Microelectronics System Design Research Group
School of Electrical, Electronic & Computer Engineering
Merz Court
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Modelling cyclic system behaviour with CPOGs

Andrey Mokhov

andrey.mokhov@ncl.ac.uk

NCL-EECE-MSD-MEMO-2011-003

January 2011

Abstract

The Conditional Partial Order Graph (CPOG) model was initially built around finite sequences of events, later generalised to partial orders. The only way to model behaviour of a cyclic system was to assume that the environment restarts the system after it completes execution of the current partial order. This assumption was crucial to create a simple yet powerful model for specification, verification, synthesis and reasoning on a large set of finite procedures.

Lack of a direct way to represent cyclic behaviour with CPOGs restricted applicability of the model and incited a multitude of ad-hoc workarounds which could not be easily generalised. In particular, although many control circuits synthesised from CPOGs happened to be speed-independent (SI) it was conceptually impossible to prove this within the model itself, thus Signal Transition Graphs (STGs) were employed to formally prove SI and other important properties.

This memo presents a new interpretation of CPOG dynamics which allows explicit specification of cyclic system behaviour. It is important that the interpretation does not require any changes in the model definition; it merely associates events with signal transitions and assigns certain conditions to them, similar in spirit to the way STGs are built on the basis of Petri nets. This analogy is reflected in the name nominated for the interpretation – Conditional Signal Graphs.

1 Introduction

This section briefly introduces Conditional Partial Order Graphs; if the reader is familiar with them he or she can safely go directly to Section 2.

The Conditional Partial Order Graph (CPOG) model [6][7] has been introduced as an alternative to Petri nets and Finite State Machines for specification and synthesis of asynchronous microcontrollers. The key features of the model are: ability to describe systems in a compact functional form, and structural synthesis methods which significantly improve performance of the whole design flow. These features make the model very efficient for representation and management of causal information in hardware and EDA software. A CPOG is a superposition of a set of partial orders which can be extracted from it by providing the corresponding codewords, see Figure 1 (centre). It can be regarded as a custom associative memory for storing cause and effect relations within a predefined set of events.

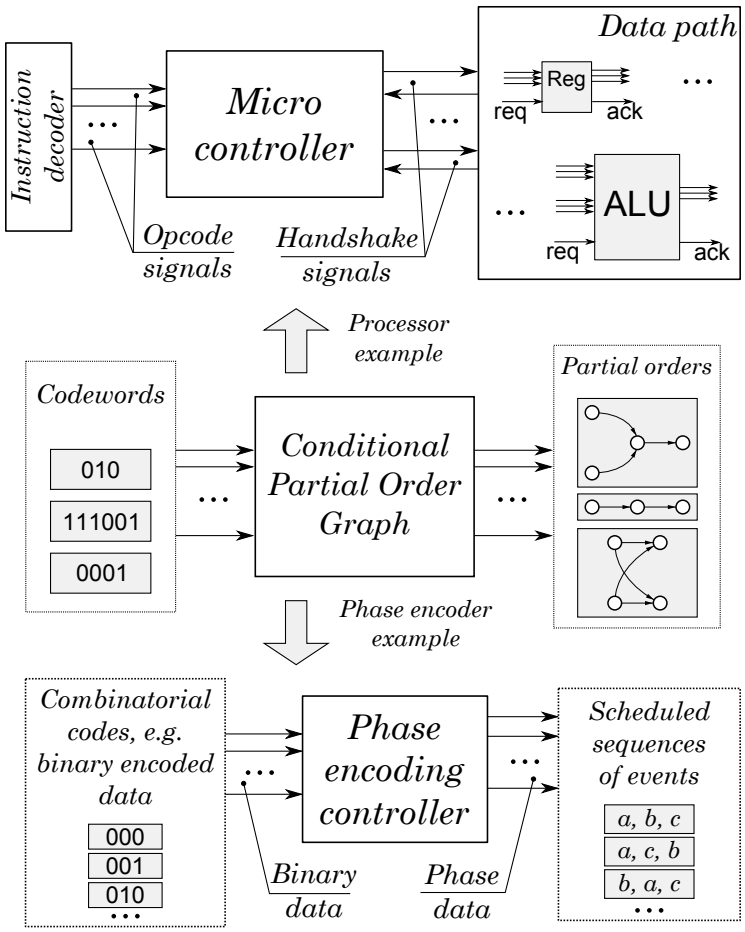
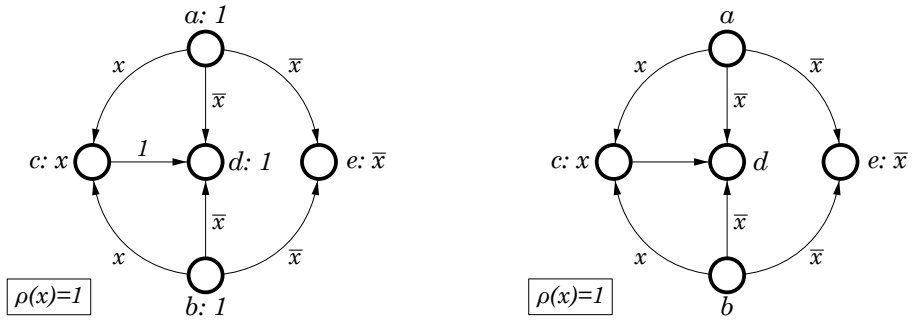


