

Conditional Partial Order Graphs: Model, Synthesis, and Application

Andrey Mokhov, *Member, IEEE*, and Alexandre (Alex) Yakovlev, *Senior Member, IEEE*

Abstract—The paper introduces a new formal model for specification and synthesis of control paths in the context of asynchronous system design. The model, called Conditional Partial Order Graph (CPOG), captures concurrency and choice in a system's behavior in a compact and efficient way. It has advantages over widely used interpreted Petri Nets and Finite State Machines for a class of systems which have many behavioral scenarios defined on the same set of actions, e.g., CPU microcontrollers. The CPOG model has potential applications in the area of microcontrol synthesis and brings new methods for modeling concurrency into the application domain of modern and future processor architectures. The paper gives the formal definition of the CPOG model, formulates and solves the problem of CPOG synthesis, and introduces various optimization techniques. The presented ideas 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, as demonstrated with several application examples.

Index Terms—Logic synthesis, concurrency, partial orders, asynchronous circuits, microarchitecture.



1 INTRODUCTION

THERE IS an issue of finding an adequate model for system description in the context of synthesis of processor microarchitectures. While the complexity of processors and multicore Systems-on-Chip has been increasing rapidly over the recent years, the conventional models reveal more and more limitations urging the development of new paradigms to facilitate automatic synthesis of microarchitecture hardware in many new applications [1]. There are a number of requirements to the model: It should be expressive enough to cover a wide range of solutions for different optimization criteria, it should capture concurrency and multiple choice, and yet be manageable, i.e., it should not overrefine the specification with unnecessary details. The latter requirement becomes especially important in designing asynchronous (or self-timed) systems [15] which do not rely on global clocking for synchronization, thus leading to massive parallelism and, in turn, to difficulties in system modeling and validation.

To date there are several design methodologies for asynchronous control logic, e.g., [14], [16]. Some approaches such as Tangram [17] and Balsa [2] use CSP-like hardware description languages (HDLs) and syntax-directed translation for synthesis. They are not well suited for control logic specification because they describe the entire system as a collection of processes and channels; control is implicit in them. Other models such as Burst-mode Finite State Machines (FSMs) [11], as well as Petri nets/Signal Transition Graphs (STGs) [4] are able to capture concurrency and

choice at a very fine level and are more suitable for control logic design: they produce more compact and faster circuits than the methods based on syntax-directed translation from HDLs. However, these models are built on explicit enumeration of all the event traces and causal relations of a system and their applicability is limited to microcontrollers with a small state space; specification of systems with many similar behavioral patterns, or event orders, is inefficient as have been demonstrated in [9] (see also Section 2 for a practical example).

In this paper, we present a new design methodology built around the Conditional Partial Order Graph model introduced recently [9]. 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 Fig. 1(center). It can be regarded as a custom associative memory for storing cause and effect relations within a predefined set of events.

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 Fig. 1(top). The order is determined by an *instruction code*—a combination of logical conditions presented to the controller by the environment [6]. 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-acknowledgment interface (data path operational units which execute the microinstructions). Thus, the condition signals dynamically reconfigure the microcontroller according to the instruction

• The authors are with the School of Electrical, Electronic, and Computer Engineering, Merz Court, Newcastle University, Newcastle upon Tyne, NE1 7RU, United Kingdom.
E-mail: {andrey.mokhov, alex.yakovlev}@ncl.ac.uk.

Manuscript received 10 May 2009; revised 29 Dec. 2009; accepted 4 Jan. 2010; published online 23 Feb. 2010.

Recommended for acceptance by I. Markov.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-05-0203.
Digital Object Identifier no. 10.1109/TC.2010.58.

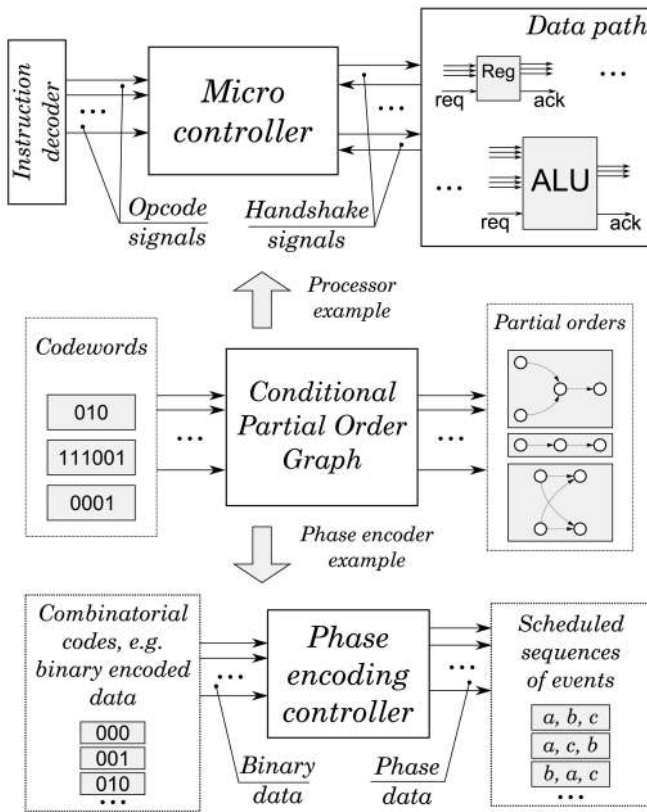


Fig. 1. Examples of dynamically reconfigurable microcontrollers.

being executed. *Microcoded control synthesis* presented in [6] 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.

Another class of controllers suitable for CPOG specification is a family of phase encoders [7]. A phase encoder is a circuit that converts data between two conceptually different information domains, see Fig. 1(bottom). The first one corresponds to the combinatorial data encoding, e.g., binary or *m-of-n* encoded data symbols [18]. The second domain is comprised of sequences of events ordered in time. CPOGs are capable of specifying such controllers without the explicit representation of all the contained behavioral scenarios thereby avoiding the combinatorial explosion of the specification (see Section 7).

Fig. 2 shows the proposed CPOG-driven flow for automated synthesis of microcontrollers. At first, the designer has to specify all the execution scenarios of the controller. A scenario is a schedule of basic events or actions that have cause and effect relationships between them. Consider a simple scenario of adding two numbers which consists of actions $\{a, b, c, d\}$:

1. Read input value X (a);
2. Read input value Y (b);
3. Compute sum $Z = X + Y$ (c); and
4. Store the result value Z (d).

One can see that action c depends on actions a and b (there is no way to compute sum Z without having values X and Y ; actions a and b are so-called passive, or material causes for

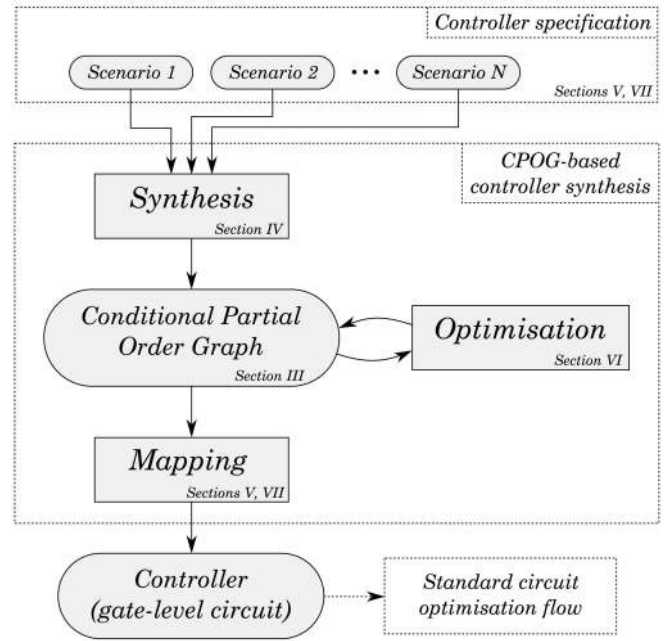
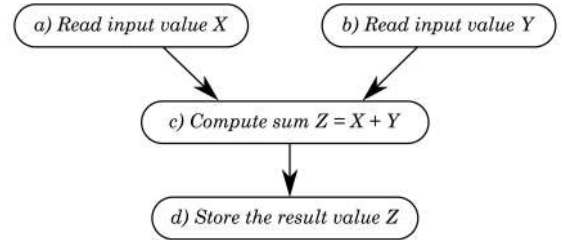


Fig. 2. CPOG-based flow for control specification and synthesis.

action c), and action d in turn cannot happen until action c is completed. This is captured by the following diagram:



The diagram depicts a *partial order* [5], a basic precedence relation on the set of actions. Every scenario is specified independently from the others.¹ This makes our approach substantially different from the STG or FSM-driven approaches that require the designer to specify the controller as a whole, often resulting in complicated and incomprehensible specifications. This is demonstrated by a simple controller with three basic scenarios used as an example in Section 2.

As soon as all the scenarios of the controller have been defined the flow proceeds to the stage of CPOG synthesis. At this stage all the scenarios are combined within a single mathematical structure—a Conditional Partial Order Graph (introduced formally in Section 3). Later on every scenario can be extracted from this structure by providing its *code*. Therefore, it is necessary to set up a correspondence between the scenarios and their codes. This correspondence is called an *encoding scheme*. Section 4 presents a formal method of synthesis and several encoding schemes which are often encountered in practice.

The obtained CPOG can be mapped into an interconnection of logic gates to produce the physical implementation

1. There are cases when many scenarios match a certain pattern and can be specified together in a functional form without their explicit enumeration as demonstrated in Section 7.1.



Fig. 3. ParSeq controller interface.

of the microcontroller as explained in Sections 5 and 7. The area and speed of the microcontroller depends on the size and structural properties of its CPOG representation. Therefore, a CPOG can undergo various optimization procedures which exploit similarities between the original scenarios and functional characteristics of their encodings (see Section 6). The obtained gate-level implementation of the controller can be further processed using the standard circuit design tools, e.g., it might require *technology mapping* for a particular gate library, or a custom technology-dependent performance optimization which are out of the scope of this paper.

