School of Electrical, Electronic & Computer Engineering

Adapting asynchronous controllers to operating conditions by logic parametrization

Andrey Mokhov, Danil Sokolov, Alex Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2011-172

September 2011

Contact:

Andrey.Mokhov@ncl.ac.uk

Danil.Sokolov@ncl.ac.uk

Alex.Yakovlev@ncl.ac.uk

Supported by EPSRC grants EP/G037809/1 and EP/F016786/1

NCL-EECE-MSD-TR-2011-172

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

`http://async.org.uk/`

Adapting asynchronous controllers to operating conditions by logic parametrization

Andrey Mokhov, Danil Sokolov, Alex Yakovlev

Microelectronics System Design Group

Abstract

Modern hardware systems are required to be robust, resilient and long-life, and thus they have to be adaptive to possibly changing requirements and operating conditions. This covers not only the data processing functions but also control, and even timing and power operation. For example, such systems will be increasingly powered by ambient sources (energy harvesting) and will experience a wide range of power modes, including not only dynamic voltage scaling but even power supplies which do not deliver a stable level of Vdd.

Asynchronous controllers can offer ultimate robustness under extreme variations caused by unstable power supply; however, the robustness often comes at the price of a heavy and slow implementation. This implementation may be the only option for the extreme operating conditions, but is unacceptable for the normal operation mode of the system (in which it may be running most of the time). There is clearly a need to choose a particular controller implementation in run-time rather than in design time. In this paper we present and investigate a new methodology which provides a way of building adaptive asynchronous controllers supporting a range of power and timing modes of operation (possibly even synchronous modes) that can be selected during run-time according to the known hardware, environmental, and power/latency budget requirements.

1 Introduction

Widening application areas of modern digital systems bring them to the domain of extreme operating conditions, new process technologies, and dynamically changing requirements. For example, new systems will be increasingly powered by ambient sources, e.g. through energy harvesting, and thereby will experience a wide range of modes in which they are powered, including not only modes of dynamic voltage scaling but even power supplies which do not deliver a stable level of Vdd at any particular moment or interval of operation. Ideas of building systems that adapt to the ambient operating and power supply conditions have been expressed in [15], where this approach was given a collective term of *energy-modulated computing*. It was emphasized that the timing and power supply issues are inseparable in adapting to such conditions, and the role of asynchronous or self-timed design methods will become more prominent.

As further discussed in [15], self-timed systems offering qualities of delay insensitivity and high tolerance to Vdd fluctuations can be more *power-proportional* (i.e. operating on a wide range of power modes and delivering *proportional* quality of service) than their synchronous counterparts, but they are not *power-efficient* (i.e. delivering the *best performance* for invested power) under the stable operating conditions. The problem is, however, that the existing synthesis techniques typically optimise the system design for a well-defined operation mode, and as a result most of the research in logic synthesis has been compartmentalized in specific timing domains. Even for systems that are globally asynchronous and locally synchronous the division between synchronous and asynchronous domains has been structural or spatial, but not temporal and therefore not adaptive to the operating conditions. Even within the domain of asynchronous design, there has typically been the way of advocacy of a certain class of design, such as delay insensitive (DI), quasi delay insensitive (QDI), speed independent (SI), relative timing,

and fundamental mode design [14]. Each particular design class was associated with a particular style of modelling actions and signal events, normally captured by models such as Signal Transitions Graphs (STGs) [4] or Asynchronous Finite State Machines (FSMs), e.g. burst-mode FSMs [13].

Bearing in mind the 'broadband' nature of increasing variety of operating conditions, there is clearly a need to design controllers in such a way that they meet certain critical behavioural specifications, but may vary in specific ways of how fine-grain signal-ordering actions are enforced, i.e. whether they are produced by explicit causal relations (for robustness under extreme conditions) or by implicit timing assumptions (for performance in a stable and predictable mode). Therefore, there is an emerging need to choose a particular controller implementation at run-time rather than at design time.

In this paper, to the best of our knowledge for the first time, we present and investigate a new methodology which provides a way of building adaptive *parametrized controllers* which combine advantages of different timing classes and domains, both asynchronous and synchronous. These controllers in addition to their interface signals will have a set of auxiliary 'parametric' control inputs that select the most adequate implementation according to the known hardware, environmental, and power/latency budget requirements. A parametrized controller (if designed without significant overheads due to its generality – this is the main challenge!), can provide the ultimate robustness under low/unstable voltage operation in the DI mode, while enjoying a high performance of non-DI solutions under high-voltage operation, when circuit delays are more predictable and timing assumptions are likely to hold.

Our main contribution is a scalable method of composing several controller implementations into a single parametrized controller (Section 3). Crucially, the method can be fully automated using existing specification and synthesis tools. We also demonstrate that the overhead of this composition is not critical in practice and the benefits of having a parametrized, dynamically reconfigurable solution outweigh the potential penalties (Sections 4 and 5). Another important contribution of this paper is a new model, called *Conditional Signal Graphs* [10], which is capable of describing heterogeneous controllers and supports automated encoding and composition methods. It forms the backbone of the proposed methodology and is formally introduced in Section 2.

