School of Electrical, Electronic & Computer Engineering

Newcastle University

Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis

A. Mokhov and A. Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2007-119

April 2007

Contact:

Andrey.Mokhov@ncl.ac.uk

Alex.Yakovlev@ncl.ac.uk

Supported by EPSRC grant EP/C512812

NCL-EECE-MSD-TR-2007-119

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

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

Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis

A. Mokhov and A. Yakovlev

April 2007

Abstract

The paper introduces a new formal model for specifying control paths in the context of asynchronous system design. The model, called Conditional Partial Order Graph (CPOG), is capable of capturing concurrency and choice in a system's behaviour in a compact and efficient way. A problem of synthesis of a CPOG composition from a set of given CPOGs is formulated and solved in this paper in respect of a subclass of CPOGs that are partial orders. This problem can be extended to a more general class of CPOGs, which is subject to future research.

The presented model can be used for the specification of system behaviour and for synthesis of area-efficient dynamically reconfigurable controllers. The synthesis of a controller is based on a novel generic architecture, called Transition Sequence Encoder (TSE). The synthesized controllers are delay insensitive and thus very robust to parametric variations. The ideas presented in the paper can be applied for CPU control synthesis as well as for synthesis of different kinds of event-coordination circuits often used in data coding and communication in digital systems. A software tool for synthesis of self-timed circuits from CPOGs is implemented and tested on a set of benchmarks.

1 Introduction

Specification and synthesis of control circuits for systems of large complexity, such as CPU cores or on-chip routers to name but a few, remains to be a challenging problem due to inefficiency of the existing design process. Typically designers of systems of such a complexity, rely on the use of hardware description languages, such as Verilog and VHDL, and the use of RTL-based synthesis flow [6]. Within this conventional methodology, designers use finite state machines to capture control specifications. Since standard RTL flow supports a synchronous (i.e. globally clocked) design paradigm these techniques lead to synchronous FSMs for control logic. In asynchronous design there is a need for generic models which are able to capture concurrency and choice in systems with many *similar patterns* in behaviour. To date there are several design methodologies for asynchronous control logic, e.g. [12] and [10]. Some methods such as Tangram (or Haste) [13] and Balsa [2] use CSP-like HDL languages for system specification and syntax-direct translation for synthesis. They are not particularly suited well for control logic specification because they capture the entire design as a collection of processes and channels. Control is implicit in them. Other methods such as Burst-Mode FSMs [8], as well as Petri nets (PN) and STGs [11], are more suitable for control logic design because they capture concurrency at a very fine level of granularity. The latter produce circuits that are more compact and faster (e.g. in terms of latency) [10] compared to those derived from syntax-direct translations from HDLs. However, the synthesis methods for Burst Mode machines and PNs (or STGs) are typically targeted at controllers with a small number of choice options, where each option is rather unique. In many applications such as a CPU controller, the designer is often faced with a problem of modelling many different behavioural patterns, or event orders, defined on the same domain of operational units. For example, in designing a CPU core, such behavioral patterns can be constructed for instructions or groups of instructions (see Section 4.1). The control flow in the

execution of instructions is determined by the values of signals produced by the instruction operation decoder and hence is available to steer the control through a certain *partial order* of events associated with the activations of operational units. Applying Burst Mode machines or Petri nets to such systems would lead to the circuits that are area and performance inefficient due to their explicit notion of *control state transitions*. Such models perform explicit state tracking which requires significant amount of logic and internal memory resources.

In this work we tried to come up with a new model that would retain the advantages of the existing behavioural models Petri nets (or STGs) and FSMs. The former are advantageous for modelling a high degree of concurrency while the latter for choice. This model, called Conditional Partial Order Graph (CPOG), builds on the order relation between actions or events from a certain set, which is determined by the combination of logical conditions presented to the controller by the environment. To this end, the controller can be seen as an entity which communicates with two parts of the environment, one part is the source of logical condition signals (in the case of a CPU, an instruction operation decoder) and the other part is the a set of controlled objects with request-acknowledgement interface (operational units) (see Figure 1). Thus the condition signals dynamically reconfigure our controller according to the instruction being executed.

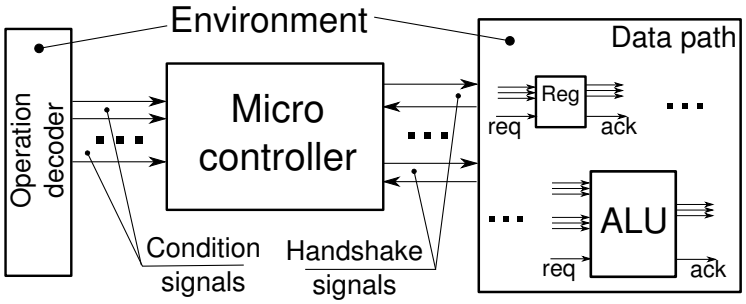


Figure 1: Dynamically reconfigurable controller

Bearing in mind the practical aspect of using such a model in designing real-life controllers, we believe that the model itself presents a source of interesting formalisation and automation problems, and to the best of our knowledge it is original and worth independent investigation.

2 Theoretical Background

The section introduces the basic notations, definitions and models that are used throughout the paper.

2.1 Partial order

A *partial order* $PO(S, R)$ is a binary relation R over a set of elements S which satisfies the following three conditions [3, 7]:

1. *Irreflexivity*: $\forall a \in S, \neg(aRa)$;
2. *Asymmetry*: $\forall a, b \in S, (aRb) \Rightarrow \neg(bRa)$;
3. *Transitivity*: $\forall a, b, c \in S, (aRb) \wedge (bRc) \Rightarrow (aRc)$.