The CPOG-based synthesis flow requires the designer's involvement only at the stage of system specification and scenario encoding. Therefore, the rest of the stages can be automated and the designer might even be unfamiliar with CPOGs and the underlying theory. It is also important to note that all the CPOG-related stages (synthesis, optimization, and mapping) rely only on structural methods and do not require exploration of the entire controller state space or explicit enumeration of all its behavioral scenarios, which results in high efficiency of the whole design flow.

The paper is organized as follows: Section 2 presents a motivational example demonstrating limitations of the existing specification models. The CPOG model and CPOG-based synthesis approach is formally introduced in Sections 3-4 which are followed by application of the proposed method to the previously discussed motivational example in Section 5. Various CPOG optimization techniques are presented in Section 6. The paper is concluded with a practical CPOG application example (Section 7) and Conclusions (Section 8).

2 MOTIVATIONAL EXAMPLE

There are many models targeted at microcontrol specification and synthesis and a strong reason is demanded if yet another model is to be introduced into this well-studied and established domain. This section demonstrates limitations of the existing control specification models, in particular STGs [4] and FSMs [11]. The limitations arise in situations when a specified system contains a mixture of data and control path interfaces. This leads to combinatorial explosion in the size of specification because data path modeling requires exploration of all possible combinations of signal arrivals within a single data codeword which in fact can be avoided by using different abstraction levels for data and control related events.

For example, consider a generalized *ParSeq controller*,² which manages two handshakes $A = (req_a, ack_a)$ and $B = (req_b, ack_b)$ on its right side according to the

2. ParSeq controller and its variations have several practical applications, e.g., in Balsa synthesis flow [2] or in phase encoding controllers [7].

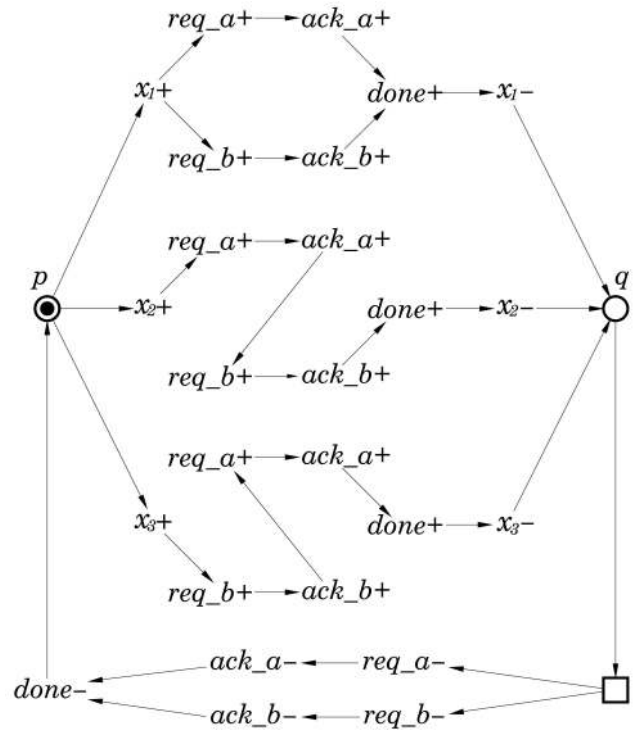


Fig. 4. An STG specification of one hot ParSeq controller.

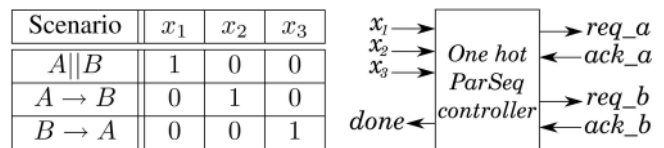
operation code (opcode) provided by an asynchronous data path interface on its left side as shown in Fig. 3.

Depending on the opcode signals the handshakes are to be initiated either in parallel (concurrent events $A||B$) or in sequence (in two possible event orders $A \rightarrow B$ or $B \rightarrow A$), hence the name of the controller. As soon as both handshakes are completed the controller issues signal *done*. The reset phase is similar but the handshakes are always reset concurrently regardless of the opcode.

The three operational scenarios can be encoded (i.e., given distinct opcodes) in different ways. The following sections demonstrate how a chosen data path encoding affects the controller specification.

2.1 One Hot Encoding

One hot encoding [18] is the most natural data path encoding in this case. It uses three signals $\{x_1, x_2, x_3\}$ to select one of the three scenarios:



Note, that combination $(0, 0, 0)$ represents a *spacer* value which separates two consecutive data symbols.

An STG specification of ParSeq controller with one hot interface is shown in Fig. 4. The STG has a global choice (place p) and the three scenarios are specified as three independent branches starting with input signals x_1+ , x_2+ , and x_3+ . The upper branch corresponds to parallel handshakes $A||B$; the two lower branches correspond to sequential handshakes $A \rightarrow B$ and $B \rightarrow A$. After the global merge (place q) the handshakes are reset concurrently and the

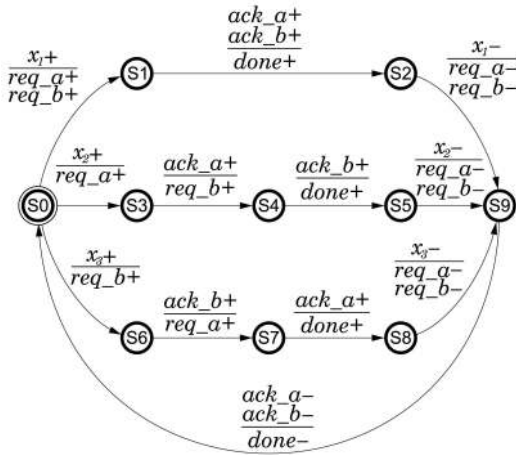


Fig. 5. An FSM specification of one hot ParSeq controller.

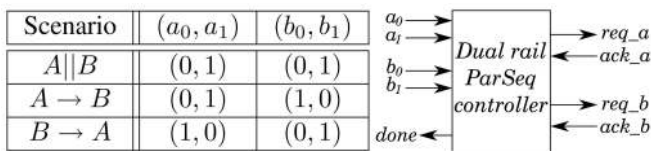
system returns to the initial state. This STG seems to be convenient, understandable, and can be designed by hand but it duplicates events in different branches which can cause exponential explosion in the size of the specification for larger controllers. This particular issue, however, is not addressed in this paper as it has been investigated in [9].

Fig. 5 shows an FSM specification of one hot ParSeq controller. Its structure is similar to the STG: it has a global choice in the initial state S_0 , three separate branches describing different scenarios, and the concurrent reset of the handshakes via state S_9 . Apart from the same event duplication issue, this specification is compact and can be easily obtained manually.

The presented specifications do not exhibit any mentioned problems of data path interface communication because of the fact that one hot encoding transfers all the opcode information within a single input transition x_k+ (which is the essence of one hot encoding, being its advantage and disadvantage at the same time). Unfortunately, one hot encoding is generally not an option for data path interfaces as it requires too many wires for data transmission. The next section shows the effect of using binary (dual rail) encoding for the opcodes.

2.2 Dual Rail Encoding

Dual rail encoding [18] uses two wires (a_0, a_1) for one data bit a encoding: signal combination $(1, 0)$ stands for Boolean value 0; $(0, 1)$ represents 1; and $(0, 0)$ is a spacer value. The three scenarios can be encoded with two dual rail signals $\{a, b\}$ as shown below:



Here, bit a can be interpreted as a permission for handshake A to happen without waiting for handshake B (bit b has a symmetric interpretation).

As can be seen from Fig. 6, the STG specification changes dramatically due to this modification of the data path interface. The reason is that the new data encoding uses two concurrent transitions to transfer an opcode (instead of only

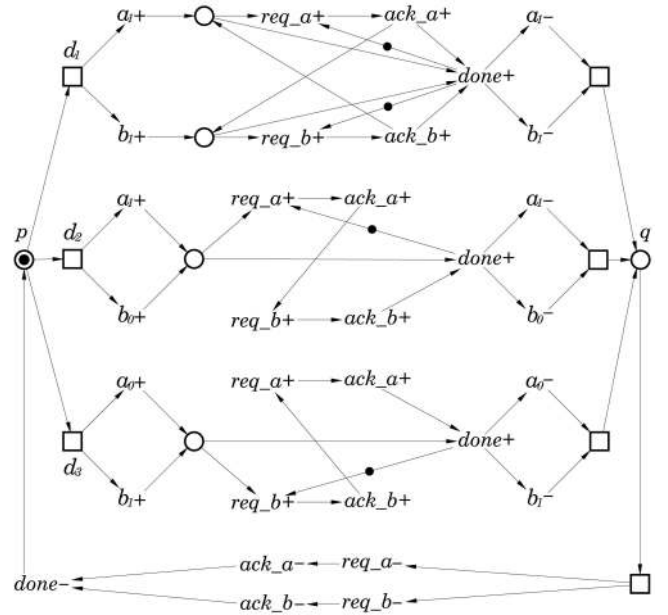


Fig. 6. STG specification of dual rail ParSeq controller.

one in one hot encoding) and all the arrival scenarios of these two transitions have to be explicitly reflected in the specification. Dummy events d_1 , d_2 , and d_3 help to simplify the specification: they do not correspond to any hardware signals but rather represent the choice of the environment, e.g., arrival of signals a_1+ and b_1+ signifies that the environment has chosen the first scenario associated with the d_1 branch of the STG, etc. (note that signal a_1+ alone is not enough to deduce the choice).

