don't cares" are described as monotonic inputs that can concurrently change with outputs, where they may appear through multiple bursts with the direction $*$ until they terminate in a burst with the direction $+$ or direction $-$.

Additionally, "don't cares" must always appear with a compulsory input, and they must correctly terminate based on their original value before appearing as a "don't care", due to the well-formed requirement of correct termination of "don't cares" described in Section 2.2.2. For example, if the signal was low (high) before appearing as a "don't care" then it must terminate with a $+$ $(-)$ event to ensure that the XBM specification is well-formed. Note that if the "don't care" appears in a loop then we assume it remains a "don't care" unless it terminates, in which we must add a transition of the opposite direction to reset the "don't care" when exiting the loop.

When we translate "don't cares" to their STG counterpart, the "don't cares" are turned into explicit STG transitions that connect from the set of input transitions that they appear with to the set of output transitions that they terminate by [15]. Due to the concurrency of STGs, the transitions of "don't cares" can fire at any time between the aforementioned sets of input transitions and output transitions.

For example, Figure 4.20 shows an XBM specification with a "don't care" and its translated STG with an explicitated "don't care" transition.



(a) XBM Specification (b) Resulting STG

Figure 4.20: Translating "don't cares" to explicit delayed STG transitions

By observing the XBM specification, we can see that the "don't care" j* appears in burst i+, j* / x+ until it terminates with direction + at burst j+, l+ / z+. Now, if we observe the resulting STG, we can also see that the input transition j+ is connected from the places corresponding to state $s0$ to the output transition z+, where j+ can fire concurrently with transitions i+, x+, k+, y+ and l+.

Conditionals

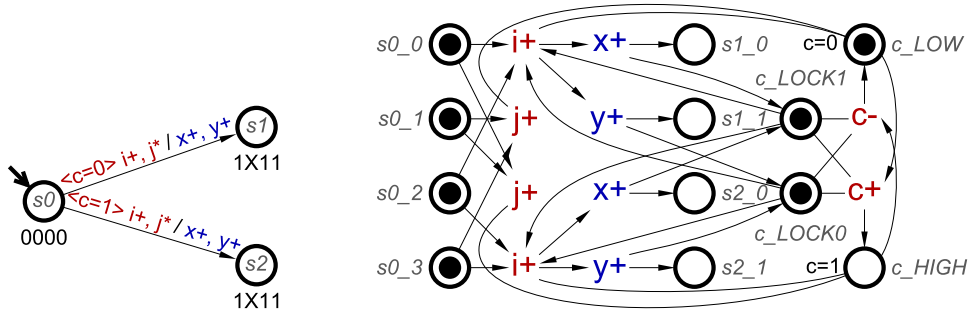
In Section 2.2.2, conditionals are described as level-sensitive inputs that determine the system's control flow based on their signal values, which may continuously change between high and low values until the conditional is sampled. When conditionals are sampled, they must stabilise once a compulsory input appears and hold their value until all subsequent outputs are produced.

Notably, conditionals can be translated into *elementary cycles* [12], which are a pair of places and a pair of rising and falling transitions used to explicitly represent the signal's current value, where places depict when the signal is high or low and transitions depict when the signal rises or fall. However, elementary cycles do not have restrictions that prevent the signal from firing, meaning the signal can continuously fire. This contradicts the sampling of conditionals, as they must be stabilised.

To ensure that the elementary cycle's signal remains stable like conditionals, we add some so-called 'lock' places [15] that are generated by the number of compulsory inputs found in the corresponding burst. These lock places contain a token, and are connected to every transition labelling the burst's compulsory inputs and connected from transition labelling the burst's outputs. Additionally, these lock places are connected with the rising and falling transitions of the elementary cycle in both directions using *read arcs* (i.e. a shorthand expression for an arc connecting from node x to node y, and an arc connecting from y to x).

When one of the transitions labelling the compulsory inputs is fired, it removes the associating lock place's token and disables the elementary cycle from firing its rising and falling transitions, imitating the conditional's setup time (i.e. the time until a compulsory input arrives). The firing of the elementary cycle's rising and falling transitions can only resume, once the tokens from all lock places have returned after the transitions labelling the outputs have fired, imitating the conditional's hold time (i.e. the time that the signal remains stable until all outputs are produced).

For example, Figure 4.21 shows an XBM specification with a conditional and its translated STG with an elementary cycle.



(a) XBM specification (b) Resulting STG

Figure 4.21: Translating conditionals to elementary cycles with lock places

By observing the XBM specification, we can see that there is a choice between bursts i+, j* / x+, y+ that are determined by the conditional c's value. If the value of c is 0 (1) then we fire burst ⟨c=0⟩ i+, j* / x+, y+ (⟨c=1⟩ i+, j* / x+, y+).

Now, if we observe the resulting STG, we can see that the elementary cycle's signal can continuously rise and fall until a transition labelling a compulsory input (i.e. i+) fires, where the tokens from its associating lock places are removed (i.e. places $s1\_0$ and $s1\_1$ if c=0 or places $s2\_0$ and $s2\_1$ if c=1) and are only returned once the transitions labelling the outputs (i.e. x+ and y+) have fired. Note that the transitions labelling j+ do not have connections, as it is a "don't care" and the STG is translated from a BM specification segment that only contains j* appearing.

4.4 Distribution Methodology

In this section, we will cover the definition of distributed FSMs (DFSMs) for this thesis, and the distribution criteria used as a guideline to determine how the group of FSMs communicate with each other. We will then cover the composition method for BAs, including how it is verified and synthesised into a composed SI (QDI) circuit.

4.4.1 Definition of Distributed Finite State Machines

For this thesis, we will define DFSM as a collection of FSMs that are connected to each other through common signals, where these common signals are sent and received by each FSM via the four-phase request-acknowledgement handshake protocol, such that, for some signal x, FSM M sends a request $xreq$ to FSM M' and M' sends an acknowledgement $xack$ to M. for some signal x.

4.4.2 Distribution Criteria

Due to the inherent behaviours and structure of any distributed specification, where each sub-specification may have its own set of behaviours, timing assumptions and interfaces (e.g. signals), we will consider the following distribution criteria as a guideline to determine the behaviour of our DFSM:

1. All connected FSMs are assumed to be concurrent with each other, e.g. there may be multiple FSMs that are running at the same time or are waiting for a response from another FSM before continuing.

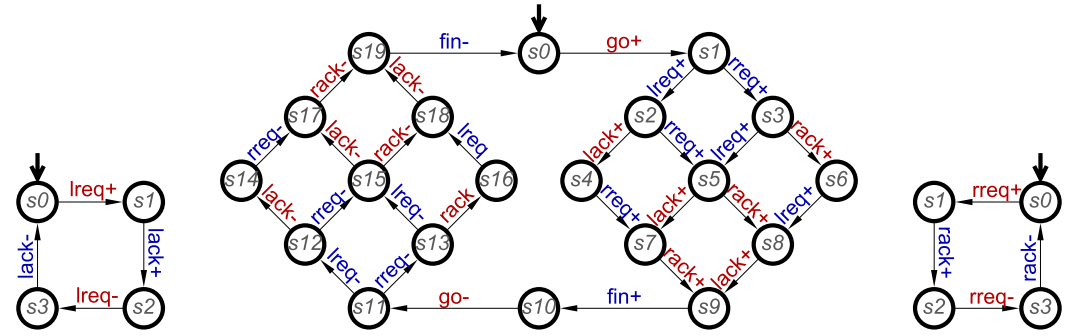
2. All communication made between the connecting FSMs will follow the four-phase request-acknowledgment handshake protocol, such that the sending FSM

will wait for a response from the receiving FSM to continue and vice versa (unless their next set of signals are mutually exclusive). Note that the names of requests and acknowledgements can differ between each FSM.

3. The connecting FSMs must be conformant with each other, such that the FSMs do not contradict each other.

4.4.3 Composition of Burst Automata

Using DFSMs' definition in Section 4.4.1 and the distribution criteria in Section 4.4.2, Figure 4.22 shows how we compose a controller BA with the two other BAs.

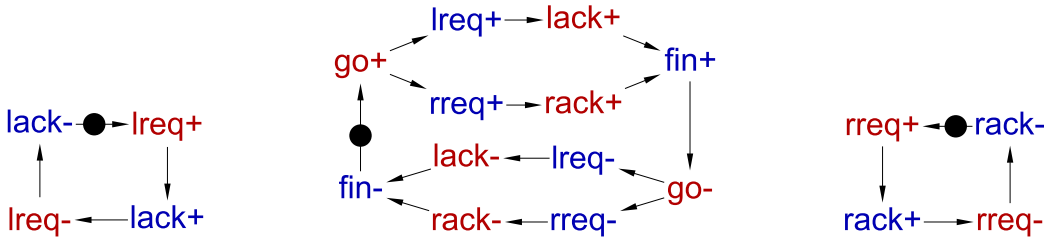


(a) Left part (b) Controller (c) Right part

Figure 4.22: BAs of a controller and the left and right environment parts

Here, we can see that the BAs are concurrent with each other, and that each BA communicates with another BA using the four-phase request-acknowledgment handshake protocol, e.g. the controller interacts with the left (right) part via handshakes lreq/lack (rreq/rack). Note that the controller receives go from and sends fin to the environments that are not modelled in this example.

Because BAs can be translated to STGs using any of the three translations in Section 4.3 and STGs are easy to compose using PCOMP, we can translate our three BAs to the STGs shown in Figure 4.23 for subsequent composition.



(a) Left part (b) Controller (c) Right part

Figure 4.23: Translated STGs of the controller and the environment parts

Using WORKCRAFT, we can verify that these three STGs satisfy the standard STG implementability properties (defined in Section 2.2.3) that include consistency, deadlock-freeness, input-properness, output-persistency, and output determinacy.

Next, we will review the three resulting STGs to ensure that they satisfy each of our distribution criterion:

1. In Section 2.2.3, it is described that STGs can express input-output concurrency, meaning they are concurrent with other STGs. Thus, this satisfies criterion 1.

2. In Section 4.3, it is shown that the BA's signals do not change after it is translated to an STG, and that the STG preserves the language of the original BA (or its reachability graph is bisimilar to the original BA), meaning the STGs also retain the communication (via the four-phase request-acknowledgement handshake) between the BAs. Thus, satisfying criterion 2.

3. With access to the STG's well-established tools, we can verify that the three resulting STGs are conformant with each other by using WORKCRAFT to check N-way conformation between the STGs. Thus, satisfying criterion 3.

Now, we can compose the three resulting STGs into the composed STG shown in Figure 4.24 by using PCOMP. Again, using WORKCRAFT, we can verify that this composed STG also satisfies the aforementioned STG implementability properties.

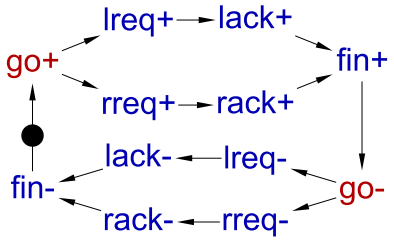


Figure 4.24: Composed STG of the controller and the environment parts

We can then synthesise this composed STG using either PETRIFY or MPSAT backends in WORKCRAFT to produce a possible circuit implementation like the SI (QDI) circuit shown in Figure 4.25.

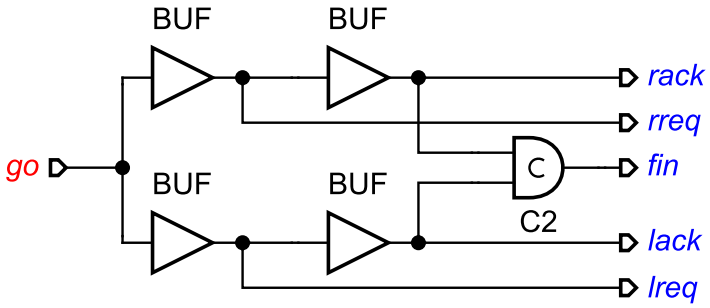


Figure 4.25: Possible circuit implementation from synthesis of composed STG

Alternatively, we can generate the reachability graph of this composed STG shown in Figure 4.26 and interpret it as a BA. Note that this reachability graph is essentially the controller in Figure 4.22b, except signals lack and rack are now outputs.

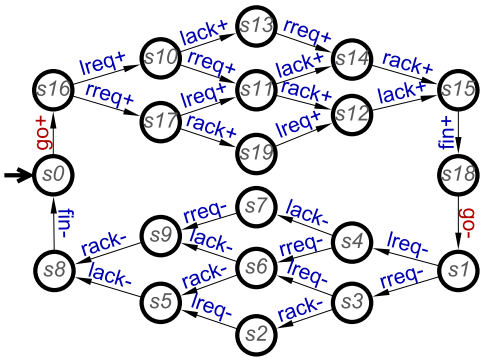


Figure 4.26: Composed BA via the reachability graph of the composed STG

4.5 Summary

In this chapter, we cover the model description of BAs, where we provide a textbook definition of BAs and use an example to compare their design with the design of (X)BM specifications and STGs. We then cover the mathematical definitions of BA, where we formally define what BAs are and what their reachability graph are, such that the latter is essentially an FSM and is used to check the model equivalence with other formalisms, e.g. STG, through their reachability graph.

Next, we cover three translations from BAs to STGs, where the first translation produces an STG with "fork" and "join" dummy transitions that preserves the language, the second translation produces an STG with "join' dummy transitions that preserves weak bisimulation, and the third translation produces an STG with neither "fork" nor "join" dummy transitions that preserves strong bisimulation. Furthermore, we also cover the XBM specification's components to their STG counterparts including the BM specification's "fake" outputs and the XBM specification's conditionals and "don't cares".

Lastly, we cover the distribution methodology of BAs, where we provide a textbook definition of DFSMs and a distribution criteria that acts as a guideline for composing the BAs. We translate the BAs to STGs, where we can compose them with Pcomp before we verify and synthesise the composed STG with Petrify or Mpsat to produce an SI (QDI) circuit, or generate its reachability graph, which can be interpreted as a composed BA.

Chapter 5

Design Automation

In this chapter, we will cover the implemented Workcraft plugin that supports the design automation of Burst Automata (BAs).

Firstly, we will review the design flow of BAs, where a new design route is created to enable the co-design between Burst-Mode (BM) specifications and Signal Transition Graphs (STGs). We will then highlight how our Workcraft plugin achieves this design flow, where the process to design, verify and synthesise STGs by translating the (X)BM specifications via BAs is automated.

Next, we will cover the features that are implemented in the Workcraft plugin, where we can design and simulate BAs, validate BAs based on (X)BM well-formed requirements if they are an (X)BM-like model, automatically translate BAs to STGs through the available translation options, and finally verify and synthesise the BAs via translated STGs using Petrify and Mpsat.

Lastly, we will study the benchmark results of our experiment using the implemented Workcraft plugin, where we analyse the model size growth of our translations from (X)BM specifications to STGs using BAs, and the literal counts from circuit implementations produced by Minimalist, 3D, Petrify and Mpsat.

5.1 Automation of Burst Automata Design Flow

In this section, we will cover the co-design between BM specifications and STGs that is achieved by using BA, where we review the design flow that is automated and implemented in our WORKCRAFT plugin. Note that in this plugin, the graphical notation of the BM specification's bursts are supported and used when bursts contain both inputs and outputs, though it follows the firing semantics of BAs where each arc (i.e. burst) is fired as two-steps such that the arrival of an input burst is followed by the subsequent production of an output burst.

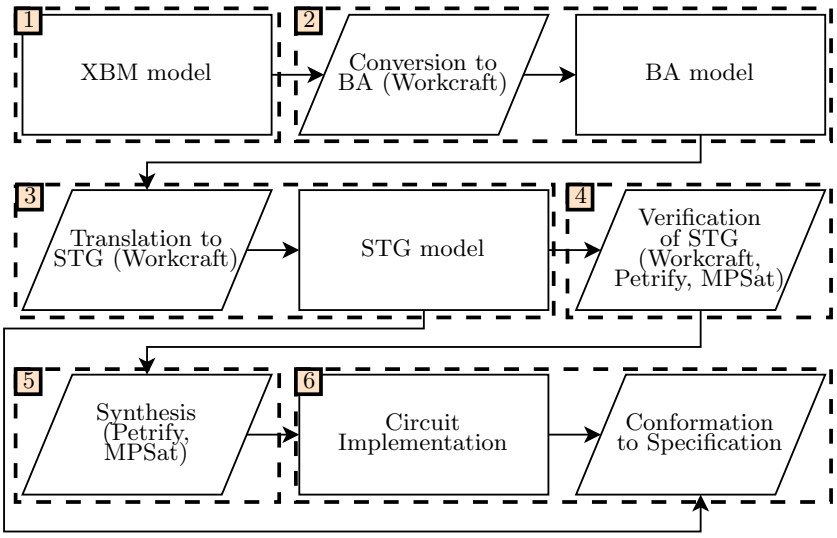


Figure 5.1: Design flow of Burst Automaton

To help us understand how this design flow works and how the co-design between BM specifications and STGs is achieved, Figure 5.1 shows a step-by-step procedure of how a circuit designer can create, validate, and translate their (X)BM specification in WORKCRAFT, where they can then verify the translated STG and synthesise it into a speed-independent (SI) (or similarly, quasi-delay-insensitive (QDI)) asynchronous circuits. Note that the annotated boxes in Figure 5.1 corresponds to the steps below:

1. Firstly, we can design our initial (X)BM specification by using WORKCRAFT or by reading a (X)BM file. We can then validate it by using the interactive simulation feature, and verify it based on the (X)BM's well-formed requirements described in Section 2.2.2 using the interactive commands in the plugin.

2. Next, we can translate our (X)BM specification into a BA, where the BM timing assumption and the (X)BM's well-formedness properties are relaxed.