2 Background on CSGs

In this paper we use two behavioural models to specify and synthesise parametrized asynchronous controllers: Signal Transition Graphs (STGs) [4] and Conditional Signal Graphs (CSGs) [10]. STGs have been introduced a long time ago and have been extensively studied and used in this context, hence we assume the reader to be familiar with them. CSGs, however, is a new formalism derived from the Conditional Partial Order Graph (CPOG) model [9][12] and in this section we will describe it in detail.

2.1 Conditional Signal Graphs

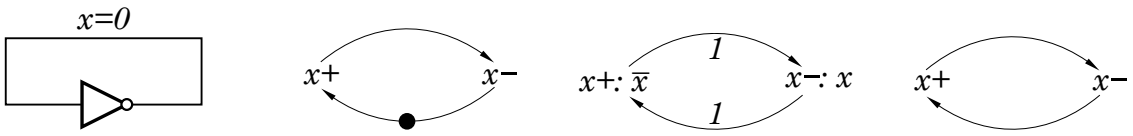
Similarly to the STG model, Conditional Signal Graphs can formally describe behaviour of asynchronous circuits by associating vertices with signal transitions and using arcs to represent causality between them (albeit in a different way).

A Conditional Signal Graph is a tuple $G = (V, E, S, \phi)$, where S is a set of signals, $V \subseteq S \times \{+, -\}$ is a set of *vertices* or *events* representing *signal transitions:* namely, events s^+ and s^- correspond to the rising and falling edges of a signal s, respectively, and $E \subseteq V \times V$ is a set of *arcs* of the graph. Together vertices and arcs $V \cup E$ are called *elements* of G. Each element $z \in V \cup E$ has a *condition* $\phi(z) \in \mathcal{F}(S)$ associated with it, wherein $\mathcal{F}(S)$ stands for a set of Boolean functions defined on domain S.

The purpose of conditions ϕ is to activate/deactivate elements of the graph according to the current state of its signals. A vertex is *active* iff its condition evaluates to 1; an arc is *active* iff its condition evaluates to 1 and it connects two active vertices; all other elements are *inactive*.

An active vertex with no active incoming arcs is *enabled* and its corresponding signal transition can fire changing the current state of the circuit. Thereby an arc $v \to u$ can be considered as a *conditional dependency* between v and u.

Example 1. Consider one of the simplest sequential circuits – a 1-inverter ring oscillator. The circuit and its STG specification are shown in Fig. 1(a,b). The circuit behaviour is very simple: events x^+ and x^- occur alternatingly starting from the initial state when $x = 0$[1].

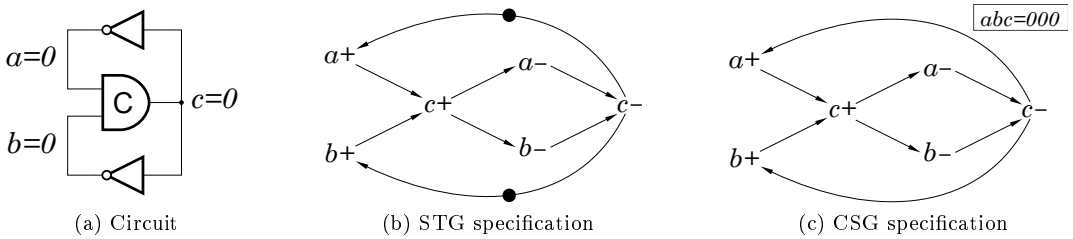


(a) Circuit (b) STG specification (c) CSG specification (full) (d) CSG (simplified)

Figure 1: Simple 1-inverter ring oscillator

Fig. 1(c) shows a CSG specification of the circuit. A vertex is represented with its signal transition $s^{\pm}$, a colon ':', and condition $\phi(s^{\pm})$. An arc $v \to u$ is drawn as an arrow between vertices v and u with condition $\phi(v \to u)$ next to it.

In the initial state event x^+ is active because its condition evaluates to 1: $\phi(x^+) = \overline{x} = \overline{0} = 1$, while x^- is inactive: $\phi(x^-) = x = 0$. Since x^+ has no active dependencies, it is enabled to fire. As soon as it fires, the circuit state changes to $x = 1$ in which x^+ is no longer active but event x^- becomes enabled, etc. In general, it is reasonable to assume[2] that for any signal $s \in S$ events s^+ and s^- are active only when $s = 0$ and $s = 1$, respectively, thus conditions $\phi(s^+) = \overline{s}$ and $\phi(s^-) = s$ may be omitted in a diagram for clarity; constant 1 conditions can also be omitted, see the simplified version of the specification in Fig. 1(d). The only visual difference between specifications in Fig. 1(b) and Fig. 1(d) is that the latter is 'token-free'. The fundamental difference between STGs and CSGs is deeper and will become clearer in the next example. Interestingly, there are many cases when a CSG specification is isomorphic to its STG equivalent.