Unexpectedly, the new STG has to model *OR-causality* [20]: handshake A can be initiated as soon as at least one of signals a_1+ or b_0+ is received. As a result the opcode decoding process propagates further into the controller specification, and it is already impossible to clearly separate data and control related event flows in the STG.

The situation with the FSM specification is even worse because FSMs are not well suited for modeling concurrency in the arrival of dual rail bits, which, coupled with OR-causal behavior, leads to a very complicated FSM shown in Fig. 7.

2.3 Dual Rail Encoding (Concurrency Reduction)

In order to simplify the specification of dual rail ParSeq controller one can try to get rid of OR-causality by concurrency reduction: the controller can be restricted to wait for both dual rail signals to arrive before generating handshakes. This greatly simplifies both STG and FSM specifications (see Fig. 8) bringing their sizes back to those of one hot controller (cf. Figs. 4 and 5).

The presented examples demonstrate a high degree of sensitivity of STG and FSM specifications to minor changes in the data path interface protocol, yet from a high-level designer perspective an actual data encoding may be unimportant at all, or sometimes may even be unknown (or undecided) until the later design stages. However, even such a simple controller with three basic behavioral scenarios becomes a real challenge for manual design in

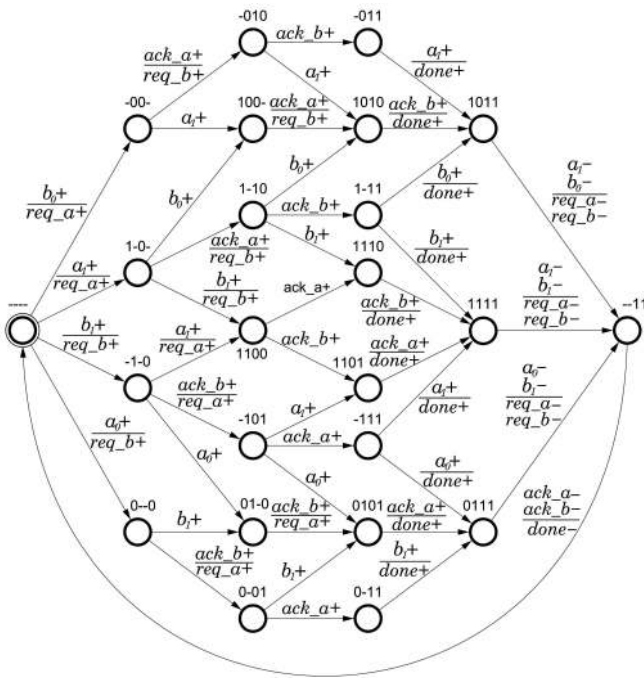


Fig. 7. FSM specification of dual rail ParSeq controller.

certain cases, and a subtle modification of data encoding requires its complete redesign.

This motivated the authors of the paper to propose a new specification model that has different levels of abstraction for data and control events. The model, called Conditional Partial Order Graph, separates control events flow within scenarios from the encoding of these scenarios and associated data path interface events. This allows specification to stay structurally unchanged under different data encodings. Moreover, the model provides an opportunity to synthesize encodings of the scenarios targeting various design optimality criteria, e.g., controller latency, average length of encodings, power balancing, etc. The model is introduced in the next section.

3 CONDITIONAL PARTIAL ORDER GRAPHS

Conditional Partial Order Graph [8], [9] is a quintuple $H(V, E, X, \rho, \phi)$, where V is a set of *vertices*, E is a set of *arcs* between them, and X is a set of *operational variables*. An *opcode* is an assignment $(x_1, x_2, \dots, x_{|X|}) \in \{0, 1\}^{|X|}$ of these variables; X can be assigned only those opcodes which satisfy the *restriction function* ρ of the graph, i.e., $\rho(x_1, x_2, \dots, x_{|X|}) = 1$. Function ϕ assigns a Boolean *condition* $\phi(z)$ to every vertex and arc $z \in V \cup E$ of the graph.

Fig. 9a 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—see Fig. 9b.

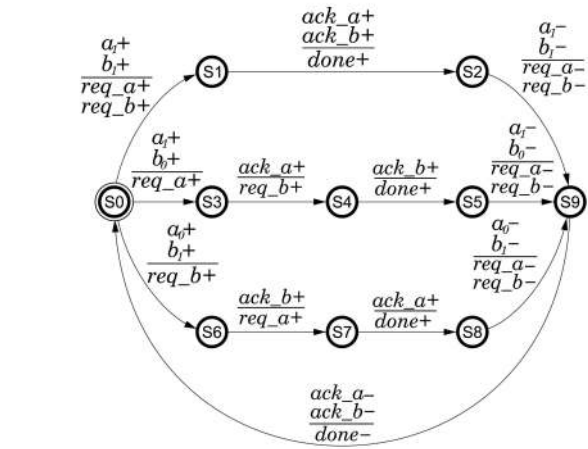
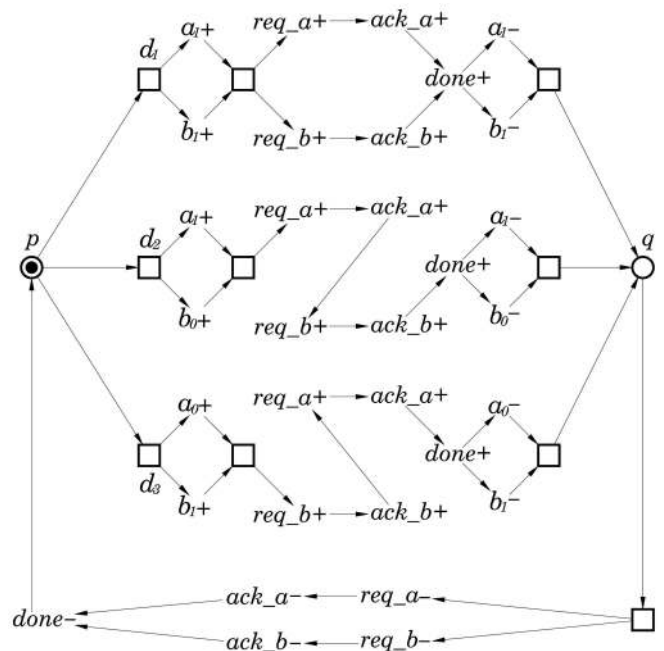


Fig. 8. STG and FSM specifications after concurrency reduction.

The purpose of vertex and arc conditions is to “switch off” some vertices and/or arcs in the graph according to the given opcode. This makes CPOGs capable of containing multiple *projections* as shown in Fig. 10. The leftmost projection is obtained by keeping in the graph only those vertices and arcs whose conditions evaluate to Boolean 1 after substitution of

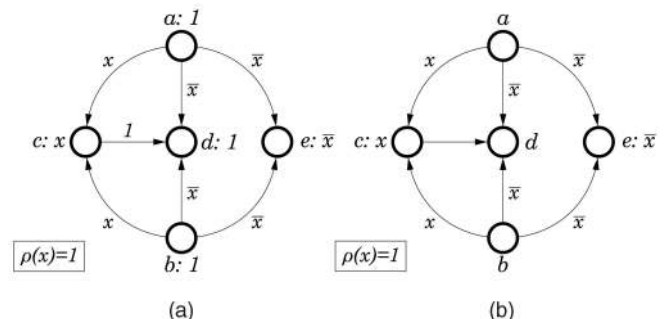


Fig. 9. Graphical representation of Conditional Partial Order Graphs: (a) full notation, and (b) simplified notation.

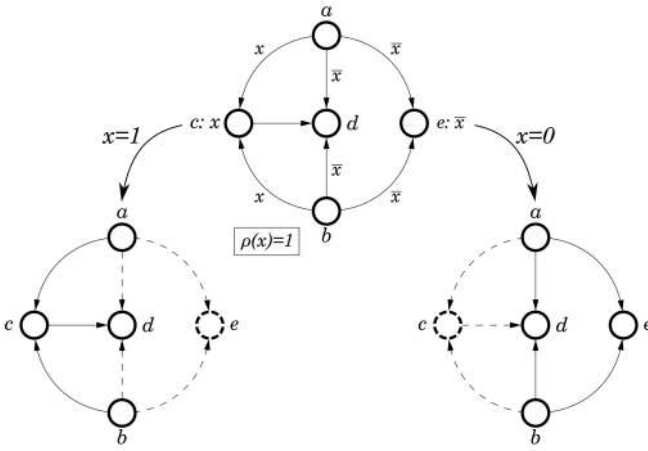


Fig. 10. Multiple projections contained within a single CPOG.

the operational variable x with Boolean 1. Hence, vertex e disappears, because its condition evaluates to 0: $\phi(e) = \bar{x} = \bar{1} = 0$. Arcs $\{(a, d), (a, e), (b, d), (b, e)\}$ disappear for the same reason. The rightmost projection is obtained in the same way but the opcode is $x = 0$. Note also that 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 behavioral scenario of the modeled system. For example, the leftmost projection corresponds to the operation of addition (cf. the partial order in Section 1). 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.

We will use notation $H|_{\psi}$ to denote a projection of a CPOG H under opcode $\psi = (x_1, x_2, \dots, x_{|X|})$. A projection $H|_{\psi}$ is called *valid* iff opcode ψ is allowed by the restriction function, i.e., $\rho(x_1, x_2, \dots, x_{|X|}) = 1$, and the resultant graph is acyclic. The latter requirement is needed to guarantee that the graph defines a partial order of events (otherwise it would not be possible to schedule them without deadlocks).

A graph H is *well-formed* iff its every allowed opcode ψ generates a valid projection $H|_{\psi}$. Let the set of all well-formed CPOGs be denoted as \mathcal{W} . The graph H in Fig. 10 is well-formed, because both projections $H|_{x=1}$ (left) and $H|_{x=0}$ (right) are valid.

