School of Electrical, Electronic & Computer Engineering

SAT-based Verification of Conditional Partial Order Graphs

A. Mokhov and A. Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2008-126

January 2008

Contact:

Andrey.Mokhov@ncl.ac.uk

Alex.Yakovlev@ncl.ac.uk

Supported by EPSRC grant EP/C512812

NCL-EECE-MSD-TR-2008-126

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

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

SAT-based Verification of Conditional Partial Order Graphs

A. Mokhov and A. Yakovlev

January 2008

Abstract

The Conditional Partial Order Graph (CPOG) Model introduced recently is a novel technique for specification and synthesis of asynchronous controllers. It combines advantages of both Petri nets and FSM-based approaches and is capable of modelling systems with a high degree of concurrency and multiple choice.

The paper extends the basic model of CPOGs to handle dynamic evaluation of internal control signals and introduces behavioural semantics for CPOGs. The extended model has a strong need for verification support, e.g. reachability analysis, deadlock detection etc. and this paper presents SAT-based characterisations for the most important CPOG verification problems.

1 Introduction

Specification and synthesis of self-timed control circuits exhibiting a high-degree of concurrency and choice (e.g. CPU cores, on-chip data transfer and routing controllers) is a challenging problem within the conventional asynchronous design flow. RTL-based synthesis flow [8] supports a synchronous design paradigm which leads to synchronous finite state machines for control logic. There are several existing design methodologies for asynchronous control logic. There are several existing design methodologies for asynchronous control logic, e.g. [15] and [13]. Methods like Tangram (or Haste) [16] and Balsa [2] use CSP-like HDL languages for system specification and syntax-direct translation for synthesis; they describe the entire sysem as a collection of processes and channels, control is implicit in them. Burts-mode FSMs [12] and Petri nets/Signal Transition Graphs (STGs) [14] are able to capture concurrency and choice at a very fine level and are more suitable for control design: they produce more compact and fast circuits than methods based on syntax-direct translations from HDLs. However these models are typically targeted at systems with a small number of choice options and specification of systems with many similar behavioural patterns, or event orders, defined on the same domain of operational units in inefficient (see Section 4).

The Conditional Partial Order Graph model introduced recently [11] is a new model that combines the advantages of the existing behavioural models Petri nets (or STGs) and FSMs and avoids the use of the explicit notion of state.

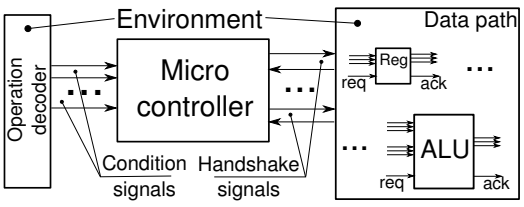


Figure 1: Reconfigurable controller

The model 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.

The basic definition of the CPOG model in [11] restricted the control signals from the environment to be constant throughout the execution of the controller which significantly limited the class of the modelled systems. The paper extends the definition and introduces CPOGs with dynamic control signal evaluation. This allows modelling systems where the control flow conditions can change during the execution of actions in the CPOG.

While the static control model from [11] exhibited only syntactic correctness conditions (most of the semantics was "correct by construction"), the extended model has more sophisticated behavioural semantics. It requires an automated verification support which is developed in Section 7.

2 Theoretical Background

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

2.1 Partial orders

A *partial order* (PO) $P(S, \prec)$ is a binary relation $\prec$ over a set of elements S which satisfies the following three conditions [3, 9]:

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

Partial orders are very natural for specification of order of events in a system when some of the events are constrained to happen before others. These constraints can be specified with partial order $P(S, \prec)$ such that if $a \prec b$ for some events $a, b \in S$ 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.

2.2 Directed acyclic graphs

A *directed graph* is an ordered pair $G(V, E)$ where V is a set of *vertices* (or *nodes*) and $E \subseteq V \times V$ is the set of ordered pairs of vertices, called *arcs* [4, 9].

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

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

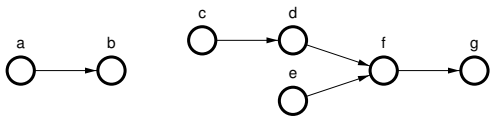
An arc $(a, b) \in E$ of a graph $G(V, E)$ is called *transitive* iff $\exists v \in V \setminus \{a, b\}, \langle a, v \rangle \in \mathcal{P}(G) \wedge \langle v, b \rangle \in \mathcal{P}(G)$.

The *transitive closure* of a graph $G(V, E)$ is the smallest graph $G*(V, E*)$ such that:

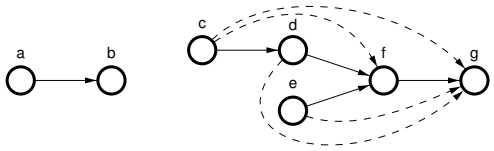
- $\forall a, b \in V, (a, b) \in E \Rightarrow (a, b) \in E*$;
- $\forall a, b, c \in V, (a, b) \in E* \wedge (b, c) \in E* \Rightarrow (a, c) \in E*$ (transitivity condition);

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

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



(a) Initial DAG



(b) Transitive closure

Figure 2: DAG and its transitive closure

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

3 Conditional Partial Order Graphs

This section defines conditional partial order graphs (CPOGs) formally and introduces CPOG algebra.

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

3.1 Addition

The result of *addition* of $H_1(V_1, E_1, X_1, \rho_1, \phi_1)$ and $H_2(V_2, E_2, X_2, \rho_2, \phi_2)$ is CPOG $H(V_1 \cup V_2, E_1 \cup E_2, X_1 \cup X_2, \rho_1 + \rho_2, \phi)$, where $\forall z, \phi(z) = \phi_1(z) + \phi_2(z)$. Here $f_1 + f_2$ stands for Boolean disjunction of functions f_1 and f_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 $H(V, E, X, \rho, \phi)$ can be *multiplied* by a Boolean function $f \in \mathcal{F}(Y)$ (which in our context can be called *scalar*). The resultant CPOG is $H'(V, E, X \cup Y, f\rho, \phi')$ where $\forall z, \phi'(z) = f\phi(z)$ ($f_1 f_2$ stands for Boolean conjunction of functions f_1 and f_2). 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 $H(V, E, X, \rho, \phi)$ under constraint $x = \alpha$ $(x \in X)$ is denoted as $H|_{x=\alpha}$ and is equal to CPOG $H'(V, E, X \setminus \{x\}, \rho|_{x=\alpha}, \phi|_{x=\alpha})$ where notations $\rho|_{x=\alpha}$ and $\phi|_{x=\alpha}$ mean that variable x is substituted with constant Boolean value α in ρ and all the functions $\phi(z), z \in V \cup E$ (which implies that $\rho|_{x=\alpha}$ and $\phi|_{x=\alpha}(z)$ belong to $\mathcal{F}(X \setminus \{x\})$). 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 restriction function and vertex/arc conditions are only Boolean constants $\rho|_\psi$ and $\phi|_\psi$ (either 0 or 1).

