Automated translation of asynchronous concepts to Signal Transition Graphs

Jonathan Beaumont

j.r.beaumont@ncl.ac.uk

NCL-EEE-MICRO-MEMO-2016-013

December 2016

Abstract

Asynchronous circuits are becoming increasingly important in system design for Internet-of-Things, where they orchestrate the interface between big synchronous computation components and the analogue environment, which is inherently asynchronous and has high uncertainty with respect to power supply, temperature and long-term ageing effects. However, wide adoption of asynchronous circuits by industrial users is hindered by a steep learning curve for asynchronous control models, such as Signal Transition Graphs, that are developed by the academic community for specification, verification and synthesis of asynchronous circuits.

Previously, we have introduced a novel high-level description language for asynchronous circuits, which is based on behavioural *concepts* – high-level descriptions of asynchronous circuit requirements, that can be shared, reused and extended by users. In this paper we will discuss more examples using concepts, and an algorithm to automatically translate these to Signal Transition Graphs for further processing by conventional asynchronous and synchronous EDA tools, such as PETRIFY and MPSAT. Our aim is to simplify the process of capturing system requirements in the form of a formal specification, and to promote behavioural concepts as a means for design reuse. The proposed design flow is fully automated in open-source toolsuite WORKCRAFT.

1 Introduction

Concepts have been presented in order to provide a more compact, adaptive and intuitive method of designing asynchronous circuits, using a fully compositional text based description method. This was born out of the use of algebra for designing systems, such as the model Conditional Partial Order Graphs [1][2][3], the algebra of switching networks [4]. Composition is an important part of concepts and the use of composition in some algebraic representations, such as with DI algebra [5] and Conditional Signal Graphs [6] helped to inspire this.

As discussed in [7], concepts are a useful language for specifying the behaviours of asynchronous circuits, in the preferred form of the user. This can be as low-level signal-level concepts, or higher-level gate- or protocol-level concepts. It also allows the definition of their own concepts, which can be reused within the same or any other specification that they wish to increase the speed of designing a system, and future systems.

With concepts, we aim to solve the problems that can arise from the more commonly used monolithic approach, where a user must design each system in the form of an STG from a blank page. The scalability of this is poor: as the system grows in complexity its monolithic specification becomes challenging to comprehend and debug. The problem becomes particularly severe when designing multi-mode systems, such as power regulators, where capturing all aspects of system behaviour in a consistent specification is a major design challenge [8][9]. Moreover, the STG models of components and operating modes are difficult to reuse when designing other specifications, and thus each new design must be built from the ground up. This is particularly undesirable for industry, as this increases the design time of each design greatly.

STGs [10][11] are commonly used for the specification, verification and synthesis of asynchronous control circuits as they are supported by multiple EDA tools, such as PETRIFY [12], MPSAT [13], VERSIFY [14], WORKCRAFT [15][16], and others. These tools take an STG specification of a complete controller and can formally verify its correctness, as well as synthesise an asynchronous circuit implementation that is *speed-independent*, i.e. guaranteed to work correctly regardless of component delays [17].

Concepts are not supported by these tools, and rather than reinvent the wheel by authoring tools to verify and synthesize concepts directly, we can *translate* concepts to STGs, for use with these existing tools. Previously, we have introduced a prototype algorithm for translating concepts to STGs. However, this version would not allow for some important features of circuits to be translated.

In this paper we display all concept types necessary for a complete specification, and some example concept specifications. These will display the possibilities of concepts, as well as the problems with the prototype translation algorithm, namely *OR-causality*, which is much more complex than AND-causality which is standard within concepts. This will lead us to a new translation algorithm, which has been implemented as a tool.

Our contributions are as follows:

- We detail the concepts required for a successful translation in Section 3
- We introduce some new gate examples and their specifications in Section 4, presenting what can be acheived with concepts, and how OR-causality becomes necessary in specifications.
- We present the implemented algorithm for translating concepts to STGs including OR-causality in Section 5.

We start with a brief recap of asynchronous concepts as a specification language in Section 2, and discuss the interoperability of the provided tool with existing standard STG based tools in Section 5.2.

2 Asynchronous Concepts

The abstract base of the concepts, on which these asynchronous specific circuits is discussed in [7].

Signal-level concepts: Asynchronous circuit specifications are mainly composed of signal transitions, and interactions between these, to show causal relationships. Signal transitions are denoted as a^+ and a^-, where a is any signal name, at least one character, and the $+$ or $-$ indicates which way the this signal transitions, $+$ denoting a low-to-high or 0 to 1 transition, and $-$ denoting a high-to-low, 1 to 0 transition.

Signal-level concepts are the base level of concepts, and are the type all other concepts are built on. Here we display the standard concepts available at this level.

A key concept in asynchronous circuits is *causality*: one signal transition *causes* another signal transition, a cause and an effect. This is denoted in the form:

$$a^+ \rightsquigarrow c^+$$

This is read as a^+ causes c^+, meaning that for the c^+ transition to occur, a^+ must have occurred previously. The $\rightsquigarrow$ operator is used to show causal relationships between signals.