The set of all partial orders defined by a well-formed graph H is denoted as $\mathcal{P}(H)$. Graphs $H_1 \in \mathcal{W}$ and $H_2 \in \mathcal{W}$ are called *equivalent* (denoted as $H_1 \sim H_2$) iff they define the same set of partial orders:

$$(H_1 \sim H_2) \stackrel{\text{df}}{=} \mathcal{P}(H_1) = \mathcal{P}(H_2).$$

Fig. 11 shows three equivalent graphs $H_a \sim H_b \sim H_c$. Graph H_a in Fig. 11a is taken from the previous example. Fig. 11b shows graph H_b with the modified operational vector. It contains two variables $X = \{x, y\}$ which are restricted in the one hot manner: only opcodes $\psi_1 = (0, 1)$ and $\psi_2 = (1, 0)$ are allowed with the restriction function $\rho(x, y) = x \oplus y$. Graph H_c in Fig. 11c does not contain any arc conditions (which are in fact redundant), and it also has inverted encodings compared to H_a . In spite of the seeming difference between the three graphs, they are equivalent as they define the same set of two partial orders $\mathcal{P}(H_a) = \mathcal{P}(H_b) = \mathcal{P}(H_c)$.

It is useful to introduce a measure of complexity of graphs in order to be able to compare them within the same equivalence class. For instance, graph H_c in Fig. 11 has simpler description in comparison with graphs H_a and H_b and is preferred in most cases.

The *complexity* (or *size*) $C(H)$ of graph $H(V, E, X, \rho, \phi)$ is measured in the number of literals contained in the restriction function ρ and conditions $\phi(z), z \in V \cup E$:

$$C(H) \stackrel{\text{df}}{=} C(\rho) + \sum_{v \in V} C(\phi(v)) + \sum_{e \in E} C(\phi(e)),$$

where $C(f), f \in \mathcal{F}(X)$ denotes the literal count [19] of a Boolean function f . Looking at graphs in Fig. 11 one can see that $C(H_a) = 0 + 2 + 6 = 8$, $C(H_b) = 2 + 2 + 6 = 10$, and $C(H_c) = 0 + 2 + 0 = 2$. So, graph H_c can be called *optimal* in this context. Methods for graphs size optimization are addressed in Section 6. Size of the physical controller implementation closely correlates with complexity $C(H)$ of its specification H . Therefore, we use $C(H)$ as an adequate estimate of CPOG H efficiency and aim to minimize $C(H)$ in the presented optimization procedures. The removal of all the redundant conditions from a graph and use of a canonical representation of Boolean functions leads to the canonical CPOG description of a set of encoded partial orders.

Two well-formed graphs H_1 and H_2 are said to be *in conflict* w.r.t. their restriction functions ρ_1 and ρ_2 iff $\rho_1 \rho_2 \neq 0$. A conflict implies the existence of an encoding ψ such that

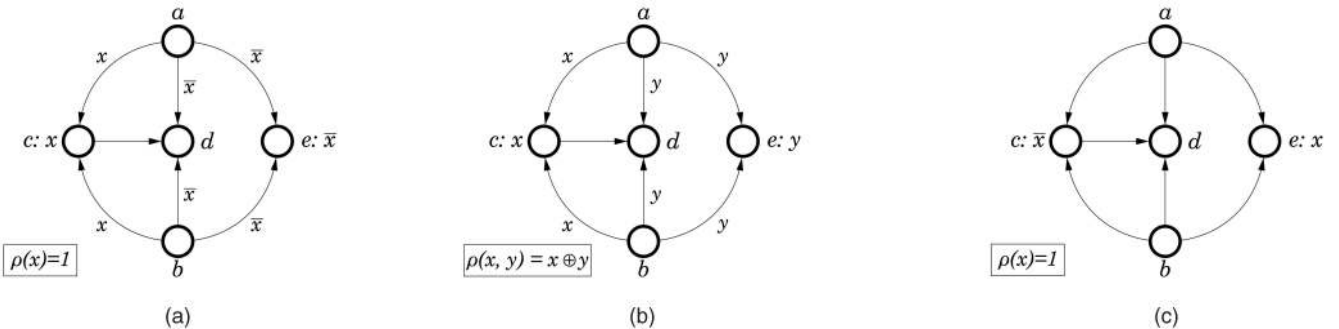


Fig. 11. Three equivalent graphs: (a) example graph (8 literals), (b) graph with two control variables (10 literals), and (c) no redundant conditional arcs (2 literals).

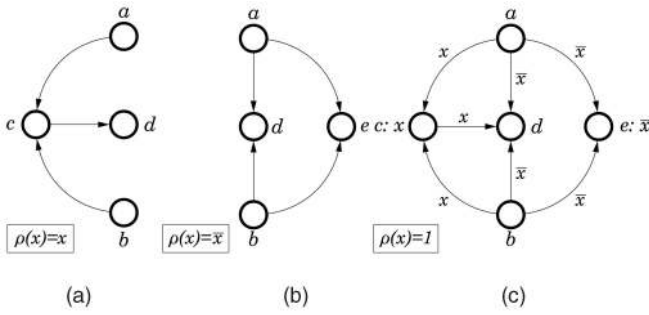


Fig. 12. Graph addition: (a) H_1 , (b) H_2 , and (c) $H_1 + H_2$.

both the restriction functions are satisfied. This leads to an ambiguity in some cases (e.g., in case of graph addition introduced in Section 3.1), when two graphs describe different behavior under the same encoding ϕ .

3.1 Addition

The result of addition of graphs $H_1(V_1, E_1, X_1, \rho_1, \phi_1)$ and $H_2(V_2, E_2, X_2, \rho_2, \phi_2)$ is graph $H(V_1 \cup V_2, E_1 \cup E_2, X_1 \cup X_2, \rho_1 + \rho_2, \phi)$, where the vertex/arc conditions ϕ are defined as

$$\forall z \in V_1 \cup V_2 \cup E_1 \cup E_2, \quad \phi(z) \stackrel{\text{df}}{=} \rho_1 \bar{\rho}_2 \phi_1(z) + \bar{\rho}_1 \rho_2 \phi_2(z).$$

Addition is denoted using the standard notation $H = H_1 + H_2$.

Theorem 1. Pair $(\mathcal{W}, +)$ is a commutative semigroup [5], i.e., set of well-formed graphs \mathcal{W} is closed under addition $+$, which is an associative and commutative operation [8].

Corollary 1. When adding more than two graphs the redundant brackets can be omitted without any ambiguity: $H_1 + H_2 + H_3$.

In the same way as graphs H_1 and H_2 are considered to be specifications of certain behavioral scenarios over event domains V_1 and V_2 , graph $H_1 + H_2$ is considered to be specification of the scenarios from both the graphs over the joint event domain $V = V_1 \cup V_2$. This is formally stated in the following theorem.

Theorem 2. If H_1 and H_2 are well-formed graphs that are not in conflict then $\mathcal{P}(H_1 + H_2) = \mathcal{P}(H_1) \cup \mathcal{P}(H_2)$, i.e., graph $H_1 + H_2$ contains partial orders from both H_1 and H_2 , preserving their encodings [8].

Consider an example of addition in Fig. 12. Each of graphs H_1 and H_2 specifies a single scenario. The graphs are not in conflict ($\rho_1 \rho_2 = x \bar{x} = 0$), the result of their addition $H_1 + H_2$ is shown in Fig. 12c. It contains both of the scenarios (as was demonstrated in Fig. 10). Another example of graph addition is shown in Fig. 13.

3.2 Scalar Multiplication

Graph $H(V, E, X, \rho, \phi)$ can be multiplied by a Boolean function $f \in \mathcal{F}(Y)$ (which can be called scalar). The resultant graph is $H'(V, E, X \cup Y, f\rho, \phi)$. The standard notation will be used for scalar multiplication: $H' = fH$.

Theorem 3. For every Boolean function f and well-formed graph H , graph $H' = fH$ is also well-formed and $\mathcal{P}(H') \subseteq \mathcal{P}(H)$ [8].

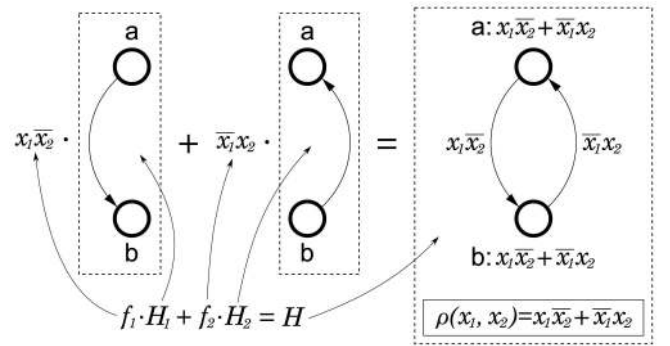


Fig. 13. One hot CPOG synthesis.

A linear combination of $n \geq 1$ graphs H_1, H_2, \dots, H_n and scalars f_1, f_2, \dots, f_n is

$$\sum_{1 \leq k \leq n} f_k H_k \stackrel{\text{df}}{=} f_1 H_1 + f_2 H_2 + \dots + f_n H_n.$$

Note that any linear combination of well-formed graphs is also well-formed due to the closure of addition and scalar multiplication operations over \mathcal{W} (Theorems 1 and 3).

Fig. 13 shows linear combination $H = f_1 H_1 + f_2 H_2$ of graphs (H_1, H_2) w.r.t. scalars (f_1, f_2) .

4 CPOG SYNTHESIS

The previous section showed that a CPOG can contain several partial orders in a compressed form and thus can be used to specify a system with several behavioral scenarios. Mokhov and Yakovlev [9] showed how to synthesize a compact CPOG system specification given its description as a set of partial orders corresponding to different scenarios in the modeled system.