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

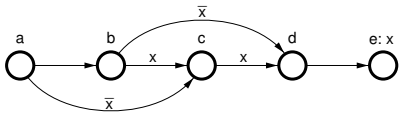
Let $H|_\psi$ be a complete projection of CPOG $H(V, E, X, \rho, \phi)$. We can construct a directed graph $G(V_G, E_G)$ such that

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

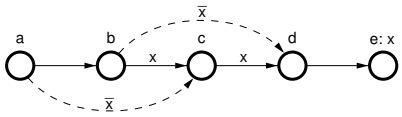
In other words G contains only those vertices and arcs whose conditions in H are constant 1.

A complete projection $H|_\psi$ is *valid* iff its corresponding graph G is a DAG and its restriction function is satisfied: $\rho|_\psi = 1$. So the purpose of a restriction function ρ is to restrict domain of the assignment functions ψ applicable to a CPOG: out of $2^{|X|}$ different possible assignment functions ψ only those satisfing ρ ($\rho|_\psi = 1$) are allowed. The other assignment functions ψ (and the corresponding complete projections $H|_\psi$) are considered invalid and meaningless in relation to the modelled system.

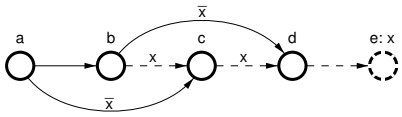
The obtained DAG $G(V_G, E_G)$ can be further converted into the corresponding partial order $P(V_G, E_G*)$. Let this operation of partial order construction from a valid CPOG complete projection $H|_\psi$ be shortly denoted as **po**: $P = \mathbf{po}(H|_\psi)$ and the inverse operation as $\mathbf{po}^{-1}$: $H = \mathbf{po}^{-1}(P)$ (here the obtained H does not contain any control variables i.e. $X = \emptyset$).



(a) Conditional partial order graph H



(b) Complete projection $H|_{x=1}$



(c) Complete projection $H|_{x=0}$

Figure 3: CPOG and its projections

An example of a CPOG and its projections is shown in Figure 3. Subfigure (a) shows the initial graph. The conditional 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 3(b) shows the complete projection under $x = 1$. The dotted arcs are those that turn to have constant 0 conditions after the projection and therefore will be excluded from the resultant partial order. The solid arcs have constant 1 conditions. The partial order defined with the projection is a simple series of events: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$.

Complete projection under constraint $x = 0$ (Figure 3(c)) results in the following partial order. Events b and c can happen only after a. There is no constraint between them, thus they can be concurrent. Event d can only happen after event b. Event e is excluded from the partial order; note, that this implies exclusion of arc (d, e) as well.

4 CPOGs synthesis

A CPOG can potentially contain an exponential number of different partial orders in a compressed form as was demonstrated in the previous section. The natural question is how to synthesise such a compact specification given the system description as a set of partial orders corresponding to the system's different behavioural scenarios. And the consequent generalisation of the synthesis problem is to use not only partial orders as building blocks but CPOGs themselves (observe that partial orders is a subclass of CPOGs that do not have conditions). This can be very useful if a designer wants to add a new scenario to an already existing CPOG without its complete resynthesis, or merge two different systems into one without their preliminary decomposition into distinct projections.

The first simple example demonstrates the advantages of the CPOG model for system specification in comparison with the conventional specifications using STGs.

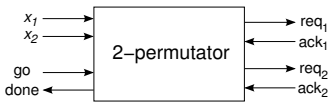


Figure 4: 2-permutator interface

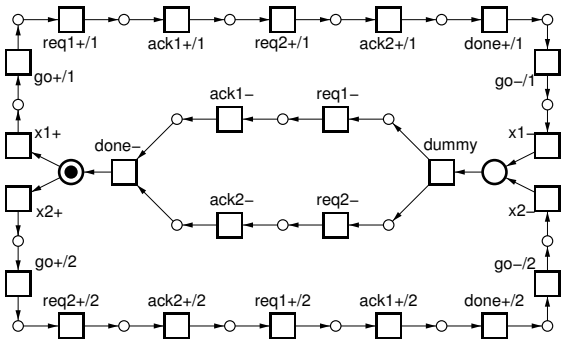
The example is the smallest non-trivial *n-permutator* [11], a circuit able to generate $n!$ different handshake sequences according to the control signals from the environment. The interface of 2-permutator is shown in Figure 4. Depending on the control signals $\{x_1, x_2\}$ the controller has to initiate two handshakes either ordered as $1 \rightarrow 2$ or as $2 \rightarrow 1$. The start of the handshake sequence is prompted by signal *go* and as soon as the handshakes are completed the controller issues signal *done*.

The first approach to the specification of the controller with an STG is shown in Figure 5(a). It has a global choice and the two scenarios are specified as two independent branches: the upper branch corresponds to the first scenario (handshake sequence $1 \rightarrow 2$); the lower branch corresponds to the second scenario ($2 \rightarrow 1$). After the global merge the handshakes are reset concurrently and the system returns to the initial state.

