Contextual Net Unfolding and Asynchronous System Verification

Alex Semenov*and Alex Yakovlev†
University of Newcastle upon Tyne, United Kingdom

Abstract

A finite truncated unfolding of a Petri net is known to be a useful tool for verifying correctness of asynchronous systems. It may however suffer from an unnecessary exponential explosion when applied to Petri net models of hardware with *level-based* signals (e.g., circuits built of logic gates). The paper proposes the use of *contextual* nets instead, by presenting an algorithm for constructing a finite contextual net unfolding segment. The method is applied to the verification of an asynchronous control structure implementing a four-slot asynchronous communication mechanism, intended for use in real-time systems. The paper shows when the use of contextual net unfolding is most advantageous compared to the ordinary Petri net unfolding.

1 Introduction

Petri nets (PNs) and their interpreted version, called Signal Transition Graphs (STGs), have already demonstrated their usefulness and practicality in synthesis and verification of asynchronous circuits and systems (e.g., [21, 7, 11, 20, 19]). The synthesis tools exploiting PNs/STGs as their underlying formalism include SIS [15], ASSASSIN [23], Forcage [5] [1] and relatively recently, petrify [2]. The verification tools, like versify from Technical University of Catalunya, define a circuit mainly in terms of Boolean equations and use Petri nets to specify the circuit's environment. The designer of an asynchronous system may, however, need to model his/her design at a higher level of abstraction than the level of logic gates, e.g. in hardware-software co-design. It is therefore generally more appropriate to consider both the system and its environment to be represented by a Petri net. For the cases when the system *to be verified* is already implemented at the logic gate level, it is possible to convert each gate into a corresponding fragment of the so-called Circuit PN [18], which is syntactically a subclass of STGs. The above mentioned synthesis tools can perform some verification of an STG but approach it as the initial behavioural specification of a circuit *to be synthesised*.

*Supported by EPSRC grant GR/J52327.

†Supported by EPSRC grants GR/J52327 and GR/K70175.

[1] This work actually introduces the model of Change Diagrams, which are very close to STGs both in syntax and semantics.

All these tools perform their verification using the State Graph (SG) model generated by the STG model by means of explicit or symbolic (e.g., BDD [11]) reachability graph traversal. Such a way is known to be prone to exponential explosion. An alternative and often more efficient technique, based on Petri net unfolding, was presented in [7] as a way to verify Circuit PNs. Recently, several modifications of the McMillan's basic PN unfolding algorithm have been proposed – most of them trying to improve on the size of the truncated unfolding by elaborating the cut-off condition [6, 14, 3]. Those modifications did not however take into account another aspect critical for the verification of asynchronous systems. This aspect, considered in this paper, often has much stronger effect on the size of the PN's truncated unfolding than the use of a particular cut-off condition.

A characteristic feature of STGs which fall into the category of Circuit PNs is the presence of a *contextual* dependence between places and transitions associated with different logic elements of the circuit. The intuitive meaning of such dependence happens to be exactly the same as that of the *context flow-relation* in contextual nets [9]. However, the "standard" STG uses the "ordinary" PN model, and those contextual links are usually represented in terms of the usual flow-relation as *self-loop* arcs (often in their short-hand form of dual-headed arcs). The application of the "ordinary" PN semantics to STGs with self-loops, as in all existing unfolding algorithms, often results in the same sort of exponential blow-up as with the use of SGs (see an example below).

The root of this problem, informally, is in the different semantic interpretations applied to self-loops in ordinary PNs and in contextual nets. This appears to be critical, from the size point of view, for the partial order (or non-sequential) semantics [1]. Indeed, the PN unfolding is a partial order technique which proves to be efficient for a large class of PN models – it can keep concurrency in its natural, "non-interleaving", form. Unfortunately, the classical partial order semantics, underlying the existing unfolding methods, loses it efficiency when the paradigms of concurrency and conflict are mixed up. Consider a well-known example of a pair of transitions $t1$ and $t2$ sharing a single place p by means of a self-loop flow relation as shown in Figure 1.(a). The place p is thus both a predecessor and a successor to transitions $t1$ and $t2$. The classical partial order semantics does not allow those transitions to fire simultaneously, since the token in the shared place is treated as a *single resource* (structural conflict on p). It will unfold this net into two interleaving sequences (Figure 1.(b)), resulting in a combinatorial blow-up. For an asynchronous system's modelling, we often use self-loops with shared places but apply a different understanding to them. Their meaning is not resource-based but rather *condition-based* – a transition, associated with a gate's output, can fire if a place, associated with another gate's output, is marked with a token. In that sense, the above example should be interpreted as the fact that two gates share the same input wire (the False or True state of the wire is modelled by place p), which when set to logical 1 (or 0) enables both gates (the events on the gates' outputs are represented by transitions $t1$ and $t2$), and so the latter can fire completely independently of each other.