While this concept is called *causality*, this doesn't necessarily imply timings, such as any *cause* transition immediately forcing the *effect* transition it applies to. Causality can be used in order to list all possible *cause* transitions which need to occur in order for an *effect* transition.

One can compose any concepts using the $\diamond$ (diamond) operator, and this applies to concepts of any level, whether predefined or user-defined. For example, two causality concepts can be composed.

$$a^+ \rightsquigarrow c^+ \ \diamond\ b^+ \rightsquigarrow c^+$$

In words, a^+ *and* b^+ must occur before c^+ can occur. This corresponds to so-called AND-causality in the fact that several cause transitions *must all* have occurred before an event can occur. AND-causality is commonly used to imply behaviours in circuits, for specific requirements of effect transitions.

The above notation can cause long-winded specifications when lots of AND-causality is involved. To solve this we provise a listing option. The following will acheive the same results:

$$[a^+, b^+] \mathrel{\sim\!\&\!\!>} c^+$$

This form of concept can be composed as usual with any other concepts.

A less common, but still useful form of causality is OR-causality. This is where an effect transition can have several possible cause transitions. Only one cause transition is required to occur to allow the effect transition to occur.

With OR-causality, the notation used lists all possible causes for the stated effect:

$$[x^+, y^+] \mathrel{\sim\!|\!\!>} z^+$$

This is, *either* x^+ *or* y^+ must occur in order for z^+ to occur.

The interactions when an effect transition is included in both AND- and OR-causality are interesting, and an example of when this occurs can be found in Section 5, along with a description of the implemented algorithm, which discusses the issues OR-causality poses with translation.

Gate-level concepts: Using the causality concept we can express the behaviour of gates in asynchronous circuits. For example, a *buffer* is a gate with one input signal and one output signal , whose output transitions causally depend on the input ones:

$$\mathsf{buffer}(a,b) = a^+ \rightsquigarrow b^+ \ \diamond\ a^- \rightsquigarrow b^-$$

An *inverter* has a similar conceptual specification, but the output transition is inverted:

$$\mathsf{inverter}(a,b) = a^+ \rightsquigarrow b^- \ \diamond\ a^- \rightsquigarrow b^+$$

A *C-element* is a gate with two inputs, in this example a and b and one output c, which synchronises both rising and falling input transitions via AND-causality:

$$\mathsf{cElement}(a,b,c) = a^+ \rightsquigarrow c^+ \diamond b^+ \rightsquigarrow c^+ \diamond a^- \rightsquigarrow c^- \diamond b^- \rightsquigarrow c^-$$

An alternative way to express the same concept is to reuse the buffer concept:

$$\mathsf{cElement}(a,b,c) = \mathsf{buffer}(a,c) \diamond \mathsf{buffer}(b,c)$$

A C-element combines the constraints imposed on the output transitions by two 'virtual' buffers. An expanded example of a C-element can be found in [7].

Behaviour of other gates can be similarly defined using concepts. We will dicuss this in Section 4.

Protocol-level concepts: In addition to gate-level concepts described above it is often important to specify *protocols* of interaction between multiple gates, components or signals.

Here we demonstrate how one can use concepts to specify asynchronous handshakes and mutual exclusion mechanisms.

Given two signals a and b, a *handshake* between them is the following composition of causality concepts:

$$\mathsf{handshake}(a,b) = a^+ \rightsquigarrow b^+ \diamond b^+ \rightsquigarrow a^- \diamond a^- \rightsquigarrow b^- \diamond b^- \rightsquigarrow a^+$$

Intuitively, we have a two-way asynchronous communication channel, where one party sends transitions a^+ and a^- and the other party responds by corresponding b^+ and b^- transitions. Note that the four causality concepts match those found in the buffer and inverter concepts, which leads to an alternative way to express a handshake between a and b:

$$\mathsf{handshake}(a,b) = \mathsf{buffer}(a,b) \diamond \mathsf{inverter}(b,a)$$

This conceptual understanding of a handshake as being composed from a buffer and an inverter is often used by circuit designers as a convenient way of reasoning.

The last standard concept is *mutual exclusion* between two signals a and b:

$$\mathsf{me}(a,b) = a^- \rightsquigarrow b^+ \diamond b^- \rightsquigarrow a^+$$

The concept states that, in terms of causality, rising transitions a^+ and b^+ can only occur after the opposite falling signal transitions. Therefore, even if all other cause-transitions for either a^+ or b^+ have occurred, while the one signal is high, this blocks the other from transitioning high. This guarantees that a and b are never set to 1 at the same time, i.e. they are mutually exclusive.

We can now specify a *mutual exclusion element* [18] that receives asynchronous requests r_1 and r_2 to a shared resource and grants access to it by corresponding mutually exclusive signals g_1 and g_2:

$$\mathsf{meElement}(r_1,r_2,g_1,g_2) = \mathsf{buffer}(r_1,g_1) \diamond \mathsf{buffer}(r_2,g_2) \diamond \mathsf{me}(g_1,g_2)$$

3 Concepts for translation

The concepts we have discussed so far are aimed at specifying the behaviour of signals in a circuit. When translating these to STGs however, these behaviours do not necessarily result in a correct specification, meaning they will not be verifiable by standard tools, and therefore not useful in further operations, such as synthesis.