Note that in some cases the partial order defined above is called a *strict* (or *irreflexive*) partial order. In these cases a *weak* (or *reflexive*) partial order is defined as a binary relation that is *reflexive* i.e. every element in S is R-related to itself: $\forall a \in S, aRa$. In the paper we focus only on strict partial orders and the qualifier *strict* will be omitted for clarity.

Partial orders are very natural for specification of order of events in a system when some of the events are constrained to happen before others. These constraints can be specified with partial order $PO(S, R)$

such that if aRb for some events $a, b \in S$ then event a must happen strictly before event b. If neither aRb nor bRa holds then the events a and b can happen in any order, possibly simultaneously.

2.2 Directed acyclic graphs

A *directed graph* is an ordered pair $G(V, E)$ where V is a set of *vertices* (or *nodes*) and $E \subseteq V \times V$ is the set of ordered pairs of vertices, called *arcs* [4, 7].

A sequence of vertices $(v_0, v_1, ..., v_n), v_k \in V, k = 0...n$ such that $(v_{k-1}, v_k) \in E, k = 1...n$ and $n \geq 0$ is called a *path* from v_0 (start vertex) to v_n (end vertex) and is denoted as $\langle v_0, v_n \rangle$. The set of all paths of a graph G is denoted as $\mathcal{P}(G)$. A *cycle* is a path $\langle v_0, v_n \rangle$ whose start and end vertices coincide: $v_0 = v_n$.

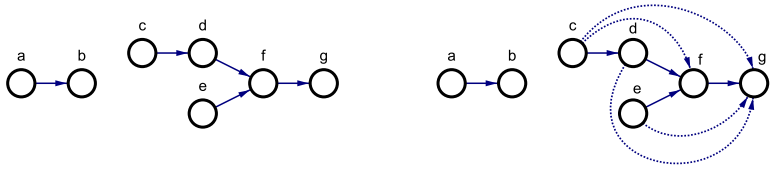
Directed acyclic graph (DAG) is a graph that does not contain any cycles.

An arc $(a, b) \in E$ of a graph $G(V, E)$ is called *transitive* iff $\exists v \in V \setminus \{a, b\}, \langle a, v \rangle \in \mathcal{P}(G) \wedge \langle v, b \rangle \in \mathcal{P}(G)$.

Transitive closure of a graph $G(V, E)$ is a graph $G*(V, E*)$ such that:

- $\forall a, b \in V, (a, b) \in E \Rightarrow (a, b) \in E*$;
- $\forall a, b, c \in V, (a, b) \in E* \wedge (b, c) \in E* \Rightarrow (a, c) \in E*$ (transitivity condition).

Figure 2 shows a DAG and its transitive closure. Transitive arcs are drawn dotted.



(a) initial DAG (b) transitive closure

Figure 2: DAG and its transitive closure

Note that there is a strong correspondence between partial orders and DAGs: every partial order is a DAG, and the transitive closure of a DAG is both a partial order and a DAG itself. The graph in Figure 2(b) directly defines a partial order relation E over the set of vertices $V = \{a...g\}$ while the graph in Figure 2(a) does not because it violates the transitivity condition. For instance, it contains arcs (d, f) and (f, g) while the corresponding transitive arc (d, g) is not present.

This correspondence between partial orders and DAGs provides an intuitive way of partial order specification. A DAG $G(V, E)$ defines a corresponding partial order $PO(V, E*)$. Note that there can be more than one DAG with the same corresponding partial order, for example, both of the DAGs in Figure 2 define the same partial order because they have the same transitive closure. The graph on Subfigure (a) is simpler, however, and is more preferable in some cases. *Hasse diagrams* [3] are widely used as a compact way of partial order specification.

3 Conditional Partial Order Graphs

Conditional partial order graph (CPOG) is a tuple $CPOG(V, E, A, \lambda, X, \phi)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of arcs, A is the set of *actions* (or events), $\lambda : V \rightarrow A$ is a *labelling function* which establishes correspondence between vertices of the graph and actions happening in the modelled system. X is the set of Boolean variables and function $\phi : E \rightarrow \mathcal{F}(X)$ assigns a *condition* to every arc in the graph. A condition on an arc $e \in E$ is a Boolean function $\phi(e) \in \mathcal{F}(X)$ where $\mathcal{F}(X)$ is the set of all Boolean functions over variables in X.

A *projection* of a CPOG on a variable $x \in X$ having value $x = \alpha$ is denoted as $CPOG|_{x=a}$ and is equal to $CPOG(V, E, A, \lambda, X \setminus \{x\}, \phi|_{x=a})$ where notation $\phi|_{x=\alpha}$ means that variable x is substituted with constant Boolean value α in all the functions $\phi(e), e \in E$. Projection is a *commutative operation* i.e.

$$(CPOG|_{x=\alpha})|_{y=\beta} = (CPOG|_{y=\beta})|_{x=\alpha}$$

A *complete projection* of a CPOG is a projection on all the variables in X. It is denoted as $CPOG|_\psi$ where $\psi : X \to \{0,1\}$ is an *assignment function* that assigns Boolean values to all the variables in X. Complete projection is a CPOG whose arc conditions are only Boolean constants $\phi|_\psi$ (either 0 or 1).

Let $CPOG(V, E, A, \lambda, \phi)$ be a complete projection. We can construct a labelled graph $G(V, E_G, A, \lambda)$ such that

$$E_G = \{e \in E | \phi(e) = 1\}$$

In other words G contains only the arcs whose conditions are constant 1.

