High-level asynchronous concepts at the interface between analogue and digital worlds

Jonathan Beaumont, Andrey Mokhov, Danil Sokolov, Alex Yakovlev
{j.r.beaumont, andrey.mokhov, danil.sokolov, alex.yakovlev}@ncl.ac.uk
School of Engineering, Newcastle University, UK

***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.**

In this article we introduce 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, and can be automatically translated 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, and is applied to the development of an asynchronous power regulator.

I. INTRODUCTION

Analogue and mixed-signal circuits become more tightly integrated with digital systems, in particular in mobile and autonomous applications, such as wearable consumer electronics and self-powered Internet-of-Things nodes, where it is essential to have intelligent timing control and power regulation [1][2][3]. An on-chip power management system is an illustrative example: it relies on analogue circuitry for power regulation and conversion, and its behaviour is characterised by many operating modes with complex interplay and high-level decision logic that is digitally controlled. Asynchronous circuits are event-driven, i.e. they react to changes in a system at the rate they occur [4]. This makes them particularly useful for interacting with analogue world, where the ability to quickly respond to non-digital input, e.g. dynamically changing loads across the chip, is essential for reliable operation and efficiency [5].

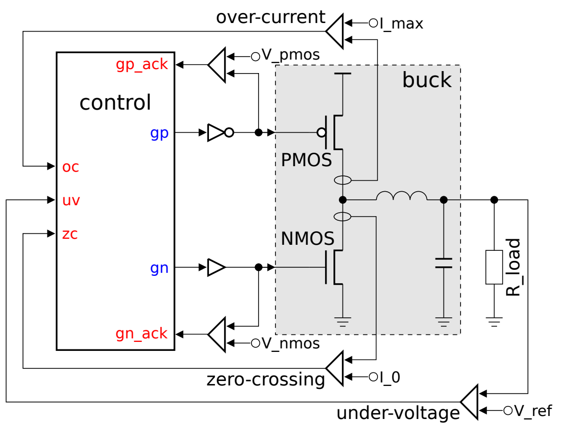
Signal Transition Graphs (STGs) [6][7] are commonly used for the specification, verification and synthesis of asynchronous control circuits as they are supported by multiple EDA tools, such as PETRIFY [8], PUNF/MPSAT [9][10], VERSIFY [11], WORKCRAFT [12][13], and others. These tools take an STG specification of a complete asynchronous 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 [14]. Such a monolithic approach to designing asynchronous circuits has poor scalability: as the system grows in complexity its monolithic specification becomes challenging to comprehend, debug and maintain. 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 [15][16]. The STG models of components and operating modes developed for one asynchronous system are difficult to reuse when designing others, and thus each new design must be built from the ground up. This further adds to the design time, hence making asynchronous circuits costly for use in industry.

In this paper we address this issue by proposing a new method for design of asynchronous circuits based on *behavioural concepts*. The method splits a specification into several parts corresponding to operational modes of the circuit (*scenarios*). The features, constraints and requirements of each scenario (*concepts*), are described in a formal notation, which we implemented as a domain specific language embedded in Haskell [17]. Concepts can be defined at several levels; specifically, we give examples of signal-, gate- and protocol-level concepts. It is possible to compose basic concepts into more complex ones, thus supporting the design reuse at the level of system specification. Concepts can be automatically translated into equivalent STGs and formally verified using standard tools, e.g. WORKCRAFT. When all scenarios have been translated to STGs and verified, they can be combined to produce a complete specification, and synthesised into an asynchronous circuit implementation targeting a chosen technology library.

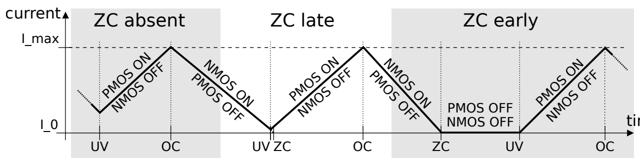
Our contributions are as follows:

- We introduce asynchronous concepts as a specification language in Section III and compare it to STGs in Section IV.
- We present an algorithm for translating concepts to STGs, which is implemented in the open-source tool PLATO [18] and integrated into the open-source asynchronous circuit design toolsuite WORKCRAFT [13], see Section V.
- We demonstrate the proposed design method on a case study of an asynchronous power regulator, Section VI.

We start with a motivational example in Section II, review related work in Section VII, and discuss future research in Section VIII.



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



(b) Informal description of three behavioural scenarios.

Fig. 1: Buck converter and its informal description.

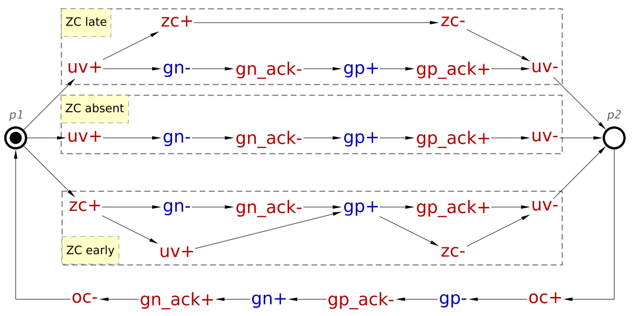


Fig. 2: STG specification of a simple buck converter.

II. Motivating Example: Buck Converter

On-chip power management is essential for energy efficiency and reliability of IoT computation nodes [1]. According to a 2016 report by Research & Markets, the global power management market will reach $34.86 billion by 2022, driven by the increasing demand in various applications, including traditional IoT sectors such as wearable, automotive, industrial, retail and building control electronics. Unlike conventional digital components, power management units need to directly interface with the analogue part of the system for sensing and controlling it, and can therefore strongly benefit from asynchronous implementation that allows them to react to changes in the system at the rate they occur, instead of sampling various analogue parameters by clock, which is slow and energy-wasteful [15][16]. As a compelling example, consider an asynchronous power management controller with 1ns response time to stimuli from the analogue world. To match its performance with a synchronous equivalent, one would need to clock it at 2-3GHz, as two clock cycles are required just for a conventional two flip-flop synchroniser [19]. Bearing in mind that power management controllers spend most of their time doing nothing and waiting for relatively slow changes in the analogue environment, such a high-speed clock solution would clearly be inadequate.

In this section we first provide the background on asynchronous buck converters Section II-A, our main motivation example and case study. Buck converters rely on analogue circuitry for power regulation and conversion and their behaviour is typically characterised by many operating modes with complex interplay and high-level decision logic that is digitally controlled. We then discuss challenges arising in the design of asynchronous buck converters with Signal Transition Graphs (STGs) [6][7], a commonly used mathematical model for the specification of asynchronous circuits (Section II-B). Finally, we outline our new design approach (Section II-C) wherein the behaviour of an asynchronous buck converter is decomposed into simple behaviours, that we later refer to as *concepts*. The approach is described at an intuitive level, and is formalised in Section III.

A. Background on buck converters

Our motivating example comes from the power management domain, a *multi-scenario power regulator* [15]. A basic power regulator comprises an analogue buck and a digital controller, as shown in Figure 1a. The controller operates the power regulating PMOS and NMOS transistors of the buck (using gp and gn outputs) as a reaction to *under-voltage* (UV), *over-current* (OC) and *zero-crossing* (ZC) conditions (uv, oc, and zc inputs, respectively). These conditions are detected by a set of sensors that compare the measured current and voltage with some reference values (V_ref, I_max, I_0). Note that in order to avoid a shortcircuit, the PMOS and NMOS transistors of the buck must never be On at the same time. Therefore, the controller is explicitly notified (by gp_ack and gn_ack) when the power transistor threshold levels (V_pmos and V_nmos) are crossed.

The operation of a power regulator is usually described in an intuitive, but rather informal way, e.g. by enumerating the possible sequences of detected conditions and describing the intended reaction to these events, as shown in Figure 1b. The diagram shows that UV should be handled by switching the NMOS transistor Off and PMOS transistor On, while OC should revert their state – PMOS Off and NMOS On (**ZC absent** scenario). Detection of the ZC after UV does not change this behaviour (**ZC late** scenario). However, if ZC is detected before UV then both the PMOS and NMOS transistors remain Off until the UV condition (**ZC early** scenario).

B. Monolithic STG-based design approach

Using the informal description of three behavioural scenarios, the designer can produce a formal specification, typically a monolithic STG describing the complete behaviour of the circuit. Figure 2 shows such a monolithic STG specification for the simple buck controller described above. Nodes of the STG correspond to signal rising ($+$) and falling ($-$) transitions, arcs model *causality*, parallel branches correspond to concurrency, and a circle *place* with a black dot (*token*) represents the choice of the current scenario. We formally introduce STGs in Section IV.