This paper proposes a new STG unfolding algorithm, to allow explicit contextual arcs in the underlying PN. This algorithm may find its theoretical justification in the work on non-sequential semantics of contextual nets by Busi and Pinna [1]. We have implemented this algorithm in our asynchronous circuit verification/synthesis tool PUNT, and applied it to a number of characteristic examples of models with contextual arcs. Those examples include the verification of asynchronous circuits for hazard-freedom (checking *output signal*

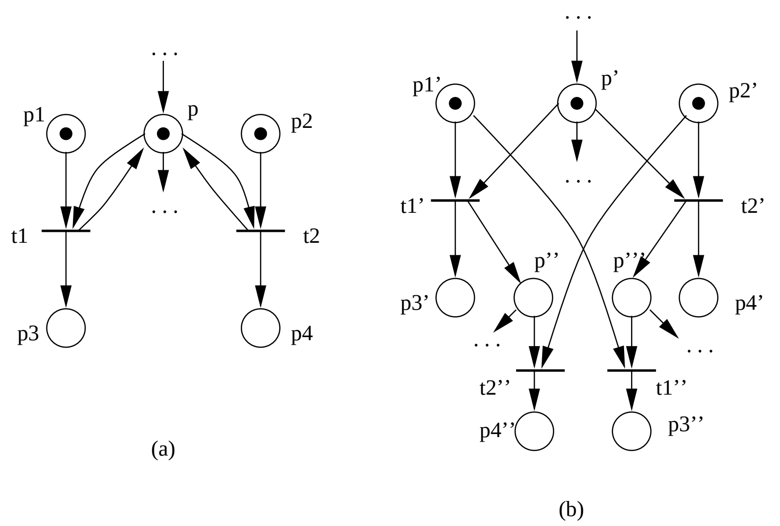


Figure 1: An example of a Petri net with self-loops (a) and its unfolding according to the requirement of structural conflict (b)

persistence) and asynchronous communication mechanism for coherence (checking *freedom from simultaneous access to data for read and write*). In the latter example, we were unable to construct the truncated unfolding for an ordinary PN interpretation, due to the large number of self-loop arcs, while its contextual net unfolding appeared to be fairly compact. Note that it is often the case that even initial STG specifications of asynchronous systems (see example in Section 4) may contain contextual arcs and therefore the use of the new unfolding algorithm will be productive. The use of contextual nets in the modelling of asynchronous systems substantially increases the efficiency of the unfolding (i.e. partial order) approach, and thus enhances the power of PNs as a modelling/verification tool in general.

2 Positive Contextual Nets

This section introduces the basic concepts required to develop the algorithm for a contextual net unfolding. For simplicity, we consider here a subclass of contextual nets in which only the so-called *positive* [1] context is allowed. This restriction mainly has practical reasons since this class is sufficient to deal with net models of asynchronous systems, which would otherwise be represented by ordinary PNs with self-loops. For example, Circuit PNs, mentioned in the introduction, are a class of 1-safe nets with self-loops. Furthermore, one can imagine that it is possible to extend this technique to nets with *negative* context (on the basis of contextual interpretation of inhibitor arcs). This can be done by the (standard) transformation of a bounded inhibitor net to an ordinary PN with complementary places and self-loop arcs [12].

Further reading on contextual Petri nets can be found, e.g., in [9, 1].

2.1 Contextual Net Definition

In order to define a contextual net we need the definition of an ordinary Petri net (see, e.g. [12, 13]).

Definition 2.1 *1. A* **Petri net** *(PN), or simply* **net***, is a tuple $\langle P, T, F\rangle$, with sets of places P, transitions T, and flow relation F, such that $P \cap T = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$.*

2. A **marked Petri net** *is a tuple $N = \langle P, T, F, m_o\rangle$, where $\langle P, T, F\rangle$ is a PN and function $m_o : P \to \omega$ (ω is a set of nonnegative integers) is the* **initial marking***.*

For $x \in P \cup T$, we define ${}^\bullet x = \{y \in P \cup T | (y, x) \in F\}$, called the set of *predecessors* of x, and $x^\bullet = \{y \in P \cup T | (x, y) \in F\}$, called the set of *successors* of x. A net is T-restricted if for all transitions t we have ${}^\bullet t \neq \emptyset \neq t^\bullet$. We shall further assume that all nets are T-restricted.

Generally, a finite multiset over the set P of places is called a *marking*. A marking m is represented with a number of tokens $m(p)$ in each place $p \in P$. The enabling and firing rules for a marked Petri net N are defined in a usual way [10].