A complete projection $CPOG|_\psi$ is *valid* iff its corresponding graph G is DAG.

The obtained DAG can be further converted into a corresponding labelled partial order $PO(V, E*, A, \lambda)$. Let this operation of partial order construction from a CPOG projection be shortly denoted as $\mathbf{po}(CPOG|_\psi)$. There are $2^{|X|}$ different assignment functions (because each of the $|X|$ variables can be assigned two different values) and therefore each CPOG can potentially represent $2^{|X|}$ different partial orders in a compressed form.

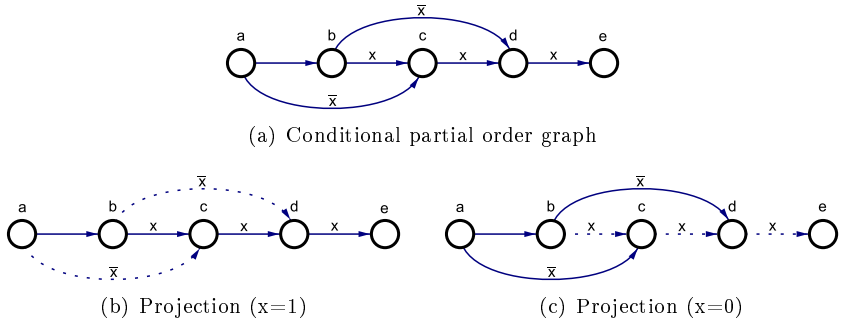
Let *assignment set* $\Psi = \{\psi_1, \psi_2, ..., \psi_m\}$ be the set of m assignment functions $\psi_k : X \to \{0,1\}$. Two Boolean functions $f, g \in \mathcal{F}(X)$ are Ψ-*equivalent* iff they evaluate to the same values over the assignment set Ψ:

$$\forall \psi_k \in \Psi, f|_{\psi_k} = g|_{\psi_k}$$

Two CPOGs $CPOG_1(V_1, E_1, A, \lambda_1, X, \phi_1)$ and $CPOG_2(V_2, E_2, A, \lambda_2, X, \phi_2)$ are Ψ-equivalent iff the assignment functions in Ψ produce the same partial orders:

$$\forall \psi_k \in \Psi, \mathbf{po}(CPOG_1|_{\psi_k}) = \mathbf{po}(CPOG_2|_{\psi_k})$$

A $CPOG$ is Ψ-*well-formed* iff every complete projection $CPOG|_\psi, \psi \in \Psi$ is valid.



(a) Conditional partial order graph

(b) Projection (x=1)

(c) Projection (x=0)

Figure 3: CPOG and its projections

An example of a CPOG and its projections is shown in Figure 3. Subfigure (a) shows the initial graph. The conditional functions are indicated over the arcs: arcs (b, c), (c, d) and (d, e) have conditional function $f = x$; the function on arcs (a, c) and (b, d) is $f = \overline{x}$; and arc (a, b) is *unconditional* i.e. its function is constant Boolean 1. Such functions are not shown on diagrams for simplicity.

Figure 3(b) shows the complete projection under $x = 1$. The dotted arcs are those that turn to have constant 0 conditions after the projection and therefore will be excluded from the resultant partial order. The solid arcs have constant 1 conditions. The partial order generated with the projection is a simple series of events: $a \to b \to c \to d \to e$.

Complete projection under $x = 0$ (Figure 3(c)) results in the following partial order. Events b and c can happen only after a. There is no constraint between them, thus they can be concurrent. Event d can

happen only after event b. Event e has no order constraints. This mean it can happen at any time. But the important fact is that there is no implication here that it *must* happen. Apparently every event can either happen or not but if it does happen it must satisfy the partial order constraints.

4 CPOG Synthesis

The previous section explained how to extract different CPOGs from a CPOG using projections. The inverse problem is much more complex: synthesize an optimal CPOG that contains all of the given CPOGs among its projections.

The CPOG optimality criteria may vary depending on the context but in this paper the main optimisation factors will be (in the order of importance):

1. minimise the number of vertices;
2. minimise the size of the control variable set X;
3. minimise the number of arcs;
4. maximise the number of unconditional arcs.

These optimisation factors follow the aim of minimisation of the size of physical implementation of CPOG-based reconfigurable controllers. The implementation details are presented in Section 6.

4.1 Synthesis from partial orders

In this paper we solve the stated synthesis problem for a subclass of CPOGs that are partial orders.

Formally, let $\mathcal{PO} = \{PO_1, PO_2, ..., PO_n\}$ be the set of n given partial orders over the same set of actions A. The objective is to synthesize a $CPOG(V, E, A, \lambda, X, \phi)$ and the assignment set $\Psi = \{\psi_1, \psi_2, ..., \psi_n\}$ such that the $CPOG$ is Ψ-well-formed and

$$\mathbf{po}(CPOG|_{\psi_k}) = PO_k, k = 1...n \tag{1}$$

To understand the process of CPOG synthesis in details let's study the following example. Consider a processing unit that has two registers p and q and can perform three operations: addition of two variables, multiplication of a variable by 2 (or doubling), and exchange of values of two variables. Instruction execution of the processing unit breaks up into six actions:

a) Instruction decoding;

b) Loading register p from memory;

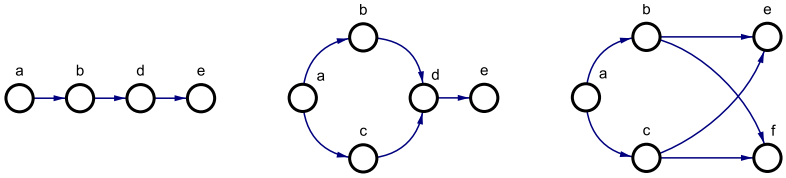
c) Loading register q from memory;

d) Addition of values loaded in registers and storing the result in register p;

e) Saving register p into memory;

f) Saving register q into memory;