This is due to various parts of an STG that we have not yet discussed, which are important for specification, namely *interface* and *initial states*.

3.1 Interface concepts

An important part of a specification is how these signals interact with the outside world, which could be another scenario or another circuit, for example. These signals can be inputs from the outside world, outputs or an internal signal, which is used only within this scenario.

To specify the type of a signal (*input*, *output* or *internal*) we introduce the interface concept:

$$\mathsf{interface}: A \to \{\mathsf{Input}, \mathsf{Output}, \mathsf{Internal}\}$$

Signal types are composed according to the following rules:

⋄	Input	Output	Internal
Input	Input	Output	Internal
Output	Output	Output	Internal
Internal	Internal	Internal	Internal

The intuition is as follows:

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

Specifying signal types is important when designing asynchronous circuits, as it helps to quickly identify errors (e.g. an input transition is caused by a hidden internal transition), and reuse existing tools for circuit simulation, verification and synthesis. Signal type information is also used in the algorithm for automated translation of concepts to STGs (Section 5).

Concepts inputs, outputs, internals are defined for specifying types of sets of signals for convenience, and to be included inline with other concepts. For example, to specify that signals a and b are inputs, c is an output, and t is internal, it is possible to write:

$$\mathsf{inputs}(\{a,b\}) \diamond \mathsf{outputs}(\{c\}) \diamond \mathsf{internals}(\{t\}).$$

3.2 Initial state concepts

Specifying the initial state is important, as it determines what the first transitions of a scenario will be. Without these, no transition can occur.

Each signal must have it's initial state declared before translation can occur. In order to specify the initial state of a handshake between signals a and b, we use the initialise concept:

$$\mathsf{initialise}(signal, value) = \mathsf{after}(signal, value)$$

The possible initial states are *high* or *low*, referred to as *0* or *1* respectively:

$$\mathsf{initialise}(a, 0) \diamond \mathsf{initialise}(x, 1)$$

A signal can only be declared as initially high or low. If the initial state of a signal is not defined, and error will occur, and the translation will not continue. Conversely, if the event a signal has it's initial state declared as both high and low, and is thus inconsistent, in a specification then the translation will also fail.

For the ease of use, and to speed up the process, initial states can also be declared in lists by on state:

$$\mathsf{initialise0}[a,b,c] \diamond \mathsf{initialise1}[x,y]$$

As an example, we can include this in the specification of a handshake, to create a handshake with built-in initial state. $\mathsf{initialise}(a,0)$ sets the state of the signal a to 0. We can compose an initial state concept with the handshake concept into a combined $\mathsf{handshake00}(a,b)$ concept as

$$\mathsf{handshake00} = \mathsf{handshake}(a,b) \diamond \mathsf{initialise}(a,0) \diamond \mathsf{initialise}(b,0)$$

The resulting concept corresponds to a handshake between signals a and b that are both initially 0.

4 Concept examples

In our previous paper, [7], we have used a C-element as a small example of how we can specify a standard gate with concepts, and how there are several ways of describing these with concepts.

In this section, we will discuss the concept specification of some different gates, a Set-Reset latch which uses AND-causality, and an OR-gate and an

AND-gate, both of which use both AND- and OR-causality, and this proves to be more difficult to translate, which we will use an example in Section .

4.1 Set-reset latch

A set-reset latch (S-R latch) is a particularly simple logic gate, used to store either a zero or one, depending on the inputs.

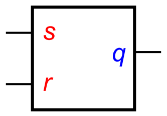

Figure 1: The symbol of an SR-latch

There are two input signals for this gate, s and r, and a single output, q:

- s: *set* - When this signal is high, it sets the output, meaning it sets it high.
- r: *reset* - When this signal is high, it resets the output, meaning it sets it low.
- q: *latched bit* - This stores the desired value, set depending on the values of the input signals.

From this information, we can start to specify the gate with concepts.

First of all, we know the types of these signals, so let us sepcify the interface concept.

$$\mathsf{interface} = \mathsf{inputs}[s,r] \diamond \mathsf{outputs}[q]$$

Now we move onto the interaction between signals. First of all, to set the output high. When s is high, this causes q to go high. However, only after s goes low can q go low. In concepts this is:

$$\mathsf{set} = s^+ \rightsquigarrow q^+ \diamond s^- \rightsquigarrow q^-$$

The interaction between the reset signal and the output will be opposite to this; r going high causes q to go low. Only after r goes low can q go low. The concept for this is:

$$\mathsf{reset} = r^+ \rightsquigarrow q^- \diamond r^- \rightsquigarrow q^+$$

The next concept to specify is the initial state. The description given mentions nothing to help with the initial state, but let's assume we would like the output to initially be 0. To then allow the S-R latch to react only to the inputs, we can set the initial states of the inputs to 0 too. This will mean that s is not trying to take the output high, and q is not blocking the output from going high.

$$\mathsf{initialState} = \mathsf{initialise0}[s,r,q]$$

Finally, we can compose all of these concepts, in order to translate them to an STG. This concept will be:

$$\mathsf{SRLatch} = \mathsf{set} \diamond \mathsf{reset} \diamond \mathsf{interface} \diamond \mathsf{initialState}.$$

It is possible reuse some previously defined concepts for the specification of an SRLatch, which may be preferred by users, as these describe their behaviour in terms of gates.

