School of Electrical, Electronic & Computer Engineering

Newcastle University

Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis

A. Mokhov and A. Yakovlev

January 2008

Contact:

Andrey.Mokhov@ncl.ac.uk

Alex.Yakovlev@ncl.ac.uk

Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis

A. Mokhov and A. Yakovlev

January 2008

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 CPOG synthesis is formulated and solved; various CPOG optimisation techniques are presented.

The introduced 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 speed independent 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.

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 or VHDL, and RTL-based synthesis flow [4]. Within this conventional methodology, designers use finite state machines to capture control specifications. Since standard RTL flow supports a synchronous 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. [9] and [7]. Some methods such as Tangram (or Haste) [10] and Balsa [1] use CSP-like HDL languages for system specification and syntax-direct translation for synthesis. They are not particularly well suited 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 [6], as well as Petri nets (PN) and STGs [8], 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) [7] 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 4.3). 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* [2] of events associated with the activations of operational

units. Applying Burst Mode FSMs or PNs 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.
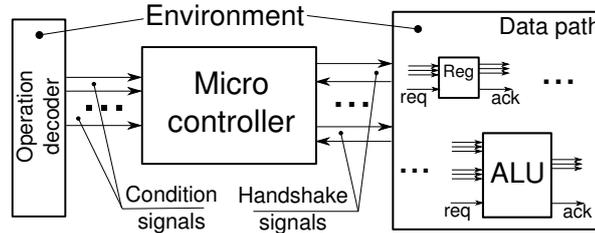
Figure 1: Reconfigurable controller

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 and avoid using the explicit notion of state. 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. The order 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 (see Figure 1), one part is the source of condition signals (operation decoder in case of a CPU) and the other part is a set of controlled objects with request-acknowledgement interface (operational units). Thus the condition signals dynamically reconfigure our controller according to the instruction being executed.

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 Motivation

This section presents an example of specification of a controller with two behavioural scenarios using STGs. The example shows that STG specification is inconvenient and inefficient in the case. Synthesis of a generalised version of this particular controller motivated our research.

Figure 2: Example controller interface

The controller's interface is shown in Figure 2. Depending on the control signals $\{x1, x2\}$ the controller has to initiate two handshakes either in order $1 \rightarrow 2$ or in order $2 \rightarrow 1$. The start of the handshake sequence is prompted by signal *go* and as soon as the handshakes are completed the controller issues signal *done*. This leads us to the STG shown in Figure 3(a). The STG has a global choice and the two scenarios are specified as two independent branches. The first scenario (the upper branch) is handshake sequence $1 \rightarrow 2$; the second one (the lower branch) corresponds to $2 \rightarrow 1$. After the global merge the handshakes are reset concurrently and the system returns to the initial state.

Although this specification seems to be convenient, understandable and can be obtained manually it duplicates events in different branches. In general there can be exponential number of different scenarios composed of linear number of events: for instance, we can generalise the exemplar controller to control order of execution of more than two events (handshakes) and for n events there will be $n!$ different

scenarios (such control circuits are called *n-permutators*). It is not efficient to have an STG specification containing $n!$ different branches and it turns from inefficient into infeasible for large values of n.

To specify the controller in a more compact way we can construct a behaviourally equivalent STG without multiple event occurrences using Petrify [3] (see Figure 3(b)). Such compositional STGs tend to be much more complicated and contain a lot of additional choice places tracking the current system state. Even for such a simple controller that was taken as our example the obtained STG is non-trivial and difficult for manual design. In practice the only way to produce an optimal STG specification is to start with an inefficient global choice STG and feed it to an optimisation tool (e.g. Petrify) but this is infeasible for large controllers. Another issue is that generation of compositional STG involves construction of state graph and examining all reachable states that is a very time and memory consuming process. It should be mentioned however that if Petrify does not run out of time and memory resources it produces very efficient gate-level implementations of the controllers. Figure 4 shows controllers synthesised using the presented CPOG-based approach and Petrify: the solutions are different and our method produces a smaller controller.