3. Subsequently, we can then translate our BA into an STG by using one of the three translations described in Section 4.3.

4. Once an STG is obtained, we can verify the standard STG implementability properties and/or some custom properties of this resulting STG, ensuring that there are no hazards and/or no unexpected behaviour.

5. Now, we can synthesise the STG into an SI (QDI) circuit by using one of the available implementation styles (i.e. complex-gate, generalised C-element, or standard C-element) with either PETRIFY or MPSAT backends.

6. Finally, we can formally verify the conformation of the resulting circuit implementation against the translated STG. While circuit synthesis is in theory correct by construction, it is also complicated and tools may have bugs, meaning it is important that we use formal verification to help increase our confidence in the correctness of the circuit implementation. Note that there is a one-to-one correspondence between BAs and the translated STGs, meaning it is possible to verify the STG, determine the violated properties (e.g. output persistency) and resolve the violation using BA (e.g. adding/removing transitions to prevent/avoid the violation). Moreover, it is also possible that the circuit may have been manually modified, which further necessitates formal verification.

From this example, there are several benefits for both state-based and event-based circuit designers, which our automated design flow achieves:

1. The design of (X)BM specifications is now automated through the design flow of BAs. Previously, the design of (X)BM specifications was mostly completed by specifying it in a text file and then visualised using one of MINIMALIST's executables without simulation capability. Here, we allow event-based circuit designers to easily specify (X)BM specifications using WORKCRAFT and then translate it into an STG, while we also allow state-based circuit designers to continue specifying their (X)BM specifications (whether as a text file or through WORKCRAFT) and make use of the plugin's features, e.g. interactive simulation, verification and translation to STG for subsequent STG verification and synthesis of an SI (QDI) circuit.

2. With BAs, state-based circuit designers may now access the STG's well-established tools to compose, verify and/or synthesise their (X)BM specification (or an equivalent FSM-based formalism) accordingly. Because the design flow of BAs and the translations from BAs to STGs are automated, state-based circuit designers do not need to understand how to design an STG, as this step is now covered by simplying translating their (X)BM specification to an STG for subsequent verification and synthesis of an SI (QDI) circuit. Thus, saving time required to re-specifying the circuit designer's specifications as STGs, while also providing the confidence that their circuits are correct, well-optimised and hazard-free via the STG's well-established tools.

3. Due to BA's interoperability between (X)BM specifications and STGs, there can be a closer collaboration between state-based circuit designers and event-based circuit designers, as one can switch from (X)BM specifications to STGs and

vice versa. This is particularly useful, especially when these circuit designers must share their work with one another (which may be in different formalisms) and when designing large systems, where some blocks are expressed as (X)BM specifications and other blocks are expressed as STGs.

5.2 Features of the Implemented Workcraft Plugin

In this section, we will cover the features that are supported in the implemented Workcraft plugin.

Firstly, we will demonstrate the modelling of a BA, where we show how its states, arcs and signals are created. We will then demonstrate two methods of assigning directions to signals, where the first method does the assignment by calculating the encoding differences between the source state and the destination state, while the second method does the assignment by the conventional method of adding directions to the signals.

Next, we will show the interactive simulation of a BA including its traces and how conditionals are changed. We will then show how the verification of the (X)BM's well-formed requirements are performed for BAs that are designed as (X)BM-like models.

Finally, we will demonstrate the translation of BAs to STGs for subsequent verification and composition, where the three translations shown in Section 4.3 are implemented. We will then demonstrate how the BAs are synthesised into SI (QDI) circuits via their translated STGs, before the produced circuit implementation is verified for conformity with the original specification.

5.2.1 Design of Burst Automaton

To design a BA or (X)BM-like model, let us first consider how we generate its states, signals and arcs. In the plugin, we have three separate buttons that each creates a new signal, a new signal and a new arc, which the latter must be connected between two existing states. These buttons are annotated and shown in Figure 5.2, where (a) annotates the button that creates states, (b) annotates the button that creates signals, and (c) annotates the button that creates arcs. Note that (d) annotates the toggle button that changes the signal type to an input, conditional or output.

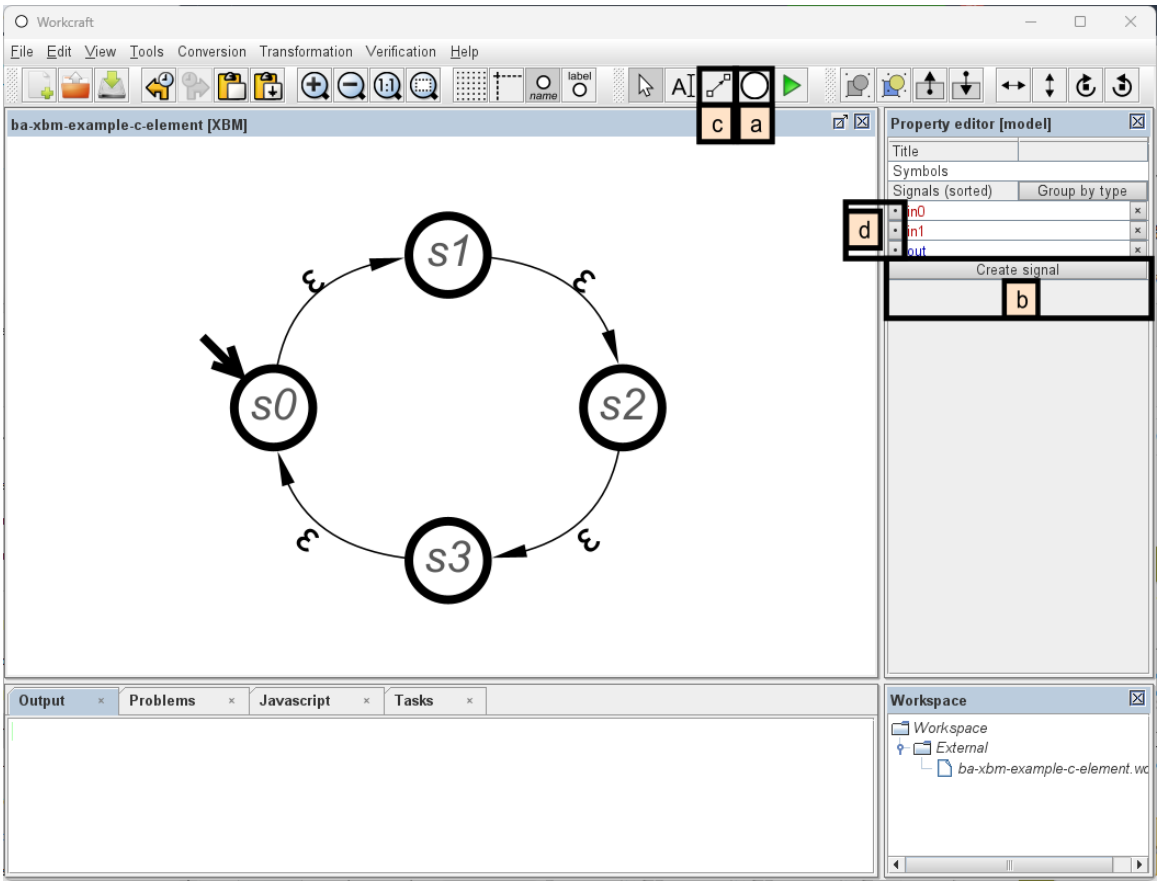


Figure 5.2: Generating the BA's states, signals and arcs

Additionally, we can select an arc and create a new input and a new output by pressing the create button labelled as $*$ under the 'Input burst' label and the 'Output burst' label respectively. We can also create a new conditional by using the conditional text field, where the user provided text is checked to ensure that it is in the form of *conditional* = 0 or *conditional* = 1, such that *conditional* is a string of alphanumerical characters. Again, these buttons and text field are annotated and shown in Figure 5.3, where (a) annotates the button that creates a new input in the input burst, (b) annotates the button that creates a new output in the output burst, and (c) annotates the text field that checks the input text and then creates a new conditional, if it satisfies the regular expression check.

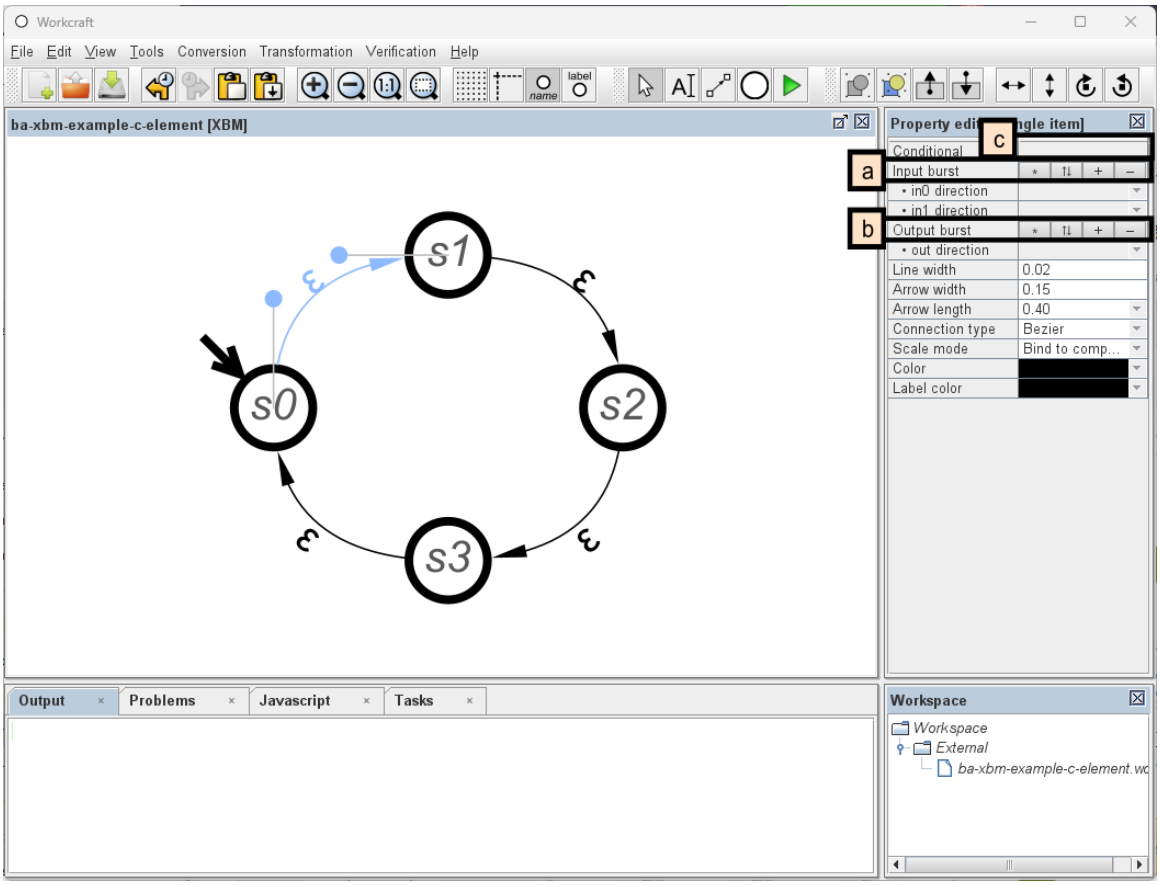


Figure 5.3: Generating conditionals with alternative way to generate signals

5.2.2 Direction Assignment to Signals

When the BA has at least two states, one signal and one arc, we can assign a direction (i.e. a $+$, $-$, or $*$) to a signal in an arc. Note that direction $*$ is included to support the design of (X)BM-like models, and that if there are no directions assigned to any of the arc's signals then an ε-symbol is shown instead.

Currently, there are two methods of assigning directions to signals in an arc. The first method calculates the state encoding difference between the arc's states, while the second uses the conventional assignment of directions to signals by selecting the associating direction.

Firstly, let us cover the method that calculates the state encoding differences shown in Figure 5.4. Here, we can select a state and assign the encoding values 1, 0, or ? to one of the state's signals, where:

- If the encoding value of some signal x is 0 in the source state and 1 in the destination state, then a $+$ event is added to x in the arc. Similarly, if the encoding value of the signal x is 1 in the source state and 0 in the destination state, then a $-$ event is added to x in the arc. Otherwise, if the encoding value of the signal x is either 0 or 1 in both the source state and destination state, then no direction event is added. If a direction was existing for x before the same encoding value was assigned, then x and its direction are removed from the arc (and are potentially pushed to the next arc).

- If the encoding value of some signal x is ? in the destination state then a $*$ event is added to x in the arc, regardless of whether the encoding value of x is 0, 1 or ? in the source state. Note that ? also sets the visual encoding of x as 'X'.

- If the encoding value of some signal x is ? in the source state, then:
  - A + event is added to x in the arc, if the encoding value of signal x is 1 in the destination state.
  - A − event is added to x in the arc, if the encoding value of signal x is 0 in the destination state.
  - A ∗ event is added to x in the arc, if the encoding value of signal x is also ? in the destination state.

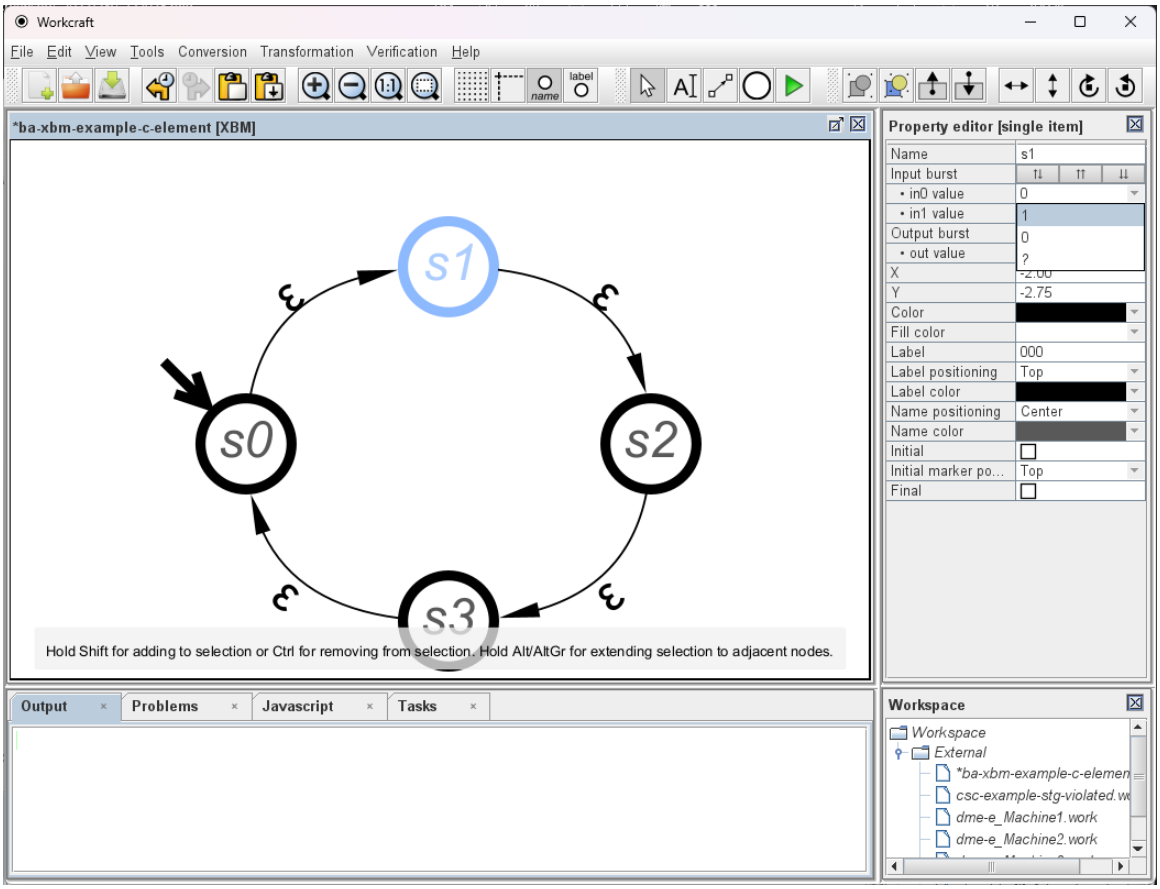


Figure 5.4: Assigning Signal Directions via State Encoding Calculation

Now, let us cover the conventional method of assigning directions to signals in Figure 5.5. Here, we can select an arc and assign the directions $+$, $-$, or $*$ to some signal x, where:

- If $+$ is assigned, then x's encoding value is set to 1 at the destination state, such that if x's encoding value is also 1 at the source state, then x is removed from the arc. Similarly, if $-$ is assigned, then x's encoding value is set to 0 at the destination state, such that if x's encoding value is also 0 at the source state, then x is removed from the arc.
- If $*$ is assigned, then x's encoding value is set to ? at the destination state.