This STG is moderately large and captures all three scenarios by overlaying their common parts. In the monolithic design approach, the designer starts with a blank page and manually inserts signal transitions and the connections between them according to the informal description of the system's operation. With there being several scenarios, the designer could choose to design each separately, and then manually compose these taking advantage of the similarities between the scenarios, as described in [15][16].

In the event that a designer needs to add or remove signals or correct a fault, editing can become difficult, due to the complexity and size. This can lead to further faults, and several iterations of design and tests until the new feature is added and deemed to be working correctly.

A designer may even have to start from a blank page in some cases, unable to reuse any of the previous design. The larger and more complex an STG is, and the more signals there are, the more difficult it is to comprehend, debug and edit. This can slow the design process of a circuit, which is undesirable in an industry where the time for a device to move from conception to market is becoming critical, and ever shorter.

C. Towards high-level asynchronous concepts

In an attempt to streamline the design process, we aimed to find a way to create STGs which allows editing and reuse at various stages of the design. This way, if something needs to be corrected, this can be done to a small part, but we can reuse everything that works correctly. These parts can then be composed to produce a full STG which contains the corrections, with a minimum amount of time spent making the corrections.

This led us to take the example of a simple buck converter, and compare what the STG shows, and what the description of operation of the system, in particular its signals, is. This allowed us to view interactions between certain signals which may not be obvious, but without which the STG would not be a correct representation of the design. For example, Figure 3 shows one scenario of the simple buck converter with some points of interest highlighted.

Studying this STG shows that there are some high-level signal interactions that we can identify:

- High levels of uv and oc are mutually exclusive; indeed, uv goes high (transition uv^+) only after oc goes low (transition oc^-), and vice versa.
- High levels of gp and gn are also mutually exclusive.
- Signals gp and gp_ack form a handshake, i.e. there is a cycle $gp^+ \to gp\_ack^+ \to gp^- \to gp\_ack^-$ etc.
- Signals gn and gn_ack also form a handshake.

These interactions can also be identified in the informal buck description: uv and oc indicate opposite conditions in the circuit and will naturally be mutually exclusive, while gp and gn are used to switch PMOS and NMOS transistors respectively, and switching these transistors ON at the same time will cause a shortcircuit and is therefore prohibited. Signals gp_ack and gn_ack are used to acknowledge the state of these transistors, and thus always follow gp and gn, respectively, forming two separate handshakes.

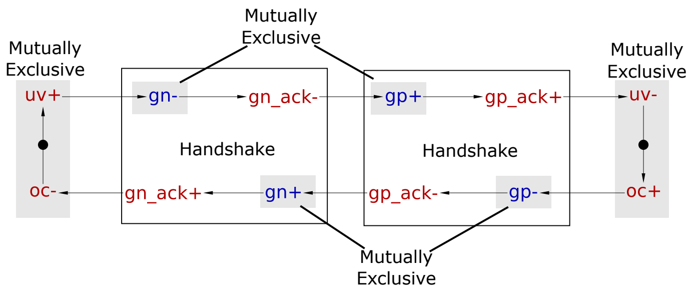


Fig. 3: Deconstructing the STG of the ZC absent scenario.

Knowing this information means that we can describe these protocols once, and include them in the design of any circuit involving these signals. Any other interactions between any of these signal will be subject to these protocols, which will prevent circuit breaking bugs in the testing phase. For example, an interaction involving gp will ensure that it is never set high at the same time as gn, as otherwise the mutual exclusion will not hold.

If one of these signals is removed, then the protocols that this affects can be removed, but the unaffected protocols can continue to be used, avoiding a major re-design which may happen when using the STG-based monolithic approach.

With this idea in mind, we need to find a way to describe these protocols, as well as other signal interactions, in order to ensure that each different interaction can be edited without needing to change every single interaction.

III. Concepts

In this section we formally introduce *concepts* that we propose to employ for the specification of asynchronous circuits. Below we list (fairly standard) definitions and notational conventions that are used throughout the paper.

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

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

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

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

A. Abstract concepts

We first describe *abstract concepts* that we use as building blocks for developing *domain specific concepts*, such as those related to asynchronous circuits (Section III-B).

Abstract concepts are parameterised by finite sets of *states* S and *events* E. The *initial state concept* captures all possible (or *permitted*) initial states of the system. In the most general form it is a function

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

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

The *event excitation concept* captures all states wherein a given event can occur (or is *excited*). In the most general form it is a function

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

that given an event $e \in E$ and a state $s \in S$ checks whether e is excited in s. In practice this concept is often realised using *interpreted graph models* such as Finite State Machines and Petri Nets [8], STGs, Conditional Partial Order Graphs [20][21], and others. A partial application of the excitation function is often useful: $\mathsf{excited}(e)$ captures all states where event e is excited; for example, if $\mathsf{excited}(e) = \mathbf{0}$ then e is never excited or *dead*. Event excitation concepts also form a commutative monoid with $e = \mathbf{1}$ and $\diamond = \wedge$. This definition corresponds to the *parallel composition* operation, a standard notion for many behavioural models [22].

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

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

that given a state $s \in S$ returns 1 if s is permitted by the invariant and 0 otherwise. Note that if for some state s the initial concept $\mathsf{initial}(s)$ holds but the invariant $\mathsf{invariant}(s)$ does not hold, then the specification is *contradictory* and cannot be satisfied by any implementation. We therefore usually assume that $\mathsf{initial}(s) \Rightarrow \mathsf{invariant}(s)$ holds for all $s \in S$. Similarly, invariant concepts form a commutative monoid with $e = \mathbf{1}$ and $\diamond = \wedge$. Intuitively, if a system comprises two subsystems then its states should be permitted in both of the subsystems.

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

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

captures all states $s \in S$ when a given event $e \in E$ cannot occur. Furthermore, one can define other useful concepts that cannot be derived from the above, e.g., the *execution concept* capturing the effects that different events have on the system state. Due to space limitations we only consider the three concepts defined above and their derivatives.

All the above concepts are monoids, hence their combinations are trivially monoids too. It is convenient to consider triples of concepts $(\mathsf{initial}, \mathsf{excited}, \mathsf{invariant})$ with $(\mathbf{1}, \mathbf{1}, \mathbf{1})$ representing the *empty specification*, and composition $(\mathsf{initial}_1, \mathsf{excited}_1, \mathsf{invariant}_1) \diamond (\mathsf{initial}_2, \mathsf{excited}_2, \mathsf{invariant}_2)$ defined as $(\mathsf{initial}_1 \diamond \mathsf{initial}_2, \mathsf{excited}_1 \diamond \mathsf{excited}_2, \mathsf{invariant}_1 \diamond \mathsf{invariant}_2)$. Note that composition of two non-contradictory specifications is always non-contradictory, that is if both $\mathsf{initial}_1(s) \Rightarrow \mathsf{invariant}_1(s)$ and $\mathsf{initial}_2(s) \Rightarrow \mathsf{invariant}_2(s)$ hold for all states $s \in S$, then $\mathsf{initial}_1(s) \diamond \mathsf{initial}_2(s) \Rightarrow \mathsf{invariant}_1(s) \diamond \mathsf{invariant}_2(s)$ holds too.

B. Concepts for asynchronous circuits

We now introduce concepts which are specific for the domain of asynchronous circuits and express them using the abstract concepts defined above.

Signal-level concepts: States and events of an asynchronous circuit are parameterised by a set of signals A. A state $s \in S$ is an assignment of Boolean values to signals, i.e. a function $s : A \to \mathbb{B}$, while an event $e \in E$ is a *signal transition*, i.e. a pair $e : A \times \mathbb{B}$ comprising a signal $a \in A$ and the value of the signal *after* the transition occurs. We call transitions $(a, 0)$ and $(a, 1)$ *falling* and *rising*, respectively, and denote them by a^- and a^+ for brevity.

The following two predicates are very useful for constructing concepts:

$$\begin{aligned} \mathsf{before} &: E \times S \to \mathbb{B} \\ \mathsf{after} &: E \times S \to \mathbb{B} \end{aligned}$$

A state $s \in S$ is said to be *before* a transition $(a, b) \in E$ if $s(a) \neq b$, i.e. in state s signal a has a value which is different from the resulting value of the transition. Similarly, s is *after* (a, b) if $s(a) = b$ (the transition has already occurred).

We are now ready to define an excitation concept called *consistency* [8]:

$$\mathsf{consistency} = \mathsf{before}$$

This concept captures the requirement that in a consistent asynchronous circuit a signal transition can only be excited in states that are before it.