The three partial orders corresponding to the operations are specified with DAGs in Figure 4. Subfigure (a) shows DAG for the operation of doubling. Four events have to be ordered sequentially: instruction decoding, loading p, addition $p = p + p$, saving p. The graph for addition of two variables is shown in Subfigure (b): instruction decoding, concurrent loading of registers p and q, addition $p = p + q$, saving p. Subfigure (c) corresponds to the operation of exchange: instruction decoding, concurrent loading of registers p and q followed by their concurrent saving into swapped memory locations. Note, that event e (saving p) must happen after both the registers p and q have been loaded (events b and c) to ensure that the value of q is not overwritten too early. The same reasoning applies to constrain the event f.



(a) doubling (b) addition (c) exchange

Figure 4: Initial partial orders

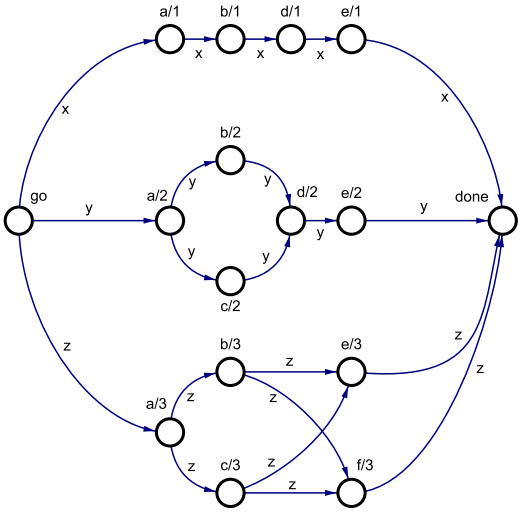


Figure 5: Naive CPOG construction

4.2 Naive approach

A naive way to construct a CPOG with the required properties is shown in Figure 5. The idea is to merge all the initial partial orders using two additional auxiliary events *go* and *done*. The labelling function λ is such that it maps vertices corresponding to the different occurrences of the same action a into the same element $a \in A$ e.g. $\lambda(a/1) = \lambda(a/2) = \lambda(a/3) = a$. In general $\lambda(a/k) = a, a \in A, k \in \mathbb{N}$.

Three variables $x, y, z \in A$ are used for selection of a particular partial order using one of the three orthogonal assignment functions $\psi_1 = (1, 0, 0)$, $\psi_2 = (0, 1, 0)$ and $\psi_3 = (0, 0, 1)$. This naive approach guarantees (1) by construction. It is equivalent to direct mapping control synthesis [9] and has all its advantages: correctness by construction, low algorithmic complexity, one-to-one correspondence between initial partial orders and parts of the obtained CPOG etc. But it is very unoptimal in terms of the optimisation criteria stated in the beginning of the section.

The assignment functions ψ_1, ψ_2 and ψ_3 can be considered as *operation codes* that are provided by the environment to dynamically reconfigure the system.

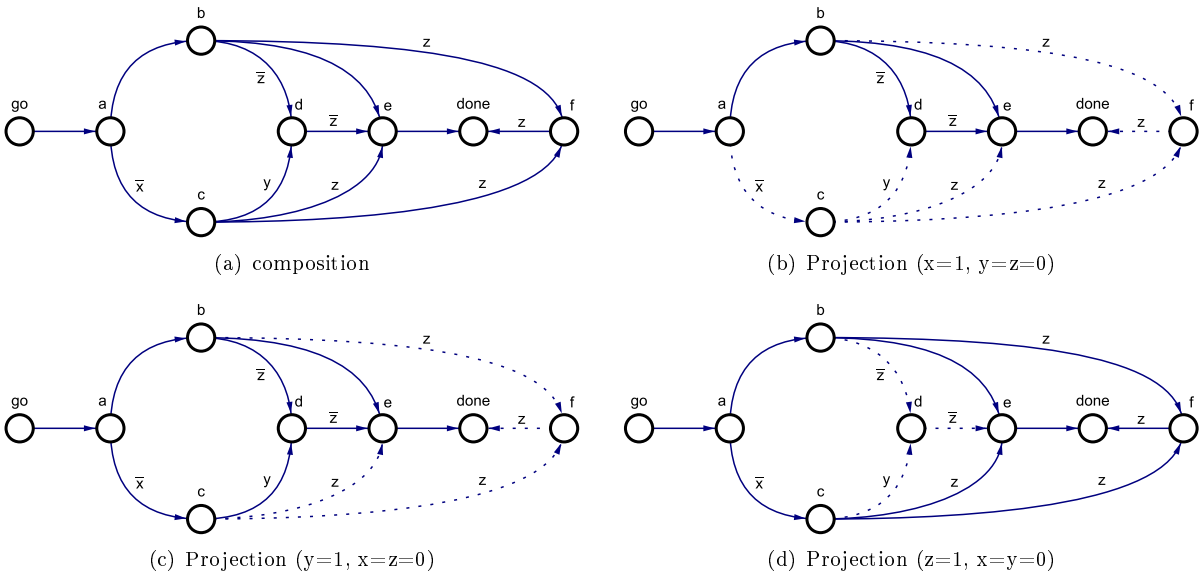
Note that such an orthogonal CPOG construction is only possible for unconditional CPOGs (partial orders) and will fail on general CPOGs because they can share control variables. The solution of the general problem is more complicated and is a subject of future work.