Figure 1: Examples of dynamically reconfigurable microcontrollers

There are different kinds of systems which can be described with the model. For example, a CPU microcontroller executes partial orders (or *instructions*) of primitive computational steps (or *microinstructions*) defined on a set of data path operational units, see Figure 1 (top). The order is determined by an *instruction code* — a combination of logical conditions presented to the controller by the environment [5]. To this end,

the microcontroller can be seen as an entity which communicates with two parts of the environment: one part is the source of condition signals (an instruction decoder) and the other part is a set of controlled objects with request-acknowledgement interface (data path operational units which execute the microinstructions). Thus the condition signals dynamically reconfigure the microcontroller according to the instruction being executed. *Microcoded control synthesis* presented in [5] is applicable to this class of systems, however, it is based on synchronous FSMs and stores the event orders separately in look-up tables, thus having a limited degree of parallelism and certain area penalties.

Formally, *Conditional Partial Order Graph* is a quintuple $H = (V,\ E,\ X,\ \rho,\ \phi)$, where V is a finite set of *vertices*, $E \subseteq V \times V$ is a set of *arcs* between them, and X is a finite set of *operational variables*. An *opcode* is an assignment $(x_1,\ x_2,\ \ldots,\ x_{|X|}) \in \{0,\ 1\}^{|X|}$ of these variables; X can be assigned only those opcodes which satisfy the *restriction function* ρ of the graph, i.e. $\rho(x_1,\ x_2,\ \ldots,\ x_{|X|}) = 1$. Function ϕ assigns a Boolean *condition* $\phi(z)$ to every vertex and arc $z \in V \cup E$ of the graph.



(a) Full notation (b) Simplified notation

Figure 2: Graphical representation of CPOGs

CPOGs are represented graphically by drawing a labelled circle ○ for every vertex and drawing a labelled arrow ⟶ for every arc. The label of a vertex v consists of the vertex name, semicolon and the vertex condition $\phi(v)$, while every arc e is labelled with the corresponding arc condition $\phi(e)$. The restriction function ρ is depicted in a box next to the graph; operational variables X can therefore be observed as parameters of ρ. Figure 2(a) shows an example of a CPOG containing $|V| = 5$ vertices and $|E| = 7$ arcs. There is a single operational variable x; the restriction function is $\rho(x) = 1$, hence both opcodes $x = 0$ and $x = 1$ are allowed. Vertices $\{a,\ b,\ d\}$ have constant $\phi = 1$ conditions and are called *unconditional*, while vertices $\{c,\ e\}$ are *conditional* and have conditions $\phi(c) = x$ and $\phi(e) = \bar{x}$ respectively. Arcs also fall into two classes: *unconditional* (arc $(c,\ d)$) and *conditional* (all the rest). As CPOGs tend to have many unconditional vertices and arcs we use a simplified notation in which conditions equal to 1 are not depicted in the graph. This is demonstrated in Figure 2(b).