Formally, let $\{P_1, P_2, \dots, P_n\}$ be a set of n given partial orders. The objective is to synthesize a CPOG $H(V, E, X, \rho, \phi)$ such that

$$\mathcal{P}(H) = \{P_1, P_2, \dots, P_n\}. \quad (1)$$

The idea behind the synthesis approach presented in [9] is to represent H as the following linear combination of acyclic graphs H_k such that $\mathcal{P}(H_k) = P_k$:

$$H = f_1 H_1 + f_2 H_2 + \dots + f_n H_n = \sum_{1 \leq k \leq n} f_k H_k, \quad (2)$$

where encoding functions $f_k \in \mathcal{F}(X)$ are orthogonal, i.e., $f_j f_k = 0, 1 \leq j < k \leq n$ and are not contradictions: $f_k \neq 0, 1 \leq k \leq n$. According to Theorems 2 and 3, this guarantees that $\mathcal{P}(H) = \mathcal{P}(H_1) \cup \mathcal{P}(H_2) \cup \dots \cup \mathcal{P}(H_n) = \{P_1\} \cup \{P_2\} \cup \dots \cup \{P_n\}$. Thus, (2) satisfies the synthesis requirement (1).

Opcode signals X and functions f_k can be selected in different ways depending on the chosen encoding scheme. The following sections describe three most commonly used encoding schemes. Note that in many practical applications a scheme is given as part of system specification and every scenario is assigned a particular opcode which cannot be changed. However, in some cases a designer can choose from several allowed encoding schemes and opcode assignments.

It should be mentioned that an encoding scheme does not necessarily have to be *delay insensitive* [18]—this depends on the opcode interface of the microcontroller that is being synthesized: if it is delay insensitive so must be the encoding scheme, otherwise non delay insensitive codes can be used allowing the microcontroller to make use of timing assumptions in order to simplify the decoding logic and decrease the number of opcode wires. This is a usual trade-off between robustness and performance of a system. The CPOG-based design flow provides support for both types of controllers.

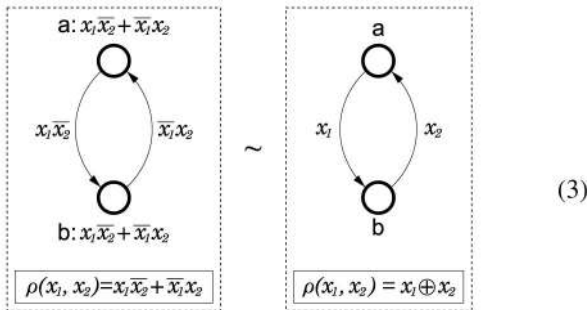
4.1 One Hot Encoding Scheme

In this scheme n operational signals $X = \{x_1, x_2, \dots, x_n\}$ are used to select a particular scenario (cf. Section 2.1). Functions f_k are set to

$$f_k = x_k \bigwedge_{\substack{1 \leq j \leq n \\ j \neq k}} \bar{x}_j$$

establishing one hot encodings of the scenarios: P_1 is encoded as $(x_1, x_2, x_3, \dots) = (1, 0, 0, \dots)$, P_2 —as $(x_1, x_2, x_3, \dots) = (0, 1, 0, \dots)$, etc.

Fig. 13 shows an example of synthesis of a CPOG containing partial orders $P_1 = \{a < b\}$ and $P_2 = \{b < a\}$. The opcode signals set is $\{x_1, x_2\}$ and the encoding functions are $f_1 = x_1 \bar{x}_2$ and $f_2 = \bar{x}_1 x_2$. The result $H = f_1 H_1 + f_2 H_2$ contains both partial orders as projections $H|_{x_1=1, x_2=0}$ and $H|_{x_1=0, x_2=1}$. It is possible to optimize it reducing the literal count from 16 to 4 (see Section 6 for details):



One hot scheme provides a simple and intuitive way of encoding partial orders but it is inefficient because of the large size of opcode signals set: $|X| = n$. It is not practical for synthesis of CPOGs containing large number of scenarios.

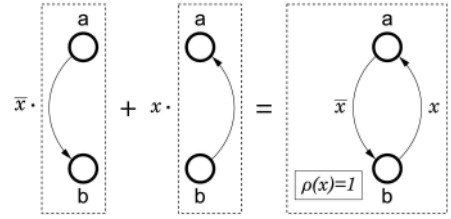
4.2 Binary Encoding Scheme

In binary scheme only $m = \lceil \log_2 n \rceil$ operational variables $X = \{x_1, x_2, \dots, x_m\}$ are used to encode n given scenarios which is the theoretical minimum. Let b_{jk} denote j th bit of integer number k . Then we can define encoding functions f_k as:

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

For example, if $n = 3$, we get $f_1 = (x_1 \Leftrightarrow 0)(x_2 \Leftrightarrow 0) = \bar{x}_1 \bar{x}_2$, $f_2 = (x_1 \Leftrightarrow 1)(x_2 \Leftrightarrow 0) = x_1 \bar{x}_2$ and $f_3 = (x_1 \Leftrightarrow 0)(x_2 \Leftrightarrow 1) = \bar{x}_1 x_2$ resulting in natural binary encodings of the three partial orders: $\psi_1 = (0, 0)$, $\psi_2 = (1, 0)$, and $\psi_3 = (0, 1)$.

Application of the binary encoding scheme to synthesis of a CPOG containing partial orders $P_1 = \{a < b\}$ and $P_2 = \{b < a\}$ leads to a very compact (only two literals) specification:



Observe the difference between this result and the optimized version of the one hot solution (3). As one can see the selected encoding scheme does not affect the structure of the synthesized CPOG. However, it affects complexity of the functions and size of the physical controller implementation.

4.3 Matrix Encoding Scheme

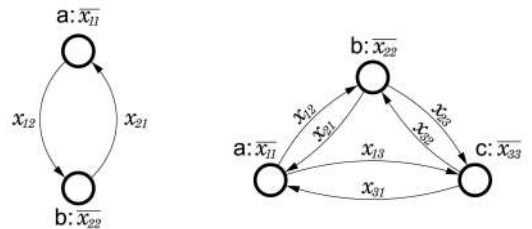
Size of the opcode signals set in this scheme does not depend on the number of scenarios. It depends only on the number of different events in the system— $|V|$. In particular, operational variables form an *operational matrix* $X = \{x_{jk}, j = 1 \dots |V|, k = 1 \dots |V|\}$. The matrix has enough information capacity to describe any partial order $P(V', <)$ on a subset $V' \subseteq \{e_1, e_2, \dots, e_{|V|}\}$ of $|V|$ events:

$$\psi(x_{jk}) = \begin{cases} 1 & \text{if } (e_j \in V') \wedge (e_k \in V') \wedge (e_j < e_k) \\ 0 & \text{otherwise,} \end{cases} \quad (4)$$

$$\psi(x_{kk}) = \begin{cases} 1 & \text{if } (e_k \notin V') \\ 0 & \text{otherwise.} \end{cases}$$

Instead of direct application of this encoding scheme to (2), we can use a generic solution with its subsequent optimization taking into account the given scenarios. The optimization removes the variables that remain constant throughout all the scenarios and makes the corresponding vertices and arcs unconditional.

Generic solutions for systems with two and three events are given below³:




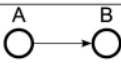
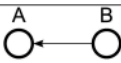
For example, using (4) to encode partial orders $P_1 = \{a < b\}$ and $P_2 = \{b < a\}$ gives us these encoding matrices:

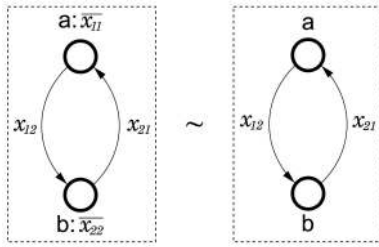
$$\psi_{1,2} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}; \quad \psi_1 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad \psi_2 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}.$$

Diagonal elements x_{kk} are constant zeros therefore vertices a and b become unconditional. As a result, the generic matrix graph is reduced to one hot solution (up to control variables renaming—see (3)):

3. A generic solution for an arbitrary number of events $V = \{e_1, \dots, e_{|V|}\}$ is a fully connected graph $K_{|V|}$ with conditions $\phi(e_k) = \bar{x}_{kk}$ and $\phi((e_j, e_k)) = x_{jk}$.

TABLE 1
Four Scenarios of a ParSeq Controller

Scenario	Partial order $P(V, <)$			
	#	V	$<$	graph
Parallel $A B$	P_1	$\{A, B\}$	\emptyset	
Sequential $A \rightarrow B$	P_2	$\{A, B\}$	$\{A < B\}$	
Sequential $B \rightarrow A$	P_3	$\{A, B\}$	$\{B < A\}$	
Spacer	P_4	\emptyset	\emptyset	(empty)



Matrix encoding scheme is general in the sense that it can be used to encode any possible behavioral scenario of a system with n events in a reasonably compact and intuitive way. It is a trade-off between one hot encoding which is straightforward but inefficient in terms of the number of opcode signals and binary encoding which has the least possible encodings length but more complicated encoding functions which are not affordable in some cases. The efficiency of matrix encoding scheme allowed us to specify and synthesize *phase encoding repeaters* (see Section 7.1) for up to 10 wires having $10! = 3,628,800$ different behavioral scenarios.

5 SYNTHESIS OF PARSEQ CONTROLLERS

This section shows how the ideas presented above can be applied to specification and synthesis of one hot and dual rail ParSeq controllers introduced in Section 2.

The event domain in this case consists of two events $V = \{A, B\}$ corresponding to the handshakes. The behavioral scenarios can be represented with partial orders as shown in Table 1. Note that the spacer scenario is explicitly defined.