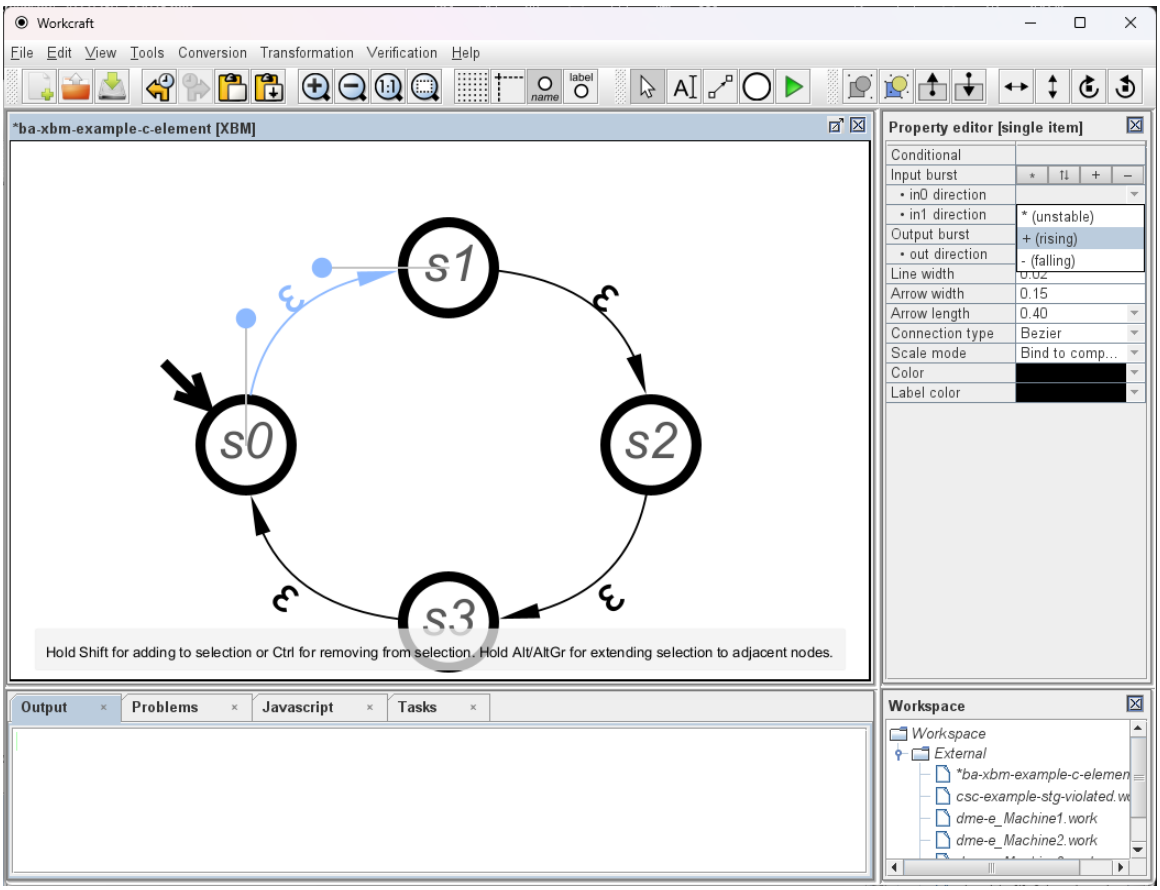


Figure 5.5: Assigning Signal Directions via Traditional Method

5.2.3 Simulation of Burst Automaton

Once we have designed a complete BA, we can simulate it in WORKCRAFT. Like other models such as FSMs and STGs, the BA becomes 'fireable' where it is displayed with highlighted arcs that are enabled and can be 'fired' by clicking the arc to generate a new trace. For example, Figure 5.6 shows the simulation of a BA specifying a C-element gate.

Here, we can see the simulation trace table on the right side of the model, where it shows that we have fired in0+, in1+, out+, in0-, in1-, out-, and in0+, in1+, such that the BA is currently on state $s1$ and out+ is enabled for firing.

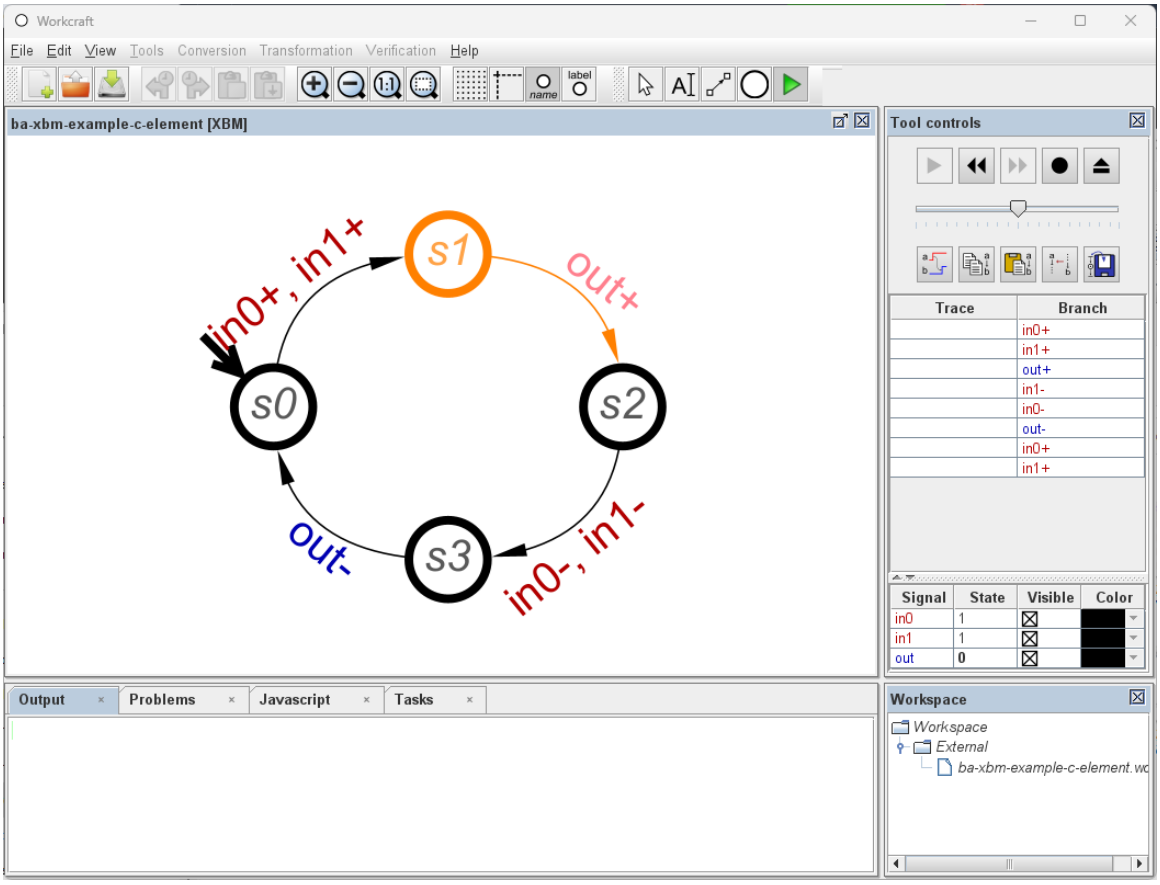


Figure 5.6: Running Simulation of the BA

If the BA is an (X)BM-like model and contains a conditional, then a checkbox for this conditional is displayed under the simulation trace table, where we can simulate the conditional rising or falling. For example, Figure 5.7 shows the simulation of an XBM-like model specifying the *biu-fifo2dma* specification [69] which contains the conditional cntgt.

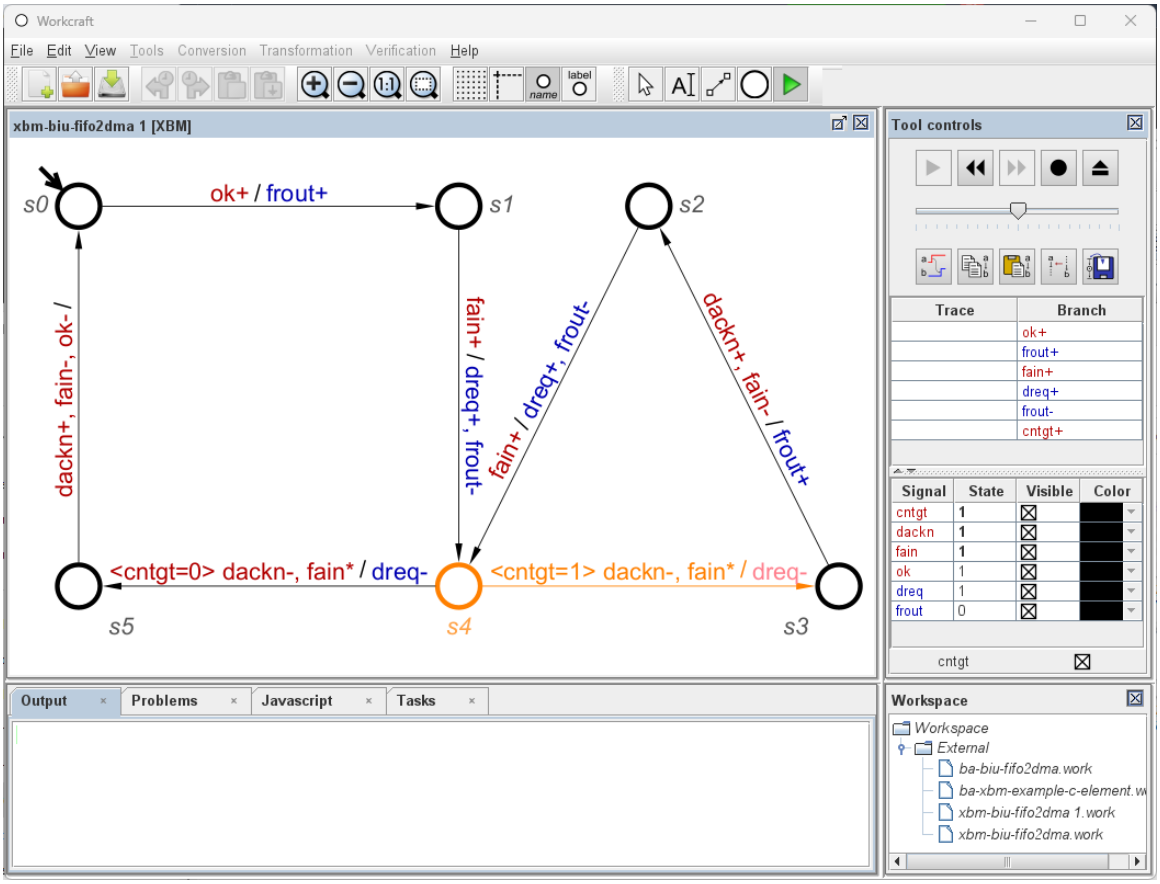


Figure 5.7: Toggling Conditionals during Simulation of an XBM-like Model

To understand how the conditional checkbox works, suppose that we reach a state where its outgoing arc(s) contain a conditional, e.g. in Figure 5.7 where state s4 has bursts ⟨cntgt=0⟩ dackn-, fain* / dreq- and ⟨cntgt=1⟩ dackn-, fain* / dreq-. Then:

- If the checkbox for the conditional is unchecked, all bursts that have the conditional as low is enabled, while all bursts that have the conditional as high is

disabled. For example, unchecking cntgt will enable the burst ⟨cntgt=0⟩ dackn-, fain* / dreq- and disable ⟨cntgt=1⟩ dackn-, fain* / dreq-.

- If the checkbox for the conditional is checked, all bursts that have the conditional as high is enabled, while all bursts that have the conditional as low is disabled. For example, checking cntgt will enable the burst ⟨cntgt=1⟩ dackn-, fain* / dreq- and disable ⟨cntgt=0⟩ dackn-, fain* / dreq-.

5.2.4 Verification of Burst-Mode Well-formed Requirements

In addition to simulation, if the BA is designed as an (X)BM-like model, then we can verify it to check if it satisfies or violates any of the (X)BM's well-formed requirements. Note that BAs do not need to satisfy the (X)BM's well-formedness requirements, as this verification is implemented for BAs specifying (X)BM specifications.

To verify the (X)BM's well-formedness requirements in the plugin, we can select the desired verification check under the 'Verification' menu as shown in Figure 5.8, where the options to verify the (X)BM-like model for BM's maximal set property, XBM's distinguishability constraint, non-empty input burst property and XBM's correct termination of "don't cares" are highlighted.

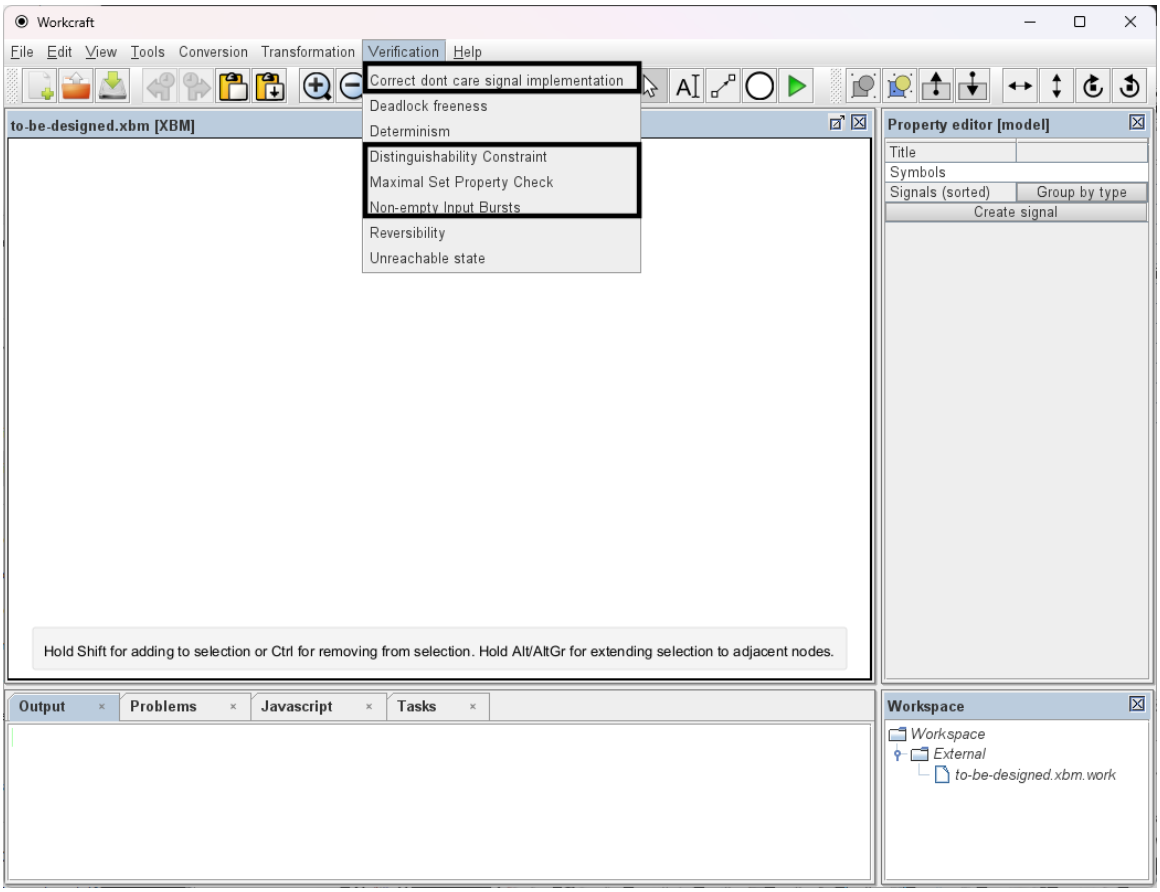


Figure 5.8: Available verification options based on (X)BM well-formed requirements

5.2.5 Translation to Signal Transition Graphs

After validating our BA through simulation and/or verifying it against the (X)BM well-formed requirements if it is specified as an (X)BM-like model, we can translate the BA into an STG using any of the three translations described in Section 4.3. Note that each translation will also translate the XBM specification's conditionals and "don't cares", and has another option that adds "fake" outputs to bursts without outputs, as described in Section 4.3.4.

These translation methods can be found under the 'Translation' menu as shown in Figure 5.9, where the available translation options are highlighted.

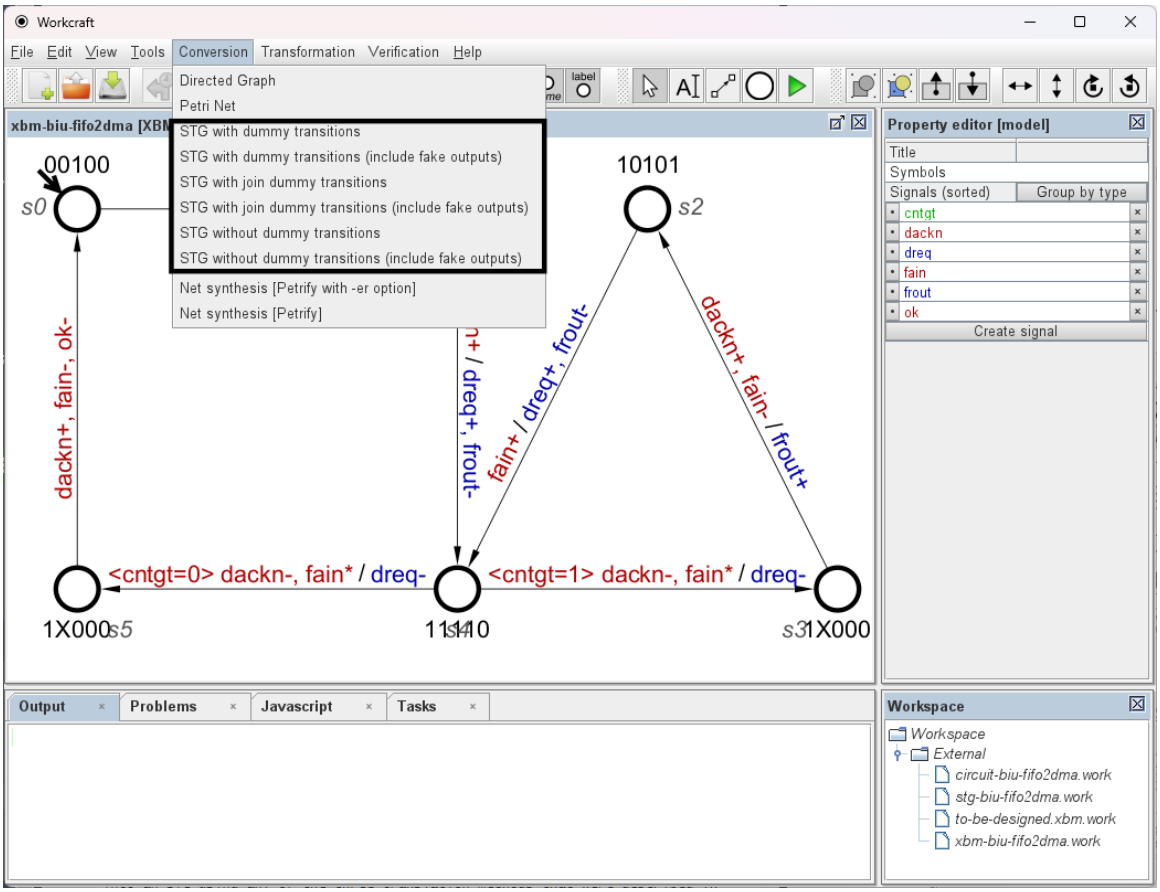


Figure 5.9: Available translations from BAs to STGs

To illustrate the plugin using one of the translation methods from BAs to STGs, Figure 5.10 shows the translation of the *biu-fifo2dma* specification to its equivalent STG, where neither 'fork' nor 'join' dummy transitions are included and "fake" output transitions are created for every empty output burst. Note that after obtaining our translated STG, we can verify it against the STG's implementability properties under the STG's 'Verification' menu (which the STG in Figure 5.10 passes) and even compose it with other STGs using PCOMP under the STG's 'Tools' menu (which is not shown in Figure 5.10 as composition is not required).