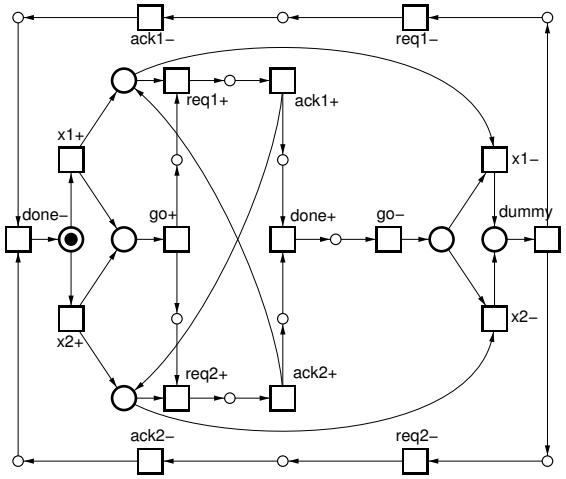
This specification is straightforward and can be easily obtained by hand but it has a very serious drawback: it duplicates events in different branches. In the general case n-permutator has $n!$ different scenarios. Clearly, a specification having $n!$ different branches is infeasible. It is possible to optimise this unoptimal STG using logic synthesis tool Petrify [5] and obtain an STG specification without event duplication as shown in Figure 5(b). But 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 as 2-permutator the obtained STG is non-trivial and difficult for manual design.

Specification of this system with a set of two partial orders and merging them into a CPOG seems to be much more natural. The two scenarios correspond directly to partial orders which are shown in Figure 6(a,b). And it is possible to merge them into the CPOG in Subfigure (c) which captures both the scenarios in a compact and understandable form.

Formally, let $\mathcal{H} = \{H_1, H_2, ..., H_n\}$ be the set of n given CPOGs. The objective is to synthesize

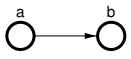


(a) Specification with global choice and multiple event occurrences

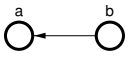


(b) Optimised specification obtained by Petrify

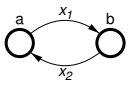
Figure 5: 2-permutator STG specifications



(a) PO for $1 \rightarrow 2$



(b) PO for $2 \rightarrow 1$



(c) synthesised CPOG

Figure 6: CPOG synthesis example

CPOG $H(V, E, X, \rho, \phi)$ and assignment set $\Psi = \{\psi_1, \psi_2, ..., \psi_n\}$ such that $\psi_k : X_C \rightarrow \{0, 1\}$ 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:

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

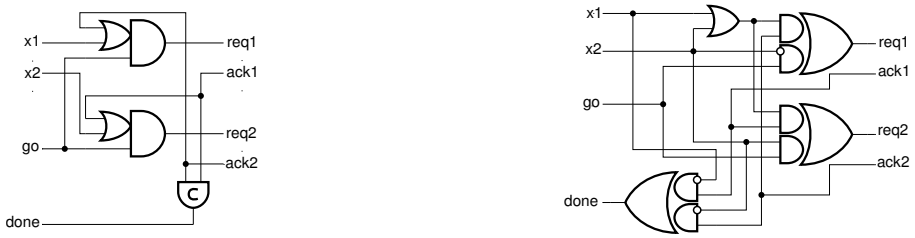
Now control set X_C, functions $f_k \in \mathcal{F}(X_C)$ and ψ_k should be selected so that $f_k|_{\psi_k} = 1$ and $f_k|_{\psi_j} = 0, j \neq k$. This can be done in different ways depending on the chosen encoding scheme [11].

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

$$x_1 \cdot \mathbf{po}^{-1}(\text{a} \rightarrow \text{b}) + x_2 \cdot \mathbf{po}^{-1}(\text{a} \leftarrow \text{b}) = \text{a} \overset{x_1}{\underset{x_2}{\rightleftarrows}} \text{b}$$

Figure 7 shows the controllers synthesised using CPOG-based approach and Petrify: the solutions are

different and our method produces a smaller controller. CPOG-based gate-level controllers synthesis uses *Transition Sequence Encoder* (TSE) generic circuit [10].



(a) CPOG-based solution (b) Petrify solution

Figure 7: Synthesised controllers

5 Dynamic control signals evaluation

The basic CPOG model assumes that control signals X remain constant during the active phase (signal $go = 1$) of the controller execution and are allowed to change only during the reset phase (signal $go = 0$). Although this greatly simplifies CPOG definition and verification it leads to difficulties in specification of large class of systems with internal choice.

Let us study the CPOG specification of a single scenario of operation $diff(a, b)$ computing the difference between two values a and b: $diff(a, b) = |a - b|$. The flow of execution of the operation breaks up into the following actions:

1. Load register a from memory ($load(a)$);
2. Load register b from memory ($load(b)$);
3. Compute the difference and store the result in a:
   (a) Compare registers a and b ($cmp(a, b)$);
   (b) If $a < b$ then swap the registers ($swap(a, b)$);
   (c) Subtract b from a, store the result in a ($sub(a, b)$);
4. Save register a into memory ($save(a)$).

Note that the result of comparison action $cmp(a, b)$ is used in the next conditional action of swapping $swap(a, b)$. The values can only be compared after they have been loaded from memory and therefore the comparison result is undefined before the execution of the comparator. The basic CPOG model does not allow such changes of the control variables during the active phase of the controller.

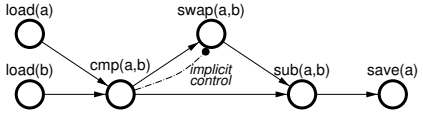
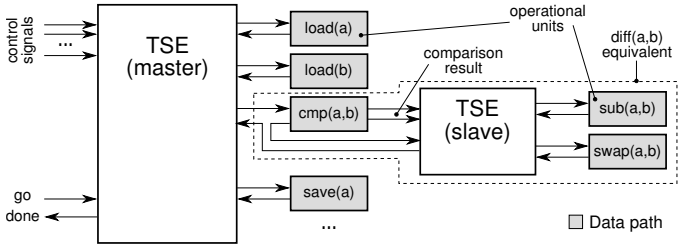
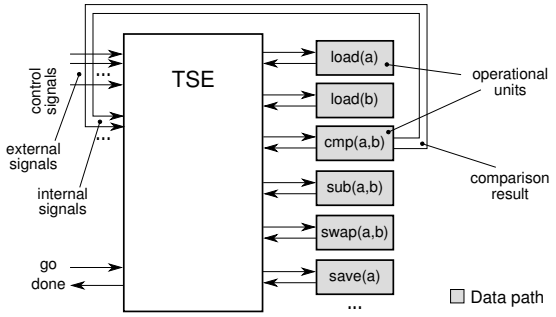


Figure 8: CPOG with implicit control

Figure 8 shows the CPOG specification for this operation with the implicit dependence between actions $cmp(a, b)$ and $swap(a, b)$ denoted with a dotted arc that can switch vertex $swap(a, b)$ on or off depending on the result of comparison. The specification problem can be resolved in two different ways: by using hierarchical TSE structure or by extending the basic CPOG model to handle dynamic signal evaluation. Both solutions are discussed below.