4.3 Composition

The main disadvantage of the naive approach presented in Subsection 4.2 is that the synthesized CPOG has multiple occurrences of the same actions. This implies that the size of the synthesized controller will be proportional to the sum of the sizes of controllers for individual partial orders.

Fortunately, it turns out that CPOG model is capable to capture all the given partial orders even if it contains only single occurrences of each action $a \in A$.

Ψ-*composition* of $CPOG(V, E, A, \lambda, X, \phi)$ is a Ψ-equivalent $CPOG'(A, E', A, \lambda', X, \phi')$ where the set



(a) composition (b) Projection (x=1, y=z=0)

(c) Projection (y=1, x=z=0) (d) Projection (z=1, x=y=0)

Figure 6: CPOG composition and its projections

of vertices equals the set of actions A (so every action occurs only once in composition) and the labelling function λ' is degraded to a trivial one-to-one mapping $\lambda'(a) = a, a \in A$. Thus Ψ-*composition* is a special case of CPOG and we will denote it as a tuple $C_\Psi(A, E, X, \phi)$.

Figure 6(a) shows the composition C_Ψ of the CPOG obtained by naive approach (Figure 5). Figures 6(b,c,d) show the three projections generated by the functions $\psi_k \in \Psi$. It is clear that the projections correspond to the three given partial orders in Figure 4. Note that arc (b, e) is unconditional. It can be assigned condition $\phi((b, e)) = z$ but this is not optimal according to the optimality criterion (4). So the condition is relaxed and this does not affect the correctness of the composition because arc (b, e) appears as transitive in projections when $\phi((b, e)) = z = 0$ and does not affect the resultant partial orders.

4.4 Algorithm

This subsection presents the algorithm for synthesis of an optimal CPOG based on the idea of Ψ-composition. The algorithm is shown in Algorithm 1. It synthesizes a CPOG C_Ψ and assignment set Ψ such that C_Ψ is the Ψ-composition of a naively constructed CPOG. The algorithm is polynomial and based on orthogonality of assignment functions in Ψ. It cannot be easily generalised for the case of CPOGs composition.

Method *RemoveTransitiveConditions*(C_Ψ, x_k) removes x_k from all the arc conditions ϕ in C_Ψ where x_k is transitive. And method *RemoveTransitiveUnconditionalArcs*(C_Ψ) removes from C_Ψ all the unconditional arcs that are transitive. The execution of the algorithm is explained below on the example of composition synthesis in Figure 6.

At first the algorithm constructs the initial composition C_Ψ using equations (2) and (3). The obtained set of arc functions is shown in Table 1. For clarity we keep the variable names from the figure $\{x, y, z\}$ instead of $\{x_1, x_2, x_3\}$ as denoted in the algorithm.

Algorithm 1 Synthesis of an optimal CPOG

Given: set of partial orders $\mathcal{PO} = \{PO_1, PO_2, ..., PO_n\}$
Result: CPOG C_Ψ and assignment set Ψ satisfying (1)

Let $PO_k = (A, R_k)$ *and* $A = \{a_1, a_2, ..., a_{|A|}\}$
Let $X = \{x_1, x_{2,}, ..., x_n\}$ *and* $\Psi = \{\psi_1, \psi_2, ..., \psi_n\}$

Construct assignment set Ψ *such that:*

$$\psi_k(x_j) = \begin{cases} 1, k = j \\ 0, k \neq j \end{cases} \quad (k, j = 1...n)$$

Construct $C_\Psi(A, E, X, \phi)$ *such that:*

$$\phi((a,b)) = \bigvee_{\substack{aR_kb \\ k=1...n}} x_k \quad (a, b \in A) \tag{2}$$

$$E = \{(a,b) | a, b \in A \wedge \phi((a,b)) \neq 0\} \tag{3}$$

Forall $x_k \in X$ *do*
C_Ψ = *RemoveTransitiveConditions*(C_Ψ, x_k);

C_Ψ = *RemoveTransitiveUnconditionalArcs*(C_Ψ);

Return (C_Ψ, Ψ) *as a result;*

ϕ	**go**	**a**	**b**	**c**	**d**	**e**	**f**	**done**
go		1	1	$y+z$	$x+y$	1	z	1
a			1	$y+z$	$x+y$	1	z	1
b					$x+y$	1	z	1
c					y	$y+z$	z	$y+z$
d						$x+y$		$x+y$
e								1
f								z
done								

Table 1: Initial set of arc functions

Then the algorithm performs removal of transitive conditions from arc functions ϕ. The result of this is shown in Table 2.

ϕ	**go**	**a**	**b**	**c**	**d**	**e**	**f**	**done**
go		1	1			1		1
a			1	$y+z$		1		1
b					$x+y$	1	z	1
c					y	z	z	
d						$x+y$		
e								1
f								z
done								

Table 2: Removal of transitive conditions

The last optimisational step is to remove the transitive unconditional arcs from both ϕ and E. The result is shown in Table 3.

ϕ	**go**	**a**	**b**	**c**	**d**	**e**	**f**	**done**
go		1						
a			1	$y+z$				
b					$x+y$	1	z	
c					y	z	z	
d						$x+y$		
e								1
f								z
done								

Table 3: Removal of transitive unconditional arcs

As a result we obtain an optimal CPOG C_Ψ and assignment set Ψ satisfying (1). Following optimisations of arc functions ϕ in C_Ψ are possible, for instance, in Figure 6 we use simpler arc functions taking into account Ψ-equivalence: $\phi((a,c)) = \overline{x}$ because function $f(x,y,z) = y+z$ is Ψ-equivalent to $f(x,y,z) = \overline{x}$.