(a) Specification with global choice and multiple event occurrences

(b) Optimised specification obtained by Petrify
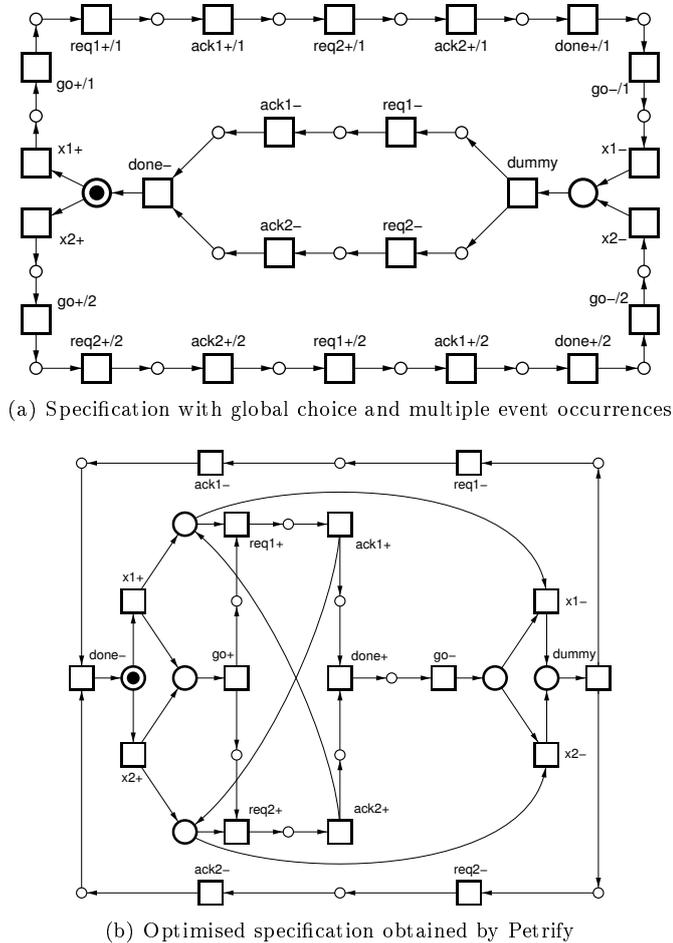
Figure 3: Controller STG specifications

These motivation factors led us to *Conditional Partial Order Graph Model* introduced in this paper. The model allows a designer to specify large systems manually as a collection of partial orders [2] and then merge them into a compact CPOG that captures all the system functionality.

A *partial order* (PO) $P(S, \prec)$ is a binary relation $\prec$ over a set of elements S which satisfies the following conditions:

1. *Irreflexivity*: $\forall a \in S, \neg(a \prec a)$;

2. *Asymmetry*: $\forall a, b \in S, (a \prec b) \Rightarrow \neg(b \prec a)$;

(a) CPOG-based solution

(b) Petrify solution
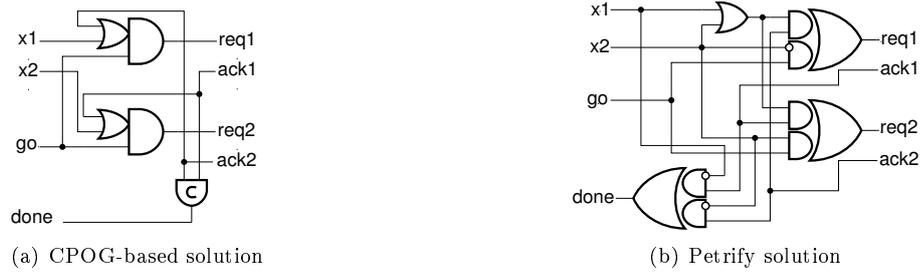
Figure 4: Synthesised controllers

3. *Transitivity*: $\forall a, b, c \in S, (a \prec b) \wedge (b \prec c) \Rightarrow (a \prec c)$.

POs 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 PO such that if $a \prec b$ then event a must happen strictly before event b. If neither $a \prec b$ nor $b \prec a$ holds then the events a and b can happen in any order, possibly simultaneously.

The concept of system specification with a set of POs is clarified by the following example. Consider a processing unit that has registers p and q and performs three operations: addition of two variables, multiplication of a variable by 2 (doubling), and exchange of two variables. Instruction execution of the processing unit breaks up into six actions:

a) Decode instruction;

b) Load register p from memory;

c) Load register q from memory;

d) Add values from registers and store the result in p;

e) Save register p into memory;

f) Save register q into memory;

The POs corresponding to the operations are shown in Figure 5 (action labels are from the list above). Subfigure (a) shows PO for the operation of doubling. Four actions are ordered sequentially: decode instruction, load p, add $p = p+p$, save p. Subfigure (b) corresponds to addition: decode instruction, load p and q concurrently, add $p = p + q$, save p. Action order of the exchange operation (Subfigure (c)) is: decode instruction, load p and q concurrently, save them concurrently into swapped memory locations.