(a) Hierarchical TSE controller

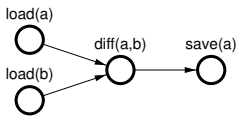


(b) TSE with dynamic control signals evaluation

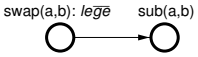
Figure 9: Hierarchcal and dynamic evaluation TSE controllers

5.1 Hierarchical TSE controllers

It is possible to use the hierarchical connection of master and slave TSE controllers as shown in Figure 9(a). Signal *go* for the slave controller is the acknowledgement signal from the comparator which ensures that the comparison result is already computed and the slave TSE can use it as the external control signal staying within the basic CPOG model. The comparison result is represented by two signals le and ge: $le = 1$ ($ge = 1$) is set iff the value in register a is less (greater) or equal to the value in register b. The slave controller performs the actions according to these control signals and sets signal *done* that is used as an acknowledgement from the comparator by the master TSE. The subsystem including the slave TSE and operational units $cmp(a, b)$, $sub(a, b)$ and $swap(a, b)$ can be treated as a complex operational unit $diff(a, b)$.



(a) Master TSE specification



(b) Slave TSE specification

Figure 10: Master and slave CPOGs

CPOG specifications for the master and slave controllers are shown Figure 10. The master CPOG is trivial and does not contain any conditions. The CPOG for the slave controller executes either sequence of operations $swap(a, b) \rightarrow sub(a, b)$ or just a single operation $sub(a, b)$. Vertex $swap(a, b)$ has condition $le \cdot \overline{ge}$ which is true iff the value in register a is strictly less then the value in register b: $le \cdot \overline{ge} = (a \leq b) \cdot \overline{(a \geq b)} = (a < b)$. Thus values a and b are swapped before the subtraction $a = a - b$ if $a < b$, so the nonnegative result is guaranteed.

5.2 CPOGs with dynamic evaluation

Another approach for specification of a system with dynamic signal changes is to extend the basic CPOG model and to allow some of the control variables to be changed during the active phase of the controller execution. In our case it is natural to allow the comparator change signals le and ge so that they can be used by the subsequent operations. Such a connection of the comparator $cmp(a, b)$ to the TSE controller is shown in Figure 9(b). CPOG specification (see Figure 11) becomes more logical and understandable in comparison with the two separated specifications in case of hierarchical design. Vertex $cmp(a, b)$ controls signals $\{le, ge\}$ (this fact is denoted below the vertex) such that after its execution condition $le + ge = 1$ holds. This condition is needed to ensure that the comparator does not produce a meaningless result $le = ge = 0$. After its execution vertex $swap(a, b)$ is included into the current operational flow only if needed.

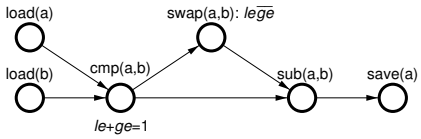


Figure 11: Dynamic CPOG

The formal description of CPOG behavior with dynamic control signals evaluation is presented in the next section.

6 CPOG behavioural semantics

Every vertex $v \in V$ in CPOG $H(V, E, X, \rho, \phi)$ is associated with an event (or action) in the modelled system. Such an event can be execution of a data path combinational logic block or enabling of a slave controller in case of hierarchical control design (see Section 5.1). To describe the dynamic behaviour of a CPOG-based controller this section introduces *firing rules* and the notions of *control vector*, *configuration* and *state*.

Control vector is an assignment function $\psi : X \to \{0, 1\}$ that assigns Boolean values to all the internal and external control variables (signals) X. The set of internal control variables is denoted as $Y \subseteq X$, and thus the set of external variables is $X \setminus Y$. The restriction function ρ bounds only the values of external control signals: $\rho \in \mathcal{F}(X \backslash Y)$.

Function $\mu : Y \to V$ assigns a *master vertex* $v \in V$ for every internal variable $y \in Y$, which means that y is generated during the execution of the action (*master action*) correspondent to $v = \mu(y)$. Before the execution the value of any internal variable $y \in Y$ is undefined and can be either 0 or 1. Set $Y_v = \{y \in Y | \mu(y) = v\}$ is called the *controlled set* of vertex v. Note, that controlled sets of any two vertices do not overlap: $Y_u \cap Y_v = \emptyset, u \in V, v \in V, u \neq v$. Every vertex $v \in V$ has its own *restriction function* $\rho_v \in \mathcal{F}(Y_v)$ associated with it which restricts values of controlled variables Y_v such that $\rho_v|_\psi = 1$ always holds after the execution of the action associated with v. For example, let v be associated with a comparator that compares two arguments and affects two variables $Y_v = \{le, ge\}$: it sets $le = 1$ ($ge = 1$) iff the first argument is less (greater) or equal to the second. Obviously le and ge cannot be both equal to 0 but the three other combinations are meaningful, thus the restriction function should be $\rho_v(le, ge) = le + ge$.

The *preset* of a vertex $v \in V$ with respect to control vector ψ is defined as $\bullet v = \{u \in V |\ \phi|_\psi(u) \wedge \phi|_\psi((u, v)) = 1\}$: it contains all the vertices u which precede vertex v in the partial order determined by the complete projection $H|_\psi$. *Postset* is defined similarly: $v\bullet = \{u \in V |\ \phi|_\psi(u) \wedge \phi|_\psi((v, u)) = 1\}$.

Configuration $C \subseteq V$ is the set of vertices whose corresponding actions have been already performed. A configuration C is *valid* iff $\forall v \in C, \phi|_\psi(v) = 1 \wedge \bullet v \subseteq C$.

Pair $\langle C, \psi \rangle$ is called *state*, as it fully describes the state of the modelled system.

In a given state $\langle C,\psi\rangle$ a vertex $v \notin C$ is *enabled to fire* iff

- it is present in the partial order determined by the complete projection $H|_\psi$: $\phi|_\psi(v) = 1$.
- its preset is a subset of the configuration: $\bullet v \subseteq C$;

A *firing* of an enabled vertex $v \notin C$ produces a new configuration $C' = C \cup \{v\}$. Control vector ψ can also be affected by the firing if vertex v is associated with an action that changes some of the internal control variables. In particular, the firing of vertex $v \in V$ affects the values of its controlled set Y_v in such a way that its restriction function holds: $\rho_v = 1$. Firing is considered to be an atomic event: $\langle C, \psi\rangle$ is changed momentarily into the new state $\langle C', \psi'\rangle$.