4.5 Control set size optimisation

Another optimisation issue is the size of the control variable set X (optimisation criterion (2)).

In our sample case it is $|X| = 3$. However, to select one of the three partial orders we need only $\lceil \log_2 3 \rceil = 2$ bits of information. So a set of two control variables is clearly enough and it can be easily constructed.

Let $Y = \{s,t\}$. Then we can encode variables $X = \{x,y,z\}$ such that $x = \overline{s} \cdot \overline{t}$, $y = s \cdot t$ and $z = s \oplus t$. A possible corresponding assignment set is $\Psi_Y = \{(0,0),(1,1),(0,1)\}$. The equivalence of these two control variables sets is visually explained in Table 4. The ideas of the control set optimisation are further developed in Section 5.

Ψ	x	y	z	
$(1,0,0)$	1	0	0	$(0,0)$
$(0,1,0)$	0	1	0	$(1,1)$
$(0,0,1)$	0	0	1	$(0,1)$
	$\overline{s} \cdot \overline{t}$	$s \cdot t$	$s \oplus t$	Ψ_Y

Table 4: Control variables set equivalence

5 Logic Optimisation

The section defines optimality criterion for control set size $|X|$ of a $CPOG(V,E,A,\lambda,X,\phi)$.

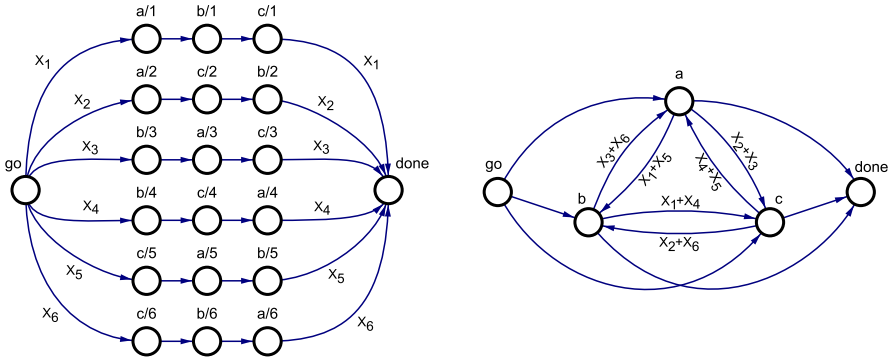
The control variable set X of $CPOG$ is Ψ-*optimal* iff

$$\lceil \log_2 |\Psi| \rceil = |X| \tag{4}$$

The idea behind 4 is that we need at least $\lceil \log_2 |\Psi| \rceil$ bits to select one of the $|\Psi|$ available partial orders encoded in $CPOG$. So if we use this minimum number of control bits then X is optimal in terms of the amount of information it gives.

It is always possible to construct a Ψ-optimal control set $Y = \{y_1, y_2, ..., y_m\}, m = \lceil \log_2 |\Psi| \rceil$ for a given $CPOG$ in a way similar to the one presented in 4.5. The following example explains it in details.

Figure 7 shows the naive and compositional specification of a 3-wire phase-encoding sender. Phase-encoding circuits and their existing implementations are thoroughly studied in [removed]. The sender is used to transmit one of the six possible permutations of the three signals $a,b,c \in A$. Clearly the control



(a) naive specification (b) composition

Figure 7: CPOGs for 3-wire phase-encoding sender

set X is not optimal here as we use 6 bits to encode one of the 6 possible scenarios instead of the required 3 bits. Let's construct a Ψ-optimal control set $Y = \{r, s, t\}$. The control variables in X can be substituted with the following functions over Y: $x_1 = \overline{r} \cdot \overline{s} \cdot \overline{t}$, $x_2 = \overline{r} \cdot \overline{s} \cdot t$, $x_3 = \overline{r} \cdot s \cdot \overline{t}$, $x_4 = \overline{r} \cdot s \cdot t$, $x_5 = r \cdot \overline{s} \cdot \overline{t}$ and $x_6 = r \cdot \overline{s} \cdot t$. Now we can use any logic optimisation tool to minimise conditions $\phi(e), e \in E$. For instance, condition $\phi((a,b)) = x_1 + x_5 = \overline{r} \cdot \overline{s} \cdot \overline{t} + r \cdot \overline{s} \cdot \overline{t}$ can be minimised into $\phi((a,b)) = (r + \overline{r}) \cdot \overline{s} \cdot \overline{t} = \overline{s} \cdot \overline{t}$. The other five arc conditions are minimised in the same way: $\phi((b,a)) = \overline{r} \cdot s \cdot \overline{t} + r \cdot \overline{s} \cdot t$, $\phi((a,c)) = \overline{r} \cdot (s \oplus t)$, $\phi((c,a)) = \overline{r} \cdot s \cdot t + r \cdot \overline{s} \cdot \overline{t}$, $\phi((b,c)) = \overline{r} \cdot \overline{s \oplus t}$, $\phi((c,b)) = \overline{s} \cdot t$.

As a result we obtained an optimal encoding of the partial orders in the CPOG. As a side effect we synthesized a phase-encoding sender able to send binary encoded signals $\{r, s, t\}$.

6 Control Gate-level Implementation

CPOG model is useful for control synthesis because it has a very area-efficient and robust gate-level implementation based on a generic circuit architecture, called Transition Sequence Encoder (TSE), which is introduced in the section.

6.1 Transition Sequence Encoder

This subsection describes the TSE, a circuit able to schedule a set of events according to the partial order specified as a DAG incidence matrix. The main feature of the circuit is that it can be dynamically reprogrammed during runtime to alter the order of the generated events. This particular feature allows the circuit to be used as a basis for the method of control structure synthesis presented in the paper.