Definition 2.2

1. *A* **(positive) contextual net** *(CN) is a tuple $\langle P, T, F, C\rangle$, where $\langle P, T, F\rangle$ is a Petri net and $C \subseteq P \times T$ is the* **context relation** *such that $C \cap F = \emptyset$.*

2. *A* **marked (positive) contextual net** *is CN augmented with an initial marking m_o of its places in P, i.e. it is a tuple $\langle P, T, F, C, m_o\rangle$.*

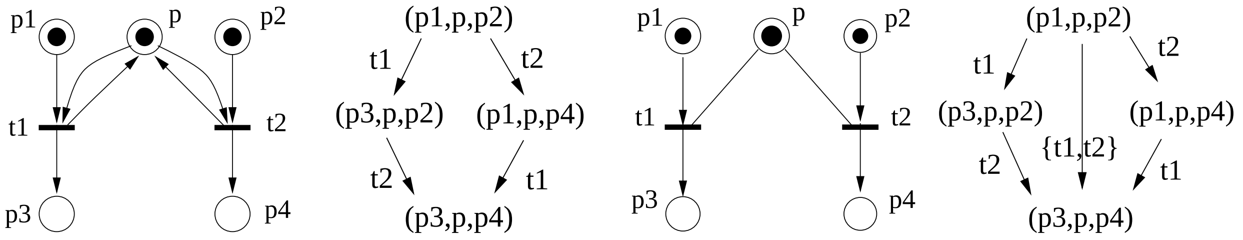
For a transition $t \in T$ we define $\widehat{t} = \{y \in P \cup T | (y, x) \in C\}$, called the *context set* of t (the places in context set of t are called *contextual places*). In graphical representation of CNs, the context relation will be depicted with a line without arrows.

To simplify terminology, unless it creates confusion, we shall be omitting term "marked" in front of the abbreviations PN and CN when referring to the marked versions of PN and CN respectively.

A transition t is called *enabled* at marking m if ${}^\bullet t \subseteq m$ and $\widehat{t} \subseteq m$. Thus, for a transition to be enabled we require that, in addition to the marking of all its predecessor places, all its contextual places are marked. We assume [2] that an enabled transition t may stay enabled for any unbounded (but finite) period of time until it either fires or becomes disabled by the firing of another transition, which changes the marking of the predecessor or contextual places of t.

The firing of a transition t enabled at marking m is defined as an *instantaneous action* producing the new marking $m' = (m \setminus {}^\bullet t) \cup t^\bullet$. Such marking m' is called *directly reachable* from marking m. If there is a (possibly empty) sequence of transition firings, called *firing sequence*, from a marking m to a marking m', the latter is called *reachable* from marking m. The set of markings that can be reached from the initial marking by all possible firings of transitions is called the *reachability set*. The graphical representation of the reachability set in which nodes are *uniquely* labelled with markings and arcs are associated with the direct reachability relation is called the *reachability graph*. The arcs of the reachability graph are usually labelled with transitions which fire between the adjacent markings.

[2] This assumption is important if we are going to model the physical asynchronous systems, where delays are finite but may be unknown.



(a) (b) (c) (d)

Figure 2: Petri net with self-loops (a) with its step reachability graph (b) and contextual net (c) with its step reachability graph (d)

2.2 Role of Conflict Relation

A pair of transitions $t1$ and $t2$ in a CN are said to be in *symmetric structural conflict* iff ${}^\bullet t1 \cap {}^\bullet t2 \neq \emptyset$. $t1$ and $t2$ are said to be in *asymmetric structural conflict* iff ${}^\bullet t1 \cap {}^\bullet t2 = \emptyset$ but ${}^\bullet t1 \cap \widehat{t2} \neq \emptyset$. A pair of transitions $t1$ and $t2$ are said to be in *symmetric dynamic conflict* (*asymmetric dynamic conflict*) if they are in *symmetric structural conflict* (*asymmetric structural conflict*) and there exist a marking in which they are both enabled.

Note that our definition of dynamic conflict is fairly strong (it is brought to match to the notion of conflict in the unfolding – see below). E.g., two transitions which share a predecessor place containing at least two tokens are considered to be in conflict as there is a chance that they will be "competing" for the same token (resource).

The above form of reachability graph defines the so-called *interleaving semantics* of a CN. This type of semantics cannot for example distinguish between two behaviours of the following kind. In the first case, a pair of transitions can fire independently (in either order or simultaneously). In the second case, two transitions are in symmetric dynamic conflict where they share a predecessor place with one token. They cannot fire simultaneously; however they are allowed to fire one after the other in either order. It is possible to apply a more refined type of semantics, called *step semantics* [4], which can differentiate between those behaviours. In the first case, we allow a transition step, consisting of *both* transitions, which will be explicitly represented in the *step reachability graph*. The step semantics of CNs is defined in [9, 1].

Example. Figure 2.(a) shows an example of a PN with self-loop flow relation between place p and transitions $t1$ and $t2$. Its reachability graph and step reachability graph are the same – they are shown in part (b) of the figure. For a "similar" contextual net, shown in Figure 2.(c), where the relation between p and $t1$ and $t2$ is contextual, the step reachability graph is different – it is depicted in part (d). Note that the same step reachability graph would be produced by the above CN (or PN) if we delete place p and its context edges.

Note that considering steps of two or more *non-conflicting* transitions, in addition to simple, single-transition steps does not create new markings in the reachability set of a

CN. It produces only new firing sequences, called *step sequences*. As has been noted in the introduction, the partial order models, such as net unfoldings, are sensitive to such semantical distinctions as well, but their representation reflects the above distinction in the structure of the semantic model. For example, the presence of dynamic conflicts in the original CN generates choice between alternative branches of the net unfolding, which consists of the instances (or occurrences) of places and transitions taken from the original CN.