(a) P_1 (doubling)

(b) P_2 (addition)

(c) P_3 (exchange)
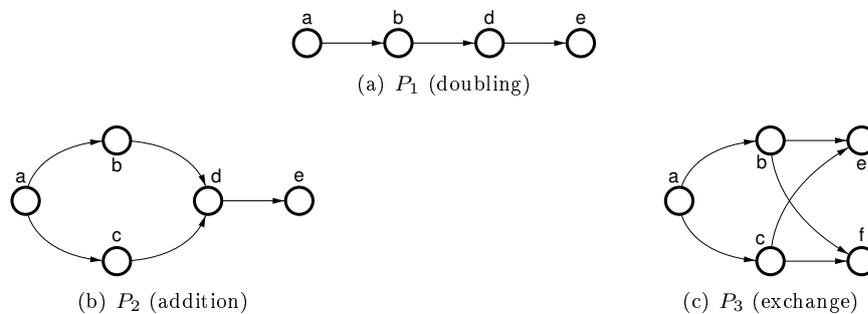
Figure 5: Operations specified as POs

As soon as the system is specified as a set of POs it is possible to synthesise a CPOG that will contain all of them in a compressed form (see Section 4.3). But first we have to introduce CPOGs formally. This is done in the next section.

3   Conditional Partial Order Graphs

Conditional partial order graph is a tuple $H(V, E, X, \phi)$ where V is the set of *vertices* (or *nodes*), $E \subseteq V \times V$ is the set of ordered pairs of vertices, called *arcs*, X is the set of Boolean variables, and function $\phi : V \cup E \to \mathcal{F}(X)$ assigns a *condition* to every vertex and arc in the graph. A condition on a vertex or arc $z \in V \cup E$ is a Boolean function $\phi(z) \in \mathcal{F}(X)$ where $\mathcal{F}(X)$ is the set of all Boolean functions over variables in X. Let's also define $\phi(z) = 0$ for $z \notin V \cup E$ in order to simplify some of the further computations.

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 fact that a CPOG H contains a path $\langle a, b \rangle$ will be denoted as $\langle a, b \rangle \in H$.

An arc $e = (a, b) \in E$ is called *transitive* iff $\langle a, b \rangle \in H \backslash \{e\}$. $H \backslash \{e\}$ means a CPOG H without arc e.

3.1   Addition

The result of *addition* of $H_1(V_1, E_1, X_1, \phi_1)$ and $H_2(V_2, E_2, X_2, \phi_2)$ is CPOG $H(V_1 \cup V_2, E_1 \cup E_2, X_1 \cup X_2, \phi)$, where $\forall z, \phi(z) = \phi_1(z) + \phi_2(z)$. Here $\phi_1 + \phi_2$ stands for Boolean disjunction of functions ϕ_1 and ϕ_2. We will use standard notation for addition: $H = H_1 + H_2$.

CPOG addition is commutative ($H_1 + H_2 = H_2 + H_1$) and associative (($H_1 + H_2) + H_3 = H_1 + (H_2 + H_3)$) and thus redundant brackets can be omitted when more than two CPOGs are added.

3.2   Scalar multiplication

A CPOG $H(V, E, X, \phi)$ can be *multiplied* by a Boolean function $f \in \mathcal{F}(Y)$ (which in our context can be called *scalar*). The resultant CPOG is $H'(V, E, X \cup Y, \phi')$ where $\forall z, \phi'(z) = f\phi(z)$ ($f\phi$ stands for Boolean conjunction of functions f and ϕ). We will use standard notation for scalar multiplication: $H' = fH$.

Scalar multiplication and addition have the following common properties:

- Left distributivity: $(f + g)H = fH + gH$;

- Right distributivity: $f(H_1 + H_2) = fH_1 + fH_2$;

- Associativity: $f(gH) = (fg)H$;

3.3   Projection

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

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

We also define a *partial assignment function* $\psi : Y \to \{0, 1\}, Y \subseteq X$ which assigns values only to a subset of X.