Given a partial order $PO(A, R)$ over a set of n events $A = \{a_1, a_2, ..., a_n\}$ the TSE circuit generates a series of request-acknowledgement handshakes $req[k]/ack[k], a_k \in A$ in the specified order as shown in Figure 8. The partial order is specified as a DAG $G(A, E)$. The "order matrix" in the figure stands for the incidence matrix of G:

$$R[i,j] = \begin{cases} 1, (a_i, a_j) \in E \\ 0, (a_i, a_j) \notin E \end{cases} \quad (a_i, a_j \in A)$$

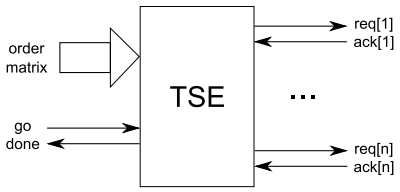


Figure 8: Transition Sequence Encoder

The initial equation for signal $req[k]$ generation contains $2^{|A|}$ complex clauses:

$$req[k] = \bigvee_{P \subseteq A} \left(\bigwedge_{a_j \in P} R[j,k] \cdot \bigwedge_{a_j \in A \setminus P} \overline{R[j,k]} \cdot \bigwedge_{a_j \in P} ack[j] \right)$$

The idea is that the request signal $req[k]$ can be generated only when the acknowledgement signals $ack[j]$ have been received for all the preceding events $a_j \in P$ (terms $\bigwedge_{a_j \in P} R[j,k]$ and $\bigwedge_{a_j \in P} ack[j]$) and no other event $a_j \in A \setminus P$ precedes event a_k (term $\bigwedge_{a_j \in A \setminus P} \overline{R[j,k]}$). To simplify the above equation it is possible to fold it into conjunction of only $|A|$ simple clauses:

$$req[k] = \bigwedge_{1 \leq j \leq n} \left(R[j,k] \cdot ack[j] + \overline{R[j,k]} \right)$$

This can be further simplified using the Boolean algebra to:

$$req[k] = \bigwedge_{1 \leq j \leq n} \left(ack[j] + \overline{R[j,k]} \right) \tag{5}$$

Note that the above equation contains a redundant term when j equals k: $ack[k] + \overline{R[k,k]}$ that is equal to 1 because $\overline{R[k,k]} = 1$ because graph G is acyclic. The term does not affect the correctness of the equation but of course in physical circuit implementation it will be omitted.

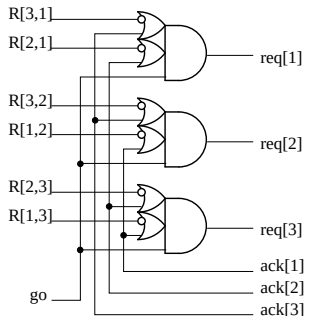


Figure 9: TSE gate-level implementation

The obtained solution can be mapped to gate-level implementation as shown in Figure 9 for the case of $|A| = 3$. Signal *go* was added which is a general "ready" signal that prompts the circuit to start generating requests. The output signal *done* can be generated as a conjunction of acknowledgement signals: $done = \bigwedge_{a_k \in A} ack[k]$.

6.2 TSE customised for a CPOG

A general TSE is able to generate control signals for every possible partial order over an event set A. However, we need only those encoded in our CPOG as complete projections. Therefore we need to build a wrapper using arc functions ϕ of the given CPOG: $R[i,j] = \phi((a_j, a_k)), j, k = 1...n$. Optimal CPOG normally has many unconditional arcs and the corresponding signals $R[i,j]$ are substituted with constant values. Boolean logic optimisation is then applied to further reduce the size of TSE circuit.

Another customisation issue is that the standard TSE generates request signals for all the events in A. But sometimes we might want a certain event not to happen at all. For simplicity we will assume that the only events happening $H \subseteq A$ are those that precondition the auxiliary event *done*: $H = \{a \in A | aRdone\}$.

These ideas are implemented in a software tool for dynamically reconfigurable control synthesis. The result of application of the tool to the compositional CPOG from Figure 6(a) is shown in Figure 10. Signals $\{x, y, z\}$ are the control signals that select one of the three available partial orders for execution.

The execution starts when the environment issues signal *go* and as soon as all the needed handshakes have been conducted the signal *done* is generated.

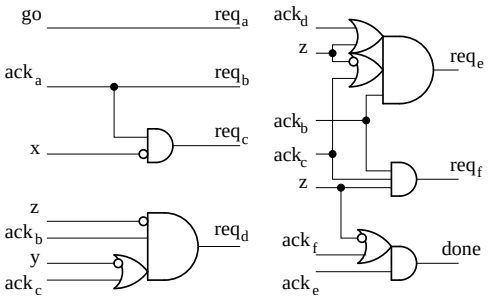


Figure 10: CPOG-customised controller

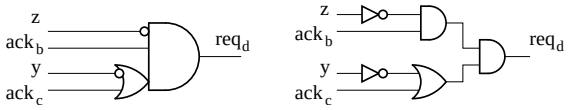
6.3 Decomposition and technology mapping

The presented gate-level control implementation is Delay Insensitive (DI) [11] w.r.t. handshake signals ($req[k]$, $ack[k]$) and thus very robust under process and environmental (parametric) variations. This is possible because the synthesized circuits do not contain any memory elements or arbitration. It makes the circuits fast and reliable because no timing assumptions or safety margins are introduced.