The purpose of conditions ϕ is to 'switch off' some vertices and/or arcs in the graph according to the given opcode – this makes CPOGs capable of specifying multiple partial orders. Figure 3 shows a graph and its two *projections*. The leftmost projection is obtained by keeping in the graph only those vertices and arcs whose conditions evaluate to 1 after substitution of the operational variable x with 1. Hence, vertex e disappears (denoted as a dashed circle ◌), because its condition evaluates to 0: $\phi(e) = \bar{x} = \overline{1} = 0$. Arcs $\{(a,\ d),\ (a,\ e),\ (b,\ d),\ (b,\ e)\}$ disappear for the same reason (denoted as dashed arrows ⇢). The rightmost projection is obtained in the same way with the only difference that variable x is set to 0. Note also that

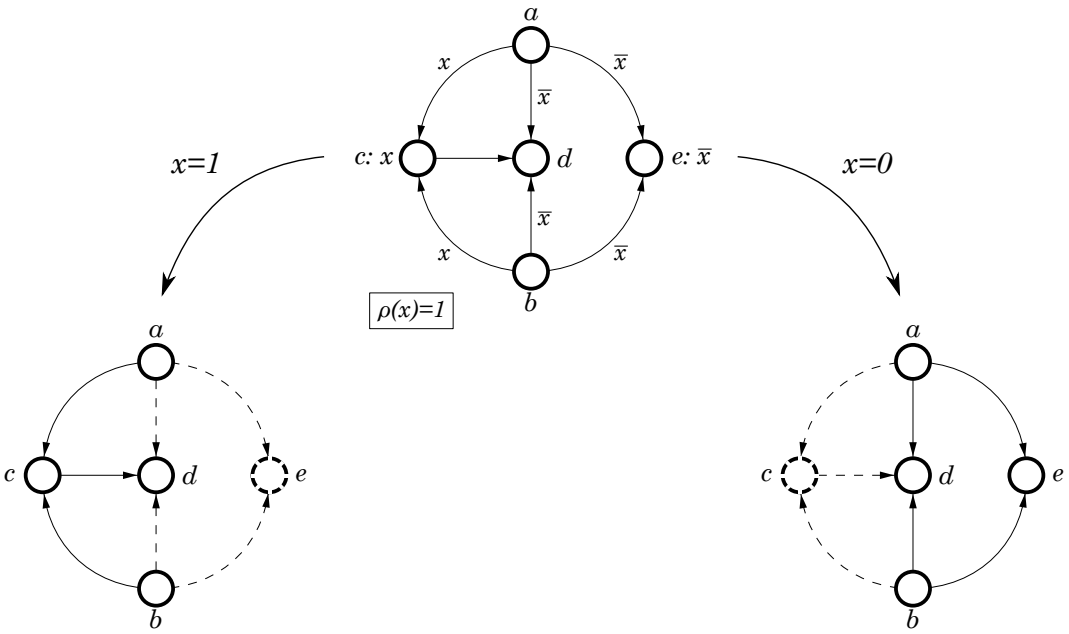


Figure 3: Multiple CPOG projections

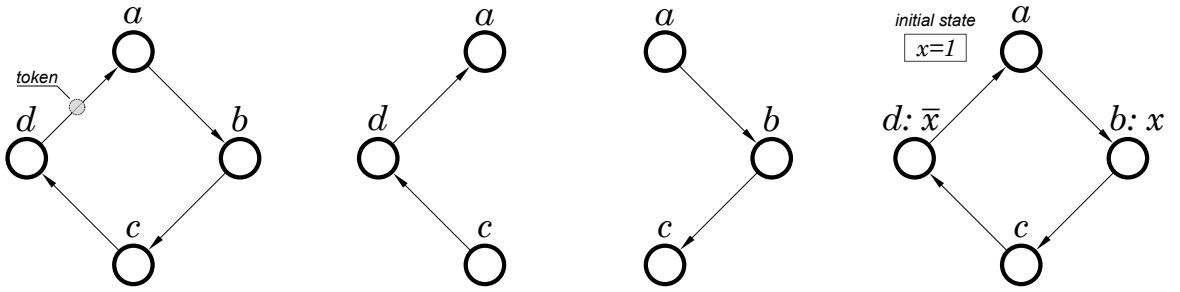
although the condition of arc $(c,\ d)$ evaluates to 1 (in fact it is constant 1) the arc is still excluded from the resultant graph because one of the vertices it connects (vertex c) is excluded and obviously an arc cannot appear in a graph without one of its vertices. Each of the obtained projections can be treated as a specification of a particular behavioural scenario of the modelled system. Potentially, a CPOG $H = (V,\ E,\ X,\ \rho,\ \phi)$ can specify an exponential number of different partial orders of events in V according to one of $2^{|X|}$ different possible opcodes.

To conclude, a CPOG is a structure to represent a set of encoded partial orders in a compact form. Synthesis and optimisation methods presented in [6][7] provide a way to obtain such a representation given a set of partial orders and their opcodes. For example, the CPOG in Figure 3 can be synthesised automatically from the two partial orders below it and the corresponding opcodes $x = 1$ and $x = 0$.

2 New interpretation

A Conditional Partial Order Graph is allowed to contain only acyclic objects – partial orders. The reason for that lies in the fact that a cyclic graph has no natural execution semantics, in particular it is not clear which event can be executed first unless some form of a 'token' is introduced as in the Petri net model [2]. In this work we propose an alternative approach wherein we interpret partial orders contained in a CPOG as acyclic *fragments* that together form behaviour of a cyclic system. The following example demonstrates this idea.