(a) Circuit (b) STG specification (c) CSG specification

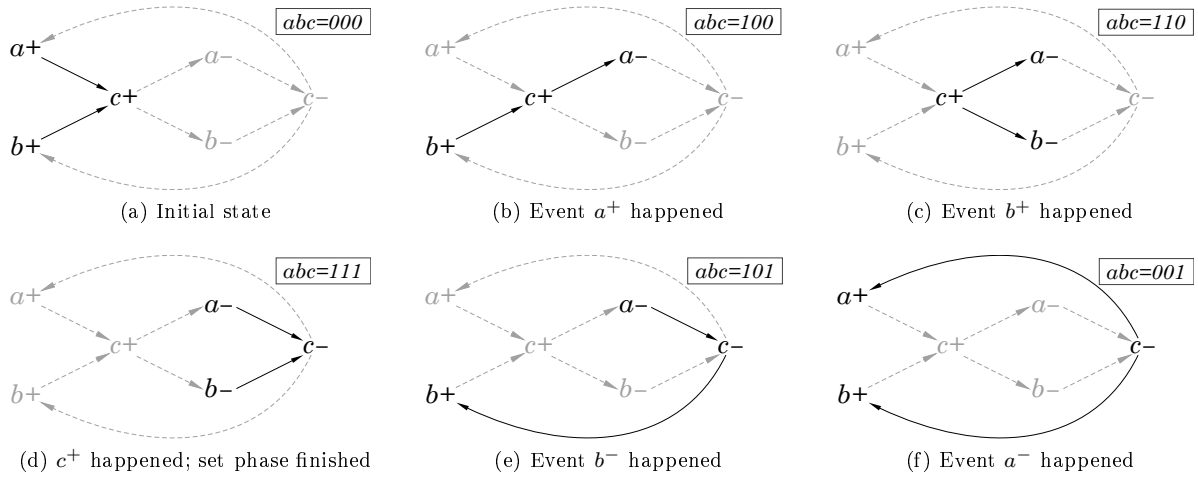
Figure 2: C-element with simple environment

Example 2. Consider a C-element with a simple environment constructed from two inverters as shown in Fig. 2(a). Initially both inverters are excited. As soon as they fire (concurrent events a^+ and b^+), the C-element becomes enabled leading to c^+. The reset phase is symmetric as captured by the STG in Fig. 2(b). The CSG specification happens to be isomorphic again – see Fig. 3(c); notice the initial state shown in the box above the diagram ($abc = 000$). The difference between the STG and CSG diagrams is manifested only in how they define the initial state of the circuit: a set of tokens (i.e. a *marking*) versus a Boolean vector. We believe the latter to be more natural because it directly corresponds to a circuit state; tokens, on the other hand, do not exist in circuits, rather they correspond to cuts in a circuit state space.

Let us simulate dynamic behaviour of the circuit using the CSG specification. Initially, fragment $\begin{smallmatrix} a^+ \to \\ b^+ \to \end{smallmatrix} c^+$ is active, so events a^+ and b^+ are allowed to happen (see Fig. 3(a), note that the active elements

[1] This is a purely digital construction. In practice at least three inverters must be connected in a ring to build a functioning oscillator.

[2] This assumption is similar to the STG property called *consistency* [4].



(a) Initial state (b) Event a^+ happened (c) Event b^+ happened

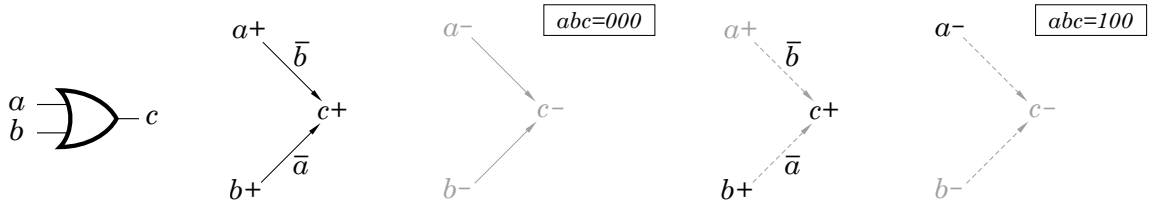
(d) c^+ happened; set phase finished (e) Event b^- happened (f) Event a^- happened

Figure 3: Simulation of CSG specification; inactive vertices and arcs are drawn grey and dashed.

are highlighted). Assuming a^+ fires first, the circuit goes to the next state shown in Fig. 3(b). As $abc = 100$ the new active fragment is $b^+ \to c^+ \to a^-$, thus the only enabled event is b^+. After it happens, the C-element becomes enabled and its firing completes the set phase of the circuit bringing it to state $abc = 111$, see Fig. 3(d). The reset phase is similar. If b^- fires first then fragment $a^- \to c^- \to b^+$ becomes active (Fig. 3(e)); a^- follows bringing the circuit to state $abc = 001$ shown in Fig. 3(f). Finally, event c^- returns the circuit to the initial state.

Although the STG and CSG specifications describe the same circuit behaviour and have isomorphic graphical representations, their firing rules are dramatically different as demonstrated above. While an STG is simulated by moving tokens around, a CSG is simulated by shifting a 'window of activity' (an acyclic fragment of system behaviour) which contains all causal dependencies relevant to the current state. In an STG an event is enabled when its every incoming arc contains a token; in a CSG an active event is enabled when it has no causal predecessors in the currently active fragment.

Example 3. The last example concerns specification of *OR-causality* [16] with CSGs. Figure 4(a) shows an OR gate whose environment has no constraints, i.e. input signals a and b can change unpredictably and we have to specify behaviour of output signal c. A CSG specification of the circuit is given in Figure 4(b). Note that we use conditional arcs $\phi(a^+ \to c^+) = \overline{b}$ and $\phi(b^+ \to c^+) = \overline{a}$ to model OR-causal enabling of event c^+. The reset phase has no arc conditions, because OR gates are asymmetric (the reset phase of an OR gate coincides with that of a C-element). The OR-causal behaviour is demonstrated in Figure 4(c) which shows the circuit after event a^+ has fired: event c^+ becomes enabled since b^+ is removed from its predecessors (arc $b^+ \to c^+$ is deactivated when $a = 1$).