Another key concept in asynchronous circuits is *causality*: we say that a transition *effect* $\in E$ causally depends on transition *cause* $\in E$, denoted as

$$\mathsf{causality}(\mathit{cause}, \mathit{effect}) : E \times E \to \mathbb{B}$$

if *effect* can occur only in states that are after *cause*. This is an excitation concept, which can be expressed as follows:

$$\mathsf{causality}(\mathit{cause}, \mathit{effect})(e) = \begin{cases} \mathbf{1} & \mathit{if}\ e \neq \mathit{effect} \\ \mathsf{after}(\mathit{cause}) & \mathit{otherwise} \end{cases}$$

In words, we do not add any constraints to events $e \in E$ that are distinct from *effect*, but *effect* is constrained to occur only after $cause$. Note that function after is used in the partially applied form. We will use a short-hand notation

$$cause \rightsquigarrow effect$$

for the causality concept for convenience. One can compose two causality concepts using the monoid composition.

$$a \rightsquigarrow c \;\diamond\; b \rightsquigarrow c$$

corresponds to so-called AND-causality: event c can only occur after both a and b have occurred. Specifying OR-causality is slightly more tricky:

$$\mathsf{orCausality}(a,b,c)(e) = \begin{cases} \mathbf{1} & \textit{if } c \neq e \\ \mathsf{after}(a) \vee \mathsf{after}(b) & \textit{otherwise} \end{cases}$$

Event c is thus excited after at least one cause has occurred.

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 $a \in A$ and one output signal $b \in A$, 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:

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

An *AND gate* has two inputs a and b and one output c, which synchronises rising transitions via AND-causality and falling transitions via OR-causality:

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

A *C-element* is a gate with two inputs a and b and one output c, which synchronises both rising and falling input transitions via AND-causality (see an example in Section IV-B):

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

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

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

Indeed, a C-element combines the constraints imposed on the output transitions by two 'virtual' buffers.

Behaviour of other gates can be similarly defined using concepts, see our open-source implementation [18].

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, as discussed in the motivating example Section II-C.

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. One can notice 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:

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

Indeed, 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.

In order to specify the initial state of a handshake between signals a and b, we use the initialise concept:

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

For example, $\mathsf{initialise}(a,0)$ 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

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

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

The last important concept that requires an introduction is *mutual exclusion* between two signals a and b:

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

The concept comprises two parts: i) in terms of causality, we say that rising transitions a^+ and b^+ can only occur after the opposite falling ones, ii) the initial states when $a = b = 1$ are forbidden. Taken together these parts guarantee that a and b are never set to 1 at the same time, i.e. they are mutually exclusive. We also add $\overline{\mathsf{after}\, a^+ \wedge \mathsf{after}\; b^+}$ to the invariant.

We can now specify a *mutual exclusion element* [19] 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:

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

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

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

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 V).

Concepts inputs, $\mathsf{outputs}$, $\mathsf{internals}$ are defined for specifying types of sets of signals for convenience. For example, to specify that signals a and b are inputs, c is an output, and t is internal, it is possible to write:

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

IV. Circuit specification with concepts

Here we present a method for deriving a circuit specification from a set of concepts that describe its different aspects. We focus on specification of *speed-independent* (SI) circuits, which is an important class of asynchronous circuits [14] that work correctly regardless of the gate delays, while the wires are assumed to have negligible delays. Alternatively, one can regard wire forks as isochronic and add wire delays to the corresponding gate delays (*Quasi-Delay Insensitive* (QDI) circuit class [23]). A convenient formalism for specification of SI circuits is STGs [6][7], which is a special kind of Petri nets [24] whose transitions are associated with signal events.

A. Petri nets and STGs

Formally, a Petri net is defined as a tuple $PN = \langle P, T, F, M_0\rangle$ comprising finite disjoint sets of *places* P and *transitions* T, *arcs* denoting the flow relation $F \subseteq (P\times T)\cup(T\times P)$ and *initial marking* M_0. There is an arc between $x \in P\cup T$ and $y\in P\cup T$ iff $(x,y)\in F$. The *preset* of a node $x\in P\cup T$ is defined as $\bullet x = \{y \mid (y,x)\in F\}$, and the *postset* as $x\bullet = \{y \mid (x,y)\in F\}$. The dynamic behaviour of a Petri net is defined as a *token game*, changing marking according to the enabling and firing rules. A *marking* is a mapping $M : P \to \mathbb{N}$ denoting the number of *tokens* in each place ($\mathbb{N} = \{0,1\}$ for *1-safe* Petri nets). A transition t is *enabled* iff $\forall p, p\in \bullet t \Rightarrow M(p) > 0$. The evolution of a Petri net is possible by *firing* the enabled transitions. *Firing* of a transition t results in a new marking M' such that

$$\forall p\ M'(p) = \begin{cases} M(p)-1 & \textit{if } p\in \bullet t\setminus t\bullet,\\ M(p)+1 & \textit{if } p\in t\bullet\setminus \bullet t,\\ M(p) & \textit{otherwise.}\end{cases}$$

An STG is a 1-safe Petri net whose transitions are labelled by signal events, i.e. $STG = \langle P, T, F, M_0, \lambda, Z, v_0\rangle$, where λ is a *labelling function*, Z is a set of *signals* and $v_0 \in \{0, 1\}^{|Z|}$ is a *vector of initial signal values*. The labelling function $\lambda : T \to Z^{\pm}$ maps transitions into *signal events* $Z^{\pm} = Z\times\{+,-\}$. The signal events labelled z^+ and z^- denote the transitions of signals $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. The labelling function does not have to be 1-to-1, i.e. transitions with the same label may occur several times in the net. To distinguish transitions with the same label and refer to them from the text an index $i\in\mathbb{N}$ is attached to their labels as follows: $\lambda(t)/i$, where i differs for different transitions with the same label. STGs inherit the operational semantics of their underlying PNs, including the notions of transition enabling and firing.

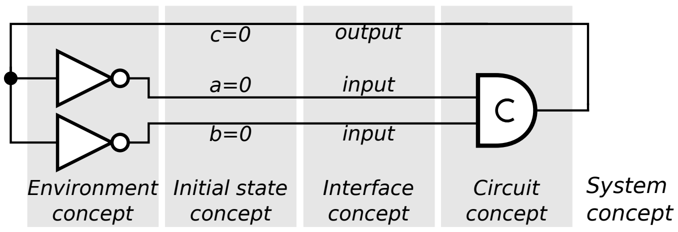


Fig. 4: Example system specified using concepts.

Graphically, the places are represented as circles, transitions as text labels, arcs are shown by arrows, and tokens are depicted by dots. For simplicity, the places with one incoming and one outgoing arc are often hidden, allowing arcs (with implicit places) between transitions.

B. Composition of concepts

A single concept can be used to describe an initial state, invariant states, a single event or a combination of these, yet describing some protocols using this method can become long winded, as these can involve multiple events. We make use of the monoid composition of concepts to describe complex systems incrementally. Importantly we can mix several levels of system description and refer to signal, gate and protocol level concepts in one specification, depending on which level is more convenient in a particular situation.

Consider an example of a C-element, which has signals a and b as inputs, and signal c is the output. The output of this C-element is connected to each of the inputs via an inverter. This is a simple example of a complete system, of a C-element with an environment, see Figure 4. The example can be described by the following script:

$$\begin{array}{lcl}
\mathsf{circuit}(a,b,c) & = & \mathsf{interface} \diamond \mathsf{outputRise} \diamond \mathsf{inputFall}\\
 & & \diamond\ \mathsf{outputFall} \diamond \mathsf{inputRise} \diamond \mathsf{initialState}\\
\mathsf{where} & & \\
\quad\mathsf{interface} & = & \mathsf{inputs}(\{a,b\}) \diamond \mathsf{outputs}(\{c\})\\
\quad\mathsf{outputRise} & = & a^+ \rightsquigarrow c^+ \diamond b^+ \rightsquigarrow c^+\\
\quad\mathsf{inputFall} & = & c^+ \rightsquigarrow a^- \diamond c^+ \rightsquigarrow b^-\\
\quad\mathsf{outputFall} & = & a^- \rightsquigarrow c^- \diamond b^- \rightsquigarrow c^-\\
\quad\mathsf{inputRise} & = & c^- \rightsquigarrow a^+ \diamond c^- \rightsquigarrow b^+\\
\quad\mathsf{initialState} & = & \mathsf{initialise}(a,0) \diamond \mathsf{initialise}(b,0)\\
 & & \diamond\ \mathsf{initialise}(c,0)
\end{array}$$

The first concepts in this list is the $\mathsf{circuit}$ concept, defined as a composition of the following five listed after the where keyword. They describe certain aspects of the system: the first four are named according to what they represent, for example, $\mathsf{outputRise}$ describes the events which cause the output to rise, and the final concept defines the initial states.