Consider a simple system with four events $V = \{a,\ b,\ c,\ d\}$, shown in Figure 4(a), having the infinite execution trace $a \to b \to c \to d \to a \to b \to c \to \ldots$ etc. We can highlight the fact that event a is executed first by putting a token on arc $d \to a$, thus transforming this simple cyclic diagram into a Petri net. Alas, however clear and natural this Petri net might be, it is still a cyclic object, so we cannot put it into the CPOG framework. Thereby, let us try to look at the given system from the partial order perspective.



(a) Simple cyclic system (b) Fragment $c \to d \to a$ (c) Fragment $a \to b \to c$ (d) Fragments merged into CPOG with auxiliary variable x

Figure 4: Example of a cyclic system, its acyclic fragments, and their CPOG composition

It is possible to break the cycle into two fragments[1] $c \to d \to a$ and $a \to b \to c$, see Figure 4(b, c). As the fragments are acyclic they can be merged into a CPOG using an auxiliary variable x as demonstrated in Figure 4(d). Two projections of the obtained CPOG coincide with the given fragments under opcodes $x = 0$ and $x = 1$, as expected. Instead of putting a token on one of the arcs, we can now specify the initial state of the system by selecting an initial value for variable x, e.g. $x = 1$ activates fragment $a \to b \to c$ in which event a happens first. As the system should not stop after executing only these three events, we have to change from one fragment to another in a seamless manner[2]. This is not possible in the traditional interpretation of CPOG dynamics, when upon completion of partial order $a \to b \to c$ the environment must change variable x and restart the system. To withdraw the environment from active participation in the cyclic process, let us assign action 'x^-' (i.e. resetting variable x to zero) to event b, and action 'x^+' (setting x to one) to event d. The execution will then go through the following steps:

1. In the initial state $x = 1$ and fragment $a \to b \to c$ is activated. Hence event a is executed first.
2. As x has not changed, the system moves on according to the active fragment and event b is executed. At the same time variable x is reset to zero and the system finds itself in fragment $c \to d \to a$.
3. We have $x = 0$ and the active fragment is $c \to d \to a$, so event c is executed.
4. Now event d is executed and x is set to one. The system returns to the initial state and the process continues forever.