The *set* concept features the same signal-level concepts as a buffer, and thus can be redefined as a buffer of s and q. Similarly, the *reset* concept can be replaced by an inverter between r and q. Two gates cannot be connected in this way in an actual circuit, however it defines their behaviours in a different way, which may be understood better in some cases.

Since this now reduces the number of concepts, and thus the length of the specification, we can define an SR-latch as follows:

$$\begin{aligned}\mathsf{SRLatch} &= \mathsf{buffer}(s,q) \diamond \mathsf{inverter}(r,q) \diamond \mathsf{inputs}[s,r] \\ &\quad \diamond \mathsf{outputs}[q] \diamond \mathsf{initialise0}[s,r,q]\end{aligned}$$

It's isn't necessary to split a scenario specification up into several concepts. Depending on the size of the specification, it may be simpler and quicker to produce a single concept like this.

Either of these concept specifications can be translated to produce an STG. The translated STG can be found in Figure 2.

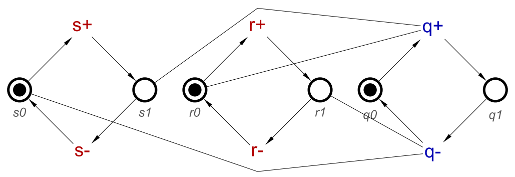

Figure 2: An SR-latch STG

4.2 OR-gate

The S-R latch features only AND-causality, and as we displayed in [7], with the examples of a C-element and a Buck controller, which only feature AND-causality, these can be translated to an STG.

OR-causality is quite different however, and causes some significant differences in a resulting STG, as we will see with the example of an OR-gate.

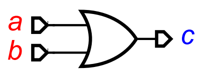

Figure 3: The circuit diagram of an OR-gate

Let's start by specifying the OR-gate. There are three signals, which for this example we will call *a* and *b* which are inputs, and *c* which is the only output:

- *a* - When either this signal or *b* is high, the output should be set high.
- *b* - When either this signal or *a* is high, the output should be set high.
- *c* - This output signal is high when at least one of the inputs is high, but low when *both* input signals are low.

As with the S-R latch, we can start by specifying the interface:

$$\mathsf{interface} = \mathsf{inputs}[a,b] \diamond \mathsf{outputs}[c]$$

Again, there is no information on the initial state. By the description, we can assume that setting all signals to be initially low will mean that the output will not be able to go high. Thus we can define the initial state as:

$$\mathsf{initialState} = \mathsf{initialise0}[a,b,c]$$

For the interaction between signals, let's start by specifying the concept which describes what causes the output to go low:

$$\mathsf{outputFall} = a^- \rightsquigarrow c^- \diamond b^- \rightsquigarrow c^-$$

In words, this means that both a^- and b^- must have occured before c^- can occur. This is AND-causality as we have used in previous examples.

For a slightly more compact specification, we can rewrite this concept as:

$$\mathsf{outputFall} = [a^-,b^-] \rightsquigarrow\!\&\!\!> c^-$$

The concept specifying the output going high is slightly different, because only one of *a* and *b* must be high in order for *c* to go high. This is OR-causality and specifying this in concepts is not too difficult:

$$\mathsf{outputRise} = [a^+,b^+] \rightsquigarrow\!|\!\!> c^+$$

This lists the possible cause transitions of c^+, namely a^+ and b^+.

This completes the specification, so let's compose these conepts for the full OR-gate specification:

$$\mathsf{orGate} = \mathsf{outputFall} \diamond \mathsf{outputRise} \diamond \mathsf{interface} \diamond \mathsf{initialState}$$

With the OR-gate it is not possible to use pre-defined concepts, but we do not need to specify the individual concepts for a similarly sized specification:

$$\begin{aligned}\mathsf{orGate}\, a\, b\, c \quad &= [a^-,b^-] \rightsquigarrow\!\&\!\!> c^- \diamond [a^+,b^+] \rightsquigarrow\!|\!\!> c^+ \\ &\diamond \mathsf{inputs}[a,b] \diamond \mathsf{outputs}[c] \diamond \mathsf{initialise0}[a,b,c]\end{aligned}$$

The difficulty with OR-causality comes with translation. The translated STG for an OR-gate can be found in Figure 4, and features one major difference.

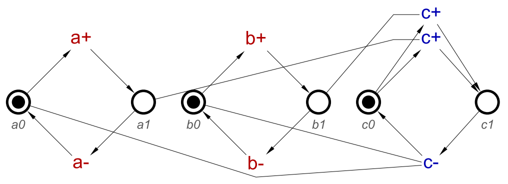

Figure 4: An OR-gate STG

This STG features 2 different c^+ transitions, due to the OR-causality. This is for each possible cause, and it is visible that one is connected via a read-arc to the $a1$ place and the other is connected via a read-arc to the $b1$ place. This allows c^+ to transition if either a^+, b^+ or both of these has occured.