Note, that every internal control variable has at most one master vertex and therefore it can change at most once. This means that system state $\langle C, \psi\rangle$ can only change *monotonically* during the system evolution: once a vertex is added to configuration it cannot be removed and once an internal control variable is changed in ψ it remains constant.

A system state $\langle C, \psi\rangle$ is *valid* iff configuration C is valid and the restriction functions of H and vertices in the configuration are not violated: $\rho|_\psi = 1 \wedge \forall v \in C, \rho_v|_\psi = 1$.

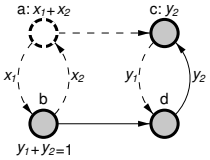
6.1 Initial states, final states and deadlocks

The *initial state* of the system is described as $\langle \emptyset, \psi_0\rangle$ which means that no actions have been performed and the control signals are set to some initial values ψ_0.

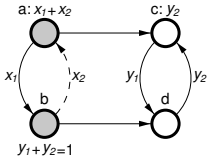
Starting from the initial state the system evolves by firing enabled vertices and finally reaches a state when no vertex is enabled. Such a state can either be a *final state* or a *deadlock*. Notice that the system will always terminate as the set of vertices V is finite and each firing adds a vertex $v \notin C$ into configuration C.

A *final state* of the system is such a state $\langle C, \psi\rangle$ that there is no vertex $v \notin C$ that is present in projection $H|_\psi$: $\forall v \notin C, \phi|_\psi(v) = 0$.

A *deadlock* is such a state $\langle C, \psi\rangle$ that there are no enabled vertices but at least one vertex $v \notin C$ is present in $H|_\psi$: $\exists v \notin C, \phi|_\psi(v) = 1$. A deadlock is a situation wherein two or more competing events are waiting for the other to happen, and thus neither ever does. Such a situation is caused by the fact that the complete projection $H|_\psi$ contains a cycle and thus invalid (see Section 3.3). A CPOG $H(V, E, X, \rho, \phi)$ is called *deadlock free* if there is no valid deadlock state which is reachable from any initial state $\langle \emptyset, \psi_0\rangle, \rho|_{\psi_0} = 1$.



(a) Final state: $C = \{b, c, d\}$, $(x_1, x_2, y_1, y_2) = (0, 0, 0, 1)$



(b) Deadlock: $C = \{a, b\}$, $(x_1, x_2, y_1, y_2) = (1, 0, 1, 1)$

Figure 12: Final state and deadlock

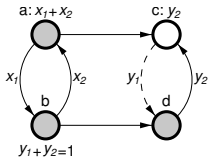
Figure 12 shows an example of a system containing both final states and deadlocks (the vertices in the configurations are marked with gray colour) . x_1 and x_2 are external control signals, while internal signals $Y = \{y_1, y_2\}$ are controlled by vertex b (its restriction function is shown below the vertex). Subfigure (a) shows the final state reachable through the following firing sequence. System starts with the initial state $\langle \emptyset, (x_1 = x_2 = y_1 = y_2 = 0)\rangle$. In this state vertex b is enabled to fire. After its firing signals $Y_b = \{y_1, y_2\}$ can change from zeroes to $(y_1 = 0, y_2 = 1)$, note that the restriction function $\rho_b = y_1 + y_2$ is satisfied. Now vertices d and then c fire and the system comes to the final state: vertex a is not present in $H|_\psi$: $\phi|_\psi(a) = 0$.

A reachable deadlock state is shown in Subfigure (b): starting from $\langle \emptyset, (x_1 = 1, x_2 = y_1 = y_2 = 0) \rangle$ system evolves by firing of vertex a which is followed by firing of b. Now if control signals $Y_b = \{y_1, y_2\}$ evaluate into $y_1 = y_2 = 1$ then system comes to the deadlock: both vertices c and d exist in projection $H|_\psi$ but none of them is enabled.

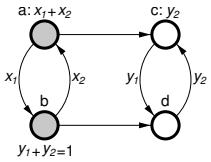
If a system does not contain any internal control variables ($Y = \emptyset$) then the final state (or deadlock) can be uniqely determined from the initial state. Otherwise the system can terminate in different states according to the different internal variable changes during the firing of their master vertices.

6.2 Valid states reachability

Not every valid state $\langle C, \psi \rangle$ is reachable from the initial state $\langle \emptyset, \psi \rangle$. The reason is that configuration C can contain such a set of vertices that there is no firing sequence leading to it. Two examples of such states are shown in Figure 13: both states have subset $\{a, b\} \subseteq C$ in the configuration but the external control signals $x_1 = x_2 = 1$ introduce mutual dependence between vertices a and b: it is impossible to fire them in any order and thus any state with configuration $\{a, b\} \subseteq C$ is unreachable. Moreover one can observe that the initial state $\langle \emptyset, x_1 = x_2 = 1 \rangle$ is a deadlock state and it is the only reachable state provided that the external signals are $x_1 = x_2 = 1$.



(a) Valid state: $C = \{a, b, d\}$, $(x_1, x_2, y_1, y_2) = (1, 1, 0, 1)$



(b) Deadlock: $C = \{a, b\}$, $(x_1, x_2, y_1, y_2) = (1, 1, 1, 1)$

Figure 13: Unreachable states

The following important property of deadlock free CPOGs is expoited in the verification algorithms presented in this paper.

Theorem I. If a CPOG H is deadlock free then every valid state $\langle C, \psi \rangle$ is reachable from the initial state $\langle \emptyset, \psi \rangle$ through a sequence of valid states.

Proof. By induction. If $C = \emptyset$ then $\langle C, \psi \rangle$ is the initial state already, otherwise let $v \in C$ be such a vertex that configuration C does not contain any vertices of the postset of v: $C \cap v\bullet = \emptyset$. If it is impossible to select such v then C must contain directed cycle $\langle v_0, v_0 \rangle = (v_0, v_1, ..., v_n = v_0), v_k \in C, k = 0...n$ which implies that the system has a deadlock reachable from state $\langle C - \bigcup_{0 \le k < n} v_k, \psi \rangle$. This contradicts the fact that H is deadlock free. Thus $v \in C, C \cap v\bullet = \emptyset$ can be selected. Now we can *unfire* vertex v and obtain a valid state $\langle C', \psi' \rangle = \langle C - v, \psi \rangle$ from which $\langle C, \psi \rangle$ is reachable by firing of vertex v. Note that the unfiring of vertex v does not change the values of the internal control variables in ψ ($\psi' = \psi$). This is allowed because before the execution of their master action v the values of variables Y_v are undefined and nothing restricts them from being equal to the result of the execution: $\psi'(y) = \psi(y), y \in Y_v$.