It is interesting to compare this approach of modelling cyclic system behaviour with the widely adopted STG-based methodology [2]. A Signal Transition Graph (STG) is a Petri net whose transitions correspond to changes of signals in a digital circuit. Similarly, we can associate every event in a CPOG with a rising (s^+) or falling (s^-) change of a signal s. Such annotated CPOGs can be called *Conditional Signal Graphs* (CSGs). The next section compares STGs and CSGs on a set of examples.

[1] There are many ways to break the cycle into two (or more) acyclic fragments; we've shown only one of them here.

[2] Intuitively, this is very similar to changing from one metro line to another: if there is no direct path between two metro stations then an indirect path can be constructed from fragments spread over two or more different metro lines.

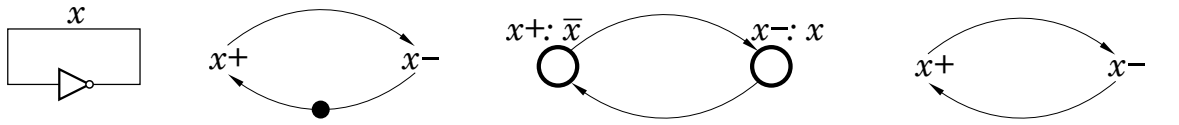
3 Case studies

In this section we study several asynchronous circuits described using the STG model and provide their equivalent CSG specifications. Our initial basic definition of CSGs is as follows.

A *Conditional Signal Graph* (CSG) is a CPOG $H = (V,\ E,\ X,\ \rho,\ \phi)$ defined on a set of signals S, such that V is a set of all signal transitions $V = S \times \{+,-\}$ and all signals are contained in the set of operational variables, i.e. $S \subseteq X$. Thus, a signal $s \in S$ can appear in a CSG in two different roles: as an event (e.g. s^+) and as a vertex/arc condition (e.g. $\phi(z) = \bar{s}$ for some $z \in V \cup E$).

3.1 Oscillator

Let us start with one of the simplest cyclic circuits – a 1-inverter ring oscillator[3]. The circuit and its STG specification are shown in Figure 5(a, b). The circuit behaviour is very simple: events x^+ and x^- occur alternatingly starting from the initial state when $x = 0$.



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

Figure 5: 1-inverter ring oscillator (initially $x = 0$)

Figure 5(c) shows a CSG specification of the oscillator. In the initial state only event x^+ is switched on and nothing prevents it from happening. As soon as it does happen, the circuit state changes to $x = 1$ in which event x^+ is switched off while event x^- becomes enabled, etc. In general, it is reasonable to assume[4] that for any signal $s \in S$ events s^+ and s^- are active only when $s = 0$ and $s = 1$, respectively, thus conditions $\phi(s^+) = \bar{s}$ and $\phi(s^-) = s$ may be omitted in a diagram for clarity; see the simplified version of the specification in Figure 5(d). Note that arcs $x^+ \to x^-$ and $x^- \to x^+$ are redundant (in fact, they are always switched off as at most one vertex can be active at any time), so they can also be removed from the diagram.

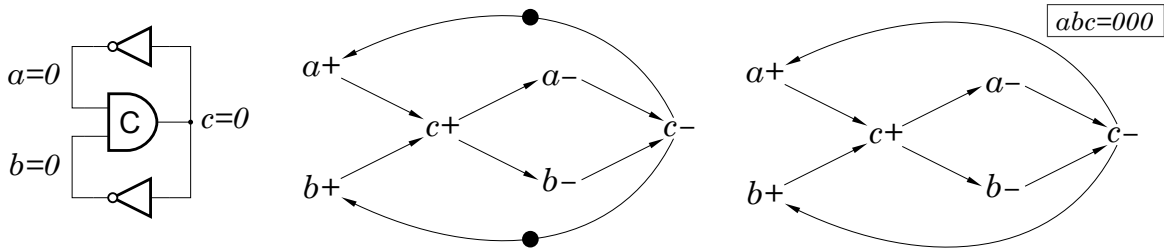
The only visual difference between specifications in Figure 5(b) and Figure 5(d) is that the latter is 'token-free'. The fundamental difference between STGs and CSGs is deeper and it will become more obvious from further examples. Interestingly, there are still many cases when a CSG specification is isomorphic to its STG equivalent as demonstrated by the next case.