(a) OR gate (b) CSG: set (active) and reset phases (c) a^+ has fired; c^+ is enabled (OR-causality)

Figure 4: OR gate with unconstrained environment

2.2 Synthesis of circuits from CSGs

An important property of the CSG model is that a CSG can be easily converted to a circuit and vice versa. For example, in Fig. 2(c) one can see that event c^+ is enabled when $a \cdot b = 1$, while c^- is enabled

when $\overline{a} \cdot \overline{b} = 1$. These equations are so called *set* and *reset functions* for signal c, denoted S_c and R_c, respectively. To synthesise an implementation for c we can use the well-known formula $c = S_c + c \cdot \overline{R_c}$ [4] leading to

$$c = a \cdot b + c \cdot \overline{\overline{a} \cdot \overline{b}} = a \cdot b + c \cdot (a + b)$$

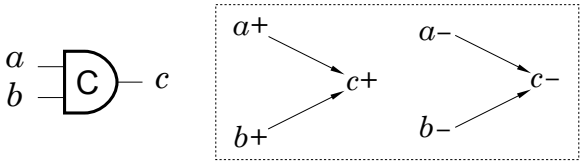
which is a standard C-element equation.

In general, set/reset functions for an arbitrary CSG can be obtained in the same way, details can be found in [12] which describes synthesis of circuits from Conditional Partial Order Graphs – the 'parent' model for CSGs.

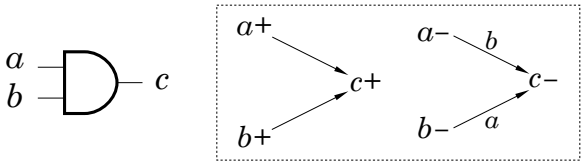
Building a CSG given a circuit is also straightforward: one has to decompose equations of each signal $s \in S$ into a pair of set/reset functions and connect events s^+ and s^- to the appropriate (conditional) predecessors. This equivalence between CSGs and circuits is very important for the methodology which we propose in this paper (Section 3).

2.3 CSG composition

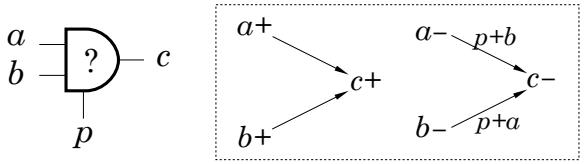
Another important feature of the CSG model is that two or more graphs can be composed into a single, *parametrized graph*, containing all of them. This feature is inherited from the CPOG model, which provides a complete framework for composition [12], encoding [11], and synthesis [9] of conditional (parametrized) *partial orders*. In this paper we extend the CPOG approach to the systems with cyclic behaviour, such as asynchronous circuits.



(a) C-element, equation: $c = a \cdot b + c \cdot (a + b)$



(b) AND gate, equation: $c = a \cdot b$



(c) Parametrized circuit, equation: $c = a \cdot b + p \cdot c \cdot (a + b)$

Figure 5: CSG composition of C-element and AND gate

Example 4. It is best to study CSG composition on a simple example. Fig. 5(a) shows a C-element with no environment and its CSG representation. Similarly, Fig. 5(b) shows an AND gate which differs from the C-element only in the reset phase: instead of waiting for both events a^- and b^- to happen, the AND gate waits only for one of them — this is another case of OR-causality. We use conditional arcs $\phi(a^- \to c^-) = b$ and $\phi(b^- \to c^-) = a$ to model OR-causal enabling of event c^-: as soon as a^- happens, arc $b^- \to c^-$ becomes inactive and c^- can happen without waiting for b^- (and symmetrically).

Let us denote the above CSGs as G_1 and G_2. Now we can compose them into a single graph G by

using CPOG multiplication and addition [12] operations:

$$G = p \cdot G_1 + \overline{p} \cdot G_2$$

where p is an auxiliary *parametric signal* which activates a particular graph in the composition as required, see Fig. 5(c): if $p = 1$ then G is equivalent to G_1 (the conditional arcs become unconditional: $1 + x = 1$), and if $p = 0$ then G is equivalent to G_2 (since $0 + x = x$). Now we can map G into a *parametrized circuit* $c = a \cdot b + p \cdot c \cdot (a + b)$, shown in Fig. 5(c), which behaves as a C-element or an AND gate depending on parameter p. It is important that the obtained composition captures similarities between the original graphs, in particular, their set phases are equal and as such do not require any parametrization. This leads to compact parametrized circuits. A poor alternative solution would duplicate the original circuits and then select the required output using a multiplexer controlled by p, thus ignoring their similar parts.

When more than two graphs are composed $\{G_1, \ldots, G_n\}$ it is not clear how to *encode* them, i.e. how to select a set of orthogonal encoding functions $\{f_1, \ldots, f_n\}$ to obtain the best composition G:

$$G = f_1 \cdot G_1 + f_2 \cdot G_2 + \cdots + f_n \cdot G_n$$

Fortunately, there are optimal encoding procedures developed for the CPOG model [11] which can be imported and applied to Conditional Signal Graphs.