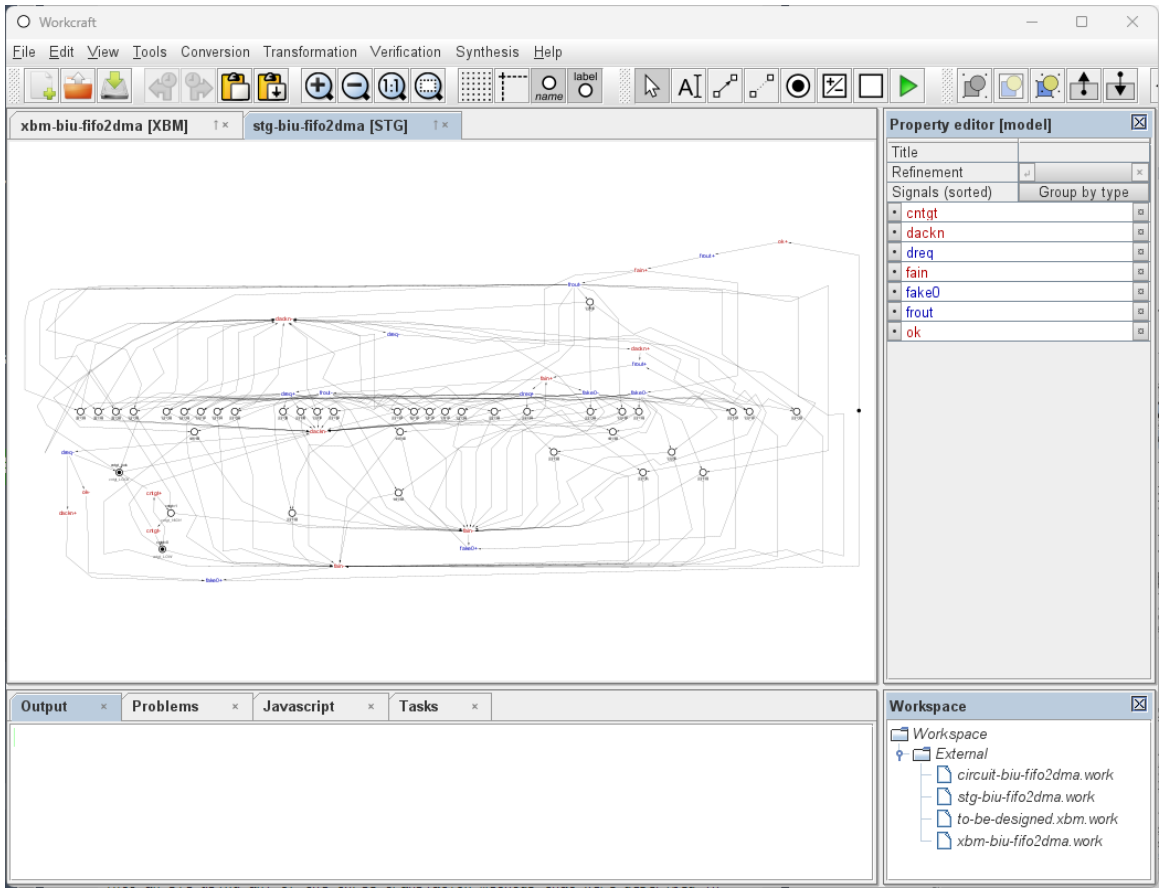


Figure 5.10: Translation from BAs to STGs with neither 'fork' nor 'join' dummy transitions (including the translation of the optional 'fake' output)

5.2.6 Synthesis of Speed-Independent Circuits

Finally, after verifying our translated STG from one of the available translations, we can synthesise it into an SI (QDI) circuit using either PETRIFY or MPSAT backends via WORKCRAFT, and then verify it for conformity with the STG.

To illustrate how the plugin synthesises the STG and produce some possible circuit implementation, Figure 5.11 shows the possible SI (QDI) circuit implementation that is produced from synthesising the translated STG in Figure 5.10, where we can then verify and see that this circuit conforms with the translated STG.

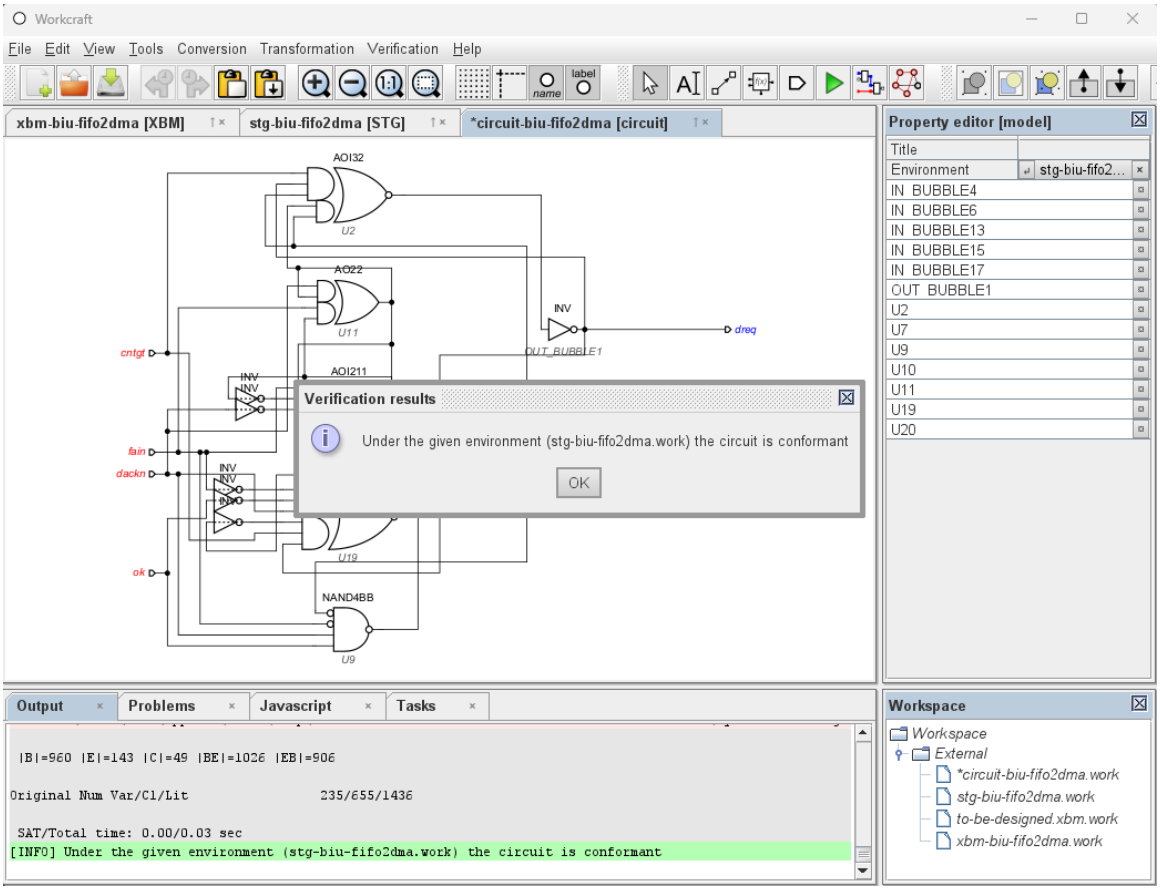


Figure 5.11: Synthesis and conformation of the translated STG

5.3 Experimental Results

In this section, we will highlight the benefits of BA and the established design workflow, where we will cover two experiments involving BAs and our implemented WORKCRAFT plugin.

Section 5.3.1 provides the table of results of the first experiment, where we analyse the size growth of the translated STGs to the original (X)BM specifications.

Section 5.3.2 provides the table of results of the second experiment, where we compare the literal counts of the circuit implementations produced by MINIMALIST and 3D with the literal counts of the circuit implementations produced by PET-

RIFY and MPSAT. To ensure fairness, we factorised the literal counts produced by both MINIMALIST and 3D using LOGIC FRIDAY [62] (a graphical frontend to the ESPRESSO logic minimiser [14]) before counting them, as they were originally in the sum-of-products form.

Note that only the number of literals are compared, as they are a traditionally used metric for analysing the performance of circuits, e.g. the experimental results in [49] only included the CPU performance and literal count. Additionally, it is shown that the power consumption and area of the controller is irrelevant in practice as some other parts of the system will be dominant in both aspects, e.g. [61] shows that the power consumption is small, but the main savings came from the fast reaction to inputs that leads to savings in the analogue part of the buck controller.

In both experiments, we used (X)BM specifications from several publications and translated them into STGs using the three translations described in Section 4.3. These specifications include the published XBM specifications that can be found in the MINIMALIST framework [54, 3] (*concur-mixer*, *dme*, *dram-ctrl*, *hp-ir*, *token-distributor*, *pe-send-ifc*, *pscsi*, *scsi*, *tangram-mixer*), in the 3D [68] tool (*biu-dma2fifo*, *diffeq*, *fifocellctrl*), and other various publications (*ack-xbm-si* [31], *biu-fifo2dma* [68], *imec-alloc-outbound* [23], *nowick-basic* [55], *po-office-sbuf-send-ctl* [22]).

Furthermore, we affix FJ, $\overline{F}J$, and $\overline{FJ}$ below the 'STG' keyword in Table 5.1 and Table 5.2 to help us distinguish the results of the three translated STGs where:

- FJ represents the STG with both 'fork' and 'join' transitions.
- $\overline{F}J$ represents the STG without 'fork' but with 'join' transitions.
- $\overline{F}\,\overline{J}$ represents the STG with neither 'fork' nor 'join' transitions.

5.3.1 Analysis of Model Sizes

In Table 5.1, from the leftmost column to the rightmost column, we have included the name of the specification, the number of signals in the specification, the model size of the (X)BM specifications, and the model sizes of the three translated STGs.

Here, we define the XBM specification's size as the total number of states, arcs and signal appearances within each arc label, and define the STG's size as the total number of explicit places (as implicit places, i.e. those with fanin 1 and fanout 1, are not drawn and so are not counted), arcs, and transitions.

Although the sizes of these two kinds of models are not directly comparable, we can use this result to observe the growth rate of the translations. In particular, we can see that all three translations scale well with the size of the original XBM specification, and that the potential exponential blowup [67] for the $\overline{F}J$ and $\overline{F}\,\overline{J}$ translations did not materialise on these benchmarks. This is particularly beneficial to ensure that any translated specification remains comprehensive, especially when synthesising them into circuits at a smaller complexity as demonstrated in [43].

5.3.2 Comparison of Literal Counts

In Table 5.2, from the leftmost column to the rightmost column, we have included the name of the specification, the number of signals in the specification, the literal counts of the circuit implementations produced by the XBM tools, and the literal counts of the circuit implementations produced by the STG tools.

In each row, we highlight the smallest literal count in bold, and if a literal count is not the smallest in their row then we include a percentage overhead with them, which is calculated with respect to the smallest literal count of that row. We also mark an 'X' for any specification that cannot be read by the tool due to incompatible

syntax, e.g. MINIMALIST cannot read the XBM extension, and mark an 'F' for any synthesis failures, e.g. unresolvable complete state coding (CSC) conflicts.

For an example of how we can interpret these results, let us consider the second row containing the literal counts of the specification *biu-fifo2dma*: Here, *biu-fifo2dma* could not be synthesised with MINIMALIST due to an incompatible syntax nor with 3D due to a synthesis failure. On the other hand, for all three of our proposed translations, *biu-fifo2dma* can be synthesised with PETRIFY and MPSAT reaching a maximum literal count of 24 and producing the smallest literal count of 22 for STG_{FJ} using MPSAT. Because the other synthesis results are not the smallest, a percentage overhead (i.e. $\frac{(24-22)}{22}$) is also included for each result.

Additionally, we calculated the overall average overhead for each synthesis tool, based on the synthesis results produced for all the XBM specifications and for all three translated STGs. This overall average overhead is calculated by totalling the percentage overhead of every literal count found in each tool column, and then dividing them by that tool's total number of successful synthesis results (i.e. the total number of non-X and non-F results). For example, let us consider the overall average overhead for 3D: Here, we total up all the percentage overheads (i.e. $120+26+34+...+36+43+50$) and divide them by the total number of successful synthesis results (i.e. $34-3$), which leads to a result of $\frac{(120+26+34+...+36+43+50)}{(34-3)}$.

5.3.3 Evaluation of Literal Counts Comparison

By evaluating our experimental results, we found that most of the specifications synthesised with PETRIFY and MPSAT had a lower literal count than when they are synthesised with MINIMALIST and 3D, despite many of them are originally (X)BM specifications. In particular, simpler specifications like *concur-mixer* and and *nowick-*

basic shown little to no improvement, as they were already the most optimal result. However, more complicated specifications like *biu-dma2fifo*, *dme-fast*, *pscsi-isend*, and *scsi-tsend-bm* all shown substantial improvements. Note that WORKCRAFT supports both PETRIFY and MPSAT, so one can synthesise the specification with both tools and select the more optimal result.

Moreover, we also found that only a few specifications required the implementation of fake outputs to be synthesiseable with PETRIFY and MPSAT. These included *biu-fifo2dma*, *pe-send-ifc*, *po-sbuf-send-ctl*, and *two-ticks-if* as they all had a signal transition $x+$ ($x-$) followed by an immediate signal transition of $x-$ ($x+$), which causes a CSC conflict.

Finally, all of the benchmarks can be synthesised as an STG when we used MPSAT. This suggests that most XBM specifications can be designed as an STG and remain synthesiseable without needing the 'burst-mode' timing assumption.

5.4 Summary

In this chapter, we cover the automated design flow of BAs and how we can use BAs to establish a connection between the design routes of BM specifications and STGs. In this automated design flow, we provide a step-by-step procedure of how (X)BM specifications are translated to BAs and then subsequently to STGs, where they can be composed with other STGs, verified, and synthesised into SI (QDI) circuits.

Next, we cover the implemented WORKCRAFT plugin that supports the design automation of BAs. In particular, we showcase the features of this plugin that include how we design BAs, how we assign directions to signals, the simulation of BAs, the optional verification of (X)BM's well-formed requirements, automated translation to STGs, and finally synthesis into an SI (QDI) circuit.

Lastly, we cover the experimental results of the design workflow with BAs, where we compare the model sizes between XBM specifications and STGs, and the literal counts between MINIMALIST, 3D, PETRIFY and MPSAT. There, it is shown that the size of the STG did not exponential explode due to the second and third translations in Section 4.3, and that both PETRIFY and MPSAT produced circuit implementations that had lower literal count than the circuit implementations produced by MINIMALIST and 3D.