3.2 C-element

Consider a C-element with a simple environment constructed from two inverters as shown in Figure 6(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 Figure 6(b). The CSG specification happens to be isomorphic and is presented in Figure 6(c); notice the initial state shown in the box above the diagram ($abc = 000$). The difference between the two 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.

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

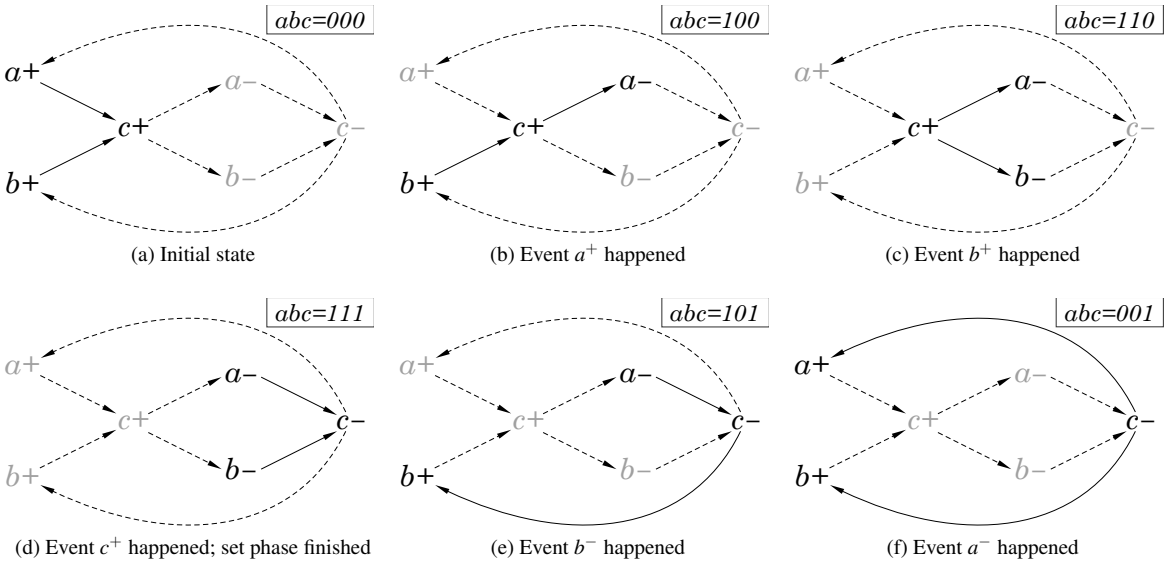
[4] Note that this assumption is similar to the STG property called *consistency* [2].



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

Figure 6: C-element with simple environment

Let us simulate dynamic behaviour of the circuit using the CSG specification. Initially, fragment $\begin{smallmatrix} a^+\rightarrow \\ b^+\rightarrow \end{smallmatrix} c^+$ is active, so events a^+ and b^+ are allowed to happen, see Figure 7(a). Assuming a^+ fires first, the circuit goes to the next state shown in Figure 7(b). As $abc = 100$ the new active fragment is $b^+ \rightarrow c^+ \rightarrow a^-$, 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 $abc = 111$ (Figure 7(d)). The reset phase is similar. If b^- fires first then fragment $a^- \rightarrow c^- \rightarrow b^+$ becomes active, see Figure 7(e); a^- follows bringing the circuit to state $abc = 001$ (Figure 7(f)). Finally, event c^- returns the circuit to the initial state.