3 Methodology

In this section we describe how the existing STG-based and CSG-based methods can be combined in order to automate specification and synthesis of parametrized controllers.

A designer is given a task: to design a controller managing interaction of components in a large system. Typically, there is a wide range of possible controller implementations with divergent characteristics in terms of latency, throughput, power consumption, robustness, etc. bearing a strong influence on the whole system. In general it is impossible to make a satisfactory decision at the design time by picking just one of the options and discarding the others, because a system can operate in varying environmental conditions and under different contradictory requirements. Therefore some form of run-time adaptability is required.

Fig. 6 shows the proposed design flow for specification and synthesis of parametrized adaptable controllers. The designer selects a set of controller implementations covering the required design space. Asynchronous implementations are formally described in the widely adopted STG model using, for example, Workcraft modelling environment [1]. The STGs are then synthesised by Petrify [3] or Punf/Mpsat [7] synthesis tools. The obtained circuits are converted into CSGs and they are encoded and composed into a single parametrized graph, which is then mapped into the final parametrized controller. This stage will be demonstrated in Section 4.

The proposed flow can also be applied to the conventional design practices for synthesis of synchronous circuits. In this case an alternative design pathway is taken through the FSM-specification of the system, its RTL synthesis, e.g. using Synopsys Design Compiler [2], and subsequent conversion into CSGs for merging with the asynchronous pathway.

The synthesised parametrized controller is then passed to the standard technology mapping and place and route (P&R) tools which are outside the scope of this paper. It is worth mentioning though, that the parametric signals are very undemanding to P&R since it is assumed that they are rarely changed. Some of them may even stay constant throughout the system lifetime after they have been initially set up at the defect management and/or product binning stages.

It may seem possible to avoid using CSGs in the described design flow and perform all the transformations entirely within the STG modelling framework. However, there are several serious practical

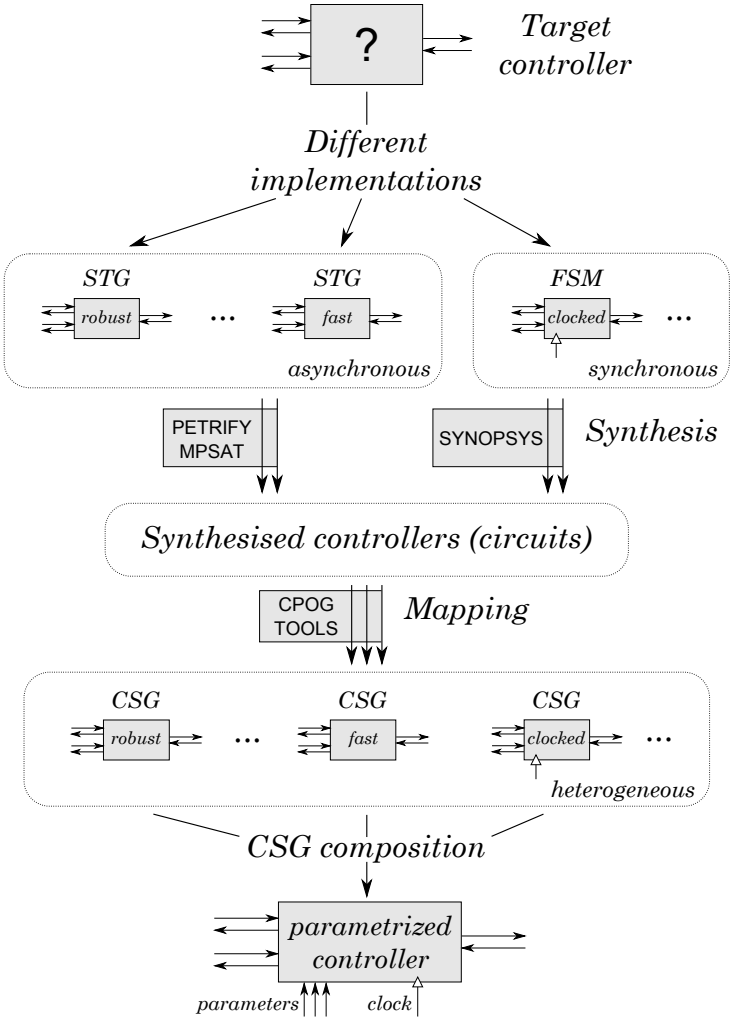


Figure 6: Design flow for parametrized controllers

difficulties for that:

- modelling parametric signals together with interface signals leads to combinatorial explosion of STG specifications as have been demonstrated in [12];
- there are no automated methods for generating optimal encoding of parametric signals in STGs;
- CSG composition is a fast structural operation while composing several STGs requires traversal of their combined state graph and that is computationally expensive.

On the other hand, it is also difficult to avoid using STGs, because specifying controllers directly in CSGs is a difficult task akin to manual circuit design. Therefore, we believe that the proposed methodology should make use of both models by exploiting their advantages and avoiding drawbacks.

4 Case studies

In this section we demonstrate the proposed methodology on two examples: a common Write/Read controller and a basic data path cell (an AND gate), both adaptable to operating conditions by logic parametrization. The obtained parametrized controllers are evaluated in Section 5.