Table 5.1: Analysis of Model Growth from STG translation

Specification		**Model Size**			
Name	**Sigs**	**XBM**	**STG**$_{FJ}$	**STG**$_{\overline{F}J}$	**STG**$_{\overline{FJ}}$
biu-dma2fifo	6	44	101	96	89
biu-fifo2dma	6	33	93	90	142
counter-6	7	18	46	38	38
concur-mixer	6	27	55	43	41
diffeq-alu1	8	50	132	130	215
diffeq-mul1	6	27	67	59	59
diffeq-mul2	6	19	47	36	35
dme	6	38	50	50	50
dme-fast	7	45	80	68	66
dram-ctrl	14	75	186	169	171
fifocellctrl	4	15	26	20	19
hp-ir	5	33	61	61	74
hp-ir-it-ctrl	12	59	158	149	191
hp-ir-rf-ctrl	11	56	92	81	80
imec-alloc-outb	7	35	41	41	41
imec-sbuf-ramw	10	34	64	56	58
martin-qelement	4	16	16	16	16
nowick-basic	5	21	51	52	51
token-distributor	8	48	48	48	48
pe-send-ifc	8	70	166	197	345
po-sbuf-send-ctl	6	35	83	76	68
pscsi-ircv	7	33	81	81	130
pscsi-isend	7	53	123	121	171
pscsi-trcv	7	33	69	70	123
pscsi-trcv-bm	8	46	114	114	191
pscsi-tsend	7	54	106	104	156
pscsi-tsend-bm	8	58	132	125	179
scsi-isend-bm	9	57	125	123	120
scsi-isend-csm	9	57	125	123	120
scsi-trcv-bm	9	57	127	121	133
scsi-trvc-csm	9	45	101	94	144
scsi-tsend-bm	9	58	112	103	102
scsi-tsend-csm	9	49	85	71	67
tangram-mixer	6	31	35	35	35
two-ticks-if	6	31	77	68	67
Average size increase by factor			2.08	1.95	2.37

Table 5.2: Comparison of Literal count between BM tools and STG tools

	XBM		**STG$_{FJ}$**	
Specification	MINIMALIST	3D	PETRIFY	MPSAT
biu-dma2fifo	X	44 (+120%)	22 (+10%)	**20**
biu-fifo2dma	X	F	24 (+10%)	**22**
counter-6	13	13	13	13
concur-mixer	15	15	15	15
diffeq-alu1	49 (+59%)	39 (+26%)	**31**	35 (+13%)
diffeq-mul1	X	28 (+34%)	24 (+15%)	24 (+15%)
diffeq-mul2	X	13 (+9%)	13 (+9%)	**12**
dme	21 (+50%)	20 (+43%)	15 (+8%)	**14**
dme-fast	27 (+69%)	27 (+69%)	23 (+44%)	**16**
dram-ctrl	41 (+3%)	**40**	46 (+15%)	41 (+3%)
fifocellctrl	X	10 (+12%)	11 (+23%)	**9**
hp-ir	8	F	8	8
hp-ir-it-ctrl	46 (+22%)	39 (+3%)	44 (+16%)	**38**
hp-ir-rf-ctrl	37 (+24%)	F	F	32 (+7%)
imec-alloc-outb	23 (+44%)	21 (+32%)	**16**	17 (+7%)
imec-sbuf-ramw	X	30 (+20%)	26 (+4%)	**25**
martin-qelement	9 (+29%)	9 (+29%)	**7**	**7**
nowick-basic	**10**	**10**	11 (+10%)	**10**
token-distributor	42 (+56%)	39 (+45%)	28 (+4%)	**27**
pe-send-ifc	81 (+89%)	52 (+21%)	49 (+14%)	**43**
po-sbuf-send-ctl	28 (+34%)	31 (+48%)	26 (+24%)	**21**
pscsi-ircv	27 (+50%)	27 (+50%)	21 (+17%)	**18**
pscsi-isend	55 (+67%)	61 (+85%)	40 (+22%)	**33**
pscsi-trcv	23 (+28%)	23 (+28%)	**18**	19 (+6%)
pscsi-trcv-bm	38 (+32%)	35 (+21%)	33 (+14%)	33 (+14%)
pscsi-tsend	43 (+35%)	45 (+41%)	36 (+13%)	**32**
pscsi-tsend-bm	52 (+45%)	50 (+39%)	46 (+28%)	37 (+3%)
scsi-isend-bm	47 (+57%)	50 (+67%)	39 (+30%)	**30**
scsi-isend-csm	47 (+52%)	50 (+62%)	39 (+26%)	**31**
scsi-trcv-bm	55 (+72%)	45 (+41%)	37 (+16%)	**32**
scsi-trvc-csm	46 (+65%)	38 (+36%)	32 (+15%)	**28**
scsi-tsend-bm	76 (+118%)	50 (+43%)	43 (+23%)	36 (+3%)
scsi-tsend-csm	41 (+71%)	36 (+50%)	40 (+67%)	25 (+5%)
tangram-mixer	**8**	**8**	10 (+25%)	**8**
two-ticks-if	13 (+19%)	**11**	12 (+10%)	**11**
Average overhead	41%	33%	15%	2%

Table 5.2: Comparison of Literal count between BM tools and STG tools (continued)

	$\mathbf{STG}_{\overline{FJ}}$		$\mathbf{STG}_{\overline{FJ}}$	
Specification	PETRIFY	MPSAT	PETRIFY	MPSAT
biu-dma2fifo	22 (+10%)	**20**	22 (+10%)	**20**
biu-fifo2dma	24 (+10%)	24 (+10%)	24 (+10%)	24 (+10%)
counter-6	13	13	13	13
concur-mixer	15	15	15	15
diffeq-alu1	**31**	35 (+13%)	**31**	35 (+13%)
diffeq-mul1	**21**	27 (+29%)	**21**	27 (+29%)
diffeq-mul2	13 (+9%)	13 (+9%)	13 (+9%)	13 (+9%)
dme	15 (+8%)	**14**	15 (+8%)	**14**
dme-fast	21 (+32%)	**16**	21 (+32%)	**16**
dram-ctrl	45 (+13%)	41 (+3%)	45 (+13%)	41 (+3%)
fifocellctrl	11 (+23%)	**9**	11 (+23%)	**9**
hp-ir	8	8	8	8
hp-ir-it-ctrl	44 (+16%)	51 (+35%)	44 (+16%)	51 (+35%)
hp-ir-rf-ctrl	F	**30**	F	**30**
imec-alloc-outb	**16**	17 (+7%)	**16**	17 (+7%)
imec-sbuf-ramw	26 (+4%)	30 (+20%)	26 (+4%)	29 (+16%)
martin-qelement	**7**	**7**	**7**	**7**
nowick-basic	11 (+10%)	**10**	11 (+10%)	**10**
token-distributor	28 (+4%)	28 (+4%)	28 (+4%)	**27**
pe-send-ifc	50 (+17%)	48 (+12%)	50 (+17%)	49 (+14%)
po-sbuf-send-ctl	26 (+24%)	27 (+29%)	26 (+24%)	26 (+24%)
pscsi-ircv	21 (+17%)	**18**	20 (+12%)	**18**
pscsi-isend	40 (+22%)	**33**	**33**	**33**
pscsi-trcv	**18**	19 (+6%)	20 (+12%)	19 (+6%)
pscsi-trcv-bm	32 (+11%)	33 (+14%)	**29**	33 (+14%)
pscsi-tsend	45 (+41%)	**32**	45 (+41%)	**32**
pscsi-tsend-bm	46 (+28%)	**36**	40 (+12%)	37 (+3%)
scsi-isend-bm	39 (+30%)	36 (+20%)	40 (+34%)	36 (+20%)
scsi-isend-csm	39 (+26%)	36 (+17%)	42 (+36%)	36 (+17%)
scsi-trcv-bm	40 (+25%)	41 (+29%)	38 (+19%)	41 (+29%)
scsi-trvc-csm	32 (+15%)	**28**	32 (+15%)	**28**
scsi-tsend-bm	43 (+23%)	38 (+9%)	**35**	39 (+12%)
scsi-tsend-csm	40 (+67%)	**24**	39 (+63%)	25 (+5%)
tangram-mixer	10 (+25%)	**8**	10 (+25%)	**8**
two-ticks-if	12 (+10%)	**11**	12 (+10%)	**11**
Average overhead	15%	7%	15%	7%

Chapter 6

Case Studies

In this chapter, we will highlight the benefits of the proposed Burst Automaton (BA) model and the implemented WORKCRAFT plugin, by covering two case studies based on real-life systems.

The first case study involves the design of the buck converter [61], where we will investigate the design and operations of the buck converter including their Signal Transition Graph (STG) models. We will then try to design the same scenarios of the buck converter using Burst-Mode (BM) Specifications, before we identify some of the shortcomings of BM specifications and design the buck's scenarios using BAs.

The second case study involves the design of the Versa Module Europa (VME) bus controller [1], where we will investigate the design and operations of the VME bus controller including their STG models. Here, we model the read, write, and combined modes the VME bus controller using BAs, before we show that the BAs and the reachability graphs of the STGs are equivalent.

6.1 Buck Controller

In this section, we will study the design of the buck converter [61], where we will first show its STGs and try to model the buck using (X)BM specifications, before we highlight the limitations of (X)BM specifications and the benefits of BAs.

6.1.1 Design and Operations of the Buck Converter

Buck converter is a direct current (DC) to DC converter that steps down the voltage and steps up the current from its input (battery) to its output (load). It comprises an analogue buck and an asynchronous controller as shown in the top-level schematic in Figure 6.1. Here, the asynchronous controller is to be specified using some formalism (e.g. STG) and then synthesised as an asynchronous circuit.

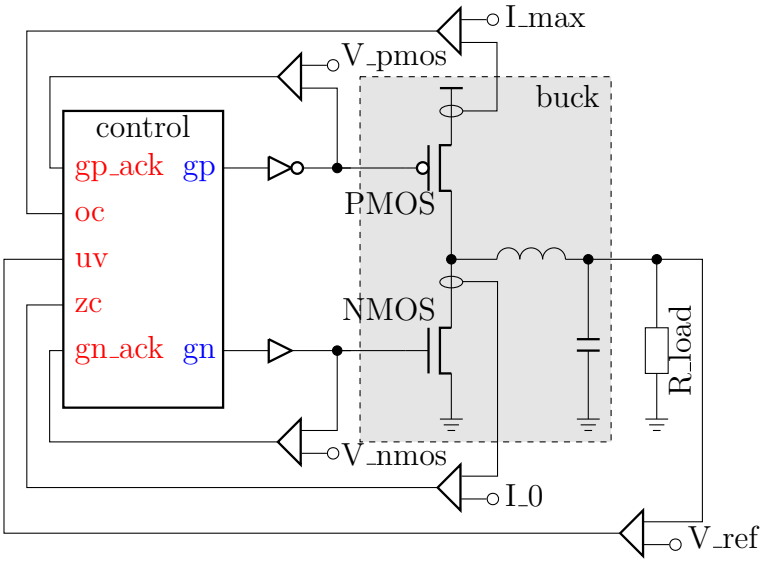


Figure 6.1: Schematic of a basic buck converter

In this schematic diagram, we can see that the controller switches the power regulating PMOS and NMOS transistors ON and OFF, as a reaction to the under-voltage (UV), over-current (OC) and zero-crossing (ZC) conditions.

These conditions are detected and signalled by a set of sensors (i.e. uv, oc, and zc) that are implemented as comparators of the measured current and voltage levels against some reference values (i.e. V_ref, I_max and I_0 respectively).

The gp and gn signals are then buffered to drive the very large power regulating transistors, and their effect on the buck can be significantly delayed. Thus, the controller is explicitly notified by the gp_ack and gn_ack signals, when the power transistor threshold levels are crossed.

As such, we can even capture this operation of the buck as a phase diagram as shown in Figure 6.2, where the intended reaction to the various sequences of the detected conditions can be seen.

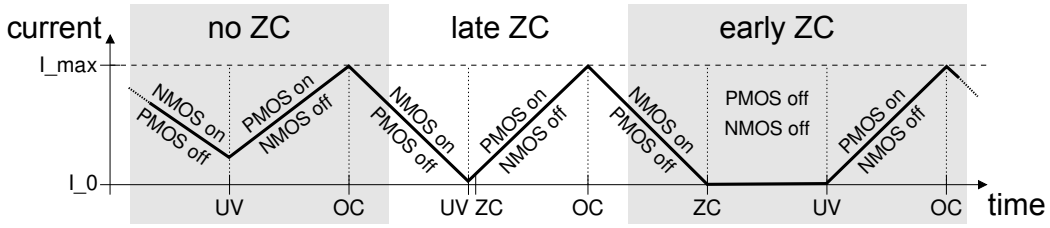
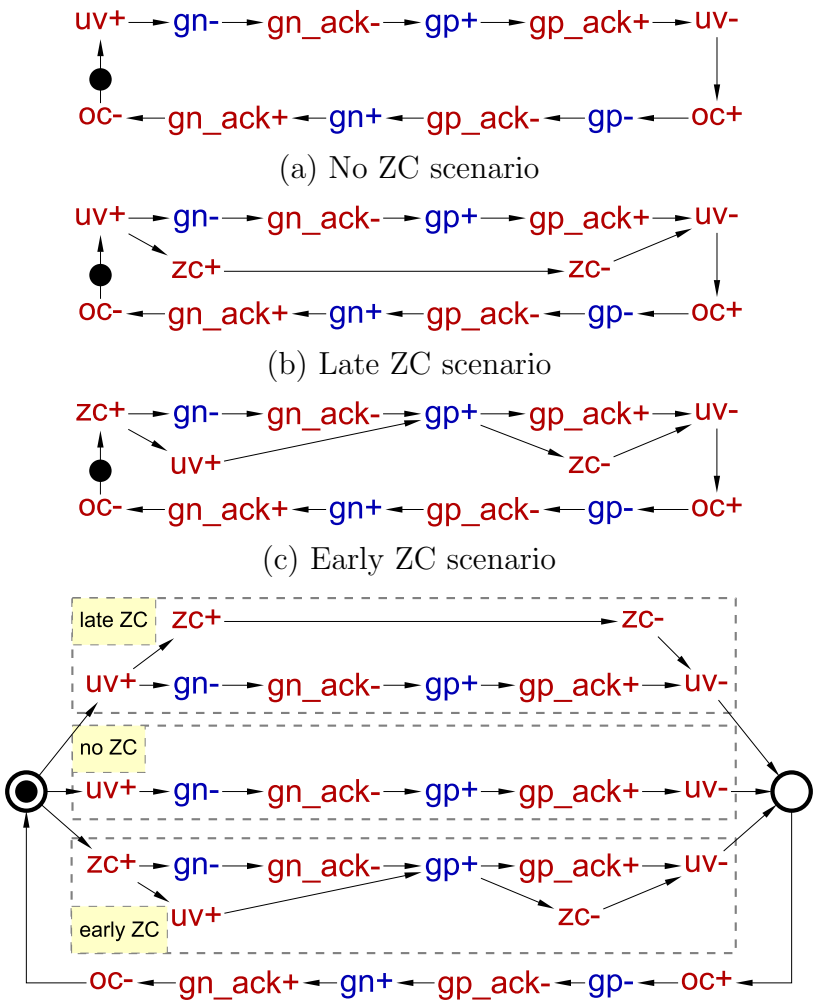


Figure 6.2: Phase diagram of buck operation

Here, we can see the alternation of the UV and OC conditions, which are handled by switching the power regulating PMOS and NMOS transistors of the buck ON and OFF. Here, the detection of the ZC condition after UV does not change this behaviour. However, if ZC is detected before UV then both the PMOS and NMOS transistors remain OFF until the UV event.

As such, we can even model this behaviour with STGs in a natural way by capturing the three possible scenarios (i.e. late ZC, no ZC, early ZC) as three separate STGs, and then combine them into a single STG, where the reset sequence that is common in all three scenarios can be compressed into one sequence for the final combined model, as shown in Figure 6.3.

Note that it is particularly beneficial to model several parts of a controller and then compose them into a single model, rather than model the whole of the controller, as shown in [47].



(a) No ZC scenario

(b) Late ZC scenario

(c) Early ZC scenario

(d) Combined STG of all three scenarios

Figure 6.3: STGs of the buck scenarios

Using WORKCRAFT, we can easily verify that the combined STG model satisfies the standard STG implementability properties that include consistency, deadlock-freeness, input-properness, output-persistency, and output determinacy, which are all defined in Section 2.2.3.

To avoid short-circuiting the battery, the PMOS and NMOS transistors of the buck must never be ON at the same time, which can be captured as a custom property that must be formally verified. Hence, we can specify this custom property as the invariant $\overline{gp} \cdot \overline{gp\_ack} + \overline{gn} \cdot \overline{gn\_ack}$, which ensures that gp and gp_ack are not on at the same time as gn and gn_ack, and we can then easily verify in WORKCRAFT that this custom property also holds for our STG above. Note that this invariant also takes care of the transient short-circuit, due to the slow switching of transistors.

Following the verification of our STG, we can then synthesise it into a speed-independent (SI) (quasi-delay insensitive (QDI)) circuit using either PETRIFY or MPSAT backends in WORKCRAFT, where a possible circuit implementation is produced that could be the SI (QDI) circuit shown in Figure 6.4.

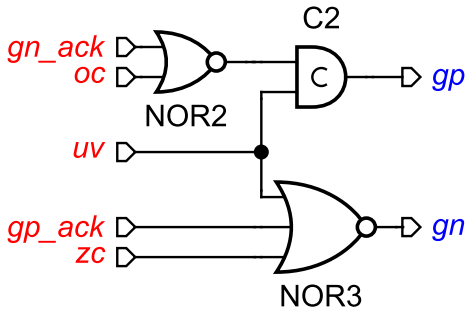


Figure 6.4: Possible circuit implementation of the buck

As the final step, we can again use WORKCRAFT to verify that this circuit is indeed a correct implementation of the above STG in Figure 6.3d (via conformation), and that it also holds the above invariant which prevents short-circuiting the battery.

6.1.2 Modelling Issues for Burst-Mode Specifications

Now, suppose that we are a circuit designer who wishes to design the buck controller using an FSM-based model, e.g. BM specification. However, this is not straightforward due to the following circumstances:

1. There is concurrency between inputs and outputs, which cannot be captured in BM specifications as explained in Section 2.2.2. In particular:

   (a) For the late ZC scenario, one may think it is possible to use "don't care" transitions from the XBM specification to capture the concurrency between transition sequences zc+ → zc- and gn- → gn_ack- → gp+ → gp_ack+. However, this does not work as there must at least be one compulsory input that appears with the "don't cares" due to the XBM's compulsory input requirement defined in Section 2.2.2, where the compulsory input uv+ must then appear with the "don't care" input zc* meaning zc+ can also fire before uv+, which breaks the scenario. This becomes even more complicated as zc- is also concurrent with the transition sequence gn- → gn_ack- → gp+ → gp_ack+, meaning expressing zc- as a "don't care" would either break the scenario (as it needs to appear and terminate with another signal) or violate the XBM's compulsory input requirement (as it would appear by itself).

   (b) Similarly for the early ZC scenario, the "don't care" transition cannot be used for the uv+ transition as, again, the compulsory input zc+ must appear with the 'don't care' input uv* meaning uv+ can fire before zc+ which too breaks the scenario.