Note: For better readability the format of concept scripts used in the article differs from the actual syntax used in our current implementation, which is embedded in Haskell programming language [17] and therefore relies on the standard Haskell syntax. The basis of the format used here is the same, but there are some minor differences. An up-to-date description of the actual syntax supported by the developed tool can be found in the manual [18].

This set of concepts is only one way of describing this C-element and the environment. Another way could be to use gate-level concepts and describe the environment explicitly. In this case the environment allows the inputs to transition in the opposite direction to the output c, as two inverters would. We can then compose this with the C-element and the same interface and initialState concepts as follows:

$$\begin{aligned}\mathsf{circuit}(a,b,c) &= \mathsf{interface} \diamond \mathsf{cElement}(a,b,c)\\ &\diamond\ \mathsf{environment} \diamond \mathsf{initialState}\\ \mathsf{where}\\ \mathsf{environment} &= \mathsf{inverter}(c,a) \diamond \mathsf{inverter}(c,b)\end{aligned}$$

This specification is equivalent to the previous one; indeed one can prove this by rearranging the primitive concepts using the commutativity and associativity of the underlying commutative monoid. Consequently, this specification will be translated to the same STG shown in Figure 5.

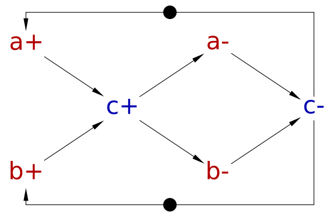


Fig. 5: STG for the example system.

Finally, the designer can also rely on protocol-level concepts, producing the following equivalent specification:

$$\begin{aligned}\mathsf{circuit}(a,b,c) &= \mathsf{interface}\\ &\diamond\ \mathsf{handshake00}(a,c) \diamond \mathsf{handshake00}(b,c)\end{aligned}$$

Indeed, at the high level the circuit can be seen simply as two handshakes synchronised on a common signal.

This example demonstrates that the presented formal notation for capturing concepts is very flexible and provides the designer with a rich selection of available levels of abstraction, which could be used not only for deriving simplest possible specifications but also for cross-checking the adequacy of specifications by *refactoring* them according to the concept composition laws.

C. *Multiple behavioural scenarios*

So far we have only considered systems operating in a single behavioural *scenario* specified by a composition of concepts. However, real-life systems often need to support multiple scenarios, e.g. start-up and normal operation, different power modes, etc. This allows each individual scenario to be designed using concepts, and tested individually to ensure they work correctly, before these are combined to produce a full system specification.

To increase the reuse of scenarios, which helps reduce design time of future systems, this method supports the use of pre-designed scenarios as concepts.

In some cases, a designer may find it easier to split the specification of operational modes further than scenarios and design certain elements separately. In this way, a model may be produced from concepts, which may not be an operational mode on its own, but can be composed and tested separately. Having several elements predefined using concepts may become useful for quickly designing systems. A predefined logic gate, for example, could be useful to quickly include in any list of concepts when designing multiple scenarios. An STG produced of this element can be referenced in a list of concepts by name (provided that the definition is appropriately imported into the current namespace). When a list of concepts is passed into the STG translation algorithm, all referenced concepts are replaced by the corresponding definitions.

V. Interoperability with STG based tools

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 [8] and Mpsat [9]. 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 present a translation algorithm (Section V-A) and discuss ways of composing multiple behavioural scenarios into a single STG specification (Section V-C).

A. *Translating concepts to STGs*

As discussed in Section IV-B, 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 such as causality. Given a specification, we can therefore break down all gate- and protocol-level constructs into 'atoms', which significantly simplifies the translation task.

Using the example of a C-element with an environment, any representation mentioned in Section IV-B can be broken down into the following list of low-level concepts:

$$\begin{aligned}\mathsf{circuit}(a,b,c) &= a^+ \leadsto c^+ \diamond b^+ \leadsto c^+ \diamond c^+ \leadsto a^- \diamond c^+ \leadsto b^-\\ &\diamond\ a^- \leadsto c^- \diamond b^- \leadsto c^- \diamond c^- \leadsto a^+ \diamond c^- \leadsto b^+\\ &\diamond\ \mathsf{inputs}(\{a,b\}) \diamond \mathsf{outputs}(\{c\})\\ &\diamond\ \mathsf{initialise}(a,0) \diamond \mathsf{initialise}(b,0)\\ &\diamond\ \mathsf{initialise}(c,0)\end{aligned}$$

One can see that the above list of primitive concepts matches the STG in Figure 5 very closely, in particular the causality concepts match the arcs of the STG. Below we describe a general translation algorithm that in particular can produce this STG fully automatically. The algorithm is implemented and integrated in the open-source tool Workcraft, where it produces STGs in commonly used *.g file format.

Given a list of primitive concepts, the algorithm starts by creating so-called *consistency loops* for each signal in the specified interface, which forces rising and falling transitions of each signal to alternate, thereby ensuring that the resulting STG satisfies the property of *consistency* [8]. The places between the transitions of a signal determine its state. Figure 6 shows consistency loops created for the example.

The next step is to connect signal transitions according to the list of causality concepts. In the above example we start

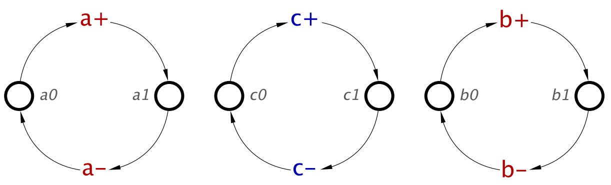


Fig. 6: The first step: create consistency loops.

with $a^+ \rightsquigarrow c^+$. To represent it in the STG, we connect the place after a^+ ($a1$ in Figure 6) to transition c^+ by a *read arc*, which allows c^+ to transition without consuming the token (consuming the token would disable a^-), see Figure 7.

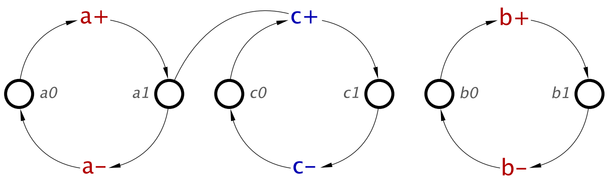


Fig. 7: The second step: add causality concept $a^+ \rightsquigarrow c^+$.

We continue translating the remaining causality concepts in the same manner obtaining the STG shown in Figure 8.

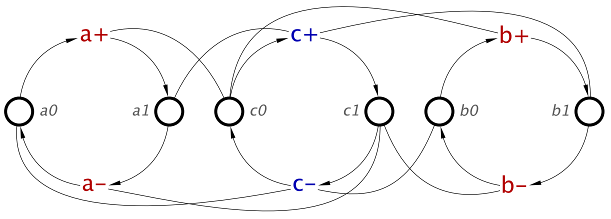


Fig. 8: Finish the second step: add all causality concepts.

The final step of the translation algorithm is to add tokens to some of the places of the resulting STG according to the initial state concepts initialise$(a, 0) \diamond$ initialise$(b, 0) \diamond$ initialise$(c, 0)$, which declare the initial states of all three signals to be 0. We therefore add tokens to places $a0$, $b0$ and $c0$, producing the fully translated STG in Figure 9.

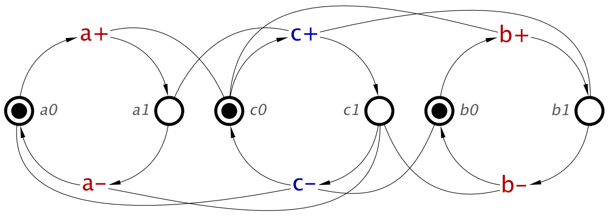


Fig. 9: The final step: set the initial state.

The resulting STG and the STG shown in Figure 5 look very different but they are behaviourally equivalent. The latter is much more compact and easier to read though, and it would be preferable to show it to the designer instead of the one obtained by our translation algorithm. Our implementation supports automated *resynthesis* of the resulting STG using PETRIFY, and can therefore seamlessly produce the STG in Figure 5 from the given specification.

The described translation algorithm is shown in pseudocode in Algorithm 1.