4.1 Write/Read controller

Fig. 7 shows a system comprised of a *writer* (a component that upon request r writes a data frame to a register in the data path and acknowledges it with signal a), three independent *readers* (components that upon request r_i read the data frame, process it, and respond with a_i), and a Write/Read controller managing their interaction. We assume that the two-phase handshake protocol [14] is used, and the component delays are denoted as d (the writer) and $d_1 - d_3$ (the readers). Let us explore several possible implementations for the controller.

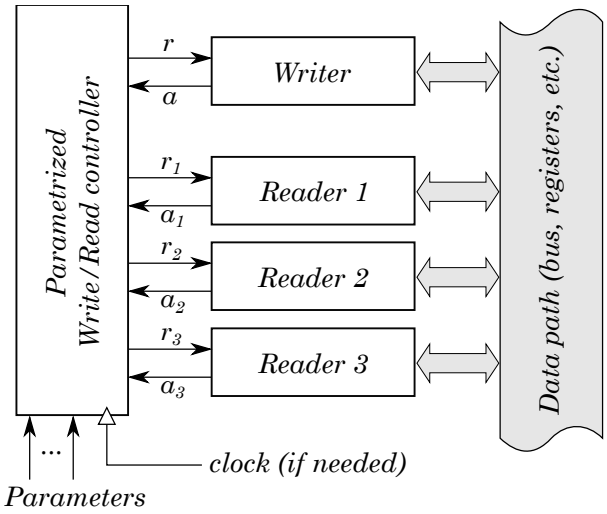
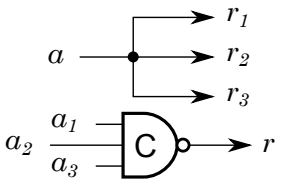


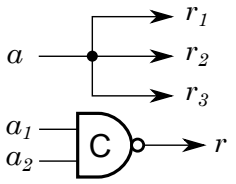
Figure 7: Write/Read controller: system-level view

Delay insensitive (DI), Fig. 8(a), is the most robust implementation; it will activate the components in the proper order regardless of their delays and under extreme operating conditions (e.g. low and/or unstable power supply). The cycle-time of this solution is rather long: $d + max\{d_1, d_2, d_3\} + C_3$, where C_n stands for delay of an n-input C-element.

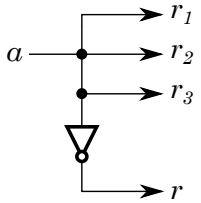
To improve the cycle-time a designer can replace (some of) the explicit causal relations by implicit timing assumptions, lowering the robustness as an unavoidable side effect.



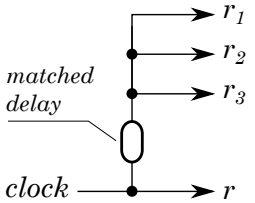
(a) Delay insensitive (DI)



(b) Partial acknowledgement (PA)



(c) Timing assumptions (TA)



(d) Synchronous (clocked, CL)

Figure 8: Different Write/Read controller implementations

An implementation with **partial acknowledgement (PA)** is shown in Fig. 8(b). In this setting we discard the acknowledgement signal a_3, simplifying the controller and reducing the cycle-time to $d + max\{d_1, d_2\} + C_2$. Possible reasons to discard a_3: i) reader 3 is faster than the others, so we should not normally wait for it; ii) the reader is unimportant, so we do not actually care if it misses the current

data frame; iii) the reader is faulty, so we do not wait for it to avoid deadlock; iv) the reader is switched off (due to, e.g., power management).

If it is known that the writer is slower than the readers, i.e. $d > d_i$, then it is possible to make a more aggressive optimisation of the cycle-time. The **timing assumption (TA)** solution, shown in Fig. 8(c), activates the writer and the readers in parallel, assuming that the readers can finish their task before the writer produces a new data frame. The cycle-time is only $d + I$, where I stands for delay of the inverter.

Finally, one can use a **synchronous clocked (CL)** solution, Fig. 8(d), which avoids any latency penalties by discarding *all* causal information (signals a, $a_1 - a_3$) and using matched delay lines instead. Its cycle-time is equal to the chosen clock period t_{clock}. However, one should remember that for correct operation t_{clock} must exceed the worst-case delays d and $d_1 - d_3$, thus when delay variability is high (data dependency, low voltage, etc.) the synchronous solution becomes inefficient.

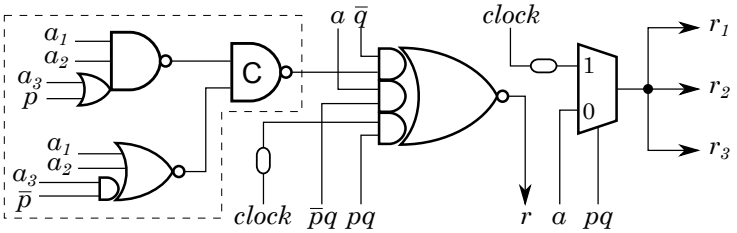


Figure 9: Parametrized Write/Read controller

Implementations in Fig. 8 can be converted to CSGs and composed into a single parametrized circuit, shown in Fig. 9 (necessary decomposition and negative logic optimisation have been performed). Note that the common logic is not duplicated: there is only one C-element (shown outlined) which works either in the 2-input or in the 3-input mode[3]; the fork issuing request signals $r_1 - r_3$ is also shared by all four modes. Parametric signals p and q have been added to activate the required implementation in run-time: combination $(p, q) = (0, 0)$ selects the DI mode, while encodings 01, 10, and 11 select the TA, PA, and CL modes, respectively.