Configuration $C' = C - v$ is smaller than C and we can conclude inductively that eventually it becomes empty and the initial state $\langle \emptyset, \psi \rangle$ is reached. During the process we construct *trace* $(\langle \emptyset, \psi \rangle, ..., \langle C'', \psi \rangle, \langle C', \psi \rangle, \langle C, \psi \rangle)$ of valid states leading from the initial state $\langle \emptyset, \psi \rangle$ to state $\langle C, \psi \rangle$. ■

The proof of the theorem is constructive and is used as a basis for the recursive algorithm of trace reconstruction for a valid state $\langle C, \psi \rangle$ which is shown in Algorithm 1. The presented algorithm directly matches the proof but it can be further optimised using *reverse topological sorting algorithm* [4] resulting in linear complexity w.r.t. to the number of arcs in H: $O(|V| + |E|)$.

Algorithm 1 Trace reconstruction

```
function TRACE(H, ⟨C, ψ⟩)
{
  if (C = ∅) then return (); // empty trace

  for all (v ∈ C) do
    if (∄u ∈ C, φ|ψ((v, u)) = 1) then
      return TRACE(C − v, ψ) + ⟨C, ψ⟩;

  // CPOG H is not deadlock free
  return deadlock_found_error;
}
```

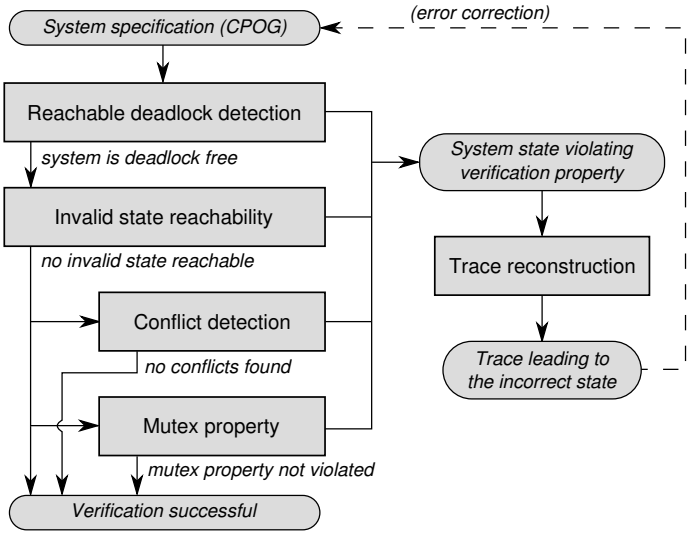


Figure 14: CPOGs verification flow

7 Verification

The overall verification flow of systems specified with CPOG model is shown in Figure 14. The given system is first checked for the absence of deadlocks (Section 7.2) and then for the impossibility to reach an invalid state from a valid one (Section 7.3). After these basic verification procedures the system can be checked for higher level properties e.g. event conflict freedom (Section 7.4) or checking that the two given events are mutually exclusive (Section 7.5). The verification tools provide the error state $\langle C, \psi \rangle$ in case of an incorrect system behaviour and this state can be given to the trace reconstruction algorithm (Section 6.2) to obtain the trace of events leading to the failure. This information can be used to correct the given system specification and rerun the verification.

One of the possible verification approaches is to convert the given CPOG into behaviorally equivalent Petri net and reuse the great deal of existing verification techniques available for them. But Petri net model is more general and verification of most of its properties is PSPACE-complete [6]. The most efficient verification algorithms are based on Petri net unfolding and subsequent use of SAT-based NP-complete techniques on the obtained prefix [7]. Because of the acyclic nature of CPOGs valid projections (each vertex can only fire at most once in every execution run) it is possible to apply SAT directly to them without the computationally expensive unfolding and to stay within the NP-complete complexity class.

7.1 SAT formulation

It is possible to formulate CPOG verification problems as instances of a *Boolean satisfiability problem* (SAT), which decides whether a given Boolean formula is *satisfiable* or not. A Boolean formula

$F(x_1, x_2, ..., x_n)$ is called *satisfiable* iff it is possible to find an assignment of Boolean values to the variables $(x_1, x_2, ..., x_n)$ that will make the formula true: $F(x_1, x_2, ..., x_n) = 1$. Our aim in this section is to build such a Boolean formula that will be satisfiable iff the property under verification holds.

The following variables are used to encode a system state $\langle C, \psi \rangle$ into Boolean domain:

- Configuration C is described with $|V|$ variables $\mathsf{conf}_v, v \in V$ such that $\mathsf{conf}_v = 1$ iff vertex v is included into the configuration: $v \in C$.
- Control vector ψ is described with $|X|$ variables $\mathsf{val}_x, x \in X$, such that $\mathsf{val}_x = 1$ iff $\psi(x) = 1$. We will also use notation $\phi_{\mathsf{val}}(z), z \in V \cup E$ to denote value of a vertex/arc function in complete projection $H|_\psi$.

Verification formula is a conjunction of a set of constraints. The constraints ensure that variables conf_v and val_x define a system state that is relevant to the particular verification problem. For instance, if we are looking only for the states with valid configurations then we have to use the following *valid configuration constraint* $\mathcal{CONF}$:

$$\mathcal{CONF} = \left(\bigwedge_{v \in V} \mathsf{conf}_v \Rightarrow \phi_{\mathsf{val}}(v) \right) \left(\bigwedge_{v \in V} \mathsf{conf}_v \Rightarrow \bigwedge_{u \in \bullet v} \mathsf{conf}_u \right)$$

It ensures that:

- a vertex $v \in V$ can be in the configuration ($\mathsf{conf}_v = 1$) only if it exists in $H|_\psi$ (the first clause);
- if a vertex $v \in V$ is in the configuration then all the vertices of its preset $\bullet v$ must be in the configuration as well (the second clause).