5.1 One Hot Encoding

At first, consider synthesis of a CPOG H for one hot ParSeq controller (Section 2.1). Operational signals are $X = \{x_1, x_2, x_3\}$. The table below shows the encodings of partial orders $\mathcal{P}(H) = \{P_1, P_2, P_3, P_4\}$ and the corresponding encoding functions f_k :

Partial order P_k	P_1	P_2	P_3	P_4
Encoding (x_1, x_2, x_3)	$(1, 0, 0)$	$(0, 1, 0)$	$(0, 0, 1)$	$(0, 0, 0)$
Encoding function f_k	$x_1 \bar{x}_2 \bar{x}_3$	$\bar{x}_1 x_2 \bar{x}_3$	$\bar{x}_1 \bar{x}_2 x_3$	$\bar{x}_1 \bar{x}_2 \bar{x}_3$

Note that the four unused encodings represent a *don't care* set [6] and can be used for logic optimization of CPOGs and final signal equations (e.g., by using ESPRESSO [13] logic minimization tool which supports don't cares). Moreover, a restriction function ρ of the synthesized graph describes this

don't care set in a very compact form: all encodings ψ such that $\rho|_{\psi} = 0$ are don't cares. See Section 6.1 for details.

The resultant graph $H = f_1 H_1 + f_2 H_2 + f_3 H_3 + f_4 H_4$ (where $\mathcal{P}(H_k) = P_k$) after logic minimization is shown in Fig. 14a. It can now be mapped into logic gates to produce a physical implementation of the controller. Event e_k is enabled to fire (req_k is excited) if all the preceding events have already fired (ack_j have been received)⁴:

$$req_k = \phi(e_k) \cdot \bigwedge_{\substack{1 \leq j \leq |V| \\ j \neq k}} (\phi(e_j) \cdot \phi((e_j, e_k)) \Rightarrow ack_j). \quad (5)$$

For example, signal req_a is mapped as

$$req_a = (x_1 + x_2 + x_3)((x_1 + x_2 + x_3)x_3 \Rightarrow ack_b),$$

which can be optimized taking into account don't cares defined by the restriction function $\rho = \bar{x}_1 \bar{x}_2 + \bar{x}_1 \bar{x}_3 + \bar{x}_2 \bar{x}_3$:

$$req_a = x_1 + x_2 + x_3 ack_b.$$

This final equation is very easy to interpret: request to event A can be generated immediately in two first scenarios ($A||B$ and $A \rightarrow B$) while in the third scenario ($B \rightarrow A$) it should happen only upon arrival of acknowledgment from event B ; opcode $(0, 0, 0)$ forces req_a to reset (the spacer scenario). Equation for req_b is similar which leads to the controller shown in Fig. 14b. Signal *done* acknowledges the completion of both handshakes. It was verified that both PETRIFY [3] and 3D [21] synthesis tools generate the same controller given the STG and FSM specifications from Figs. 4 and 5, so all the three specifications (in STG, FSM, and CPOG models) describe exactly the same controller.

5.2 Dual Rail Encoding

Dual rail ParSeq controller (Section 2.2) has four opcode signals: $X = \{a_0, a_1, b_0, b_1\}$. The scenarios are encoded as follows:

P_k	P_1	P_2	P_3	P_4
Encoding	01 01	01 10	10 01	00 00
$a_0 a_1 \ b_0 b_1$		00 10	00 01	
		01 00	10 00	
f_k	$\bar{a}_0 \bar{a}_1 \bar{b}_0 \bar{b}_1$	$b_0 + a_1 \bar{b}_1$	$a_0 + b_1 \bar{a}_1$	$\bar{a}_0 \bar{a}_1 \bar{b}_0 \bar{b}_1$

Note that scenarios P_2 and P_3 have more than one encoding: this reflects the fact that the controller can start generating events after receiving only a partial opcode (OR-causality modeling has been shifted to the stage of scenarios encoding in CPOG-based synthesis flow). Graph containing all these scenarios encoded with functions f_k is shown in Fig. 15a.

The final equations obtained using (5) are

$$req_a = a_1 + b_0 + (a_0 + b_1)ack_b,$$

$$req_b = a_0 + b_1 + (a_1 + b_0)ack_a.$$

Signal *done* should acknowledge completion of both handshakes and also the arrival of a complete opcode (this

4. Here, $a \Rightarrow b$ stands for Boolean implication indicating "b if a" relation. It shouldn't be mixed with Boolean equivalence $a \Leftrightarrow b$ or "b if and only if a" relation [5].

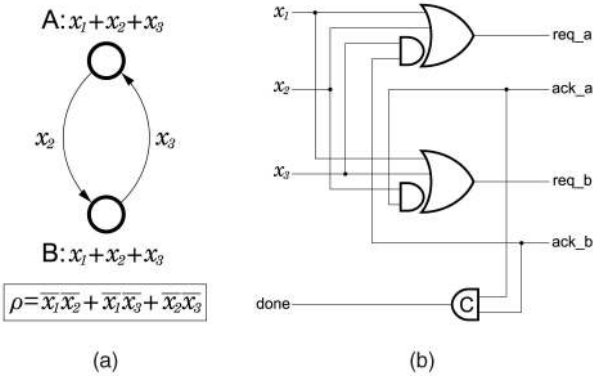


Fig. 14. Specification and implementation of one hot ParSeq controller: (a) CPOG specification, and (b) implementation.

is needed because in some cases the controller can finish the handshakes having only partial opcode information). Gate-level implementation of the controller is shown in Fig. 15b. Signal *done* is decomposed into several 2-input gates outlined with a dotted line. There can be several possible decompositions (note that complex gates generating signals *req_a* and *req_b* may have to be decomposed as well). Logic decomposition is one of the key problems of circuit synthesis [4] and is out of scope of this paper. PETRIFY and 3D produce the same controller (without signal *done* decomposition) given the specifications from Section 2.2.

Once again the CPOG specification stays structurally unchanged for different scenario encodings as can be seen from Figs. 14a and 15a. This is a very important and convenient feature: it gives designer an opportunity to change encoding without graph resynthesis because it is possible to substitute opcode variables with different ones and to rewrite the corresponding conditions.

The CPOG model has different levels of abstraction for data and control path events and is beneficial for specification and synthesis of controllers that have both kinds of interfaces. Data path events and all the choice in the system are modeled with Boolean functions, while control path events and concurrency associated with them are modeled with partial orders. This combination of comprehensively studied Boolean algebra and graphs allows a lot of powerful optimization techniques to be reused.

6 OPTIMIZATION TECHNIQUES

The size of the physical controller implementation is proportional to the size of its CPOG specification measured as the total number of literals in its conditions [9]. There are different optimization techniques reducing the size of a given CPOG by functional logic minimization and/or by exploiting structural graph properties.

All the techniques presented here preserve the equivalence class of a given CPOG (i.e., the resultant optimized graph is equivalent to the given one) and the original encodings of partial orders (i.e., the opcodes of scenarios remain the same).

6.1 Logic Minimization

The most evident optimization opportunity comes from the fact that all Boolean conditions ϕ in a CPOG $H(V, E, X, \rho, \phi)$

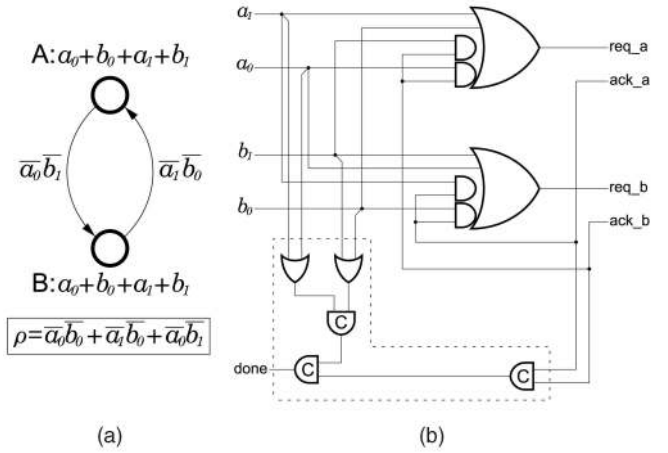


Fig. 15. Specification and implementation of dual rail ParSeq controller: (a) CPOG specification, and (b) implementation.

can be minimized by taking into account the don't care set defined with restriction function ρ .

Let $z \in V \cup E$ be an arc or a vertex in a graph $H(V, E, X, \rho, \phi)$ having condition $f = \phi(z)$. Then it is possible to substitute function f with another (possibly simpler) function g iff the following Boolean equation is a tautology⁵:

$$\rho \Rightarrow (f \Leftrightarrow g). \quad (6)$$

The intuition here is that function g must evaluate to the same value as f only under valid opcode variable assignments ψ (when $\rho|_{\psi} = 1$) and is unconstrained otherwise.

In particular, substitution $g = (\rho \Rightarrow f)$ appears to be quite good in practice (it forces the condition to evaluate to 1 in don't care entries of the truth table thus simplifying the min-terms). According to (6) it is a valid substitution, because

$$\begin{aligned} \rho \Rightarrow (f \Leftrightarrow g) &= \rho \Rightarrow (f \Leftrightarrow (\rho \Rightarrow f)) = \\ \rho \Rightarrow (\overline{f}(\overline{\rho} + f) + f(\overline{\rho} + f)) &= \rho \Rightarrow (\rho + f) = 1. \end{aligned}$$

For example, the original condition

$$\phi(A) = x_1\overline{x_2}\overline{x_3} + \overline{x_1}x_2\overline{x_3} + \overline{x_1}\overline{x_2}x_3$$

obtained after one hot synthesis of ParSeq controller in Section 5.1 is minimized into

$$\phi_{opt}(A) = (\rho \Rightarrow \phi(A)) = x_1 + x_2 + x_3,$$

which is shown in Fig. 14a. $\rho = \overline{x_1}\overline{x_2} + \overline{x_1}\overline{x_3} + \overline{x_2}\overline{x_3}$ in this case.