Algorithm 1 Algorithm for translating concepts to STGs

```
for Each defined concept do
    add signal-level concepts to conceptList
end for
for Each signal in system do
    define signal as input/output/internal
    add transition high
    add transition low
    add place signal-0
    add place signal-1
    connect (transition high, place signal-1)
    connect (place signal-1, transition low)
    connect (transition low, place signal-0)
    connect (place signal-0, transition high)
end for
for Each causlity concept do
    connect-read(place following cause, effect transition)
end for
for Each initial state concept do
    add-token(place signal-value)
end for
```

B. Partial translation and parallel composition

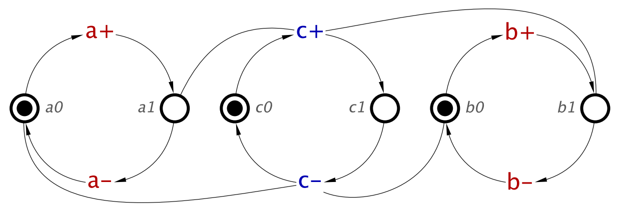
In some cases it may be useful to partially translate concepts to STGs, for example to investigate the behaviour of a circuit in different environments. An important property of the above algorithm is that it can be applied to incomplete systems, thereby translating smaller collections of concepts, or even singular concepts, which do not necessarily correspond to well-behaving STGs (e.g. the resulting STGs may contain spurious input transitions if they are not controlled by the environment). Such partial specifications can be visualised, studied, and then composed into complete specifications using the *parallel composition*, which is supported by WORKCRAFT using an integrated tool PCOMP [22].

Using the example in Section IV-B, a C-element with environment, we can translate the concepts corresponding to the C-element and the two environment inverters individually. Including the initial states, this would give us the STGs found in Figure 10. These STGs can be composed using the parallel composition, the outcome of which will be equivalent to the result obtained by translating the complete specification, bringing us back to the STG in Figure 5.

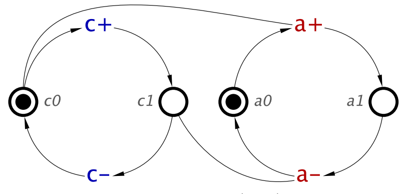
C. Scenario combination

As mentioned in Section IV-C, concepts are used to specify a single behavioural scenario. When combining scenarios, there are several things to consider in how the scenarios fit together. Depending on the application, some scenarios may need to operate in certain orders, for example, one scenario may exist simply to initialise the circuit, therefore this scenario needs to run at start-up, before any other scenario, and then never be run again while the system remains active.

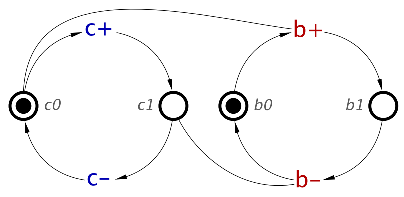
To address this, we can define *templates*, which can be used to combine scenarios in various ways. With this the designer specifies how the scenarios should be combined, and if an order is needed, the order the scenarios should be run from start up. The following are some examples of templates that can be used to combine scenarios.



(a) cElement(a, b, c)



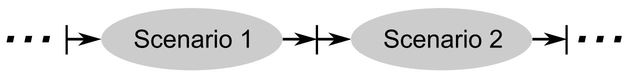
(b) inverter(c, a)



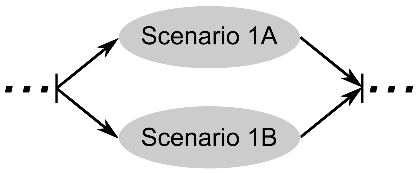
(c) inverter(c, b)

Fig. 10: Partial translation of concepts into STGs

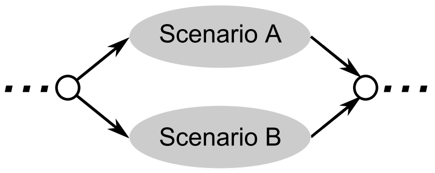
Sequential template: Sequential combination allows a designer to select the order of all scenarios, so when combined, they run in a sequence.



Concurrent template: In this case there is no order, but one or more of the scenarios run concurrently. It may be necessary to limit concurrency by specifying, for instance, the number of scenarios that can be active simultaneously:



Non-deterministic choice template: This template combines the scenarios in a way that allows any of the scenarios to run by consuming the initial token from the *choice place* and producing it in the *merge place* when the scenario completes:



It is possible to combine some scenarios using one template, then including the result in a combination with other scenarios using another template.

This method can allow for many possible scenario combination styles, and more complex systems can be combined automatically, which in comparison to manual combination, could reduce the number of errors as well as design time.

D. *Tool integration*

The concepts tool PLATO [18], which translates asynchronous concepts to STGs has been integrated into open-source toolsuite WORKCRAFT [13]. This allows a designer to visualise their concept designs as STGs, to simulate, verify and synthesise them using other tools integrated in WORKCRAFT, and 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.

Scenario combination is not yet implemented and is performed manually by the designer using WORKCRAFT's graphical user interface. This design stage can also be automated to further streamline the design process using concepts.

VI. CASE STUDY

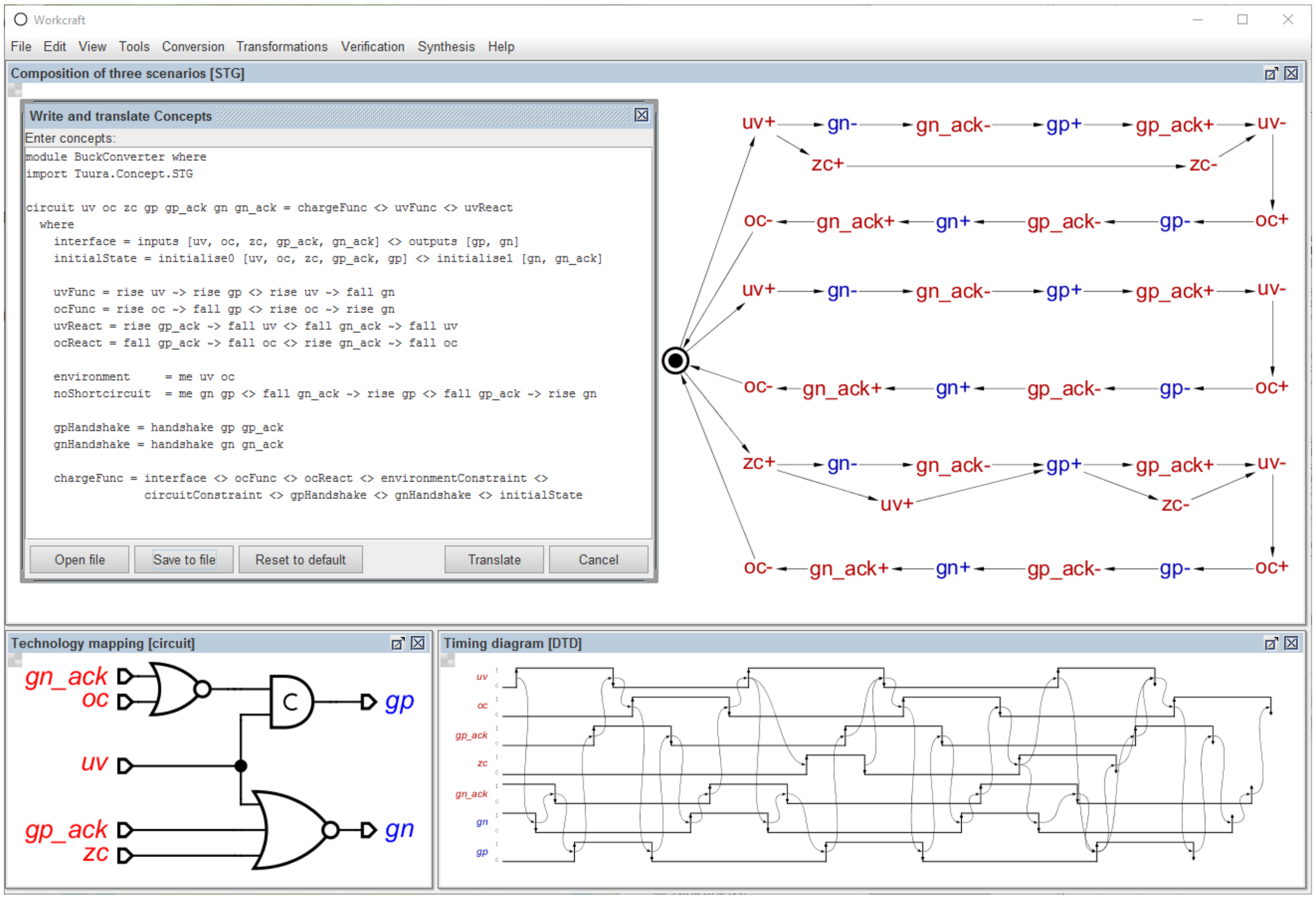
In Section II we introduced the motivating example for this article, a simple buck converter, and discussed the challenges of monolithic approach to specifying asynchronous circuits. We have also introduced concepts in Section III, and with examples explained how they can be used to design simple circuits, and how these can be translated to STGs for subsequent synthesis of asynchronous gate-level implementation. In this section we use the buck converter as a case study to detail the whole design process in WORKCRAFT, see Figure 11.