The synthesized circuits may contain large complex gates which are not present in the gate library, so certain decomposition and technology mapping issues have to be addressed in order to stay within DI class after technology mapping [5].

A very important feature of the TSE-based controllers is that both the request ($req[k]$) and acknowledgement ($ack[k]$) signals are generated in a monotonic way: once a signal goes high it will remain high until all the other signals become high. The same behaviour is observed during the reset phase. This allows a simple structural decomposition to be used. Suppose that we use a gate library that only contains two-input positive gates and inverters and we need a 4-input complex gate shown in Figure 11(a). We can safely decompose the complex gate into several library gates as shown in Subfigure (b). The decomposition is structural in the sense that every large gate can be decomposed independently of the others. This will introduce no hazards into the circuit because of the monotonic signals behaviour.

It may be observed that the circuit is not DI w.r.t. inverters on control variables. However, we assume that these control signals are issued before signal *go* arrives that guarantees appropriate timing relations. These signals remain constant during the operation of the handshakes.



(a) complex gate (b) decomposition

Figure 11: Structural decomposition

7 Benchmarks

The algorithm was implemented in a software tool and its performance and optimisation abilities were checked against a set of benchmarks.

7.1 MSP430 CPU controller synthesis

MSP430 processor [1] was selected as one of the real-life benchmarks to test how the introduced CPOG model can describe such complicated systems as a general purpose microprocessors. The processor has

16 registers, 7 addressing modes and 27 core instructions.

26 partial orders $\mathcal{PO}$ were extracted from the synchronous specification of instructions provided in [removed]. The set of actions A contains 18 actions (including dummy actions *go* and *done*). Two control circuits were synthesized and compared: the first one used a naively constructed CPOG composition while the second one was constructed using the presented CPOG optimisation algorithm. The results of comparison are shown in Table 5.

Parameter	Naive CPOG	Optimised CPOG
Number of vertices	170	18
Number of arcs	241	62
Number of gates	652	118
Average gate complexity	1.48	3.25
Circuit size estimation (inverter area units)	964.0	309.5

Table 5: Comparison of naive and optimised compositional CPOG

The table shows that the number of arcs in the optimised CPOG composition is significantly reduced. The number of gates is also reduced because each arc and the conditional function on it corresponds to a gate in the physical circuit implementation . However the average gate size increased as the arc conditions became more complicated. Estimation of the control circuit size measured in inverter area units shows that the optimised composition is approximately three times more area-efficient.

We can compare these results with the synchronous control for MSP430 processor synthesized using Synopsys toolkit (DC compiler) from Verilog specification in RTL level. The control area is approximately 1900 inverter area units. It should be mentioned however that the comparison of this result with ours is not fair enough because our approach delegates some of the control functions to the controlled blocks e.g. ALU block has its own TSE-based controller to decide which of the 16 registers to add in a particular instruction.

7.2 Phase-encoding senders

This set of benchmark circuits was used as a stress test for the tool. Phase-encoding senders are a class of systems with highly variable behaviour that match CPOG model perfectly. One of the examples – 3-wire phase-encoding sender was presented in Section 5.

The tool performance results on several phase-encoding senders are presented in Table 6. Note that the size of naive system specification grows exponentially w.r.t. the number of data wires of the sender. However the size of the composition generated by the tool remains polynomial (the number of vertices grows linearly, the number of arcs – quadratically). Consequently, the size of the optimised control circuit grows quadratically and the gap between naive and optimal solutions increases dramatically.

Benchmark	Number of vertices	Number of arcs	Circuit size
3-wire sender (naive/opt)	20 / 5	24 / 12	96 / 32
4-wire sender (naive/opt)	98 / 6	120 / 20	480 / 84
6-wire sender (naive/opt)	4322 / 8	5040 / 42	20160 / 1913

Table 6: Tool performance on phase-encoding senders

8 Conclusions

A novel Conditional Partial Order Graph model is introduced. It is able to capture concurrency and choice in systems with many similar patterns in behaviour in a compact and efficient way. The problem of CPOG

composition synthesis is solved for a subclass of CPOGs that are partial orders. The generalisation of composition and further study of the model properties is the subject of future research.

A TSE-based method for dynamically reconfigurable control synthesis for systems described with CPOGs is proposed. The method is implemented in a software tool and tested on a set of benchmarks. The obtained results show that the proposed method is very robust and area-efficient. The method mainly targets large systems such as CPU microcontrollers, data encoding and transfer circuits etc.

A combination of the presented model and synthesis method offers a new way to providing a consistent design flow for control circuits specification and synthesis.

References

[1] *MSP430x4xx Family User's Guide.*

[2] Andrew Bardsley and Doug Edwards. The balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, 2000.

[3] G. Birkhoff. *Lattice Theory.* Third Edition, American Mathematical Society, Providence, RI, 1967.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[5] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1997.

[6] Luciano Lavagno, Louis Scheffer, and Grant Martin. *Electronic Design Automation For Integrated Circuits Handbook.* 2006.

[7] Art Lew. *Computer Science: A Mathematical Introduction.* Prentice-Hall, 1985.

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

[9] Danil Sokolov. *Automated synthesis of asynchronous circuits using direct mapping for control and data paths.* PhD thesis, University of Newcastle upon Tyne, 2005.

[10] Danil Sokolov and Alex Yakovlev. Clock-less circuits and system synthesis. In *IEE Proceedings, Computers and Digital Techniques*, 2005.

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

[12] Kees van Berkel, Mark B. Josephs, , and Steven M. Nowick. Scanning the technology: applications of asynchronous circuits. In *Proceedings of the IEEE*, 1999.

[13] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The vlsi-programming language tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, 1991.