Let $H(V, E, \emptyset, \phi)$ be a complete projection ($\forall z, \phi(z) \in \{0, 1\}$). We can construct a graph $G(V_G, E_G)$ such that

$$\begin{cases} V_G = \{v \in V | \phi(v) = 1\} \\ E_G = \{e = (a, b) \in E | \phi(a)\phi(b)\phi(e) = 1\} \end{cases}$$

In other words G contains only those vertices and arcs whose conditions in H are constant 1.

A complete projection H is *valid* iff its corresponding graph G is a *directed acyclic graph* (DAG) [2].

The obtained DAG $G(V_G, E_G)$ can be further converted into a corresponding PO $P(V_G, \prec)$ such that $a \prec b$ iff $\langle a, b \rangle \in G$. Let this operation of partial order construction from a valid CPOG complete projection H be shortly denoted as **po**: $P = \mathbf{po}(H)$ and the inverse operation as $\mathbf{po}^{-1}$: $H = \mathbf{po}^{-1}(P)$. There are $2^{|X|}$ different assignment functions ψ (each of the $|X|$ variables can be assigned two different values) and therefore each CPOG can potentially represent $2^{|X|}$ different POs in a compressed form.

(a) Conditional partial order graph

(b) Projection under $x = 1$

(c) Projection under $x = 0$
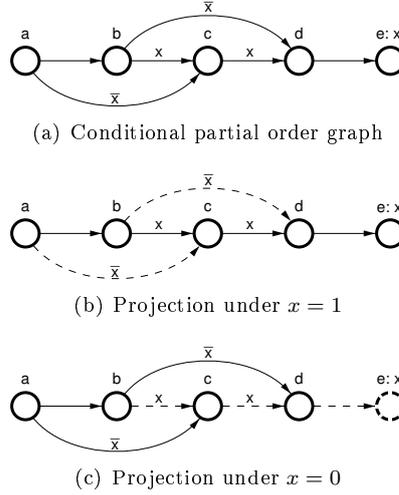
Figure 6: CPOG and its projections

An example of a CPOG and its projections is shown in Figure 6. Subfigure (a) shows the initial graph. The conditional arc functions are indicated over the arcs: arcs (b, c) and (c, d) have conditional function $f = x$; the function on arcs (a, c) and (b, d) is $f = \overline{x}$; arcs (a, b), (d, e) and vertices $a...d$ are *unconditional* i.e. their functions are constant Boolean 1. Such functions are not shown on diagrams for simplicity. The only conditional vertex e has condition $f = x$ which is shown next to its label separated by a colon.

Figure 6(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 defined with the projection is a simple series of events: $a \to b \to c \to d \to e$.

Complete projection under condition $x = 0$ (Figure 6(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 is excluded from the partial order; note, that this implies exclusion of arc (d, e) as well.

3.4   Ψ-equivalence

Let *assignment set* $\Psi = \{\psi_1, \psi_2, ..., \psi_n\}$ be the set of n 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:

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

A CPOG H is *Ψ-well-formed* iff every complete projection $H|_\psi, \psi \in \Psi$ is valid. Ψ-well-formed CPOGs H_1 and H_2 are Ψ-equivalent iff they produce the same POs:

$$\forall \psi_k \in \Psi, \mathbf{po}(H_1|_{\psi_k}) = \mathbf{po}(H_2|_{\psi_k})$$

We will use the following notation for Ψ-equivalence: $f \overset{\Psi}{\sim} g$ or $H_1 \overset{\Psi}{\sim} H_2$. Ψ-equivalence is a proper *equivalence relation* [2] as it satisfies the following properties:

- Reflexivity: $a \overset{\Psi}{\sim} a$;

- Symmetry: $a \overset{\Psi}{\sim} b \Rightarrow b \overset{\Psi}{\sim} a$;

- Transitivity: $a \overset{\Psi}{\sim} b \wedge b \overset{\Psi}{\sim} c \Rightarrow a \overset{\Psi}{\sim} c$.

4 CPOG Synthesis

The previous section introduced CPOG algebra and showed that a CPOG can contain exponential number of partial orders in a compressed form. The problem now is to synthesise a compact CPOG specification given the description of the system as a set of POs corresponding to different behavioural scenarios in the modelled system. The synthesis problem can even be more generalised: we can use not only POs as building blocks but also mix them with CPOGs and synthesise an optimal CPOG containing all of them among its projections. This can be useful if you, for instance, want to add a new scenario to already existing CPOG without its complete resynthesis from POs.