A. *Formal specification using concepts*

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

1) ZC absent scenario: A circuit that handles buck operation in absence of ZC condition is specified as follows:

$$
\begin{array}{l}
\mathsf{zcAbsentScenario}(uv, oc, zc, gp, gp\_ack, gn, gn\_ack) \\
\qquad\qquad = \mathsf{chargeFunc} \diamond \mathsf{uvFunc} \diamond \mathsf{uvReact} \\
\ \mathsf{where} \\
\begin{array}{lll}
\quad \mathsf{interface} & = & \mathsf{inputs}\ [uv, oc, zc, gp\_ack, gn\_ack] \\
 & \diamond & \mathsf{outputs}\ [gp, gn] \\
\quad \mathsf{uvFunc} & = & uv^+ \rightsquigarrow gp^+ \ \diamond\ uv^+ \rightsquigarrow gn^- \\
\quad \mathsf{ocFunc} & = & oc^+ \rightsquigarrow gp^- \ \diamond\ oc^+ \rightsquigarrow gn^+ \\
\quad \mathsf{uvReact} & = & gp\_ack^+ \rightsquigarrow uv^- \ \diamond\ gn\_ack^- \rightsquigarrow uv^- \\
\quad \mathsf{ocReact} & = & gp\_ack^- \rightsquigarrow oc^- \ \diamond\ gn\_ack^+ \rightsquigarrow oc^- \\
\quad \mathsf{environment} & = & \mathsf{me}(uv, oc) \\
\quad \mathsf{noShortcircuit} & = & \mathsf{me}(gn, gp) \\
 & \diamond & gn\_ack^- \rightsquigarrow gp^+ \ \diamond\ gp\_ack^- \rightsquigarrow gn^+ \\
\quad \mathsf{gpHandshake} & = & \mathsf{handshake}(gp, gp\_ack) \\
\quad \mathsf{gnHandshake} & = & \mathsf{handshake}(gn, gn\_ack) \\
\quad \mathsf{initialState} & = & \mathsf{initialise0}\ [uv, oc, zc, gp, gp\_ack] \\
 & \diamond & \mathsf{initialise1}\ [gn, gn\_ack] \\
\quad \mathsf{chargeFunc} & = & \mathsf{interface} \diamond \mathsf{ocFunc} \diamond \mathsf{ocReact} \\
 & \diamond & \mathsf{environment} \diamond \mathsf{noShortcircuit} \\
 & \diamond & \mathsf{gpHandshake} \diamond \mathsf{gnHandshake} \\
 & \diamond & \mathsf{initialState}
\end{array}
\end{array}
$$



```
module BuckConverter where
import Tuura.Concept.STG

circuit uv oc zc gp gp_ack gn gn_ack = chargeFunc <> uvFunc <> uvReact
  where
    interface = inputs [uv, oc, zc, gp_ack, gn_ack] <> outputs [gp, gn]
    initialState = initialise0 [uv, oc, zc, gp_ack, gp] <> initialise1 [gn, gn_ack]

    uvFunc = rise uv ~> rise gp <> rise uv ~> fall gn
    ocFunc = rise oc ~> fall gp <> rise oc ~> rise gn
    uvReact = rise gp_ack ~> fall uv <> fall gn_ack ~> fall uv
    ocReact = fall gp_ack ~> fall oc <> rise gn_ack ~> fall oc

    environment     = me uv oc
    noShortcircuit  = me gn gp <> fall gn_ack ~> rise gp <> fall gp_ack ~> rise gn

    gpHandshake = handshake gp gp_ack
    gnHandshake = handshake gn gn_ack

    chargeFunc = interface <> ocFunc <> ocReact <> environmentConstraint <>
                 circuitConstraint <> gpHandshake <> gnHandshake <> initialState
```

Fig. 11: Stages of the design flow when using asynchronous concepts in WORKCRAFT.

The zcAbsentScenario concept is a composition of buck charging and UV handling. Consider the charging function captured in chargeFunc concept. It comprises several concepts: interface specifies the types of signals and initialState defines their initial state; gpHandshake and gnHandshake specify the protocol on gp/gp_ack and gn/gn_ack interfaces respectively; noShortcircuit enforces a safety constraint to prevent a shortcircuit; ocFunc and ocReact define the interplay with OC condition; and environment captures the fact that OC and UV conditions never happen at the same time.

Note that the sequence of PMOS/NMOS switching during the charging cycle is the same for all operating modes, and therefore the chargeFunc concept can be reused in other scenarios.

The zcAbsentScenario concept is automatically translated to the STG specification in Figure 12.

2) ZC late scenario: If ZC condition is detected after UV, then buck operation is the same as in absence of ZC, i.e. ZC conditions can be ignored. This is captured by an additional zcLate concept:

$$\begin{aligned}&\mathsf{zcLateScenario}(uv, oc, zc, gp, gp\_ack, gn, gn\_ack)\\&\qquad = \mathsf{chargeFunc} \diamond \mathsf{uvFunc} \diamond \mathsf{uvReact} \diamond \mathsf{zcLate}\\&\text{where}\\&\quad \mathsf{zcLate} = uv^{+} \rightsquigarrow zc^{+} \diamond zc^{-} \rightsquigarrow uv^{-}\end{aligned}$$

The STG specification automatically produced from the zcLateScenario concept is shown in Figure 13. It looks similar to the STG in Figure 12 but features a concurrent branch for zc signal. Note that the arc $zc^{+} \rightsquigarrow zc^{-}$ is implied at translation time by consistency loops.

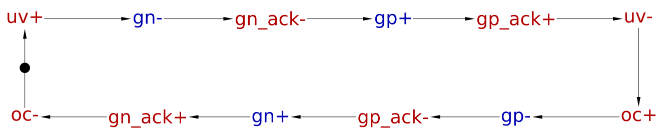


Fig. 12: STG for the zcAbsentScenario concept.

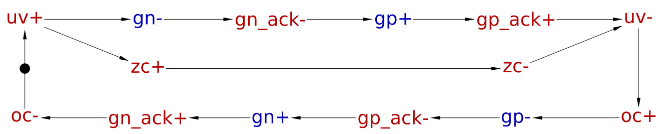


Fig. 13: STG for the zcLateScenario concept.

Note that the concept specification of zcLateScenario reuses most of the code from the zcAbsentScenario; in particular, the description of the analogue environment does not need to be duplicated, as with the STG approach, where the designer needs to redraw the specification from scratch. WORKCRAFT allows to copy and paste STGs, which mitigates the problem, but duplication and associated design problems remain – for example, if the analogue environment needs to be amended, these changes need to be done consistently in all scenarios. With concepts, only one definition needs to be changed, which increases the productivity of the designer.

3) ZC early scenario: If ZC condition is detected before UV then it needs to be explicitly handled by the control circuit:

$$\begin{aligned}
&\mathsf{zcEarlyScenario}(uv, oc, zc, gp, gp\_ack, gn, gn\_ack)\\
&\quad = \mathsf{chargeFunc} \diamond \mathsf{zcFunc} \diamond \mathsf{zcReact} \diamond \mathsf{uvFunc}'\\
&\quad \diamond\ \mathsf{uvReact}'\\
&\text{where}\\
&\quad \mathsf{zcFunc} = zc^{+} \rightsquigarrow gn^{-}\\
&\quad \mathsf{zcReact} = oc^{-} \rightsquigarrow zc^{+} \diamond gp^{+} \rightsquigarrow zc^{-}\\
&\quad \mathsf{uvFunc}' = uv^{+} \rightsquigarrow gp^{+}\\
&\quad \mathsf{uvReact}' = zc^{+} \rightsquigarrow uv^{+} \diamond zc^{-} \rightsquigarrow uv^{-}\\
&\qquad \diamond\ gp\_ack^{+} \rightsquigarrow uv^{-}
\end{aligned}$$

The obtained STG for the zcEarlyScenario concept is shown in Figure 14.

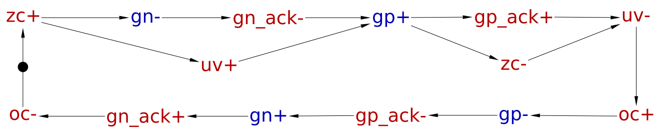


Fig. 14: STG for the zcEarlyScenario concept.