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

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

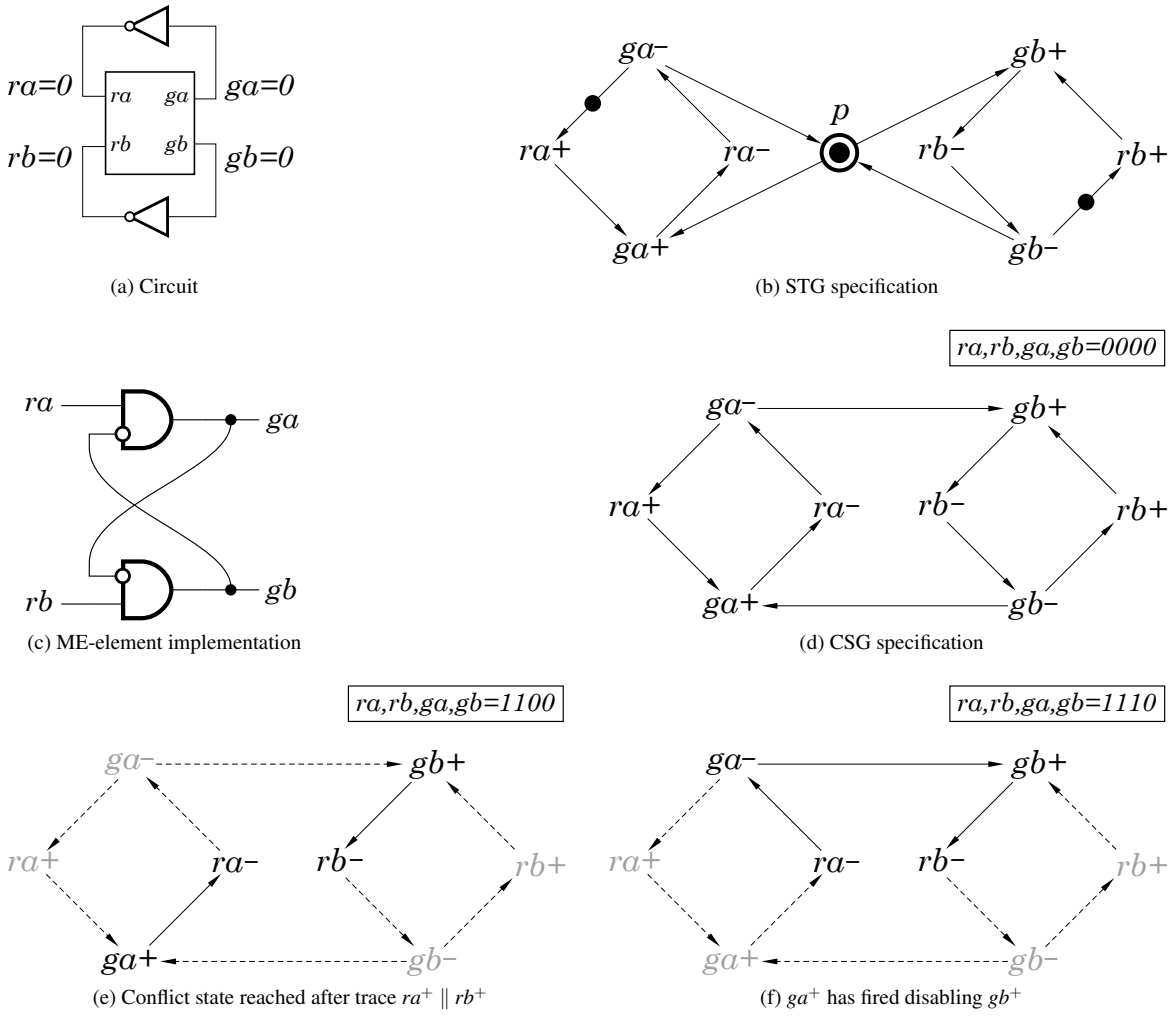
Figure 7: Simulation of CSG specification; inactive vertices and arcs are drawn grey and dashed, respectively.

Although the STG and CSG specifications shown in Figure 6(b, c) 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 event is enabled when it has no causal predecessors in the currently active fragment.

In general STGs and CSGs are not isomorphic as demonstrated in the next example.