Note that there is still only one c^- transition, as this requires both a^- and $b-$ to have transitioned.

4.3 AND-gate

An AND-gate, unlike what the name suggests also features OR-causality. For the output to go high, the input signals are in an AND-causality relationship with the output, however, only one of the inputs needs to transition low for the output to go low.

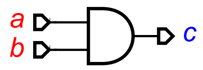

Figure 5: The circuit diagram of an AND-gate

This example also features 3 signals, *a* and *b* which are inputs, and *c* which is an output:

- *a* - When this signal and *b* are high, the output should be set high.
- *b* - When this signal and *a* are high, the output should be set high.
- *c* - This output signal is high when all of the inputs are high, but low when *at least* one of the input signals are low.

Again, we can start with the interface:

$$\mathsf{interface} = \mathsf{inputs}[a,b] \diamond \mathsf{outputs}[c]$$

With the initial state, we can again assume that setting all signals low initially will mean that no signals are causing any other signal transitions:

$$\mathsf{initialState} = \mathsf{initialise0}[a,b,c]$$

Now for signal interactions. For the output to go high, both input signals must transition high. This concept is:

$$\mathsf{outputRise} = a^+ \rightsquigarrow c^+ \diamond b^+ \rightsquigarrow c^+$$

This concept can also be written as:

$$\mathsf{outputRise} = [a^+,b^+] \overset{\&}{\rightsquigarrow} c^+$$

OR-causality is necessary for an AND-gate when specifying the output falling. Either a^-, b^- or both must occur before c^- can occur. In concepts, we list the possible causes for the effect:

$$\mathsf{outputFall} = [a^-,b^-] \overset{|}{\rightsquigarrow} c^-$$

We can now compose these for an AND-gate specification. As with the previous examples this can be in two forms.

$$\mathsf{andGate} = \mathsf{outputRise} \diamond \mathsf{outputFall} \diamond \mathsf{interface} \diamond \mathsf{initialState}$$

A compact version can be formed in this way:

$$\begin{aligned}\mathsf{andGate} &= [a^+,b^+] \overset{\&}{\rightsquigarrow} c^- \diamond [a^-,b^-] \overset{|}{\rightsquigarrow} c^- \\ &\diamond \mathsf{inputs}[a,b] \diamond \mathsf{outputs}[c] \diamond \mathsf{initialise0}[a,b,c]\end{aligned}$$

This can now be translated, and the resulting STG can be viewed in Figure 6

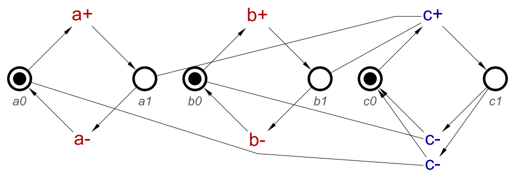

Figure 6: An AND-gate STG

This STG is similar to the OR-gate STG (Figure 4), however in this image, there are two c^- transitions, one of which connects to $a0$ via a read-arc, the other of which connects to $b0$. This way, either a^+ , b^+, or both of these will allow c^+ to occur. However, both *a* and *b* transitioning high is necessary for *c* to transition high.

We have shown the difference that OR-casuality can make to an STG in this section, but not how this affects the translation. In Section 5 we will discuss the implemented translation algorithm.

5 Concepts to STG translation algorithm

Concepts are a useful way of specifying asynchronous circuits, but specifications also need to be *verified* against certain properties to ensure their correctness, and once they are deemed to be correct, they need to be *synthesised* into efficient circuit implementations. Many software tools exist which automatically verify and synthesise STG specifications, such as PETRIFY [12] and MPSAT [13].

In order to reuse the tools developed by the community, it is necessary to be able to automatically translate concept specifications to STGs. In this section, we will display an example of how the translation algorithm operates, and present a pseudocode form of the algorithm in Algorithm 1.

5.1 A translation example

The example we will use for this is that of an OR-gate with control signal. This is similar to the OR-gate as seen in Section 4.2; inputs of a and b, and an output of c. There will however be an extra input signal, x.

This is the control signal, and interacts as follows: Only when x is high can the output c transition. This control signals allows a circuit to latch the output signal, by setting x high to let the output change, and when ready to use this signal, setting x low, so regardless of the inputs, it will remain in this state, and not change.

This is a rudimentary example, but is useful in displaying how the algorithm handles a system featuring both OR- and AND-causality. The concept for the interaction of the control signal is:

$$\mathsf{control} = x^+ \rightsquigarrow c^+ \diamond x^+ \rightsquigarrow c^-$$

We know this signal is an input, and let's assume we would like to latch the initially low output signal, and we can include this concept in the scenario for this translation:

$$\begin{aligned}\mathsf{orGateCtrl}\, a\, b\, c\, x \ = \ & \mathsf{orGate}\, a\, b\, c \diamond \mathsf{control} \diamond \mathsf{inputs}\,[a, b, x] \\ & \diamond \mathsf{outputs}\,[c] \diamond \mathsf{initialise0}\,[a, b, c, x]\end{aligned}$$

In this concept, we have reused the OR-gate concept, defined in Section 4.2. For reference, the fully translated STG can be found in Figure 7.