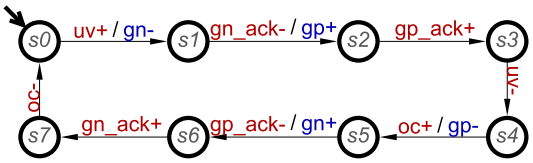
2. There are sequences of inputs without intermediate outputs. While it is possible in BM specifications to have bursts without outputs, this is interpreted as having a fake output, which may potentially cause a state change. Furthermore, this cannot be done without introducing the timing assumption that firing this fake transition and state change would happen faster than the arrival of the next input, which is difficult to guarantee.

3. Finally, it is not possible to compose the BM specifications of the three ZC scenarios into one model (like in Figure 6.3d where the three STGs were composed into one STG), as there is a non-deterministic choice between the two $uv+$ transitions that is prevented by the BM's maximal set property. Additionally, we are also not able to decouple the outputs due to the BM's inherent timing assumption, nor are we able to express bursts containing only outputs due to the BM's non-empty input burst property. Furthermore, removing this non-deterministic choice from the buck converter is non-trivial.

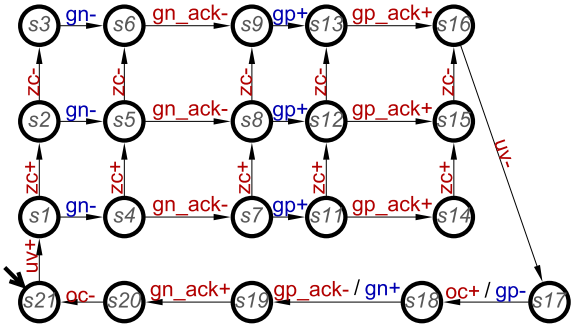
6.1.3 Relaxing Burst-Mode Well-formedness Requirements

Due to the non-straightforwardness in modelling the buck converter with BM specifications, let us instead consider modelling the converter with a 'generalised' BM specification, where we set aside some of the BM's well-formedness properties, i.e. BM's non-empty input bursts requirement is ignored to enable empty output bursts and subsequent output bursts, and BM's maximal set property and XBM's distinguishability constraint are ignored to enable non-determinism.

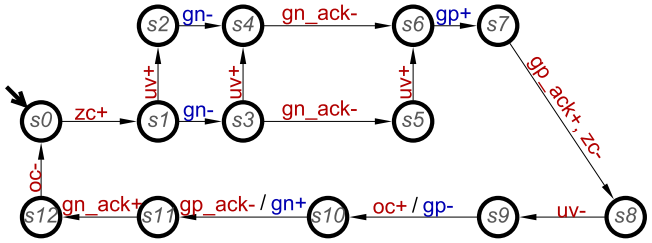
By relaxing these requirements, it is now possible for us to model the converter without any issues, as shown in Figure 6.5 where each of the three ZC scenarios can be described as a BM-like model. Note that unlike the traditional interpretation of BM specification where 'fake' outputs must be created for every burst without an output, this is not required for BAs and is instead included as another method for translating a BA into an STG with explicitated 'fake' outputs. Additionally, these three models can also be manually combined into a single model while compressing the common reset phase, as shown in Figure 6.5d.



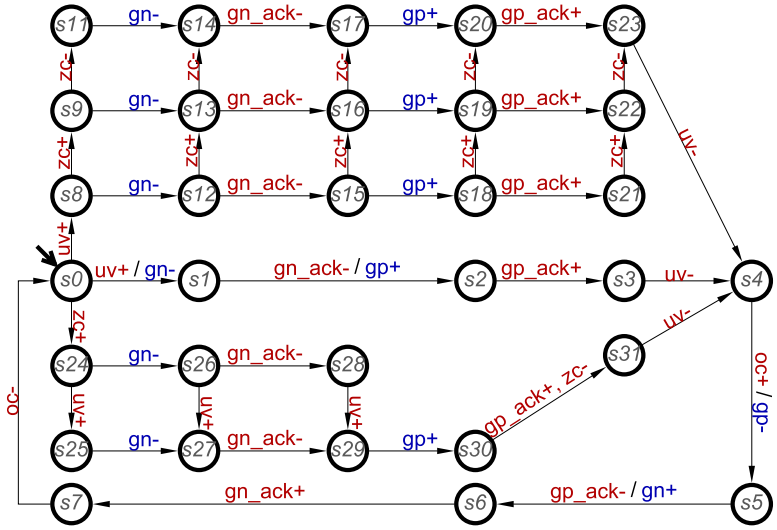
(a) No ZC scenario



(b) Late ZC scenario



(c) Early ZC scenario



(d) Combined BM-like specification of all three scenarios

Figure 6.5: BM-like specifications of the buck scenarios

When we observe the resulting model, we can see that it makes sense semantically despite not being a well-formed BM, and that it actually strongly resembles a reachability graph of the original STG model in Figure 6.3d, with the only difference being that transitions were combined into bursts where possible.

As we already have an STG produced, it would be tempting to just build its reachability graph and interpret it as a BM-like model with single-event bursts. Unfortunately, this is not viable as BM specifications are not a proper extension of FSMs, and so the state graphs of STGs are generally not well-formed BM specifications.

On the other hand, we may even opt to only use STGs to design their specifications. However, many circuit designers are still more familiar with FSMs, and there are still some inclinations that the industry finds STGs to be too complicated when compared to the very familiarised FSM-based models.

Thus, this encourages the need for BAs, as BAs can help bridge the gap between BM specifications and STGs where, on the side BM specifications, it is not possible to express certain behaviours, and on the side of STGs, many circuit designers are still not familiar with it.

For example, by removing some of the BM's well-formedness requirements and enabling a more fluid transition between different formalisms (i.e. FSMs like BM specifications and Petri nets like STGs), we can achieve the three models shown in Figure 6.5, which are all valid BAs that can be subsequently translated to STGs or FSMs, formally verified, and then synthesised into SI (QDI) circuits.

In particular, we even can translate the composed BA in Figure 6.5d automatically to the STG shown in Figure 6.6 via Workcraft.

Alternatively, we may even translate the three BAs in Figure 6.5 into STGs, apply them with net synthesis using Petrify, and then combine them, while compressing the reset phase, to achieved the same STG shown in Figure 6.3d.

Once our translated STG is ready, we can then formally verify in WORKCRAFT that our translated STG satisfies the standard STG implementability properties, and the custom property where there is no short-circuit, before we synthesis the translated STG into an SI (QDI) circuit, like the one shown in Figure 6.4.

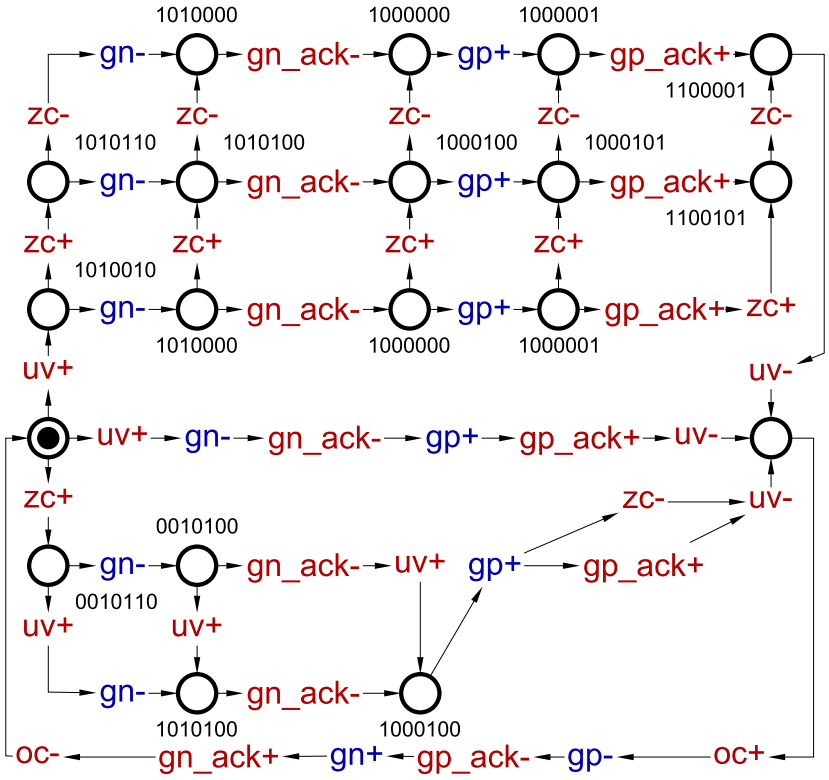


Figure 6.6: Translated STG of the overall buck converter

6.1.4 Benefits from Burst Automata Modelling

From this simple modelling exercise, there are several potential areas of improvements that we can identify for BM specifications, which BAs help achieve. These include:

- **Local input-output mode**: Sometimes, e.g. when modelling input-output concurrency or choices like the mutex element, bursts are too coarse-grained and the XBM specification's "don't care" transitions may not be applicable, so

it would be beneficial to allow the designer to fall back to the FSM-style fine-grain way of specifying behaviour. The flip side of this is that the possibility to specify a wider class of behaviours would require a more computationally expensive SI (QDI) synthesis. However, we can argue that this is a reasonable price to pay, as the alternative is not being able to model the desired behavior at all, or introducing timing assumptions which can be much more cumbersome and computationally expensive to handle.

- **Non-deterministic specifications**: By dropping some of (X)BM's well-formed requirements, in particular the non-empty input burst property and BM's maximal set property (or XBM's extension, the distinguishability constraint), we can enable modelling and synthesis of non-deterministic specifications, e.g. the BA in Figure 6.5d has a non-deterministic choice at the initial state between two $uv+$ transitions. Additionally we also allow empty bursts, which can be interpreted as ε-transitions in FSMs. The semantics of non-deterministic specifications are formally defined in [37], where it is argued that non-determinism is an extremely useful and powerful feature, e.g. as it was shown earlier in the buck converter example above, where no timing assumption was required for the late ZC scenario due to the no ZC scenario. Note that common transformations, such as hiding signals, can turn deterministic specifications into non-deterministic ones. In particular, non-deterministic specifications can also be determinised provided that output-determinacy holds, as violation of output determinacy means that there are errors.

- **Model interoperability**: There are great advantages of modelling formalisms to have compatible semantics. This allows fluid transition (via automatic translation) from one formalism to another, as well as designing large systems from

pre-existing blocks expressed in different formalisms. Thus, with BAs, interoperability can be achieved between BMs, XBMs, STGs, FSMs, Petri nets, or generally any formalism that can be automatically translated into any of these models, e.g. Waveform Transition Graphs (WTGs) [50] which have STG-based semantics.

Interestingly, these advantages are achieved in the BA not by adding 'bells and whistles' to the BM specification, e.g. like how XBM specifications added conditionals and "don't cares" that made its well-formedness properties more complicated as a result, but rather by dropping some of its restrictions, which in turn also simplified the definition of the model. In particular, BAs can be simpler to define than BM specifications as:

- There is no maximal set property (nor XBM's extension, the distinguishability constraint) that prevents non-deterministic choices.
- There is no requirement that the input bursts must be non-empty (which, included with the above bullet point, means non-determinism can be enabled).
- Bursts are not split into input and output half-bursts, but instead are just sets of actions. Note that splitting these inputs and outputs into half bursts may still be useful and can be shown graphically, but it is not a part of the model definition in Section 4.2.1.
- There is in fact no need to distinguish between inputs and outputs as BAs are defined over a homogeneous alphabet of actions, and the partitioning of the alphabet into inputs and outputs, as well as adding directions to actions (e.g. $a+$ and $a-$) with the associated consistency requirements, are instead refinements of the original BA model.

6.2 VME Bus Controller

In this section, we will study the design of the VME bus controller [1], which can be modelled as an STG. We will then attempt to model the VME bus controller's mode of operations using (X)BM specifications, which have a restricted ordering of output events, before attempting to model with BAs.

6.2.1 Design and operations of the VME Bus Controller

To understand the operation of the VME bus controller, let us consider its schematic shown in Figure 6.7.

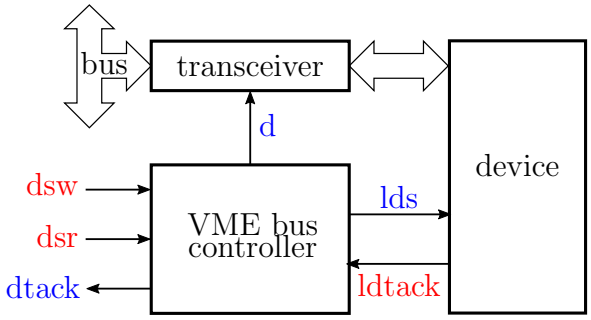


Figure 6.7: Schematic of the VME bus controller

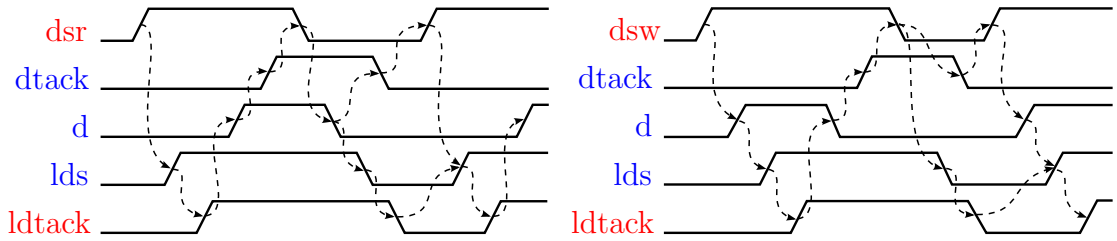
In this schematic diagram, we can see that the controller has two modes of operation, where one mode reads the data from the device into the bus by enabling the input dsr and the other mode writes the data from the bus into the device by enabling the input dsw.

When the VME bus controller is in read mode, a request to read the data from the device is sent by firing lds+. When the device is ready to send this data, it is acknowledged by the assertion of ldtack+, where the VME bus controller opens up its transceiver by asserting d+ and notifies the bus that data is ready for a transfer

by asserting dtack+. Once the read operation is complete, all signals return to their initial state.

When the VME bus controller is in write mode and when the data is stable on the bus, the VME bus controller opens up its transceiver by asserting d+ and a write request is then made by asserting lds+. Once the device acknowledges that it has received the data via the assertion of ldtack+, the VME bus controller then closes the transceiver by asserting d-. As a result, the device gets isolated from the bus, and the bus is then notified that the write operation is completed by the assertion of dtack+. Like in read mode, once the write operation is complete, all signals return to their initial state.

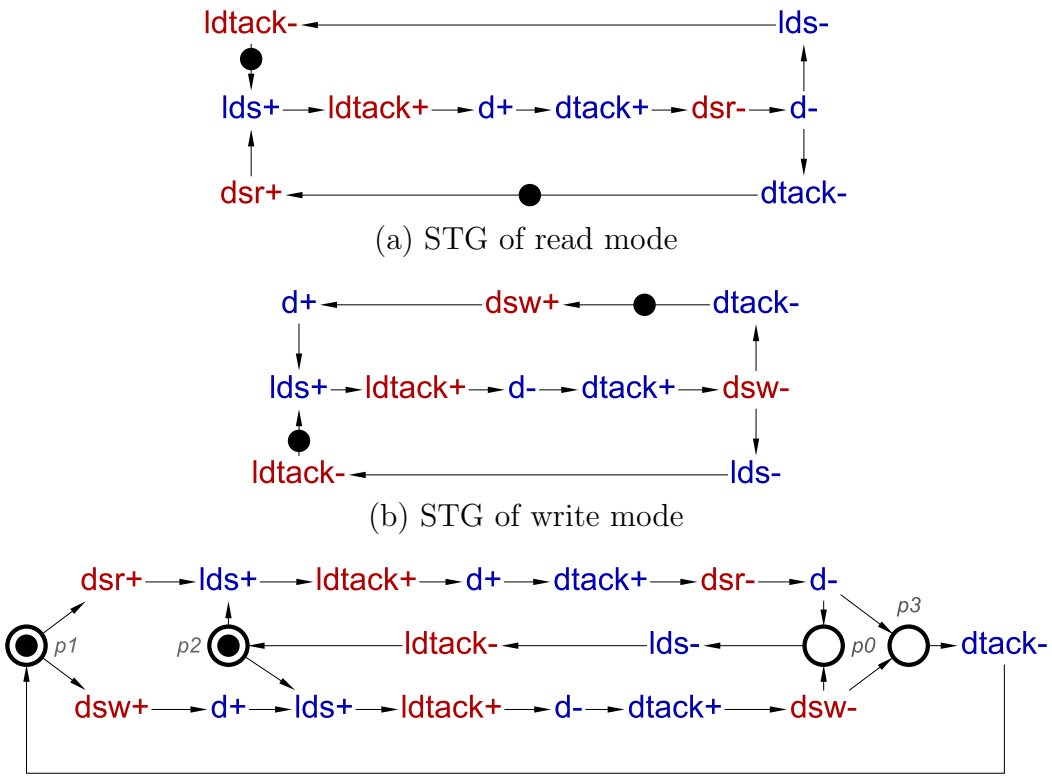
Thus, we capture each operation of the VME bus controller as timing diagrams as shown in Figure 6.8.



(a) Read mode (b) Write mode

Figure 6.8: Timing diagrams for the VME bus controller

Using these timing diagrams, we can even specify the VME bus controller's two modes as STGs as shown in Figure 6.9, where we can capture sequence of events that were described for the read mode (Figure 6.9a) and write mode (Figure 6.9b). Notably, we can even combine these STGs together by merging their initial marking and collapsing the common dtack- transition as shown in Figure 6.9c, where we now have one choice place between dsr+ and dsw+ and one choice place between top lds+ and bottom lds+, as well as two merge places from either top d- or dsw-.