A CN is said to be *finite* if sets P and T are finite. A CN is said to be *k-bounded* (or simply bounded) if there exists such a k that for any place p in any reachable marking m, we have $m(p) \leq k$. A 1-bounded CN is called *safe* CN.

A transition t is said to be *disabling* another transition t', if there exist a marking m in which both t and t' are enabled and t is not enabled in the marking m' reached from m by firing t'. We also say that in this case t is *non-persistent* transition or CN is *non-persistent with respect to* t. Otherwise CN is called persistent with respect to t. We can generalise the notion of persistence to the case of a subset of transitions. Finally, a CN is called *persistent* if it is persistent with respect to all its transitions.

It is obvious that the necessary condition for a disabling (and hence non-persistency) between transitions is the presence of dynamic conflicts between them. The opposite is not true – indeed, in the CN shown in Figure 2.(a) transitions $t1$ and $t2$ are in dynamic conflict but they do not disable each other. The property of transition persistence (or non-persistence) is crucial in the analysis of asynchronous circuit behaviour. It corresponds to the behavioural quality of a circuit to be either free from hazards on a given signal or not. In the use of PNs for analysis of hazard-freedom we would like to be able to identify hazards with some local properties of the semantical model, such as conflicts – the latter are easer to check than those properties which are based on the global information such as markings and their reachability. Therefore the cases like the one shown in Figure 2.(a) appear to be a difficulty for analysis because from the circuit semantics such behaviour can be correct (the signal transition corresponding to say $t1$ is not disabled by firing $t2$) while its analysis through the notion of dynamic conflicts raises an alarm [3]. CNs allow us to distinguish the cases of contextual dependence from those which are real dynamic conflicts and imply hazardous behaviour in the circuit terms. The CN unfolding, introduced in the next section supports this distinction. It fills the semantic gap between the notion of a dynamic conflict in the net and that of non-persistence, the gap characteristic for ordinary PNs.

In the following sections, we will be interested in the analysis of properties of the interpreted version of CN, that is the CN whose transitions are labelled with the names of actions in the modelled asynchronous system. For example, we would have to verify if a pair of actions can be enabled in the same marking or if a transition labelled with a specific action can be disabled by some other transition. While such properties can certainly be verified using the reachability graph, we would like to reduce the risk of combinatorial explosion (for highly concurrent behaviours). We shall try to exploit the succinctness of the net unfolding in representing concurrency in its true form. Indeed, most of these properties are relation-based – they can be formulated as queries about the ordering relations between instances of places and transitions, i.e., precisely the elements of the unfolding structure.

[3]This alarm can certainly be checked and identified as false by means of explicit analysis for disabling – but this would affect the overall efficiency of the analysis process and hence we would prefer to avoid it.

3 Contextual Net Unfolding

3.1 Petri Net Unfolding

Methods and algorithms and for unfolding ordinary PNs can be found elsewhere[8, 3]. This section defines the CN unfolding and suggests an algorithm for its construction in many respects using the McMillan's approach for PN unfolding. This approach is based on the following:

- The definition of an *occurrence net* , i.e. an *acyclic* PN, which when traversed from its min-elements visits each place and transition *only once*.

- The algorithm for the construction of a *labelled occurrence net* for a given (potentially cyclic) PN by means of iterative instantiation of places and transitions of the latter. This construction process begins with the instances of places belonging to the initial marking of the PN. Such place instances are the min-elements of the unfolding; let us call them *minimal places*. For each initially marked place p, it creates as many individual instances p' of this place in the occurrence net as there are tokens in p, i.e. $m_o(p)$ instances.

Following this approach, we shall build the CN unfolding as an occurrence contextual net in which places and transitions are labelled with the corresponding names of the places and transitions in the original CN. To avoid use of long terms, in the following we shall be using words "places" and "transitions" to address the objects of the CN unfolding, i.e. place instances and transition instances, respectively. Only if it creates confusion with the places and transitions of the original CN, we shall be adding word "instance".

The occurrence net is a special type of PN which must satisfy the conditions of the following definition.

Definition 3.1 *[8] A* (P, T) **labelled occurrence net** *is a tuple* $\langle P', T', F', L' \rangle$, *where* $\langle P', T', F' \rangle$ *is a* PN *and* L' *is a labelling function which maps* P' *onto set* P *and* T' *onto set* T. N' *must satisfy the following properties:*

1. *Well-foundedness: every subset of* T' *contains a minimal element with respect to* F'^* *(transitive and reflexive closure of* F'*).*

2. *No backward conflict: if* $t1^\bullet \cap t2^\bullet \neq \emptyset$ *then* $t1 = t2$.