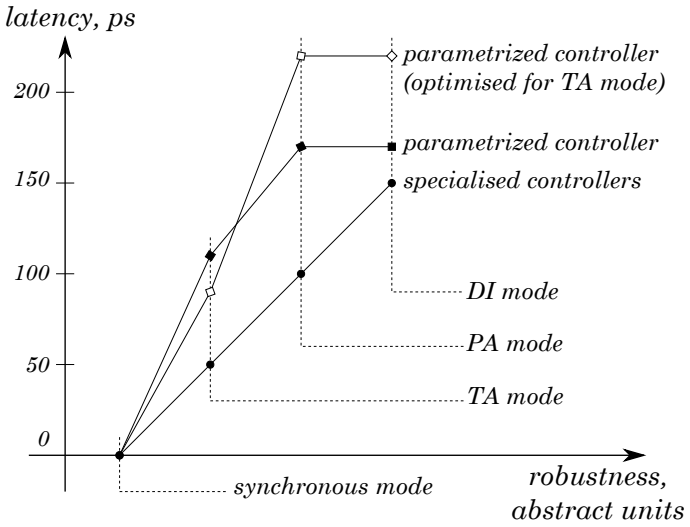


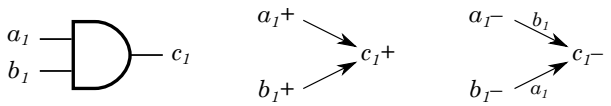
Figure 10: Comparison of Write/Read controllers

Fig. 10 shows a diagram comparing all the described implementations of the Write/Read controller against each other in terms of latency and robustness. See Section 5 for a further discussion.

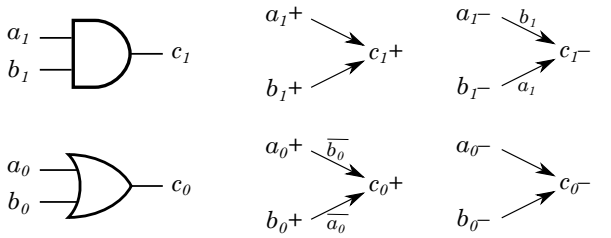
[3] In a standard gate library, a 3-input C-element can be implemented using a 3-input AND/OR pair and a 2-input C-element mappable to a majority gate.

4.2 Parametrized AND gate

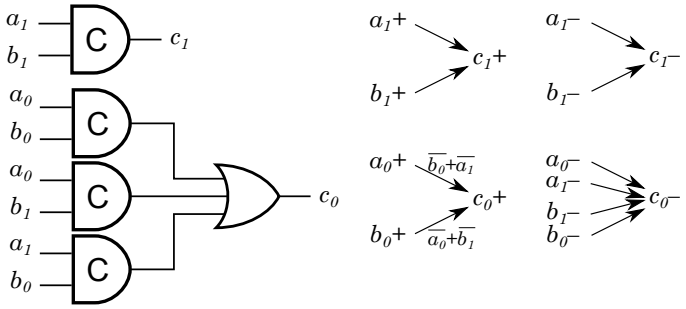
Interestingly, one can apply the same methodology to data path circuits. For example, Fig. 11(a-c) shows three well-known implementations of an AND gate: single rail (SR), dual-rail NCL-X [8], and dual-rail NCL-D [6]. All are shown together with their CSG representations which can be encoded and merged into a parametrized CSG shown in Fig. 11(d). As a result signal c_1 is implemented as an AND/C-element described in Section 2-C, while signal c_0 has a more complicated equation being a combination of the NCL implementations.



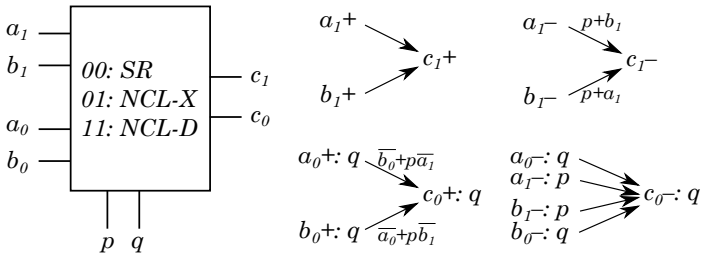
(a) Single rail implementation



(b) dual-rail, NCL-X style (early propagative)



(c) dual-rail, NCL-D style (strongly indicating)



(d) Parametrized controller (SR/NCL-X/NCL-D combined)

Figure 11: CSGs for parametrized AND gate

From the practical point of view such data path circuits may seem too heavy for general use, however, they might find an application in the class of systems requiring extreme robustness of NCL-D data path and still capable of running in a fast single rail clocked mode (albeit with 89% extra latency, see Table 1). Regarding power consumption one should note that unused parts of such data cells may be disconnected from power supply through power gating.

5 Conclusions and analysis of benchmarks

The benchmarks discussed in this paper were implemented in UMC 90nm technology using Faraday gate-level library. Latency of the controllers was measured under normal operating conditions using Synopsys PrimeTime [2].