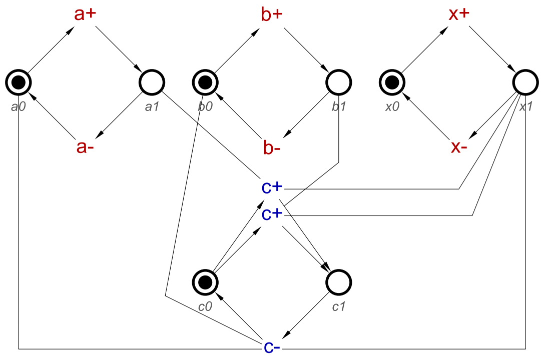

Figure 7: OR-gate with control signal STG

This STG still contains the two c^+ transitions, allowing this transition when either a^+ or b^+ has occured. However, with the inclusion of x, note that both of these transitions also require x^+ to have occured. This is the same for c^-. Through the addition of the *control* concept, we need to add AND-causality to each and every c^+ and c^- transition with x^+. The combination of signals required for each possible transition in both high and low transitions are calculated in the algorithm, and we will detail this process in this section.

As displayed in Section 4, there are multiple ways of representing a specification using concepts. However, all levels of abstraction available to the designer are built out of primitive low-level signal concepts. Given a specification, we can therefore break down all gate- and protocol-level constructs into 'atoms', which significantly simplifies the translation task.

For this example, the specification is broken down into these signal-level concepts:

$$\begin{aligned} \mathsf{orGateCtrl}\, a, b, c\, x \;=\; & [a^+, b^+] \mathrel{\rightsquigarrow\!\!\!\!\!\!\!+\!\!\!\gg} c^+ \diamond a^- \rightsquigarrow c^- \diamond b^- \rightsquigarrow c^- \\ & \diamond\ x^+ \rightsquigarrow c^+ \diamond x^+ \rightsquigarrow c^- \diamond \mathsf{inputs}\, [a, b, x] \\ & \diamond\ \mathsf{outputs}\, [c] \diamond \mathsf{initialise0}\, [a, b, c, x] \end{aligned}$$

The first step is to prepare the read-arcs, and this is done by reformating these concepts into lists of causes for effects. Each list will be of possible causes for each OR- and AND-causality signal-level concept in the specification, meaning that for each OR-causality concept, there will be 2 or more signals in the list, and for each AND-causality concept there will be a single item list.

For example, using the concept $x^+ \rightsquigarrow c^+$, for this c^+ transition, there is only one possible cause of it, x^+. Therefore, this forms a single item list, containing only x^+.

For another example, taking the concept $[a^+, b^+] \mathrel{\rightsquigarrow\!\!\!\!\!\!\!+\!\!\!\gg} c^+$, there are two possible transitions for c^+ here, a^+ and b^+. These therefore form a two item list.

This will mean there may be several different lists for each cause transition. Table 1 contains all of the lists described above.

Table 1: Lists of possible cause transitions for effect transitions

Causes	Effect
x^+	c^+
a^+, b^+	c^+
x^-	c^-
a^-	c^-
b^-	c^-

From this, we can then combine the lists by effect transition, to give us a list of lists. This can be viewed in Table 2

Table 2: Lists of lists of cause transitions for effect transitions

Causes	Effect
$[x^+], [a^+, b^+]$	c^+
$[a^-], [b^-], [x^-]$	c^-

Note how for the c^- transition there is 3 different single item lists. Each one is an AND-causality, and why they are each in a separate list will become apparent in the next step.

This step is applying the *Cartesian Product* to each the list of lists for each effect transition. This will "expand the brackets", combining each item of each list with every item of every other list, creating a single list which provides us with the number of this transition and whicharcs to connect for each transition of each type. For example, using the list of lists for c^+:

$$[a^+, b^+]\,[x^+] = [a^+x^+, b^+x^+]$$

In this example, we combine every item in the first list, $[a^+, b^+]$, with every item in the second list, which only contains x^+. This will therefore give us a new 2 item list.

This list provides us with both the number of c^+ transitions, 2, and which transitions are the causes of each of these transitions. The first c^+ transition will be caused by a^+ and x^+, the second will be caused by b^+ and x^+.

For c^-, the result of the cartesian product will be a single item, as combining each item in each list will combine all three of the items at once. This will be as follows:

$$[a^-], [b^-], [x^-] = [a^-b^-x^-]$$

There will therefore be only one c^- transition, requiring all three of a^-, b^- and x^- to have occured before c^- can occur.

Now, we can list the read-arcs to be connected for each transition of c. This can be viewed in Table 3.