3. *No self-conflict: for all* $t1, t2, t3 \in T'$, *if* $t1F'^*t3, t2F'^*t3$ *and* ${}^\bullet t1 \cap {}^\bullet t2 \neq \emptyset$, *then* $t1 = t2$.

4. *No redundancy: for all* $t1, t2 \in T'$, *if* $L'(t1) = L'(t2)$ *and* ${}^\bullet t1 = {}^\bullet t2$, *then* $t1 = t2$.

One can easily check that the above definition guarantees that the labelled occurrence net satisfies the following important property: *if every place in the set of the minimal places is marked with a token, then every transition of the labelled occurrence net can fire once and only once.*

The occurrence net derived by unfolding a PN is defined as follows.

Definition 3.2 *[8] If $\langle P, T, F, m_o\rangle$ is a marked* PN*, then the* **unfolding** *of this net is the* maximal *(P, T) labelled occurrence net (up to isomorphism) $\langle P', T', F', L'\rangle$, satisfying the following properties:*

1. *for all $t' \in T', L'({}^\bullet t') = {}^\bullet L'(t')$ and $L'(t'^\bullet) = L'(t')^\bullet$.*

2. *$L'(P' \setminus cod(F')) = m_o$, where $cod(X)$ denotes the co-domain of relation X, and thus $P' \setminus cod(F')$ stands for the minimal places of the unfolding.*

3.2 Basic relations in acyclic contextual nets

In order to define the corresponding notions for contextual nets, we need several preliminary definitions. These will impose additional conditions on the structure of a (P, T) labelled positive occurrence CN when it is augmented with the context relation between places and transitions. Those conditions must ensure that every transition in the occurrence CN can be fired once and only once if all its minimal places are initially marked.

Thus, let us first consider any (P, T) labelled positive CN $CN = \langle P', T', F', C', L'\rangle$, where $\langle P', T', F', L'\rangle$ is a (P, T) labelled occurrence net and $C' \subseteq P' \times T'$ is a context relation, satisfying $C' \cap F' = \emptyset$.

Let us define the following relations in this net.

Definition 3.3 *(Weak and Strong Precedence)*

1. *Two elements x' and x'' of CN are said to be in the* **weak precedence relation***, denoted $x' \prec x''$, iff $(x', x'') \in (F' \cup C')^*$.*

2. *Two elements x' and x'' of CN are said to be in the* **strong precedence relation***, denoted $x' \prec\!\prec x''$, iff $\exists t : (x', t) \in F'^* : t \prec x''$.*

It is clear from these definitions that:

- $\prec\!\prec \subseteq \prec$

- if the first element in a pair of elements (x', x'') of CN is a transition (i.e. $x' \in T'$), then $(x', x'') \in \prec\!\prec$. This is a simple corollary of the definition of C', which is defined on $P' \times T'$.

- if $(x', x'') \in (\prec \setminus \prec\!\prec)$ then: (1) $x' \in P'$, and (2) $\exists t \in T' : t \prec\!\prec x''$ and $x' \in \hat{t}$.

Note that the set-theoretic difference between the weak and strong precedence is a relation which reflects the specific nature of acyclic CNs. If a place p is in such a relation with another element x, this means that p must be marked with a token before x is either marked (if $x \in P'$) or fired (if $x \in T'$). However, this does not state that when x is marked or fired, the token must have been removed from p.

We will also need definitions of conflict and semi-conflict.

Definition 3.4 *(Conflict and Semi-conflict)*

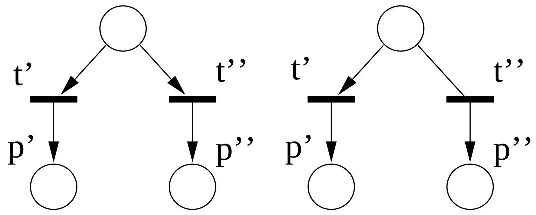


Figure 3: Conflict and semi-conflict.

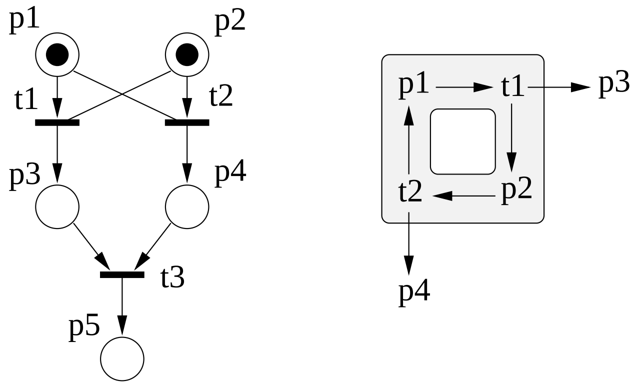


Figure 4: Example of CN which cannot be a contextual occurrence net; the graph of its *must occur before* relation.