3.3 Arbitration

A *mutual exclusion* (ME) element is a basic arbitrating circuit with two request inputs $(ra,\ rb)$ and two grant outputs $(ga,\ gb)$. An output can become high only upon receipt of the corresponding request, and at most one of the outputs can be high at any moment of time [4]. Let us study behaviour of an ME-element placed in a simple environment, see Figure 8(a). An STG specification of the circuit is shown in Figure 8(b); it consists of two request/grant loops synchronised on place p with a single token.



(a) Circuit

(b) STG specification

(c) ME-element implementation

(d) CSG specification

(e) Conflict state reached after trace $ra^+ \parallel rb^+$

(f) ga^+ has fired disabling gb^+

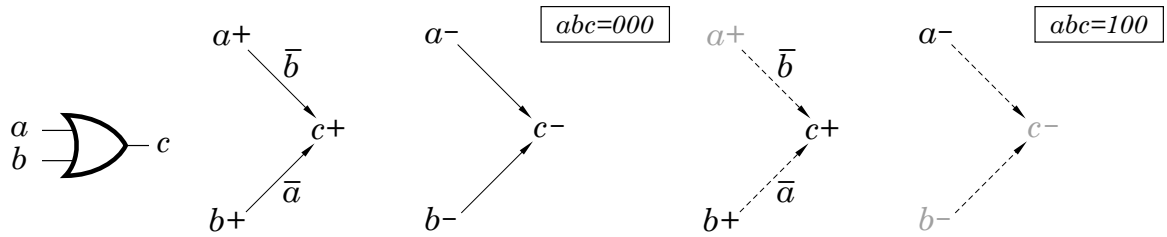
Figure 8: ME-element with simple environment

Figure 8(d) presents a CSG specification of the circuit. It is similar to the STG but the conflict between the grant outputs is modelled in a different way. The conflict state is reached when the environment generates two requests ra^+ and rb^+ so the ME-element has to decide which grant to issue – see Figure 8(e). Both outputs ga^+ and gb^+ are enabled in this state, however firing of any of them disables the other one. For example, if ga^+ fires first then the circuit comes to the state shown in Figure 8(f), where the active fragment is $ra^- \rightarrow ga^- \rightarrow gb^+ \rightarrow rb^-$, i.e. gb^+ is delayed until grant ga is removed.

It is interesting to note relation between the CSG specification and a possible implementation of an ME-element shown in Figure 8(c): event ga^+ has two incoming arcs from events ra^+ and gb^-, conformably the set function of gate ga is equal to $ga\uparrow = ra \cdot \overline{gb}$. We believe that a CSG specification of a circuit can be derived from set/reset functions of its signals and vice versa. This emphasises the fact that CSGs are much closer to circuits than STGs. In particular, it means that a CSG can be synthesised from an STG specification of a circuit using PETRIFY [1] or MPSAT [3] tool.

3.4 OR-causality

The last example concerns specification of OR-causality [8] with CSGs. Figure 9(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 9(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^+: $c\uparrow = a + b$. The reset phase has no arc conditions, because OR gates are asymmetric and $c\downarrow = \overline{a} \cdot \overline{b}$. The OR-causal behaviour is demonstrated in Figure 9(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 disabled by $a = 1$).



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

Figure 9: OR gate with unconstrained environment

4 Conclusions

In this work we presented a new approach to specification of cyclic systems using Conditional Signal Graphs which are CPOGs whose vertices are labelled with signal transitions. Several example circuits have been studied in order to compare the approach with widely adopted STG-based modelling techniques.

The future research work consists of providing a formal definition of CSGs, developing software tools for their synthesis and simulation, and applying the presented ideas to compositional CPOG-based synthesis of asynchronous controllers.

Acknowledgements

This work was supported by EPSRC grants EP/G037809/1 (VERDAD) and EP/C512812/1 (NEGUS).

References

[1] 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.

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

[3] Victor Khomenko, Maciej Koutny, and Alexandre Yakovlev. Logic synthesis for asynchronous circuits based on petri net unfoldings and incremental sat. In *Proceedings of International Conference on Applicatioon of Concurrency to System Design (ACSD'04)*, pages 16–25, 2004.

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

[5] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

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

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

[8] 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.