Table 3: List of all effect transitions and their causes

Causes	Effect
a^+, x^+	$c^+(0)$
b^+, x^+	$c^+(1)$
a^-, b^-, x^-	c^-

The numbers of the c^+ transitions are necessary for reference when defininig arcs, for example, b^+ needs to connect to $c^+(1)$, not $c^+(0)$ or just c^+ as this can cause erroneous arcs to be included.

For clarity, where there is only one transition, such as with c^- we will refer to this simply as c^-, but this is effectively $c^-(0)$.

This concludes the part of the algorithm which defines the arcs and transitions. Next, the algorithm begins to build the STGs.

First of all, we need to add places for each signal. These are used to show whether a signal has transitioned high or low at any point. A token in a 0 place for example shows that that signal has transitioned low. We add a 0 and 1 place for each signal, which for this example gives us $a0$, $a1$, $b0$, $b1$, $x0$, $x1$ $c0$ and $c1$.

The interface is defined at the start of the algorithm, by listing them as $inputs$, $outputs$ or $internals$, so now we need to include all the transitions, and connect these to the places to form consistency loops, to ensure that all signals in this system can only transition in one way when this signals previous transition was in the opposite direction.

This is done by taking each transition, and if it is a $+$ transition, connecting the 0 place for this signal to the transition, then connecting this transition to the 1 place for this signal. Conversely, if it is a $-$ transition, we connect the 1 place for this signal to the transition, and connect this transition to the 0 place. Including these will form an STG as shown in Figure 8. Note that both c^+ transitions are included in these loops.

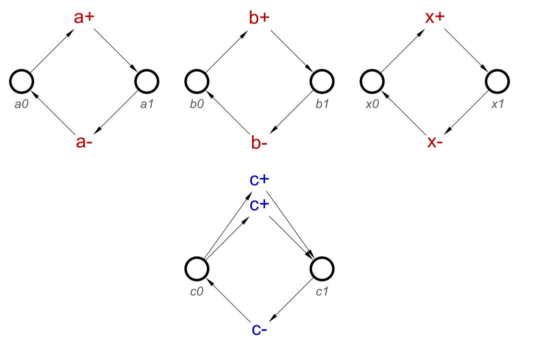

Figure 8: STG containing only consistency loops

Now the algorithm introduces the initial states. All of the signals are specified to be initially 0, meaning that the first transition will be the $+$. From the consistency loops, to allow this to happen, we need to place a token in each 0 place for each signal. This will produce the STG displayed in Figure 9.

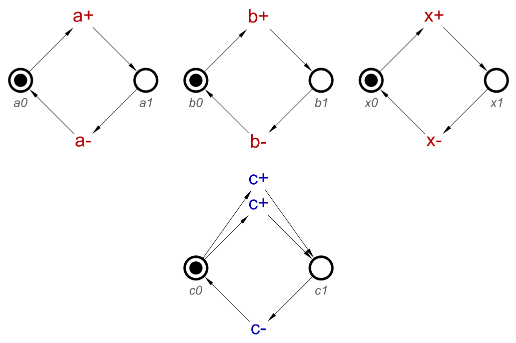

Figure 9: STG containing consistency loops and initial states

Finally, we can add the connections to the transitions, and cause the expected interaction between signals. We use read-arcs to connect the effect transitions to the places after the transition of the cause. For example, for $x^+ \rightsquigarrow c^+$, we will connect the $x1$ place to the transition of c^+. We use read-arcs for these, as for this example, only after c^+ has occured, and placed a token in $x1$, will c^+ be allowed to occur, but the read arc will not consume the token in $x1$ which would block x^- from being able to occur.

Following this, the specification is fully translated, and the resulting STG will be the same as shown in Figure 7.

Algorithm 1 Algorithm for translating concepts to STGs

```
for Signal $s$ in System do
    define interface of $s$ as Input/Output/Internal
end for
for Each effect transition $e$ do
    $allCauses$ ← Concatinate lists of possible causes for $e$
    $transitionList$ ← $Cartesian\ Product$ of $allCauses$
    for $i = 0$ to Length of $transitionList$ do
        add $e(i)$ to $transitions$
    end for
end for
for Each signal $s$ in system do
    add place $s$.Name0
    add place $s$.Name1
end for
for Each transition $t$ in $transitions$ do
    if transition is high then
        connect (place $t$.signalName0, transition)
        connect (transition, place $t$.signalName1)
        for Each cause transition $c$ for $t$ do
            read-arc (place $t$.signalName1, $c$)
        end for
    end if
    if transition is low then
        connect (place $t$.signalName1, transition)
        connect (transition, place $t$.signalName0)
        for Each cause transition $c$ for $t$ do
            read-arc (place $t$.signalName0, $c$)
        end for
    end if
end for
for Each initial state $state$ concept do
    if $state$ is low then
        add-token(signalName.place0)
    end if
    if $state$ is high then
        add-token(signalName.place1)
    end if
end for
```

5.2 Interoperability with STG based tools

The concepts tool, which translates asynchronous concepts to STGs has been integrated into open-source toolsuite WORKCRAFT [16]. This allows a designer to visualise their concept designs as STGs, to simulate, verify and synthesise them using other tools integrated in WORKCRAFT, such as PETRIFY [12] and MPSAT [13]. If there are any corrections or additions to be made these can be done either directly in the STG or in the original concept specification, which can then be re-translated into an updated STG. These automated processes can allow for a streamlined design process of asynchronous circuits.