1. *Two elements x' and x'' of CN are said to be in the* **conflict relation**, *denoted $x'\#x''$, iff there exist two distinct transitions t' and t'' such that ${}^\bullet t' \cap {}^\bullet t'' \neq \emptyset$ and $t' \prec x'$ and $t'' \prec x''$.*

2. *Two elements x' and x'' of CN are said to be in the* **semi-conflict relation**, *denoted $x'\vec{\#}x''$, iff there exist two distinct transitions t' and t'' such that $\widehat{t'} \cap {}^\bullet t'' \neq \emptyset$ and $t' \prec x'$ and $t'' \prec x''$.*

The difference between two conflict relations is illustrated in Figure 3. In the left part of the figure t' and t'' are in conflict, while in the right part in semi-conflict. The semi-conflict relation in CN corresponds to the asymmetric conflict in PNs discussed earlier. The firing of t' can disable t''. In this case, p' and p'' will not be marked together. However, if t'' manages to fire *before* t', then both p' and p'' can have tokens in the same marking. Obviously, if p' and p'' are in conflict, there is no marking in which they both can have tokens. This illustration brings us to the third type of relation between elements in CN – independence.

Example. Consider the CN shown in Figure 4. This example satisfies the conditions Definition 3.1, applied to the CN taken above. At the same time we cannot allow this CN be a valid contextual occurrence net since its transition $t3$ cannot be fired. This is despite the fact that $t3$ satisfies the “no self-conflict” condition of Definition 3.1.

Indeed, in order to fire transition $t3$ both $p3$ and $p4$ must be marked in the same marking. In order to mark $p3$, $t1$ must fire before the token in removed from $p2 \in {}^\bullet t2$. However, firing $t1$ removes token from $p1$ and thus prevents $t2$ from firing. The weak precedence relation defined as on the set $(C' \cup F')^*$ does not reflect all possible precedences that are expressed by the CN. For example, the weak precedence relation (due to context relation) between $p2$ and $t1$ in Figure 4 reflects only the fact that $p2$ must be marked with a token before $t1$ fires. On the other hand, again due to the context relation between $p2$ and $t1$ we can state that $t1$ must occur before the token is removed from $p2$, and hence before the occurrence of $t2$, the successor of $p2$. This new type of precedence combined with the transitive closure of F' gives rise to the new relation which will be important for the definition of independence relation.

Definition 3.5 *(t-Weak Precedence) Two elements x' and x'' of CN are said to be in the* **t-weak precedence relation**, *denoted $x' \prec^t x''$ if $(x', x'') \in (F'^{+} \cup C'^{-1})$, where C'^{-1} is the inverse of C'.*

This relation should be interpreted as *"must occur before"*. The prefix t designates the fact that we are using relation C'^{-1}, which originates in transitions. Let us construct a graph where vertices represent elements of the CN involved in context relation $(x', x'') \in C'$ and elements whose independence we need to check. The edges represent $\prec^t$.

On the basis of the $\prec^t$ relation we define the notion of contextual conflict in CN.

Definition 3.6 *Two elements x' and x'' of CN are said are said to be in* **contextual conflict**, *denoted $x' \widehat{\#} x''$, iff there exist two distinct transitions t' and t'' such that $(t', t''), (t'', t') \in (\prec^t)^+$ (there exists a cycle in the graph of the $\prec^t$ relation) and $t' \prec x'$ and $t'' \prec x''$.*

Example. The graph showing this relation for the example in Figure 4 is shown in the right part of this figure. As can be observed, there exists a loop in this graph which indicates that it is impossible to fire transitions belonging to this loop due to contextual dependence between them.

Thus the independence relation is introduced as:

Definition 3.7 *Two elements x' and x'' of the* CN *unfolding are said to be in the* **independence relation**, *denoted $x' \| x''$, iff the following three conditions are satisfied:*

1. *x' and x'' are not in conflict $((x', x'') \notin x' \# x'')$;*
2. *x' and x'' are not in strong precedence with each other $(x' \prec\!\prec x'')$;*
3. *x' and x'' are not in contextual conflict $((x', x'') \notin x' \widehat{\#} x'')$.*

The first two conditions are trivial extension of those used in defining independence in occurrence nets. The third condition shows that a pair of elements x' and x'' may be independent even if they are in semi-conflict but not in contextual conflict. For example, places p' and p'' are in semi-conflict in Figure 3 but are also independent. Therefore these two places can be marked together.

3.3 Contextual net unfolding

We are now ready to define the contextual occurrence net.

Definition 3.8 *A* (P,T) **labelled (positive) contextual occurrence net** *is a* CN $CN = \langle P', T', F', C', L' \rangle$*, where* $\langle P', T', F', L' \rangle$ *is a* (P,T) *labelled occurrence net,* $C' \subseteq P' \times T'$ *is a context relation, satisfying* $C' \cap F' = \emptyset$*, and for any* $t \in T' : \forall p1, p2 \in {}^{\bullet}t \cup \widehat{t} : p1 \| p2$ *(i.e. all preceding and context places for every transition are pairwise independent).*

Our definition of a contextual occurrence net is different from the one in [1] (Definition 6). It is, however, possible to prove that the conditions defined in items 3 to 5 of Definition 6 in [1] follow from our condition based on the independence relation in Definition 3.8.