6.2 Implicit Arc Exclusion

Whenever a vertex is excluded from a graph all its adjacent arcs are also excluded even if their conditions evaluate to Boolean 1. This can be exploited as follows:

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 function g iff the following relation is a tautology:

$$v_a v_b \rho \Rightarrow (f \Leftrightarrow g). \quad (7)$$

5. A Boolean function is called tautology iff it is true under any possible assignment of its parameters [5].

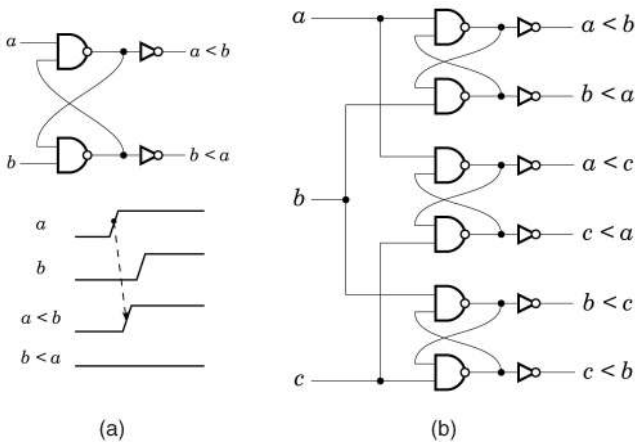


Fig. 16. Phase detection: (a) Mutex element, and (b) three-wire phase detector.

In other words, the don't care set is extended to include those opcodes in which arc e is implicitly excluded because of the exclusion of one of its vertices (cf. (6)).

For example, consider arc (A, B) in the CPOG from Fig. 15a: $\rho = \overline{a_0} \overline{b_0} + \overline{a_1} \overline{b_0} + \overline{a_0} \overline{b_1}$, $v_A = v_B = a_0 + b_0 + a_1 + b_1$, and $f = b_0 + a_1 \overline{b_1}$ (this is the original unoptimized condition on the arc). We would like to substitute f with a simpler function $g = \overline{a_0} \overline{b_1}$ (which is used in the figure). However, this is not a valid substitution according to (6) because $f \neq g$ in the spacer scenario ($a_0 = a_1 = b_0 = b_1 = 0$). On the other hand, vertices A and B are excluded from the graph in this scenario, thus the actual value of the condition on arc (A, B) is not important. This is captured in (7) which considers g to be a valid substitution of f .

6.3 Transitive Arc Reduction

Another optimization opportunity is to reduce the transitive arc conditions. For instance, arc $e = (a, b)$ in Fig. 20(left) is transitive w.r.t. path $a \rightarrow c \rightarrow b$ when $x_2 = 1$. Clearly, an indirect dependency between events a and b is enough to establish the order relation between them, hence the condition $\phi(e) = x_1 + x_2 + x_5$ can be optimized into $\phi_{opt}(e) = x_1 + x_5$. All the other arcs in the graph from Fig. 20 can be optimized in the same way leading to the graph shown to the right.

Formally, let arc $e = (a, b)$ have condition $f = \phi(e)$ and a transitive path $\langle a, b \rangle$ exist in graph $H \setminus \{e\}$ if condition t is true. Then it is possible to substitute function f with function g iff the following relation is a tautology:

$$\overline{t}\rho \Rightarrow (f \Leftrightarrow g).$$

In other words all the opcodes in which the transitive path $\langle a, b \rangle$ is activated ($t = 1$) are added to the don't care set of arc (a, b) . It is possible to combine this technique with the previous one to obtain the general arc optimization equation:

$$v_a v_b \overline{t}\rho \Rightarrow (f \Leftrightarrow g). \quad (8)$$

6.4 Common Factors Extraction

The three optimization techniques presented above can be applied to every vertex or arc independently from the others thereby increasing the efficiency of CPOG optimization

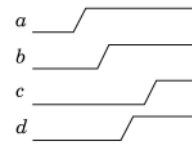


Fig. 17. Data symbol in multiple rail phase encoding channel.

algorithms. However, the actual circuit implementation of the controller may share the common factors (subexpressions) of the conditions for the purpose of area minimization. This requires the joint optimization of the conditions, which is a more time consuming procedure and may not be affordable for large designs.

The CPOG specification from Fig. 15a can serve as a simple demonstration of common factors extraction. Vertex condition $\phi(A) = a_0 + b_0 + a_1 + b_1$ can be expressed in terms of arc conditions $\alpha = \overline{a_0} \overline{b_1}$ and $\beta = \overline{a_1} \overline{b_0}$ as $\phi_{opt}(A) = \overline{\alpha\beta}$. This reduces the CPOG size by four literals. However, utilization of common factors may slow down the resultant controller in some cases, therefore, finding an appropriate trade-off between area and performance is necessary.

7 APPLICATION EXAMPLE: PHASE ENCODERS

This section demonstrates application of CPOG-based approach to synthesis of *phase encoding controllers* [7].

The *multiple rail phase encoding* protocol uses several wires for communication and data is encoded in the order of occurrence of transitions in the communication lines. Fig. 17 shows an example of a data packet transmission over a 4-wire phase encoding communication channel. The order of rising signals on wires $\{a, b, c, d\}$ indicates that permutation $abcd$ is being sent. In total it is possible to send $n!$ different permutations over an n -wire channel. This makes the multiple rail phase encoding protocol very attractive for its information efficiency [7].

Phase encoding controllers contain an exponential number of behavioral scenarios w.r.t. the number of wires and are very difficult for specification and synthesis using conventional approaches. The CPOG model suits perfectly for this class of systems and provides a compact specification and efficient synthesis method which is demonstrated below.

7.1 Phase Encoding Repeater

Phase encoding repeater is a circuit that regenerates the deteriorating phase difference between signals in the phase encoding communication channel, i.e., it receives a phase encoded data packet and retransmits it further over the channel. Hence, it consists of two functional parts: a receiver (a *phase detector*, which determines the order of the incoming transitions) and a sender (a *phase encoder* generating a series of transitions in the order they were received).

Phase detector for an n -wire communication channel consists of $\binom{n}{2}$ *mutual-exclusion (mutex) elements*. They determine the order of n transitions by comparing their arrival times pairwise (see Fig. 16 for an example of a simple mutex element implementation and three-wire phase detector).

The result of phase detection can be converted into a more common encoding domain, e.g., into binary codes,

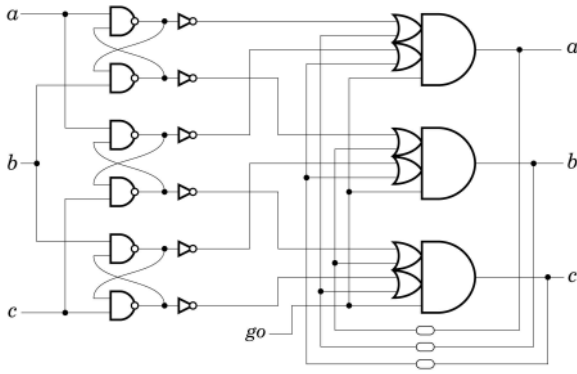
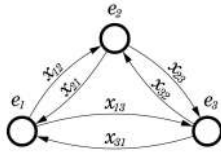


Fig. 18. Three-wire phase encoding repeater.

which can be processed using the binary sender. However, one can notice that the outputs of the mux elements form an operational matrix (4) with zeroes on its main diagonal. Therefore, it is possible to use the matrix encoding scheme directly (see Section 4.3) to avoid additional binary conversion circuitry. This gives the following CPOG specification for 3-wire matrix phase encoder (events e_k correspond to output transitions, and arc conditions x_{jk} represent the result of phase detection):



Having synthesized the CPOG we can derive Boolean equations for physical controller implementation. The controller should have $n^2 - n$ inputs $X = \{x_{jk}, 1 \leq j, k \leq n, j \neq k\}$ and n outputs $T = \{t_1, t_2, \dots, t_n\}$. Output transition t_k is enabled to fire if all the preceding (w.r.t. the partial order specified by control matrix X) transitions have already fired:

$$t_k = \bigwedge_{\substack{1 \leq j \leq n \\ j \neq k}} (x_{jk} \Rightarrow t_j) = \bigwedge_{\substack{1 \leq j \leq n \\ j \neq k}} (\overline{x_{jk}} + t_j).$$

It is possible to exploit the fact that operational matrix X specifies a *total order*.⁶ In our case, it means that $\overline{x_{jk}} = x_{kj}$:

$$t_k = \bigwedge_{\substack{1 \leq j \leq n \\ j \neq k}} (x_{kj} + t_j).$$

As the phase encoder should maintain a certain time separation Δ between the generated transitions it is necessary to modify the above equation to take this fact into account:

$$t_k = \bigwedge_{\substack{1 \leq j \leq n \\ j \neq k}} (x_{kj} + t_j^\Delta)$$

where t_j^Δ represents signal t_j delayed for Δ time units. For the purpose of resetting the controller into the initial state after generating the desired sequence of transitions we should also add signal go that would serve as an initiating and resetting signal:

6. Partial order is called *total* iff every pair of elements is ordered.

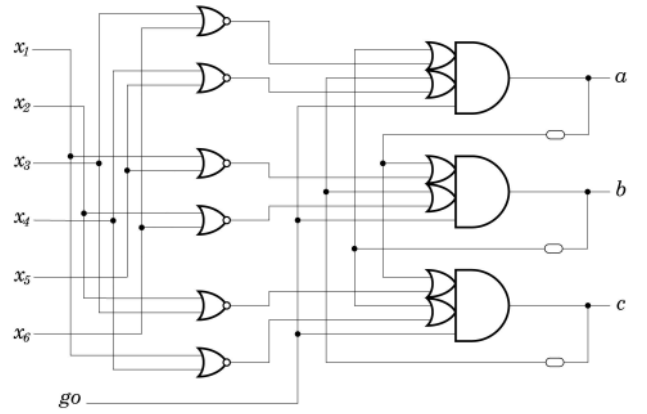


Fig. 19. Three-wire one hot phase encoder.

$$t_k = go \cdot \bigwedge_{\substack{1 \leq j \leq n \\ j \neq k}} (x_{kj} + t_j^\Delta). \quad (9)$$

The implementation of phase encoding repeater consisting of phase detector and phase encoder specified with (9) is shown in Fig. 18. Signal go can be generated in a number of ways depending on whether the repeater should be *early-propagative* or not as well as on several other criteria which are out of the scope of this paper.