Another constraint that will be often used is the *control signals constraint* $\mathcal{SIG}$:

$$\mathcal{SIG} = \rho(\mathsf{val}_{x_1}, ..., \mathsf{val}_{x_{|X|}}) \cdot \bigwedge_{v \in V} \mathsf{conf}_v \Rightarrow \rho_v(\mathsf{val}_{y_1}, ..., \mathsf{val}_{y_{|Y_v|}})$$

It captures the fact that the values $\mathsf{val}_{y_1} ... \mathsf{val}_{y_{|Y_v|}}$ of the control signals from vertex v cotrolled set Y_v must satisfy its restriction function ρ_v if $v \in C$, while CPOG restriction function ρ must be satisfied with values $\mathsf{val}_{x_1}, ... \mathsf{val}_{x_{|X|}}$.

To simplify some of the further equations we also introduce Boolean function $enabled_v, v \in V$:

$$enabled_v = \overline{\mathsf{conf}_v} \cdot \phi_{\mathsf{val}}(v) \cdot \bigwedge_{u \in \bullet v} \mathsf{conf}_u$$

Thus $enabled_v = 1$ iff vertex $v \in V$ is enabled to fire in the current state according to the definition in Section 6.

7.2 Reachable deadlock detection

Verification of deadlock freedom is different from the other verification problems because the deadlock freedom property ensures that all the valid system states are reachable from the initial state. Thus when we check for deadlock we cannot be sure that the found solution represents a reachable state. But the beauty of this property is that if a particular deadlock state which was found by the presented algorithm is unreachable it means that there must exist another reachable deadlock state that occurs before the detected one and prevents it from being reachable (this follows from the proof of Theorem I). Therefore the presented approach does not guarantee that the found deadlock is reachable but it guarantees that if it finds a deadlock then there exists a reachable deadlock in the system. Note that the reachability of the found state can be polynomially checked using the trace reconstruction algorithm (Section 6.2).

Figure 13(b) shows an example of unreachable deadlock state $\langle C = \{a, b\}, x_1 = x_2 = y_1 = y_2 = 1\rangle$. Note that this deadlock is unreachable because of the existence of reachable deadlock state $\langle C = \emptyset, x_1 = x_2 = y_1 = y_2 = 1\rangle$. To reach the former deadlock state the system should somehow resolve the latter one which can only be done by violation of one of the conflicting dependencies between vertices $\{a, b\}$.

The verification formula for this problem (named $\mathcal{RD}$ i.e. reachable deadlock) is the conjunction of constraints $\mathcal{CONF}$, $\mathcal{SIG}$ (which together define a valid state $\langle \mathsf{conf}, \mathsf{val}\rangle$) and the following constraint $\mathcal{DS}$ which is true iff there is no enabled vertex but at least one unfired vertex is present in $H|_\psi$ (a deadlock state by definition):

$$\mathcal{DS} = \bigwedge_{v \in V} \overline{enabled_v} \cdot \bigvee_{v \in V} \overline{\mathsf{conf}_v} \cdot \phi_{\mathsf{val}}(v)$$

This gives us the following SAT formula:

$$\mathcal{RD} = \mathcal{CONF} \cdot \mathcal{SIG} \cdot \mathcal{DS}$$

If SAT solver finds such an assignment of variables conf_v and val_x that $\mathcal{RD} = 1$ then there is a reachable deadlock in the system, otherwise the system is deadlock free. Note, that it is usually assumed that formula is given in *Conjunctive Normal Form* (CNF) to SAT solver but it is possible to convert any Boolean formula into CNF efficiently [17].

7.3 Invalid state reachability

The next basic CPOG property that we are going to verify is the reachability of an invalid state from a valid one. Example of such a situation is shown in Figure 15. The current system configuration is $C = \{a, c\}$, the only internal control variable y is set to zero and its master vertex is d (this fact is denoted below the vertex) as shown in Subfigure (a). This state is legal by definition because vertex b is not present in the current complete projection $H|_\psi$: $\phi_{\mathsf{val}}(b) = y = 0$. However, vertex d is enabled to fire and it can change the value of variable y producing an invalid state: $C = \{a, c, d\}$, $y = 1$ (Subfigure (b)). This state violates validity of the configuration because now vertex $c \in C$ has vertex b in its preset and it is not in the configuration.

(a) $C = \{a, c\}$, $y = 0$ (b) $C = \{a, c, d\}$, $y = 1$

Figure 15: Reachable invalid state example

The verification formula $\mathcal{ISR}$ (i.e. invalid state reachability) is the conjunction of constraints $\mathcal{CONF}$, $\mathcal{SIG}$ (which together define a valid state $\langle \mathsf{conf}, \mathsf{val}\rangle$) and the following constraint $\mathcal{IS}$ which is true iff there is an invalid state $\langle \mathsf{conf}', \mathsf{val}'\rangle$ reachable from $\langle \mathsf{conf}, \mathsf{val}\rangle$:

$$\mathcal{IS} = \bigvee_{v \in V} enabled_v \cdot \mathcal{FIRE}(v) \cdot \mathcal{SIG}' \cdot \overline{\mathcal{CONF}'}$$

where $\mathcal{SIG}'$ is the control signal constraint for variables conf'_v and val'_x (we assume that vertex v fires correctly i.e. it sets its controlled variables according to its restriction function and thus $\mathcal{SIG}'$ constraint is not violated); term $\overline{\mathcal{CONF}'}$ ensures that state $\langle \mathsf{conf}', \mathsf{val}'\rangle$ has invalid configuration. Function $\mathcal{FIRE}(v)$ defined below constructs state $\langle \mathsf{conf}', \mathsf{val}'\rangle$ by firing of vertex v in state $\langle \mathsf{conf}, \mathsf{val}\rangle$:

$$\mathcal{FIRE}(v) = \mathsf{conf}'_v \cdot \bigwedge_{u \neq v} \mathsf{conf}'_u \Leftrightarrow \mathsf{conf}_u \cdot \bigwedge_{x \notin Y_v} \mathsf{val}'_x \Leftrightarrow \mathsf{val}_x$$

It ensures that configuration conf' differs from conf only with vertex v and the only control signals that are allowed to change are those belonging to its control set Y_v.