B. Combining the scenarios

The scenario STGs can now be combined as described in Section V-C to produce a multi-scenario specification. As buck operating modes are active one at a time, the control scenarios are combined using the non-deterministic choice template.

Figure 15 shows the resulting STG specification. This can be further simplified by merging the common parts of the scenarios, thus producing a more compact model similar to that shown in Figure 2.

There is an explicit place in this model which holds a token initially and allows any of the scenarios to run. This free-choice place has no control over which scenario can run, and only allows one of them to run at a time.

C. Verification and simulation

To be combinable by our method, the scenario STGs need to satisfy certain properties [8]:

- Complete State Coding (CSC): each state of the models with different behaviour has differing signal encodings to avoid problems during synthesis. Note that in most cases it is possible to automatically resolve CSC conflicts.
- Deadlock freedom: no state is reachable from which no progress can be made.
- Output persistence: there are no race conditions in the STG (i.e. no glitches).
- Signal consistency: in any trace the rising and falling transitions of each signal alternate.

These properties can be automatically checked in WORKCRAFT using the MPSAT [9] backend tool. It is also possible to verify custom safety properties expressed using syntax similar to SystemVerilog Assertions, which is familiar to many hardware designers. Figure 16 contains an example custom assertion !(gp && gn) that we created for this case study. In this we formally verify that there is no shortcircuit in the system, by checking that gp and gn

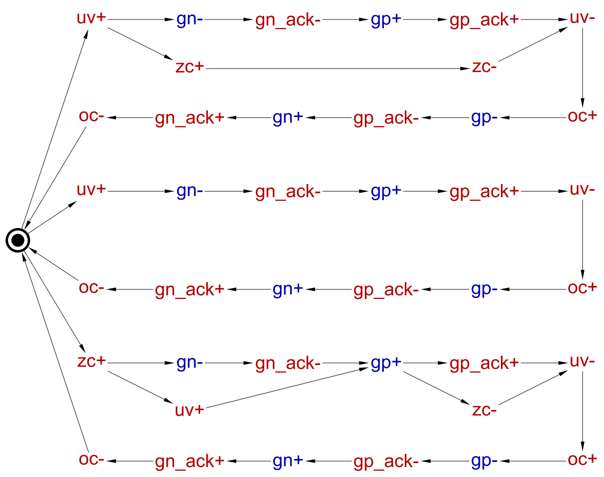
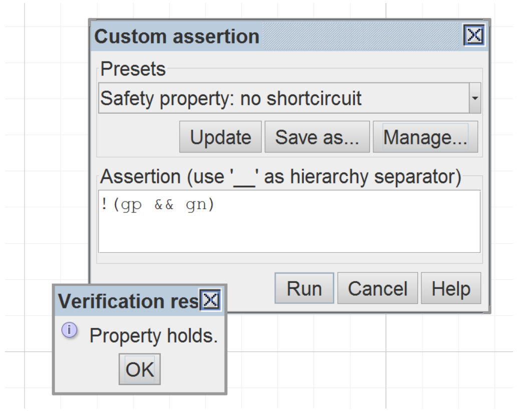


Fig. 15: Complete STG for a buck converter.


Custom assertion

Presets

Safety property: no shortcircuit

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

Assertion (use '__' as hierarchy separator)

```
!(gp && gn)
```

Run | Cancel | Help

Verification res

Property holds.

OK

Fig. 16: Custom verification of no shortcircuit.

are never high at the same time. As seen in this image, the property correctly holds, as expected.

In the event that one of these properties does not hold, unless it can be corrected automatically, which MPSAT can attempt, the combination of scenarios fails. In this case a problematic concept can be identified and diagnostic information is printed out to help a designer correct the issue.

Correctly produced scenarios may not necessarily work as the designer expects, and this needs to be validated before using these scenarios in any further designs. In addition to formal verification, WORKCRAFT features a simulation tool, and this can be used by a designer to check that the signals can transition according to the initial requirements. If the simulation produces undesirable results, a designer can work to fix the error of the scenario STG, or correct the design at the concept level. The latter is the preferable method if the design is to be reused either as a predefined concept, or as a scenario in another system.

After all scenarios have been combined, the result will be the STG specification for the full system. If each scenario used in the combination process satisfies the verification properties, it is possible no errors will exist within the full specification. However, unforeseen errors may be produced and thus, it is necessary to verify the resulting specification. This can be performed using MPSAT as with the scenarios to ensure the same properties are held. The simulation tool built into WORKCRAFT can also be utilised for testing the functionality of a full system specification.

D. *Synthesis of a speed-independent controller*

The final step in this design flow is to synthesise the STG specification into a speed-independent circuit. Synthesis is the process of finding Boolean equations to calculate the next state of the output signals based on the input signals and the current state of the circuit [8]. Circuits can be synthesised using PETRIFY or MPSAT, both of which are integrated in WORKCRAFT. There are different types of synthesis with their own specifics. For example, Figure 17 shows the result of *Complex gate* synthesis of the buck STG.

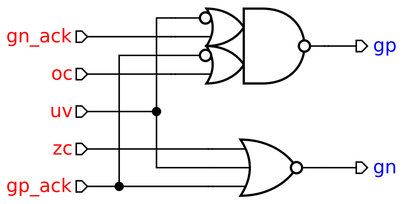


Fig. 17: Complex gate implementation.

Complex-gate synthesis does not use a gate library and yields Boolean functions of arbitrary complexity. These functions are often too large to be implemented by a single gate available in the gate library. Unfortunately, breaking up a complex-gate into smaller ones, when performed naïvely, generally yields an incorrect circuit – this happens due to the delays associated with the outputs of the newly introduced gates and can lead to glitches or deadlocks.

In fact, logic decomposition in the context of speed-independent circuits is a very difficult problem, that cannot always be solved. PETRIFY and MPSAT backend tools do a good job in many situations, but occasionally they fail to converge to a solution and a manual intervention by the designer is required.

Another type of synthesis is *Technology mapping*. This performs *Logic decomposition* and uses a gate library to map the logic only onto gates available in the library.

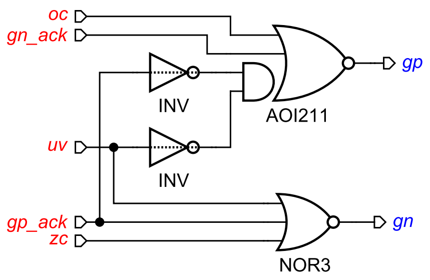


Fig. 18: Technology mapped implementation.

Figure 18 is the technology mapped implementation. The gate labels correspond to the gate names in the library. The two inverters are shown with a dotted line through. This is because they have the *Zero delay* property expressing the assumption that their output wire delays are negligible. This is usually unproblematic as long as such input inverters are placed close to the main gate; this can be enforced during the place-and-route stage.

Within WORKCRAFT we can further decompose the implementation of *gp* into two-input gates, as shown in Figure 19. This is achieved by restricting the synthesis back end (in this case MPSAT) to use the gates with at most two inputs. Such 2-input decompositions may be beneficial for designs targeting low operating voltages.

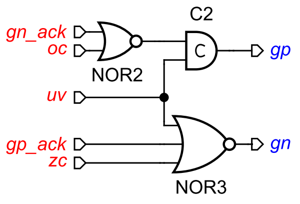


Fig. 19: Technology mapping into 2-input gates/latches.

More information on the synthesis of this example can be found in the tutorials of the WORKCRAFT website [13].

VII. RELATED WORK

There are several existing methodologies which are similar to the one being proposed in this paper, and there are a common set of features between these which can be compared. In some cases, these cannot be used for asynchronous designs, but some of their other key features can still be compared. Table I contains a comparison of these features.

We have found the related work limited in certain aspects, and we discuss this below.

A common approach of designing asynchronous circuits, **Balsa** [25], uses an RTL language to specify operations for asynchronous circuits for both big-digital, data featuring multiple bits, and little-digital, control systems. A Balsa specification is initially converted into a format describing a network of handshake components, which can be used for simulation and circuit diagrams. This can then be used in synthesis, by mapping handshake components on to library components [26]. RTL languages are used for synchronous design, and thus designers can adapt to Balsa more easily, however specifying a control system can lead to a complicated program which can be difficult to comprehend, in comparison to the STGs produced in the proposed approach, in which signal interactions can be visualised.