The translation tool itself produces a *.g* document format, which is a graph format used to store STG information. This means that it is not necessary to view the STG after translation, and a concept specification can be used directly after translation with these tools.

6 Conclusions and future work

In this work we show that it is possible to design asynchronous control circuits at the interface between analogue and digital worlds by splitting their specification into operational modes, scenarios, and describing signal interactions and requirements of each scenario using high-level asynchronous concepts. These can then be translated into STGs that represent these operational modes, which can be used with existing verification and synthesis tools. STGs can be further combined to produce a complete model for the system specification.

In this work, we show some more possibilites available when desingning asynchronous control circuits using concepts. OR-causality can be a useful tool to include in a circuit, and providing a seemless method of translation of both this, and the more common AND-causality allows for concepts to be used in the design of a wider range of asynchronous circuits.

Using concepts, a user can reduce the time of designing an asynchronous control circuit from the ground up, as well as allow reuse of components either as part of a scenario or entire scenarios to reduce the design-time of future projects. Composition of concepts and scenarios can help reduce errors and save time in comparison to performing these manually. This method can help to make asynchronous circuits more appealing to industrial designers.

Currently, this method works with Signal Transition Graphs, however it can be applied to other modelling disciplines, such as Finite State Machines (FSM).

Process mining can also be used for various purposes in conjunction with designing asynchronous circuits. For example, process mining can discover a behavioural model when none currently exists, and can be used to check that an existing specification is realistic, or find less complex models. All of this can be performed automatically, by tools such as PG-MINER [19], given an event log with observations of a real analogue or digital system, and aid a designer in reducing design time and errors. We aim to test the possibility of producing concepts directly from the mining of these event logs.

The translations tool we have discussed is available from [20], and as stated, is integrated into WORKCRAFT [16]. A manual is included with the tool, which features descriptions of the features. We host a regularly updated blog which discusses some interesting properties of concepts, and new ideas we find for concepts, and this is available at [21].

References

[1] Andrey Mokhov and Alex Yakovlev. Conditional partial order graphs: Model, synthesis, and application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[2] Andrey Mokhov. *Conditional Partial Order Graphs*. PhD thesis, PhD thesis, Newcastle University, 2010.

[3] A. Mokhov and V. Khomenko. Algebra of Parameterised Graphs. *ACM Transactions on Embedded Computing*, 13(4s), 2014.

[4] Andrey Mokhov. Algebra of switching networks. *IET Computers & Digital Techniques*, 2015.

[5] M.B. Josephs and J.T. Udding. An overview of d-i algebra. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume i, pages 329–338 vol.1, Jan 1993.

[6] A. Mokhov, D. Sokolov, and A. Yakovlev. Adapting asynchronous circuits to operating conditions by logic parametrisation. In *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, pages 17–24, May 2012.

[7] J. Beaumont A. Mokhov D. Sokolov A. Yakovlev. Compositional design of asynchronous circuits from behavioural concepts. In *ACM-IEEE International Conference on Formal Methods and Models for System Design MEMOCODE15*, June 2015.

[8] D. Sokolov, A. Mokhov, A. Yakovlev, and D. Lloyd. Towards asynchronous power management. In *IEEE Faible Tension Faible Consommation (FTFC)*, pages 1–4, May 2014.

[9] Danil Sokolov, Victor Khomenko, Andrey Mokhov, Alex Yakovlev, and David Lloyd. Design and verification of speed-independent multiphase buck controller. In *Asynchronous Circuits and Systems (ASYNC), 2015 21st IEEE International Symposium on*, pages 29–36. IEEE, 2015.

[10] T.-A. Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[11] A. Yakovlev L. Rosenblum. Signal graphs: from self-timed to timed ones. *International Workshop on Timed Petri Nets*, pages 199–206.

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

[13] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting state encoding conflicts in stg unfoldings using sat. *Fundamenta Informaticae*, 62(2):221–241, 2004.

[14] Oriol Roig i Mansill. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Citeseer, 1997.

[15] I. Poliakov, D. Sokolov, and A. Mokhov. Workcraft: a static data flow structure editing, visualisation and analysis tool. In *Petri Nets and Other Models of Concurrency*, pages 505–514. 2007.

[16] Workcraft. www.workcraft.org.

[17] W. Bartky D. Muller. A theory of asynchronous circuits. *International Symposium of the Theory of Switching*, 1959.

[18] D. J. Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley and Sons, 2008. ISBN: 978-0-470-51082-7.

[19] Andrey Mokhov, Josep Carmona, and Jonathan Beaumont. Mining Conditional Partial Order Graphs from Event Logs. In *Transactions on Petri Nets and Other Models of Concurrency XI*, pages 114–136. Springer Berlin Heidelberg, 2016.

[20] Concepts repository. https://github.com/tuura/concepts, 2016.

[21] https://jrbeaumont.github.io/concepts blog/. Concepts-blog.