Formally, let $\mathcal{H} = \{H_1, H_2, ..., H_n\}$ be the set of n given CPOGs. The objective is to synthesize CPOG $H(V, E, X, \phi)$ and assignment set $\Psi = \{\psi_1, \psi_2, ..., \psi_n\}$ such that $\psi_k : X_C \to \{0, 1\}$ are partial assignment functions over the control set $X_C \subseteq X$ and

$$\forall \psi_k \in \Psi, H|_{\psi_k} = H_k$$

The idea behind our synthesis approach is to represent H as the following sum of given CPOGs:

$$H = f_1 H_1 + f_2 H_2 + ... + f_n H_n = \sum_{k=1}^{n} f_k H_k$$

Now control set X_C, functions $f_k \in \mathcal{F}(X_C)$ and ψ_k should be selected so that $f_k|_{\psi_k} = 1$ and $f_k|_{\psi_j} = 0, j \neq k$. This can be done in different ways depending on the chosen encoding scheme. There are many encoding schemes; two commonly used (one-hot and binary) are presented below.

4.1 One-hot encoding scheme

In this scheme we use n additional control variables $X_C = \{x_1, x_2, ..., x_n\}$. Functions f_k and ψ_k ($k = 1...n$) are trivial: $f_k = x_k$, $\psi_k(x_k) = 1$, $\psi_k(x_j) = 0, j \neq k$.

One-hot scheme provides a simple and intuitive way of encoding but it is inefficient because of the large size of control variables set X_C: $|X_C| = n$. Synthesis of the example controller CPOG using one-hot encoding is trivial:

4.2 Binary encoding scheme

In binary scheme only $m = \lceil \log_2 n \rceil$ control variables $X_C = \{x_1, x_2, ..., x_m\}$ are used which is the theoretical minimum. Let b_{jk} denote j-th bit of integer number k. Then we can define functions f_k and ψ_k ($k = 1...n$) as:

$$\psi_k(x_j) = b_{(j-1)(k-1)}, \quad f_k = \bigwedge_{j=1}^{m} (x_j \Leftrightarrow \psi_k(x_j))$$

For example, if $n = 3$ we will get $\psi_1 = (0,0)$, $\psi_2 = (1,0)$ and $\psi_3 = (0,1)$. Functions f_k are: $f_1 = (x_1 \Leftrightarrow 0)(x_2 \Leftrightarrow 0) = \overline{x_1}\,\overline{x_2}$, $f_2 = x_1\overline{x_2}$ and $f_3 = \overline{x_1}x_2$.

4.3 Synthesis from partial orders

CPOG synthesis from POs is a special case of general synthesis problem presented above. Given a set of n POs $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ we can convert them to CPOGs $H_k = \mathbf{po}^{-1}(P_k), k = 1...n$ and then use the general method to synthesise CPOG containing $\mathcal{H} = \{H_k | k = 1...n\}$.

The result of synthesis of CPOG containing the three behavioural scenarios from Figure 5 is shown in Figure 7(a). One-hot encoding scheme with control set $\{x, y, z\}$ was used. You can check that its projections $\psi_1 = (1,0,0)$, $\psi_2 = (0,1,0)$ and $\psi_3 = (0,0,1)$ produce POs P_1 (doubling), P_2 (addition) and P_3 (exchange) respectively.