The occurrence net derived by unfolding a CN is defined as follows (in a similar way as in [8]).

Definition 3.9 *If* $\langle P, T, F, C, m_o \rangle$ *is a marked* CN*, then the* **unfolding** *of this net is the* maximal (P,T) *labelled contextual occurrence net (up to isomorphism)* $\langle P', T', F', C', L' \rangle$*, satisfying the following properties:*

1. *for all* $t' \in T', L'({}^{\bullet}t') = {}^{\bullet}L'(t'), L'(t'^{\bullet}) = L'(t')^{\bullet}$ *and* $L'(\widehat{t'}) = \widehat{L'(t')}$.
2. $L'(P' \setminus cod(F')) = m_o$*, where* $P' \setminus cod(F')$ *stands for minimal places of the unfolding (they are mapped onto the initial marking).*

The notion of independence relation plays critical part in the algorithm of the unfolding as it allows to determine if a particular transition is instantiated or not. A new instance t' is constructed when there exists a set of mutually independent instances of places in $({}^{\bullet}t) \cup \widehat{t}$. Similar to the PN unfolding, the following "cornerstones" are defined:

- A *configuration* $\mathcal{C}$ of CN unfolding is defined as a transitively backwards closed (with respect to $F' \cup C'$) set of transitions (i.e. if $t' \prec t''$, then $t'' \in \mathcal{C}$ implies $t' \in \mathcal{C}$) such that none of two instances t' and t'' in $\mathcal{C}$ are in conflict or contextual conflict. A local configuration of t', denoted as $\lceil t' \rceil$, is a minimal configuration including t'.
- A *postset of configuration* $\mathcal{C}$, denoted as $\mathcal{C}\bullet$, is a set of places which are successors but not predecessors of the transition instances in $\mathcal{C}$. The postset consists of instances of places reachable after firing of all transitions in $\mathcal{C}$.
- A *final state of* $\mathcal{C}$, denoted as $F_s(\mathcal{C})$, is a set of places which is obtained by mapping the instances in $\mathcal{C}\bullet$ onto places of the original CN.

A local configuration $\lceil t' \rceil$ also has a postset and a final state. Note that places in $\widehat{t'}$ are not consumed by t' and, therefore, instances in $\widehat{t'}$ are always included into the postset of its local configuration.

In a PN unfolding, if a configuration $\mathcal{C}1$ is a subset of another configuration $\mathcal{C}2$, then the $F_s(\mathcal{C}2)$ is always reachable from $F_s(\mathcal{C}1)$ by firing all transitions whose instances are in $\mathcal{C}2 \backslash \mathcal{C}1$. Firing all such transitions is always possible because, by the definition of configuration of a

PN unfolding [8], all their instances are not in conflict with any transitions in $\mathcal{C}2$ and hence, if enabled cannot be disabled.

In a CN unfolding, it is possible that transition instances that are in semi-conflict belong to one configuration. Let the above-mentioned $\mathcal{C}2$ be such a configuration and let $t2 \in \mathcal{C}2$ and $t1 \in \mathcal{C}1$ be in semi-conflict $t2 \vec{\#} t1$. Then transition $t1$ may be fired before $t2$, and thus disable $t2$. At the same time, if $t2$ fires before $t1$, the latter is not disabled, and thus configuration $\mathcal{C}2$ becomes valid. There always exists at least one firing sequence which includes all transitions of any configuration in the contextual CN unfolding.

3.4 Contextual net unfolding segment

The CN unfolding can be infinite. To obtain a finite fragment of CN unfolding (called a truncated unfolding in [8]) we introduce a cutoff condition.

Definition 3.10 *A transition instance t' is called a* cutoff point *if there exits another instance t'' such that :*
- $|\lceil t'' \rceil| < |\lceil t' \rceil|$, *and*
- $F_s(\lceil t'' \rceil) = F_s(\lceil t' \rceil)$.

Note that the cutoff definition imposes the same conditions on the newly generated instance t' as in [8]. Choice of these conditions may introduce redundancy into the unfolding fragment, i.e. some transitions will be added although their successors do not add any information about the CN behaviour. A number of works, e.g. [14, 3], suggest cutoff conditions which are tuned to battle this redundancy. However, in this work we are not concerned with this redundancy. The CN unfolding algorithm in given in Figure 5.

The algorithm uses the QUEUE to order the transitions before adding them into the unfolding. Thus each transition is only considered *after* all transitions with smaller local configuration were already added to the unfolding. Each newly generated transition instance is checked if it is a cutoff point. If it is, then the instances and place instances in $t'\bullet$ are marked so that they cannot be chosen as inputs of another instance. The algorithm starts from considering transitions enabled at the initial marking m_0. For other transitions t a set of *mutually independent* instances of places in $(\bullet t) \cup \hat{t}$ is found. If such a set exists and instance t' has not been constructed for this set, then the transition is added to the QUEUE as a candidate for a new instance. Eventually, for a bounded and finite CN the QUEUE will be exhausted due to cutoff point transitions. The procedure will then terminate producing a CN unfolding segment N'.