To conclude, there is an invalid state reachable from a valid state in the given CPOG iff the following Boolean formula is satisfiable:

$$\mathcal{ISR} = \mathcal{CONF} \cdot \mathcal{SIG} \cdot \mathcal{IS}$$

In the example from Figure 15 SAT solver finds the following assignment of variables: $\langle(\mathsf{conf}_a = \mathsf{conf}_c = 1, \mathsf{conf}_b = \mathsf{conf}_d = 0), (\mathsf{val}_y = 0)\rangle$ and $\langle(\mathsf{conf}'_a = \mathsf{conf}'_c = \mathsf{conf}'_d = 1, \mathsf{conf}'_b = 0), (\mathsf{val}'_y = 1)\rangle$.

If formula $\mathcal{ISR}$ is unsatisfiable then there is no reachable invalid state in the system.

7.4 Event conflict detection

Two events are said to be in *conflict* iff there is a reachable state $\langle C, \psi\rangle$ when both of them are enabled to fire but firing of one of them disables the other. Note that a conflict does not necessarily lead to a deadlock or an invalid state, and the monotonicity of configuration C and control signals ψ is not violated. Conflicts only affect the monotonicity of function $enabled_v, v \notin C$ and eventually lead to glitches or hazards in gate-level implementation of the controller.

(a) $C = \emptyset$, $y = 0$, $enabled_c = 1$

(b) $C = \{a\}$, $y = 1$, $enabled_c = 0$

Figure 16: Conflict between events a and c

Example of a simple conflict is shown in Figure 16. The system is in initial state $\langle\emptyset, y = 0\rangle$ and all the three vertices $\{a, b, c\}$ are enabled to fire. But if vertex a fires and sets its controlled variable y into 1 then vertex c becomes disabled due to the appearance of an arc (b, c) (see Subfigure (b)).

Verification formula $\mathcal{EC}$ (i.e. event conflict)

$$\mathcal{EC} = \mathcal{CONF} \cdot \mathcal{SIG} \cdot \mathcal{CS}$$

is constructed from the valid state constraint $\mathcal{CONF} \cdot \mathcal{SIG}$ and constraint $\mathcal{CS}$ which is satisfiable iff there is a state $\langle\mathsf{conf}', \mathsf{val}'\rangle$ reachable from $\langle\mathsf{conf}, \mathsf{val}\rangle$ by firing of an enabled vertex v such that another enabled vertex u becomes disabled:

$$\mathcal{CS} = \bigvee_{\substack{v,u \in V \\ u \neq v}} enabled_v \cdot enabled_u \cdot \mathcal{FIRE}(v) \cdot \mathcal{SIG}' \cdot \overline{enabled'_u}$$

As before terms $\mathcal{SIG}'$ and $enabled'_u$ operate on variables $\langle\mathsf{conf}', \mathsf{val}'\rangle$ of the new state constrained with function $\mathcal{FIRE}(v)$. The solution for the example in Figure 16 is: $\langle(\mathsf{conf}_a = \mathsf{conf}_b = \mathsf{conf}_c = 0), (\mathsf{val}_y = 0)\rangle$ and $\langle(\mathsf{conf}'_a = 1, \mathsf{conf}'_b = \mathsf{conf}'_c = 0), (\mathsf{val}'_y = 1)\rangle$.

7.5 Mutex property checking

A CPOG specification can contain more than one vertex corresponding to the same action in the modelled system. Figure 17 shows an example of MPS430 (a general purpose microprocessor [1]) instruction specification. The CPOG represents the operational flow for an ALU operation with addressing mode **#123 to Rn/PC** e.g. adding a constant to a register Rn or program counter (PC). *PC++* is the increasing of the counter, *IR* is the instruction reading, *ALU* is an arithmetic operation; *pca* (program counter access) is the control variable that is set to one (zero) if the second operand is PC (register Rn).

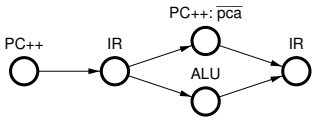


Figure 17: Multiple actions occurrence

The scenario is to increase PC (action $PC{+}{+}$) and to load the constant (which occupies the next word in the code memory after the instruction itself) into the instruction register (IR). After that the constant is added to the second operand (ALU). If the second operand is not PC then it is increased concurrently (the second occurrence of $PC{+}{+}$). The last step is to load the next instruction into the instruction register (the second occurrence of IR).

To avoid arbitration it is necessary to be sure that there are no concurrent requests to the same action. Looking at Figure 17 one can observe that the two occurrences of actions $PC{+}{+}$ (and IR as well) are mutually exclusive, but there can be much more sophisticated CPOGs and an automated procedure is needed to be sure that the mutex property is not violated for a pair of given events.

The SAT-based verification formula for checking the mutex property for given vertices $v, u \in V$ is

$$\mathcal{MUTEX}(v,u) = \mathcal{CONF} \cdot \mathcal{SIG} \cdot enabled_v \cdot enabled_u$$

It is satisfiable iff there is a valid state $\langle \mathsf{conf}, \mathsf{val} \rangle$ such that both vertices v and u are enabled to fire.

8 Conclusions

The paper introduced the extended Conditional Partial Order Graph model able to describe systems with dynamic changes of internal control signals. The complexity of the model requires software tools for efficient CPOGs synthesis and verification. This work provides theoretical base for the verification tool which is the part of the whole CPOG tool flow that is currently under development. The creation of this tool flow would enable us to create a set of real-life examples of synthesis and verification including specification of processors and NoC routers that are very suitable for CPOG characterisation.

Acknowledgement

Authors would like to thank Victor Khomenko for his expert advice on SAT theory and Danil Sokolov for the useful discussions. This work was supported by EPSRC grant EP/C512812/1.

References

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

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

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

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

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

[6] J. Esparza. Decidability and Complexity of Petri Net Problems - an Introduction. In *Lectures on Petri Nets I: Basic Models, W. Reisig and G. Rozenberg (Eds.).*, 1998.

[7] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting State Coding Conflicts in STG Unfoldings Using SAT. In *Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)*, 2003.

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

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

[10] Andrey Mokhov and Alex Yakovlev. Transition Sequence Encoder. Technical report, University of Newcastle upon Tyne, September 2006.

[11] Andrey Mokhov and Alex Yakovlev. Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis. Technical report, University of Newcastle upon Tyne, January 2008. (The paper accepted for DATE 2008).

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

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

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

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

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

[17] Ingo Wegener. *The Complexity of Boolean Functions*. Johann Wolfgang Goethe-Universitat, 1987.