Our first benchmark is the Write/Read controller: four 'specialized' and two parametrized implementations of the controller were analysed, see Table 1. The first parametrized version has been presented in Section 4-A, while the second has been obtained from it by optimising the path activated in the DI mode, thus trying to make it work faster in this mode by sacrificing performance in the other modes.

The results confirm the intuitive assumption that a specialized implementation has a lower latency than a generic design working in the corresponding mode. However, in case of the second version of the parametrized controller the penalty is only 20% w.r.t. to the specialized DI circuit and there is no extra delay in the synchronous mode of operation[4]. This means that we can run the system in a fast clocked mode with no penalty in terms of latency and still have a robust DI solution available as a 'backup' with only 20% latency increase.

In other cases the penalty is more significant, e.g. in the TA mode the parametrized controller exhibits 71% higher latency than the specialized circuit. However, this latency is still much lower than that of the specialized DI controller, thus justifying this mode of operation. Moreover, remember that global system cycle-times will differ much more significantly, namely 87.1ps $+ d$ vs 148.1ps $+ d + max\{d_1, d_2, d_3\}$. This system-level reasoning can justify *all* other penalties.

Benchmark	Controller	Mode	Latency, ps	Overhead
Write/ Read	specialized	DI	148.1	-
		PA	97.1	-
		TA	50.9	-
		CL	$0^{(*)}$	-
	parametrized	DI	233.3	57.7%
		PA	234.1	141.0%
		TA	87.1	71.1%
		CL	$0^{(*)}$	-
	parametrized opt. for DI	DI	177.9	20.1%
		PA	178.8	84.1%
		TA	113.1	122.2%
		CL	$0^{(*)}$	-
AND gate	specialized	SR	42.5	-
		NCL-X	86.5	-
		NCL-D	209.8	-
	parametrized	SR	80.2	88.7%
		NCL-X	233.4	169.8%
		NCL-D	317.6	51.4%
AND/C-element	specialized	AND	42	-
		C	74.7	6.1%
	parametrized	AND	70.4	67.6%
		C	70.4	-

Table 1: Summary of simulation results

The second benchmark is the AND gate example. Note that both NCL implementations are based on dual-rail encoding and require a codeword/spacer switching protocol [5]. Therefore their cycle delay is a combined latency of two phases: switching from a codeword to spacer and backwards. Similarly to the previous benchmarks, the configurable implementation trades the circuit latency for flexibility and suffers from 89% (in case of single-rail) up to 170% (in case of NCL-X) delay penalties compared to the specialized implementations.

Finally, the table shows analysis of the AND/C-element cell which, surprisingly, can work even slightly faster than a dedicated C-element due to peculiar transistor-level effects (this is a pure coincidence). The AND/C-element is still 68% slower than an AND gate; a careful transistor-level implementation can, however, eliminate this gap.

[4],(*) In the synchronous mode any controller latency can be compensated by adjusting (shifting) the phase of the clock signal.

To conclude, parametrized controllers, although suffering from natural penalties due to generality, are justified by their adaptability to wide range of operating conditions. The proposed methodology thus paves the way to both power-proportional and power-efficient systems, which combine advantages of heterogeneous timing and power domains.

Acknowledgement

This work was supported by EPSRC grants EP/G037809/1 and EP/F016786/1.

References

[1] The Workcraft framework homepage. http://www.workcraft.org, 2009.

[2] H. Bhatnagar. *Advanced ASIC chip synthesis: using Synopsys Design Compiler, Physical Compiler, and PrimeTime.* Kluwer Academic Publishers, 2002.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[4] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. *Logic synthesis of asynchronous controllers and interfaces.* Advanced Microelectronics. Springer, 2002.

[5] K. Fant and S.A. Brandt. Null Conventional Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis. In *Int'l Conf. Application-Specific Systems, Architectures, and Processors*, 1996.

[6] Cheoljoo Jeong and Steven M. Nowick. Technology mapping and cell merger for asynchronous threshold networks. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(4):659–672, 2008.

[7] Victor Khomenko, Maciej Koutny, and Alexandre Yakovlev. Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT. In *Proc. of International Conference on Applicatioon of Concurrency to System Design (ACSD'04)*, pages 16–25, 2004.

[8] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous CAD tools. *IEEE Transactions on Computers*, 2002.

[9] Andrey Mokhov. *Conditional Partial Order Graphs.* PhD thesis, Newcastle University, 2009.

[10] Andrey Mokhov. Modelling cyclic system behaviour with CPOGs, Technical Memo. Newcastle University (2011).

[11] Andrey Mokhov, Arseniy Alekseyev, and Alex Yakovlev. Automated synthesis of instruction codes in the context of micro-architecture design. In *Proceedings of International Conference on Applicatioon of Concurrency to System Design (ACSD'10)*, pages 3–12, 2010.

[12] Andrey Mokhov and Alex Yakovlev. Conditional Partial Order Graphs: Model, Synthesis and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[13] Steven Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers.* PhD thesis, Stanford University, 1993.

[14] Jens Sparsø and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001.

[15] A. Yakovlev. Energy-modulated computing. In *Design, Automation and Test in Europe (DATE) Conference*, pages 1340–1345, 2011.

[16] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with OR causality. *Formal Methods in System Design*, pages 189–234, 1996.