(a) STG of read mode

(b) STG of write mode

(c) Combined STG

Figure 6.9: STGs of the VME bus controller

If we verify the composed STG model, we will see that it satisfies the standard STG implementability properties that includes deadlock-freeness, input-properness, output-persistency, and output determinacy, which are all defined in Section 2.2.3 and can be easily checked in WORKCRAFT.

We can then synthesise the composed STG into an SI (QDI) circuit using either of the PETRIFY or MPSAT backends in WORKCRAFT, which produces a possible circuit implementation like the SI (QDI) circuit shown in Figure 6.10.

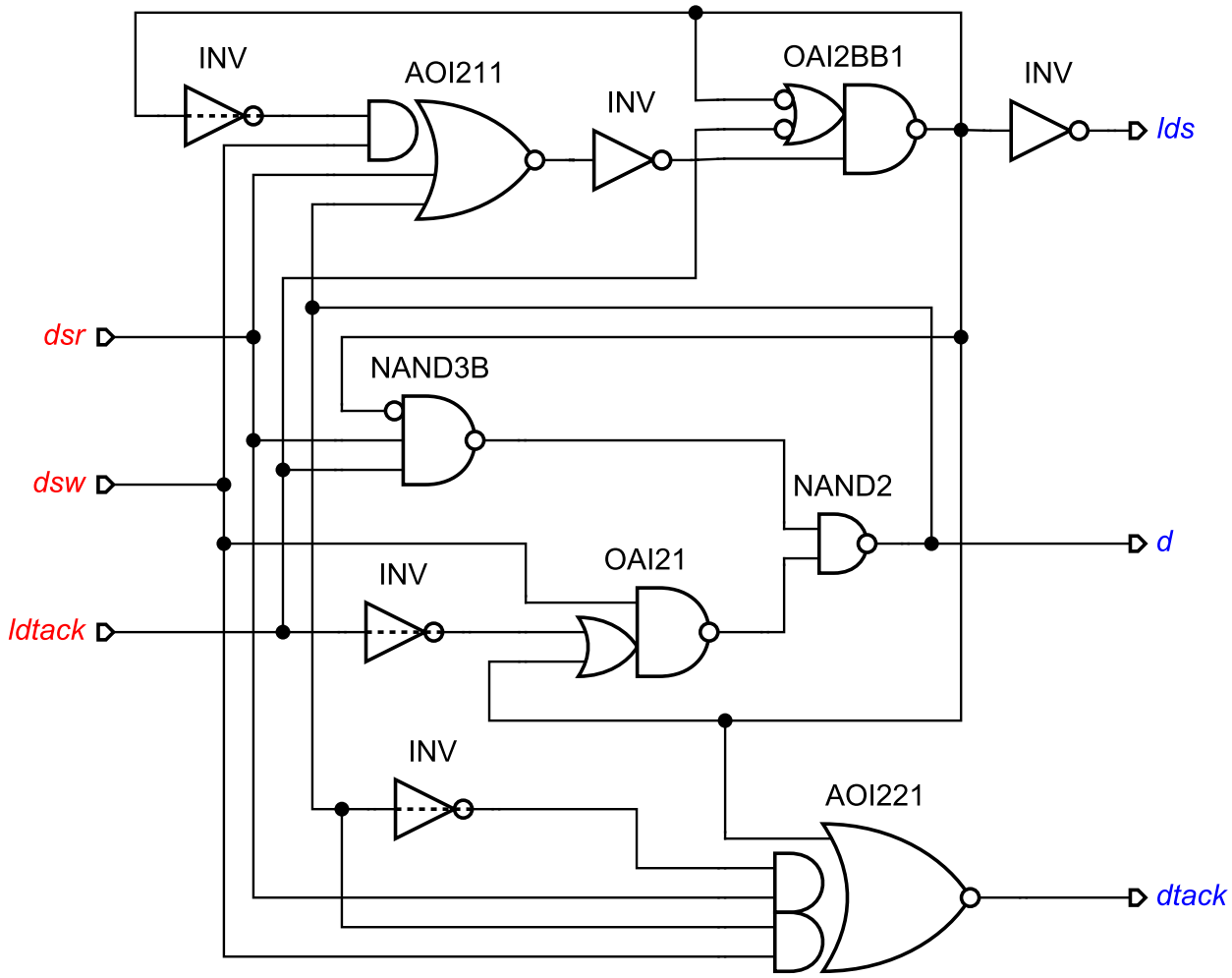


Figure 6.10: Possible circuit implementation of the VME bus controller

6.2.2 Modelling with Burst Automata

Now, if we try to model the VME bus controller using (X)BM specifications, it is unfortunately not possible as:

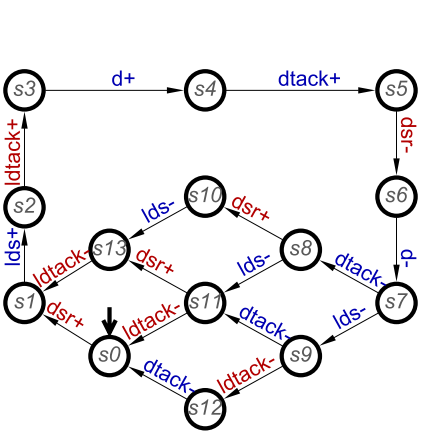
- There is input-output concurrency, which as explained in Section 2.2.2 cannot be expressed in BM specifications. While XBM specifications can express input-output concurrency using its conditionals and "don't cares", neither conditionals nor "don't cares" can be applied here, as none of the inputs in the

VME bus controller appears with another input (even if we turned it into a BM specification), meaning conditionals would have appeared on their own if we substituted any of the inputs (which is incorrect), and the compulsory input requirement (for "don't cares") would likely be violated.

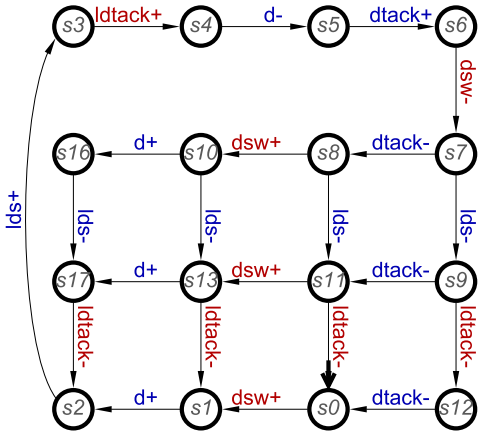
- There are output choices, which BM specifications cannot express because of their non-empty input burst requirement. Although one might consider grouping up these output transitions into bursts, e.g. d+ and dtack+, this would break the scenario, as the ordering of these outputs in the VME bus controller is important, e.g. dtack+ is dependent on d+.

- There is merging of specifications, which can be a non-trivial operation for BM specifications. While there are decomposition methods like BM Decomp that are available for BM specifications, there does not seem to be a composition method for BM specifications. Instead, we can just consider combining these specifications together like how we do for STGs, it becomes complicated once we need to factor in situations, such as input-output concurrency, output choices and even non-deterministic choices (which are not possible in BM specifications, due to its maximal set property).

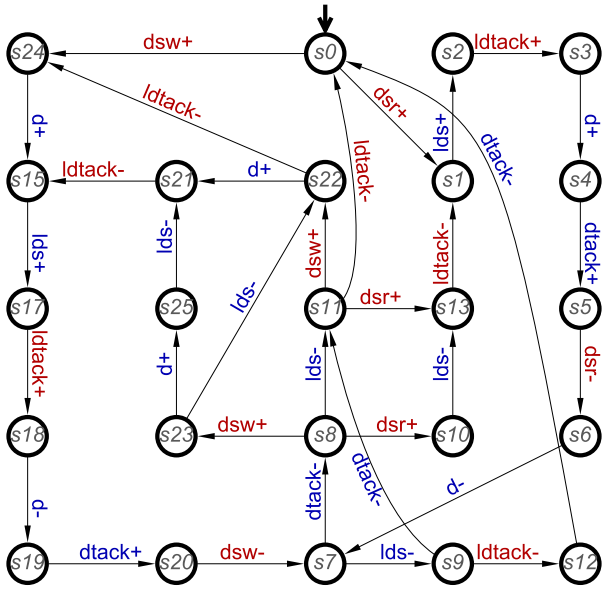
Instead, suppose that we use BAs to model the VME bus controller. Here, because there are no well-formed requirements to be considered like in (X)BM specifications, we can model the VME bus controller easily using BAs as shown in Figure 6.11, where concurrent inputs and outputs, as well as output choices, can be expressed in BAs.



(a) BA of read mode



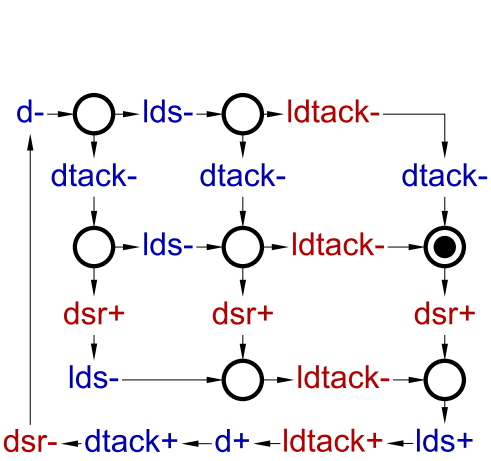
(b) BA of write mode



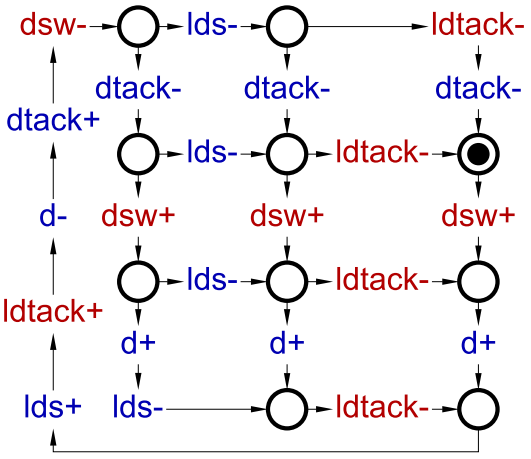
(c) Combined BA

Figure 6.11: BAs of the VME bus controller

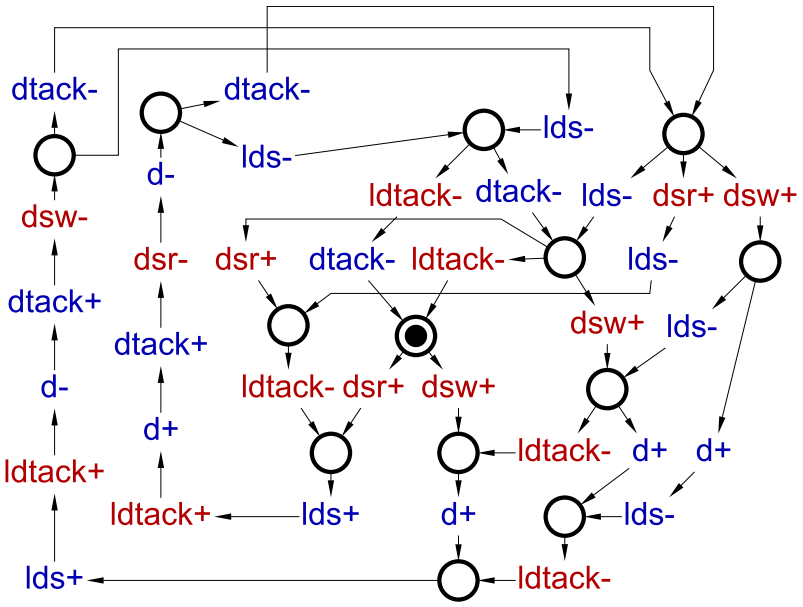
Once our BAs are modelled, we can then translate them into the STGs shown in Figure 6.12, where we can subsequently verify and determine that they pass all of the STG's implementability properties, and synthesise the combined STG to produce the same possible circuit implementation of the VME bus controller shown in Figure 6.10. Note that we can also apply 'net synthesis' to these STGs with PETRIFY to achieve same STGs shown in Figure 6.9.



(a) Translated STG of read mode



(b) Translated STG of write mode

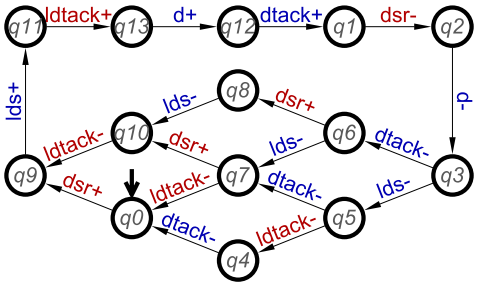


(c) Combined translated STG

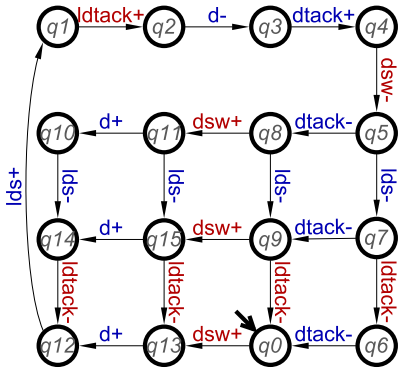
Figure 6.12: Translated STGs of the VME bus controller

To ensure that our BAs are equivalent to original STGs in Figure 6.9, we can also build their reachability graphs as shown in Figure 6.13 to check for model equivalences.

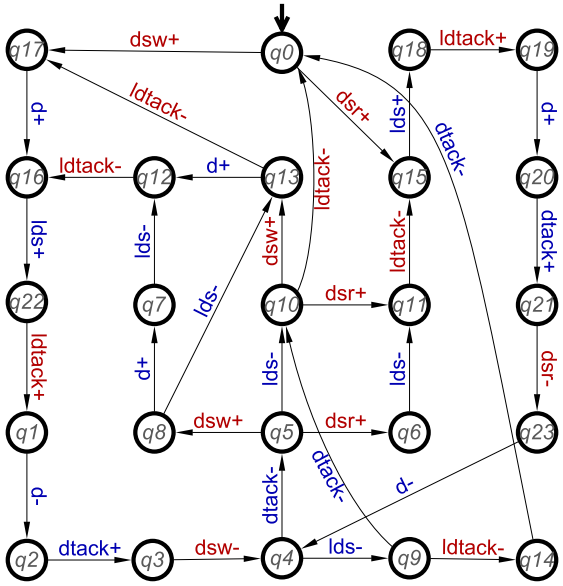
Note that by observing the BAs in Figure 6.11, we can see that they exactly resemble the STG's reachability graphs in Figure 6.13 and that each state can be matched accordingly, as the transitions are common in both BAs and STG reachability graphs.



(a) Reachability graph of STG specifying read mode



(b) Reachability graph of STG specifying write mode



(c) Reachability graph of Combined STG

Figure 6.13: Reachability graph of the STGs specifying the VME bus controller

Chapter 7

Conclusion

This thesis presents a new formal model called Burst Automaton (BA) that can capture the design, verification, and synthesis of composed Finite State Machine (FSM)-based models, while also providing the necessary framework for enabling interoperability of many different models like Burst-Mode (BM) specifications and Signal Transition Graphs (STGs), which helps establish a new design route that bridges the gap between the 'legacy' design route and 'disruptive' design route of specifying asynchronous circuits.

In this chapter, we summarise the main ideas that are developed and discussed throughout this thesis, where we highlight the key contributions and discuss their impact in Section 7.1, and then outline the future research area that can be considered for further investigation and improvement to the project in Section 7.2.

7.1 Summary of Thesis Contributions

Burst Automaton model: In Chapter 4, we introduce the BA model, where we provide its model description (Section 4.1), its formal definition (Section 4.2.1) and its defined semantics as an asynchronous state graph (Section 4.2.2), which the latter is comparable to the reachability graph of STGs.

With the support of several methods that can translate BAs to STGs (Section 4.3),

BAs enable interoperability between many models including BM specifications, Extended BM (XBM) specifications, STGs, FSMs, or generally any formalism that can be automatically translated into any of these models, e.g. Waveform Transition Graphs (WTGs) [50] which have STG-based semantics. As a result, we also enable the composition of (X)BM specifications via BA (Section 4.4), which can be verified and synthesised into a speed-independent (SI) (quasi-delay insensitive (QDI)) circuit.

As shown in Section 3.3, this particularly helps establish a new 'co-design' route that bridges the gap between the 'legacy' design route of BM specifications and the 'disruptive' design route of STGs, where we grant the state-based circuit designers access to the well-established tools that are available for STGs by automated translation via BA, and similarly, enable event-based circuit designers to make use of 'legacy' designed BM specifications via a convenient import and export mechanism.

The benefits of BAs can also be found in Chapter 6, where we cover two case studies involving the design of a buck converter and the design of a VME bus controller, and show how BAs can enhance the modelling of BM specifications to express similar behaviours to the STGs.

Automated design flow based on the Burst Automaton's 'co-design' route: In Chapter 3, we discuss the issues that are posed for the 'legacy' design route and the 'disruptive' design route of specifying asynchronous circuits, where the former produces circuit implementations that are not well-optimised and the latter is avoided by state-based circuit designers due to their unfamiliarity with STGs (Section 3.1). These issues are then highlighted and shown in our motivational example, where we first specify the handshake decoupler as an STG and then try to specify it as a BM specification (Section 3.2).

This results in the proposal of a new 'co-design' route that is established by using BAs, where it bridges the gap between the 'legacy' design route and the 'disruptive'

design route (Section 3.3), and allows us to then easily specify the handshake decoupler using BAs, which shown several potential areas of improvements for both the 'legacy' design route and the 'disruptive' design route (Section 3.4).

Translation methods from Burst Automata to Signal Transition Graphs: In Section 4.3, we present three translations from BAs to STGs, where the first translation is linear and preserves the language (Section 4.3.1), the second is exponential and preserves weak bisimulation (Section 4.3.2), and the third is exponential and preserves strong bisimulation (Section 4.3.3).