Biscotti [27] is an approach which uses a C-style language, which can be easy to adapt to as designers are likely to have programming experience. It features *forever* blocks, in which code runs sequentially, but all these blocks run concurrently to each other. This design method starts by specifying a circuit which is then *compiled* into a Petri Net for verification with existing tools including WORKCRAFT and if successful, synthesis, similar to our approach with conversion of concepts to allow use with multiple existing back-end tools based on STGs. Biscotti is aimed at designing data-driven asynchronous systems however, and as with Balsa, specifying an asynchronous control system is difficult, especially as the number of signals involved increases.

TABLE I: A comparison of features of related work and the proposed method

Method	Asynchronous support	Tool support	Gate-level	Event-level	Protocol-level	Design focus
Concepts	Yes	Yes	Yes	Yes	Yes	Little digital
Balsa	Yes	Yes	Yes	No	Yes	Big digital
Biscotti	Yes	Yes	Yes	No	No	Big digital
Lava	No	Yes	Yes	No	Yes	Big digital
Cλash	No	Yes	Yes	No	Yes	Big digital
Snippets	No	No	No	No	No	Little digital
DI algebra	Yes	No	No	Yes	No	Little digital
Structural design	No	Yes	No	No	No	Modular

[28] introduces **Lava**, a Haskell tool. This features its own design flow, using predefined functions for structures like logic gates, allowing users to define functions using these structures. A collection of these functions can be used to design a circuit and be verified within Lava, however for steps such as simulation and synthesis, Lava generates VHDL code to be used by other software. This has similar ideals to concepts, allowing users to define functions from predefined structures, which can be reused. However, Lava is more focussed on designing synchronous circuits for data processing featuring wires with multiple bit widths, rather than asynchronous control circuits.

Cλash, introduced in [29], is also a Haskell based tool, similar to Lava, focussed on synchronous data processing circuits. It has some cross-over features with Lava, such as built-in verification, and the ability for users to define functions. Cλash also features built in synthesis and simulation, avoiding the need to export VHDL, however this feature remains. This allows a simpler way of specifying synchronous circuits, which may be more natural for designers, however Haskell as a language features huge differences to programming languages like C, which may be an issue for adoption. Cλash is also focussed on synchronous circuits.

The current implementaion of asynchronous concepts also uses Haskell as the host language, but our focus is on asynchronous control circuits, therefore the designer only works with a very small subset of Haskell using predefined domain-specific primitives, i.e. no advanced Haskell knowledge is required. We use Haskell because it provides powerful functional programming abstractions, significantly simplifying our implementation, and allowing us to use algebraic approaches to the specification of event-interaction graphs, e.g. see [30].

Snippets [31], similar to concepts, are smaller state graph models which are used to compose full state graphs of larger systems. Snippets describe the operation of a part of a system in terms of input and output alphabets, and in which ways these snippets can fail. When composed with other snippets it can produce a working system state graph model. With our design methodology however we want to go deeper and decompose a component into concepts responsible for capturing signal behaviours for system features, such as handshakes, mutual exclusion, synchronisation and others.

DI algebra [32] is a method of describing systems as algebraic equations. Each equation represents an operation of the specification, similar to scenarios, and composing these can be simplified for the most compact version of the equation. These can then be composed to find an equation for the whole specification and again simplified for the most compact version. Our method is similar to DI algebra, however concepts are described textually, which is different to DI algebra and as such, simplification does not occur at concept level, but during the composition and combination steps, and the most compact form of the model is automatically produced.

To the best of our knowledge there are no EDA tools supporting compositional design of asynchronous circuits using snippets or DI algebra, which makes these approaches unsuitable for use in an industrial setting.

Structural design features re-usability of modular components [33]. Here, a component design can be used multiple times across full device designs in conjunction with several other circuit modules. These modules can be changed in some way without affecting how they are used in the full device designs. The ideas of this method are similar to that of the design methodology we are proposing to reduce design time. However, this method is at a much higher level, using fully designed and tested components, whereas we propose to allow re-usability at gate and even signal/event levels.

Asynchronous concepts presented in this paper have been inspired by a series of research works dedicated to the compositional specification of asynchronous circuits. Specifically, we build on Conditional Partial Order Graphs [20], Conditional Signal Graphs [21] and Parameterised Graphs [34]. This work is different in that we focus on developing a textual specification language that can be used by a designer directly, without the need to explicitly describe large graph-based models.

Concepts have many advantageous features, such as reuse, description and composition at multiple levels of abstraction. Several of the abovementioned approaches feature similar features which make them beneficial in certain circuit design applications, but we believe that concepts address an important gap in the state of the art: compositional design of little digital asyncrhonous control logic, with industrial-strength open-source EDA tool support.

VIII. Conclusions and future research

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.

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 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 PGMINER [35], 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 include process mining in the design flow for the proposed approach.

As the Internet-of-Things becomes ubiquitous, asynchronous circuits will be key in designing energy-efficient and reliable IoT nodes, particularly at the interfaces between analogue and digital domains, such as power management units. We believe that high-level asynchronous concepts could be instrumental in the design of these systems.

Acknowledgements

The authors would like to thank the reviewers for their constructive comments. This research was supported by EPSRC research grant ‘A4A: Asynchronous design for Analogue electronics’ (EP/L025507/1) and the Royal Society Research Grant ‘Computation Alive: Design of a Processor with Survival Instincts’. Jonathan Beaumont is sponsored by a PhD scholarship from the School of Engineering, Newcastle University, UK.

References

[1] J. Myers, A. Savanth, R. Gaddh, D. Howard, P. Prabhat, and D. Flynn. A subthreshold arm cortex-m0+ subsystem in 65 nm cmos for wsn applications with 14 power domains, 10t sram, and integrated voltage regulator. *IEEE Journal of Solid-State Circuits*, 51(1):31–44, Jan 2016.

[2] Andrew Talbot. Holistic mixed signal design in ultra deep sub-micron technologies. *NMI R&D Workshop: Analog and Mixed-Signal Design*, 2016.

[3] Y. Lee, Y. Kim, D. Yoon, D. Blaauw, and D. Sylvester. Circuit and system design guidelines for ultra-low power sensor nodes. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1037–1042, June 2012.

[4] Jens Sparsø and Stephen B Furber. *Principles of asynchronous circuit design: a systems perspective*. Springer Netherlands, 2001.

[5] Jonathan Audy. Navigating the path to a successful IC switching regulator design. *Tutorial at IEEE International Solid-State Circuits Conference (ISSCC)*, 2008.

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

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

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

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

[10] V. Khomenko and A. Mokhov. An algorithm for direct construction of complete merged processes. In *International Conference on Application and Theory of Petri Nets and Concurrency*, pages 89–108, 2011.

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

[12] D. Sokolov, V. Khomenko, and A. Mokhov. Workcraft: Ten years later. In *This asynchronous world. Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday*, pages 269–293. Newcastle University (UK), 2016.

[13] Workcraft website. https://www.workcraft.org.

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

[15] 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.

[16] 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.

[17] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, 1996.

[18] Concepts repository. https://github.com/tuura/plato, 2016.

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

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

[21] Andrey Mokhov, Danil Sokolov, and Alex Yakovlev. Adapting asynchronous circuits to operating conditions by logic parametrisation. *2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems*, pages 17–24, 2012.

[22] Arseniy Alekseyev, Victor Khomenko, Andrey Mokhov, Dominic Wist, and Alex Yakovlev. Improved parallel composition of labelled Petri nets. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 131–140, 2011.

[23] A. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing, vol. 1(4)*, pages 226–234, 1986.

[24] C. Petri. *Kommunikation mit automaten (Communicating with automata)*. PhD thesis, University of Bonn, 1962.

[25] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[26] Kees Van Berkel. *Handshake circuits: an asynchronous architecture for VLSI programming*, volume 5. Cambridge University Press, 1993.

[27] Gang Jin, Lei Wang, and Zhiying Wang. A new description language for data-driven asynchronous circuits and its design flow. In *Circuits, Communications and Systems, 2009. PACCS ’09. Pacific-Asia Conference on*, pages 322–325, May 2009.

[28] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[29] Christiaan Baaij. Cλash: From Haskell to Hardware. 2009.

[30] A. Mokhov. Algebraic Graphs with Class (Functional Pearl). In *Proceedings of the International Haskell Symposium*. ACM, 2017.

[31] Igor Benko and Jo Ebergen. Composing snippets. In Jordi Cortadella, Alex Yakovlev, and Grzegorz Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 1–33. Springer Berlin Heidelberg, 2002.

[32] Mark B Josephs and Jan Tijmen Udding. An overview of di algebra. In *Hawaii International Conference on System Sciences (HICSS)*, volume 1, pages 329–338. IEEE, 1993.

[33] Craig Armenti. Get to market faster with modular circuit design. *Electronic Engineering Journal*, 2015.

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

[35] 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.