(a) Synthesised CPOG

(b) Vertex conditions optimisation

(c) Arc conditions optimisation

(d) Projection $y = 1, x = z = 0$
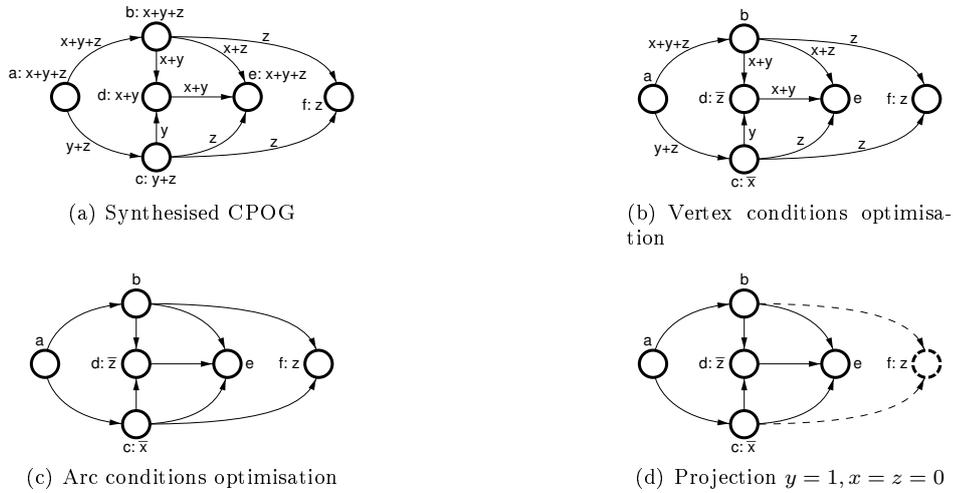
Figure 7: CPOG synthesis and optimisation

5 Logic Optimisation

The size of the physical controller implementation is proportional to the size of CPOG specification measured as the total number of literals in its conditional functions. It is possible to use logic optimisation techniques and exploit structural properties of a CPOG in order to minimise the number of literals and thus to obtain an area efficient controller.

5.1 Vertex conditions optimisation

Vertex conditions optimisation is based on the notion of Ψ-equivalence of functions. If a function $\phi(z), z \in V \cup E$ in a CPOG $H(V, E, X, \phi)$ is replaced with a Ψ-equivalent function $\phi'(z)$ then the modified CPOG H' is Ψ-equivalent to H. This property provides a powerful optimisation technique: a conditional function can be optimised independently of the others. Moreover optimisation under the Ψ-equivalence relation allows logic optimisation tool to consider the variable assignments not in Ψ as *don't cares*.

Ψ	x	y	z	$f = x + y + z$	$g = 1$	$f = x + y$	$g = \overline{z}$
ψ_1	1	0	0	1	1	1	1
ψ_2	0	1	0	1	1	1	1
ψ_3	0	0	1	1	1	0	0

Table 1: Ψ-equivalence of functions

Let's take the CPOG in Figure 7(a) as an example. Vertices $\{a, b, e\}$ have conditional function $f = x + y + z$. This function is Ψ-equivalent to function $g = 1$. It is demonstrated in Table 1: both f and g give the same result 1 for the variable assignments in Ψ. Thus the condition on vertices $\{a, b, e\}$ can be optimised from $f = x + y + z$ to $g = 1$. Table 1 also shows that $x + y \overset{\Psi}{\sim} \overline{z}$. This is used for optimisation of conditions of vertices c and d.

The result of optimisation of the CPOG in Figure 7(a) is shown in Figure 7(b). In total 11 literals were saved.

5.2 Arc conditions optimisation

Functional Ψ-equivalence can also be applied for arc conditions optimisation. But in addition to that it is possible to exploit two other potentialities discussed below.

Let arc $e = (a, b)$ have condition $f = \phi(e)$ and connect vertices with conditions $v_a = \phi(a)$ and $v_b = \phi(b)$. It is possible to substitute function f with another (possibly simpler) function g if the following relation is satisfied:

$$(v_a v_b f \Leftrightarrow v_a v_b g) \overset{\Psi}{\sim} 1$$

In other words if arc e exists in the original CPOG (term $v_a v_b f$) then it exists in the modified CPOG (term $v_a v_b g$) and vice versa. Note, that these terms have to be equal only up to the assignment set Ψ and thus we use $\overset{\Psi}{\sim} 1$ and not $= 1$ relation. The above equation can be simplified into

$$\overline{v_a} + \overline{v_b} + (f \Leftrightarrow g) \overset{\Psi}{\sim} 1 \tag{1}$$

For example, let's take arc (a, c) in Figure 7(b). We have $v_a = 1$, $v_c = \overline{x}$ and $f = y + z$. Using (1) you can see that it is possible to substitute f with $g = 1$: $\overline{1} + \overline{\overline{x}} + ((y + z) \Leftrightarrow 1) = x + y + z \overset{\Psi}{\sim} 1$. It saves us two literals.

Another opportunity is to optimise transitive conditions in CPOGs. For instance, arc (b, e) in Figure 7(b) is transitive with respect to path $b \rightarrow d \rightarrow e$ (this path exists in the graph if $(x + y)\overline{z}$ is true) and this fact can be exploited.