In particular, these proposed translations enable BAs to be easily composed (Section 4.4), verified and synthesised into speed-independent (SI) (quasi-delay insensitive (QDI)) circuit, using the STG's established tools. Because (X)BM specifications can be translated into a BA by generalising some of its well-formed requirements (Section 6.1.3), these translation methods also directly enables composition, verification, and synthesis of SI (QDI) circuits for (X)BM specifications.

Moreover, the translation of the XBM specification's components to their STG counterparts (Section 4.3.4) allows the interpretation of (X)BM-like models via BAs.

Implementation of the automated design flow as a Workcraft plugin: In Chapter 5, we show the implementation Workcraft plugin that is based on our automated design flow of BA's 'co-design' route (Section 5.1), where it enables design automation support for BAs and (X)BM specifications featuring graphical-based designs (Sections 5.2.1 and 5.2.2), simulation (Section 5.2.3), (X)BM-based verification (Section 5.2.4), translation to STG (Section 5.2.5), and STG-based verification and circuit synthesis using Petrify and Mpsat backends (Section 5.2.6).

The results of the Workcraft plugin are then shown in Section 5.3, where all three translation methods from BAs to STGs scaled well (Section 5.3.1) and their produced circuit implementations achieved smaller literal counts (Section 5.3.2).

7.2 Future research and Development

The contributions of this thesis have shown how our proposed BA model helps establish a new 'co-design' route between the 'legacy' route of designing BM specifications and the 'disruptive' route of designing STGs, where several benefits can be particularly found for both 'legacy' circuit designers and 'disruptive' circuit designers. However, there are still improvements that can be made and considered for the future research and development of this BA model and its design flow.

Firstly, there are still optimisations that can be done to the presented BA model, where better circuit results can be produced e.g. reducing the impact of interleaving, reducing the concurrency of output bursts, and optimising the ordering of outputs.

The presented translation methods, in particular the two exponential-sized translations that preserves weak bisimulation and strong bisimulation, in Section 4.3 can also be improved due to the creation of redundant places. For example, recent work in [35, 36] shows how these exponentially-sized translations can be significantly improved to polynomial-sized translations by replacing the Cartesian product construction with a new construction, based on edge clique covers of graphs.

New features can also be added to the presented WORKCRAFT plugin in Section 5, such as a method that groups up 'burstable' signals (e.g. interleaving arcs containing only inputs/outputs) in composed BAs when signal order is not important, and a method that decouples/recouples bursts into optimised subsets of signals.

Finally, the presented distributed methodology for composing BAs in Section 4.4 can be extended to include the modelling of distributed environments, where it can be used to investigate the possibility of allowing multiple timing assumptions to coexist in a circuit implementation, e.g. one part operating in input-output mode and another part operating in the 'burst-mode' timing assumption.

Bibliography

[1] Design of vme bus controller. [Online]. Available: https://workcraft.org/tutorial/design/vme_bus/start, 2006. Accessed: 28-12-2022.

[2] Workcraft. [Online]. Available: `https://workcraft.org/`, 2006. Accessed: 28-12-2022.

[3] CaSCADE tools. [Online]. Available: `http://www.cs.columbia.edu/~nowick/asynctools/`, 2007. Accessed: 15-07-2022.

[4] Verification properties. [Online]. Available: `https://workcraft.org/help/verification`, 2014. Accessed: 20-05-2023.

[5] Modelling with finite state machines: Vending machine. [Online]. Available: https://workcraft.org/tutorial/model/vending_machine/start, 2015. Accessed: 28-12-2022.

[6] Petri net synthesis: Concurrent vending machine. [Online]. Available: https://workcraft.org/tutorial/method/petri_synthesis/start, 2015. Accessed: 28-12-2022.

[7] M. Agyekum and S. Nowick. A cycle-based decomposition method for burst-mode asynchronous controllers. In *Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 129–142, 2007.

[8] M. Akrarai, N. Margotat, G. Sicard, and L. Fesquet. An asynchronous hybrid pixel image sensor. In *Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 55–61, 2021.

[9] A. Alekseyev, V. Khomenko, A. Mokhov, D. Wist, and A. Yakovlev. Improved parallel composition of labelled petri nets. In *Int. Conf. Application of Concurrency to System Design (ACSD)*, pages 131–140, 2011.

[10] S. Ataei and R. Manohar. Amc: An asynchronous memory compiler. In *Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 1–8, 2019.

[11] J. Beaumont, A. Mokhov, D. Sokolov, and A. Yakovlev. High-level asynchronous concepts at the interface between analog and digital worlds. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 37(1):61–74, 2018.

[12] J. Beister, G. Eckstein, and R. Wollowski. From STG to Extended-Burst-Mode Machines. In *Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 145–158, 1999.

[13] C. Van Berkel, M. Josephs, and S. Nowick. Applications of Asynchronous Circuits. *Proc. IEEE*, 87(2):223–233, 1999.

[14] R. Brayton, A. Sangiovanni-Vincentelli, C. McMullen, and G. Hachtel. Logic minimization algorithms for vlsi synthesis. In *Kluwer Academic Publishers*, 1984.

[15] A. Chan, D. Sokolov, V. Khomenko, D. Lloyd, and A. Yakovlev. Synthesis of SI Circuits from Burst-Mode Specifications. In *Design, Automation and Test in Europe Conf. (DATE)*, pages 366–369, 2021.

[16] A. Chan, D. Sokolov, V. Khomenko, D. Lloyd, and A. Yakovlev. Burst Automaton: Framework for Speed-Independent Synthesis Using Burst-Mode Specifications. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 42(5):1560–1573, 2023.

[17] A. Chan, D. Sokolov, V. Khomenko, and A. Yakovlev. Formal Modelling of Burst-Mode Specifications in a Distributed Environment. In *Forum on Specification & Design Languages (FDL)*, pages 1–8, 2022.

[18] T. Chu. Synthesis of Self-Timed VLSI Circuits From Graph-Theoretic Specifications. Technical report, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1987.

[19] W. A. Clark. Macromodular computer systems. In *Proc. Spring Joint Computer Conference*, AFIPS, page 335–336, New York, NY, USA, 1967. Association for Computing Machinery.

[20] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Trans. Information and Systems*, E80-D:315–325, 1997.

[21] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis for Asynchronous Controllers and Interfaces*. Springer Publishing Company, Incorporated, 2002.

[22] A. Davis, B. Coates, and K. Stevens. The post office experience: Designing a large asynchronous chip. In *Proc. of the Twenty-sixth Hawaii Int. Conf. on System Sciences*, volume i, pages 409–418 vol.1, 1993.

[23] J. De San Pedro, T. Bourgeat, and J. Cortadella. Specification mining for asynchronous controllers. In *Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 107–114, 2016.

[24] H. Delsoto, D. Oliveira, G. Batista, D. Silva, and L. Romano. A Tools Flow for Synthesis of Asynchronous Control Circuits from Extended STG Specifications. In *Latin American Symp. on Circuits Systems (LASCAS)*, pages 225–228, 2019.

[25] D. Domingos and M. Santos. Asynchronous controller design and simulation using signal transition graphs. In *Conference on Design of Circuits and Integrated Circuits (DCIS)*, pages 01–06, 2022.

[26] J. C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, dec 1991.

[27] R. Fuhrer, B. Lin, and S. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *Proc. IEEE Int. Conf. on Computer Aided Design (ICCAD)*, pages 604–611, 1995.

[28] A. Gill et al. Introduction to the theory of finite-state machines. 1962.

[29] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, pages 299–309, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[30] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, 1954.

[31] H. Jacobson, C. Myers, and G. Gopalakrishman. Achieving fast and exact hazard-free logic minimization of extended burst-mode gc finite state machines. In *Int. Conf. on Computer Aided Design (ICCAD)*, pages 303–310, 2000.

[32] V. Khomenko. Unfolding tools. [Online]. Available: http://homepages.cs.ncl.ac.uk/victor.khomenko/home.formal/tools/UnfoldingTools/current/, 2006. Accessed: 28-12-2022.

[33] V. Khomenko. Efficient automatic resolution of encoding conflicts using stg unfoldings. In *Int. Conf. Application of Concurrency to System Design (ACSD)*, pages 137–146, 2007.

[34] V. Khomenko, M. Koutny, and A. Yakovlev. Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT. In *Int. Conf. Application of Concurrency to System Design (ACSD)*, pages 16–25, 2004.

[35] V. Khomenko, M. Koutny, and A. Yakovlev. Avoiding exponential explosion in petri net models of control flows. In L. Bernardinello and L. Petrucci, editors, *Application and Theory of Petri Nets and Concurrency*, pages 261–277, Cham, 2022. Springer International Publishing.

[36] V. Khomenko, M. Koutny, and A. Yakovlev. Slimming down petri boxes: Compact petri net models of control flows. In B. Klin, S. Lasota, and A. Muscholl, editors, *Proc. of CONCUR*. Leibniz International Proceedings in Informatics (LIPIcs), 2022.

[37] V. Khomenko, M. Schaefer, and W. Vogler. Output-Determinacy and Asynchronous Circuit Synthesis. *Fundamenta Informaticae*, 88(4):541–579, 2008. Special Issue on Best Papers from ACSD'07.

[38] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Design Automation Conference (DAC)*, page 67–70. Association for Computing Machinery, 1996.

[39] L. Lavagno and A. L. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, USA, 1993.

[40] G. Mao, A. Yakovlev, F. Xia, T. Lan, S. Yu, and R. Shafik. Automated synthesis of asynchronous tsetlin machines on fpga. In *IEEE Int. Conf. Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2022.

[41] A. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 2005.

[42] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proc. MIT Conf. on Advanced Research in VLSI*, AUSCRYPT, page 263–278, Cambridge, MA, USA, 1990. MIT Press.

[43] P. Mattheakis and C. Sotiriou. Polynomial Complexity Asynchronous Control Circuit Synthesis of Concurrent Specifications Based on Burst-Mode FSM Decomposition. In *Int. Conf. on VLSI Design (VLSID)*, pages 251–256, 2013.

[44] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[45] F. Mendes, T. Curtinhas, D. Oliveira, H. Delsoto, and L. Faria. A Novel Tool for Synthesis by Direct Mapping of Asynchronous Circuits from Extended STG Specifications. In *Int. Conf. VLSI Design (VLSID)*, pages 451–452, 2018.

[46] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., USA, 1989.

[47] A. Mokhov, M. Rykunov, D. Sokolov, and A. Yakovlev. Towards reconfigurable processors for power-proportional computing. In *IEEE Faible Tension Faible Consommation (FTFC)*, pages 1–4, 2013.

[48] E. F. Moore. . gedanken-experiments on sequential machines. automata studies, edited by c. e. shannon and j. mccarthy, annals of mathematics studies no. 34, litho-printed, princeton university press, princeton1956, pp. 129–153. *Journal of Symbolic Logic*, 23(1):60–60, 1958.

[49] A. Moreno and J. Cortadella. State-based encoding of large asynchronous controllers. *IEEE Access*, 6:61503–61518, 2018.

[50] A. Moreno, D. Sokolov, and J. Cortadella. Synthesis from Waveform Transition Graphs. In *Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 60–67, 2019.

[51] D. Muller and W. Bartky. A Theory of Asynchronous Circuits. In *Int. Symp. Theory of Switching*, pages 204–243, 1959.

[52] C. J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, Inc., 2001.

[53] C. J. Myers and T. H. Y. Meng. Synthesis of timed asynchronous circuits. *Proc. IEEE Int. Conf. Computer Design: VLSI in Computers & Processors*, pages 279–284, 1992.

[54] S. Nowick. Minimalist homepage. [Online]. Available: `http://www.cs.columbia.edu/~nowick/minimalist/minimalist.html`, 1999. Accessed: 16-12-2021.

[55] S. Nowick and D. Dill. Synthesis of Asynchronous State Machines Using a Local Clock. In *Int. Conf. Computer Design (ICCD)*, pages 192–197, 1991.

[56] S.M. Nowick, N.K. Jha, and F.-C. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1514–1521, 1997.

[57] C. A. Petri. *Kommunikation mit automaten (Communicating with automata).* PhD thesis, 1962.

[58] L. Rosenblum and A. Yakovlev. Signal Graphs: From Self-Timed to Timed Ones. In *Int. Workshop on Timed Petri Nets*, pages 199–206, 1985.

[59] C. L Seitz. Ideas about arbiters. *Lambda*, 1(1):10–14, 1980.

[60] C. L Seitz and C Mead. System timing. *Introduction to VLSI systems, Chapter 7*, pages 218–262, 1980.

[61] D. Sokolov, V. Khomenko, A. Mokhov, V. Dubikhin, D. Lloyd, and A. Yakovlev. Automating the Design of Asynchronous Logic Control for AMS Electronics. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 39(5):952–965, 2020.

[62] Sontrak. Logic friday. [Offline]. Archive: `https://web.archive.org/web/20131216141103/http://sontrak.com/index.html`, 2007. Archive Accessed: 20-07-2022.

[63] J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design.* Springer, 2002.

[64] M. Theobald and S.M. Nowick. An implicit method for hazard-free two-level logic minimization. In *Proc. Symp. Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 58–69, 1998.

[65] M. Theobald, S.M. Nowick, and Tao Wu. Espresso-hf: a heuristic hazard-free minimizer for two-level logic. In *Design Automation Conference (DAC)*, pages 71–76, 1996.

[66] J. T. Udding. *Classification and composition of delay-insensitive circuits.* PhD thesis, Mathematics and Computer Science, 1984.

[67] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, pages 429–528, 1996.

[68] K. Yun and D. Dill. Automatic Synthesis of Extended Burst-Mode Circuits. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 18(2):101–132, 1999.

[69] K. Y. Yun and D. L. Dill. A High-Performance Asynchronous SCSI Controller. In *Int. Conf. Computer Design (ICCD)*, pages 44–49, 1995.

Appendix A

Composed BA Model

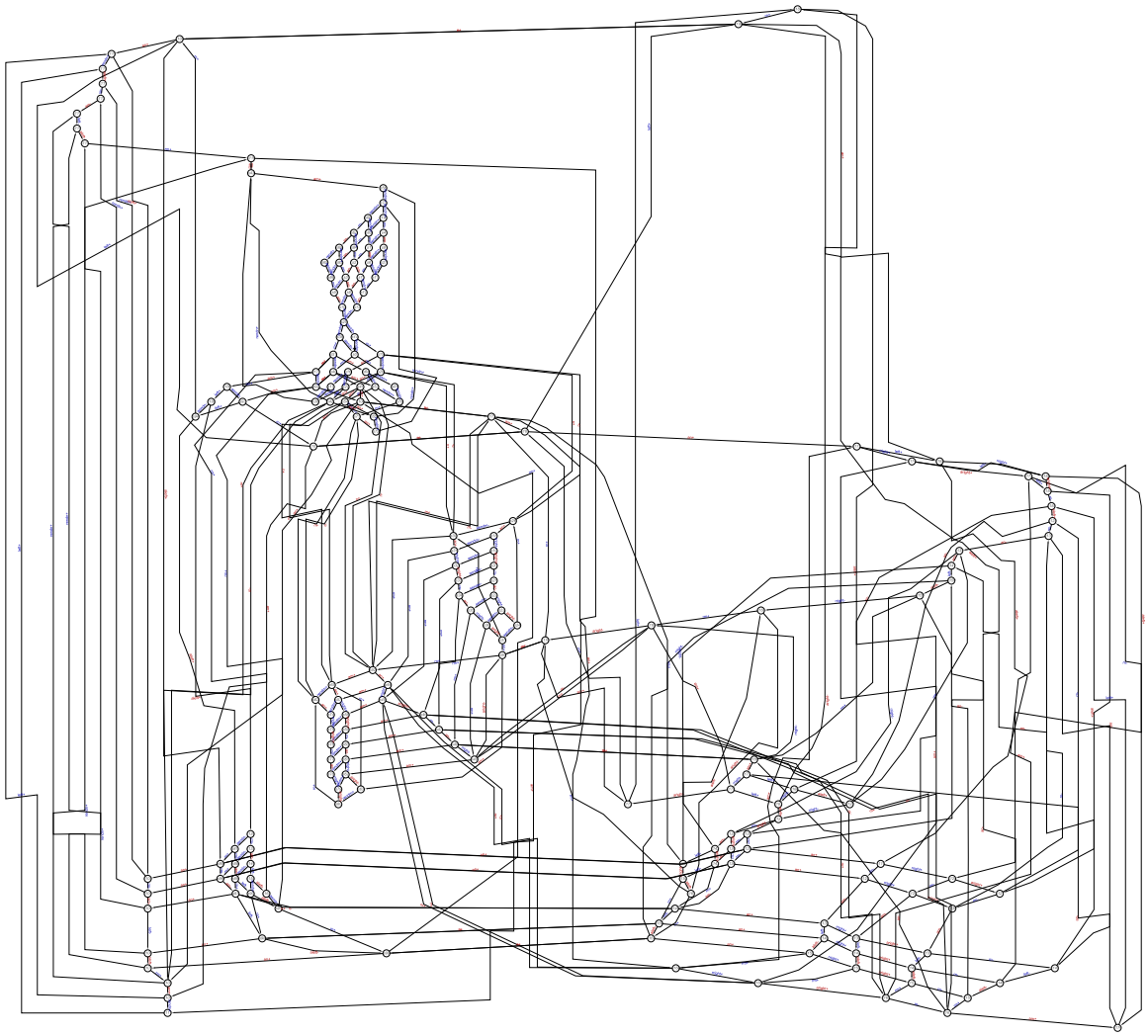


Figure A.1: Composed BA of the Handshake Decoupler's Controller in Chapter 3 interpreted from STG reachability graph in Figure 3.11

Index