Note that application of a nonstructural synthesis approach would lead to exploration of the whole state space of the controller which is huge (its size is proportional to $n \cdot n!$ even if we assume all the input signals to arrive simultaneously!). However, the CPOG-driven approach demonstrated here performs only a number of operations on objects of polynomial size ($n \times n$ matrices) and results in circuits of polynomial area. This allowed us to synthesize phase encoding repeaters for up to 10 wires in a matter of seconds with negligible memory consumption, while the STG-driven approach could not cope even with the case of five wires running out of available memory.

7.2 One Hot Phase Encoder

One hot encoding can be used to specify the order of output signal transitions for small values of n (for large values of n the method is inappropriate because it needs $n!$ input wires).

Consider 3-wire one hot phase encoder. The six behavioral scenarios ($a < b < c$, $a < c < b$, etc.) are encoded using six opcode variables $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ resulting in the CPOG shown in Fig. 20 (to the left); it is possible to simplify it into a slightly smaller CPOG using the transitive conditions reduction (to the right). The final gate-level implementation of the controller specified with the obtained optimal CPOG is shown in Fig. 19.

The obtained controller is not *speed-independent* [10] and operates correctly only under the timing assumptions

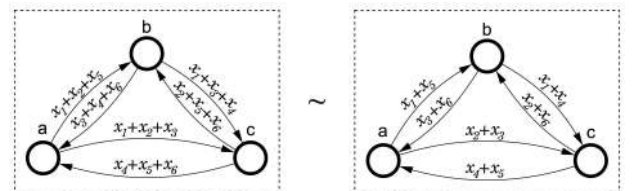


Fig. 20. Transitive arc reduction in one hot phase encoder CPOG.

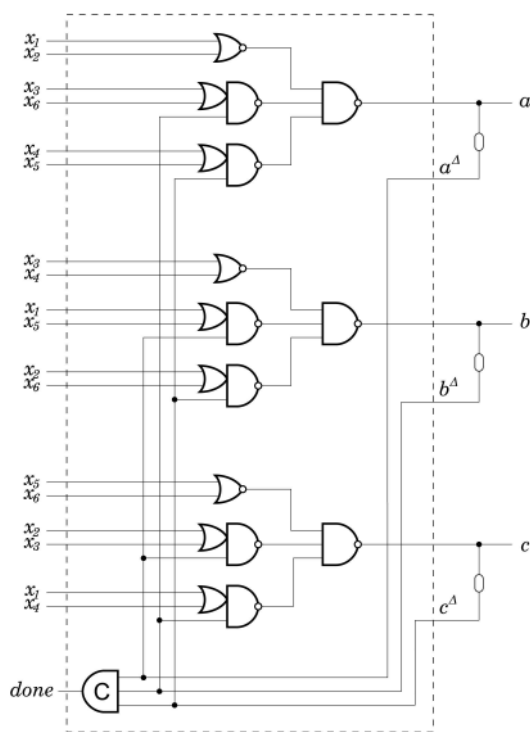


Fig. 21. Three-wire one hot phase encoder (a speed-independent solution).

imposed on opcode signals X and request signal go (similar to the *bundled data* protocol). In order to establish a proper speed-independent communication protocol between the controller and the environment signal go should be removed (the start and reset functions are delegated to one hot signals $x_1 \dots x_6$) as shown in Fig. 21. Instead a new signal $done$ should be introduced to prompt the environment that the controller has sent the phase encoded data and is ready for the next data packet. The delay elements should also be moved outside the controller and become part of the environment (the controller is separated from the environment with a dotted line). The complex gates generating the output signals are decomposed into two and three-input logic gates with a subsequent negative logic optimization. The delayed output transitions are synchronized with a C-element to produce signal $done$. The circuit was formally verified for the compliance with the environment interface and the absence of hazards using WORK-CRAFT [12] framework.

8 CONCLUSIONS

The paper presented the Conditional Partial Order Graph model and CPOG-based methodology for specification and synthesis of asynchronous controllers. It contains an extensive set of examples demonstrating the advantage of the proposed approach over the conventional methods. The model is beneficial for the specification of a certain class of systems which have many behavioral scenarios defined on the same set of events.

The methods presented in this paper can be applied whenever a controller has to react to different control codes by initiating different event sequences. The most natural examples include: CPU microcontrollers, which receive an

instruction code and activate a set of data path operational units (adders, multipliers, etc.) in a proper partial order; NoC routers, which receive a routing code (source/destination address) and perform a set of routing procedures in the requested sequence.

ACKNOWLEDGMENTS

This work was supported by EPSRC grants EP/F016786/1 and EP/C512812/1.

REFERENCES

- [1] International Technology Roadmap for Semiconductors (ITRS '07). <http://www.itrs.net/Links/2007ITRS/Home2007.htm>, 2007.
- [2] A. Bardsley and D. Edwards, "The Balsa Asynchronous Circuit Synthesis System," *Proc. Forum on Design Languages*, 2000.
- [3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers," *IEICE Trans. Information and Systems*, vol. E80-D, no. 3, pp. 315-325, 1997.
- [4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Logic Synthesis of Asynchronous Controllers and Interfaces," *Advanced Microelectronics*, Springer-Verlag, 2002.
- [5] A. Lew, *Computer Science: A Math. Introduction*. Prentice-Hall, 1985.
- [6] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.
- [7] A. Mokhov, C. D'Alessandro, and A. Yakovlev, "Multiple rail phase encoding circuits," Technical Report, NCL-EECE-MSD-TR-2008-133, May 2008.
- [8] A. Mokhov, "Conditional Partial Order Graphs," PhD thesis, Newcastle Univ., Sept. 2009.
- [9] A. Mokhov and A. Yakovlev, "Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis," *Proc. Design, Automation and Test in Europe (DATE) Conf.*, 2008.
- [10] D. Muller and W. Bartky, "A Theory of Asynchronous Circuits," *Proc. Int'l Symp. Theory of Switching*, pp. 204-243, 1959.
- [11] S. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers," PhD thesis, Stanford Univ., 1993.
- [12] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, and A. Yakovlev, "Automated Verification of Asynchronous Circuits Using Circuit Petri Nets," *Proc. Int'l Symp. Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, 2008.
- [13] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization," *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 727-750, Sept. 1987.
- [14] D. Sokolov and A. Yakovlev, "Clock-Less Circuits and System Synthesis," *Proc. IEE Proc. Computers and Digital Techniques*, 2005.
- [15] A. Taubin, J. Cortadella, L. Lavagno, L. Lavagno, A. Kondratyev, and A.M.G. Peeters, "Design Automation of Real-Life Asynchronous Devices and Systems," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 1, pp. 1-133, 2007.
- [16] K. van Berkel, M. Josephs, and S. Nowick, "Scanning the Technology: Applications of Asynchronous Circuits," *Proc. IEEE*, 1999.
- [17] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalijs, "The VLSI-Programming Language Tangram and Its Translation into Handshake Circuits," *Proc. European Conf. Design Automation (EDAC)*, 1991.
- [18] T. Verhoef, "Delay Insensitive Codes—An Overview," *Distributed Computing*, vol. 3, no. 1, pp. 1-8, 1988.
- [19] I. Wegener, *The Complexity of Boolean Functions*. Johann Wolfgang Goethe-Universität, 1987.
- [20] 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*, vol. 9, pp. 189-234, 1996.
- [21] K.Y. Yun, D.L. Dill, and S.M. Nowick, "Synthesis of 3D Asynchronous State Machines," *Proc. Int'l Conf. Computer Design (ICCD)*, pp. 346-350, 1992.



Andrey Mokhov studied computing science at Kyrgyz-Russian Slavic University from 2000 to 2005. After graduation, he joined the Asynchronous Research Group at Newcastle University as a PhD student and in 2009 he successfully defended the PhD dissertation. He is a research associate in the School of Computing Science, Newcastle University. His research interests include different levels of electronic design

automation: from formal models for system specification and verification to logic synthesis and application-specific optimization. He is a member of the IEEE.



Alexandre (Alex) Yakovlev received the MSc and PhD degrees from St. Petersburg Electrical Engineering Institute in 1979 and 1982, respectively, where he worked in the area of asynchronous and concurrent systems since 1980 and the DSc degree from Newcastle University in 2006. In the period between 1982 and 1990, he held positions of assistant and associate professor at the Computing Science Department. Since 1991, he has been at the Newcastle

University, where he worked as a lecturer, reader, and professor at the Computing Science Department until 2002, and is now heading the Microelectronic Systems Design Research Group (<http://async.org.uk>) at the School of Electrical, Electronic and Computer Engineering. His current interests and publications are in the field of modeling and design of asynchronous, concurrent, real-time, and dependable systems on a chip. He has published four monographs and more than 200 papers in academic journals and conferences, has managed over 25 research contracts. He has chaired program committees of several international conferences, including the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), Petri nets (ICATPN), Applications of Concurrency to Systems Design (ACSD), and is currently a chairman of the Steering committee of the Conference on Application of Concurrency to System Design. He is a senior member of the IEEE and a member of the IET. In April 2008, he was general chair of the 14th ASYNC Symposium and 2nd International Symposium on Networks on Chip, and tutorial chair at Design Automation and Test in Europe (DATE) in 2009.

▷ **For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.**