Let arc $e = (a, b)$ have condition f and path $\langle a, b \rangle \in H \backslash \{e\}$ exist in the $H \backslash \{e\}$ if condition t is true. Then it is possible to substitute function f with function g if

$$((f + t) \Leftrightarrow (g + t)) \overset{\Psi}{\sim} 1$$

In other words if a path between vertices a and b (either via arc (a, b) or via path $\langle a, b \rangle \in H \backslash \{e\}$) exists in the original CPOG (term $f + t$) then it exists in the modified CPOG (term $g + t$) and vice versa. After simplification we get

$$t + (f \Leftrightarrow g) \overset{\Psi}{\sim} 1 \tag{2}$$

We can apply (2) to arc (b, e) in Figure 7(b) and check that it can be also done unconditional ($g = 1$). We have $f = x + z$ and $t = (x + y)\overline{z}$ which gives us $(x + y)\overline{z} + ((x + z) \Leftrightarrow 1) = (x + y)\overline{z} + x + z = x + y + z \overset{\Psi}{\sim} 1$.

It is possible to combine these two techniques and get the general optimisation relation (simplified):

$$\overline{v_a} + \overline{v_b} + t + (f \Leftrightarrow g) \overset{\Psi}{\sim} 1 \tag{3}$$

Let's use (3) to optimise arc (c, e) in Figure 7(b). We have $v_c = \overline{x}$, $v_e = 1$, $t = \overline{x} y \overline{z}(x + y)$ and $f = z$. Trying to substitute f with $g = 1$ we get $\overline{\overline{x}} + \overline{1} + \overline{x} y \overline{z}(x + y) + (z \Leftrightarrow 1) = x + \overline{x} y \overline{z} + z \overset{\Psi}{\sim} 1$ and thus $g = 1$ is a proper substitution of f for arc (c, e). The result of arc conditions optimisation of CPOG in Figure 7(b) is shown in Figure 7(c). Interestingly enough the resultant CPOG does not contain conditional arcs at all and has only 3 conditional vertices. During the both optimisations the total literals count was reduced

from 29 to 3. The obtained CPOG is Ψ-equivalent to the original one in Figure 7(a) and has the same projections. For example, projection $\psi_2 \in \Psi$ is shown in Figure 7(d) and it defines PO P_2 for addition operation from Figure 5(b).

As a result of logic optimisation the obtained controller (see Figure 8) is quite small. Speed-independent implementation of controllers is based on *Transition Sequence Encoder* circuit [5] and consists of mapping of CPOG logic conditions into logic gates and equipping the resultant circuit with *go/done* interface.
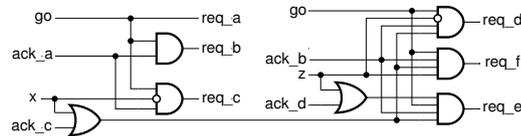
Figure 8: Gate-level implementation

6   Results and conclusions

The presented approach for control specification and synthesis was tested on a set of benchmarks that included CPU core specification and n-permutators synthesis (see Section 2). The results are compared with Petrify solutions obtained from STGs. Benchmarks show (Table 2) that CPOG specification size grows polynomially for systems with exponential number of behavioural scenarios while STGs size and Petrify runtime grow exponentially.

The presented model, CPOG synthesis and optimisation techniques and TSE-based controller implementation provide a consistent design flow for asynchronous control path specification and synthesis.

number of actions	number of scenarios	STG size (file size)	Petrify time	CPOG size (literals)	Synthesis time
3	6	1.3 Kb	1.4 sec	6	< 1 sec
4	24	5.8 Kb	191 secs	12	< 1 sec
5	120	34.3 Kb	-	20	< 1sec

Table 2: Synthesis of n-permutators

References

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

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

[3] J. Cortadella et al. 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] G. Martin L. Lavagno, L. Scheffer. *Electronic Design Automation For Integrated Circuits Handbook*. 2006.

[5] A. Mokhov and A. Yakovlev. Transition sequence encoder. Technical report, Newcastle University, 2006.

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

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

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

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

[10] K. van Berkel et al. The VLSI-programming language Tangram and its translation into handshake circuits. In *European Conference on Design Automation*, 1991.