Example. Consider an example in Figure 6. In order to instantiate $t'8$ we would build a graph to verify independence of $p'8$ and $p'9$ shown in Figure 6. As can be observed, these places are not independent due to a contextual conflict between them. Hence $t'8$ will not be instantiated.

Let us refer to the arguments used in [8] to prove that the truncated PN unfolding fully represents the reachability set of the PN. Those arguments can be applied in an analogous way to the CN unfolding and its finite segment built by the algorithm given in Figure 5. We can therefore extend the main result of [8] to the CN unfolding segment.

```
proc Build segment(N = ⟨P, T, F, C, m_0⟩)
   Initialise N' with instances of places in m_0
   Initialise QUEUE with t enabled at m_0
   while QUEUE not empty do
      Pull t from QUEUE
      Add t' and t'• to N'
      if t' is a cutoff then do
         Mark t' and t'• as cutoff points
      end do
      for each t in T do
         Find unused set of independent instances of places in •t ∪ t̂
         if such set exists then do
            Add t to QUEUE in order if its |⌈t'⌉|
         end do
      end do
   end do
   return N'
end proc
```

Figure 5: Algorithm for obtaining CN unfolding segment.

Proposition 3.1 *Let CN' be the unfolding segment for a marked* CN *CN. Then the following two statements hold:*

1. *A marking m is reachable in CN iff there exists a finite configuration in the unfolding segment CN' such that its final state $F_s(C)$ is equal to m.*

2. *If CN is bounded, then its unfolding segment is finite.*

The proof of this proposition follows from the lemmata in [8], which can be applied without changes because the firing rule of CNs does not remove tokens from the contextual places of a firing transition.

4 Application: Modelling and Analysis of Asynchronous Systems

Asynchronous circuit verification is an important task in the list of the problems of asynchronous system design. It involves the construction of a formal model for a circuit together with its environment and checking this model with respect to circuit correctness properties. The latter have to be formulated in terms of the properties of the model. When using PNs as a formal language for modelling asynchronous systems [4], we identify PN elements, such

[4]Note that we are concerned mainly with modelling the so-called control circuits – the signals in such circuits have obvious "binary" nature. E.g., if $x = 1$, this implies that signal x in its active state controls

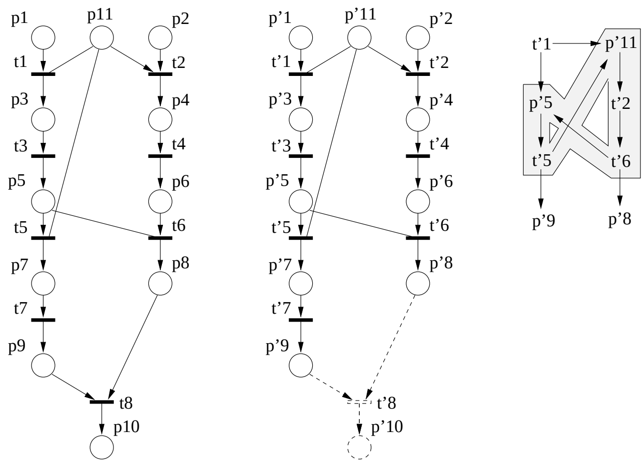


Figure 6: Another example of CN and its unfolding.

as places and transitions, with the corresponding conditions (e.g. the states of the circuit's components) and switching events (e.g. the rising and falling edges of binary signals).

In this section, we first show how one can construct a contextual net model for an asynchronous circuit design by means of interconnecting the fragments of CNs corresponding to individual circuit components. We then present an example of the use of CNs in modelling an implementation of asynchronous communication mechanism described in [16].

4.1 Asynchronous Control Structure

An *asynchronous control structure* (ACS) [22] is an interconnection (by wires) of a collection of components, each of which has a finite number of inputs and finite number of outputs. The wires, therefore, connect (1) the inputs of some elements to the outputs of other elements or (2) to the primary inputs of the circuit, or (3) the outputs of elements to the primary outputs of the circuit. Thus an ACS structure can represent an arbitrary interconnection of modules, with the only restriction that no two modules can drive a single signal. Formally, an Asynchronous (Discrete) Control Structure (ACS) is a directed graph $\langle V, H, Y, \rho \rangle$, where V is a finite set of *nodes*, associated with the abstract discrete components of the ACS, H, $H \subseteq V \times V$ is a finite set of *arcs*, standing for the *interconnections* between the components, $Y = \{y_1, ..., y_n\}$ is a finite set of finite-state *variables*, or *signals*, and $\rho : H \to Y$ is a *labelling* (total) function, associating every arc with a variable. Any two arcs labeled with the same variable must have the same source node (i.e. they represent a branching interconnection), so formally $\forall (v_1, v_2), (v_1', v_2') \in H$ we must have $\rho(v_1, v_2) = \rho(v_1', v_2') \Rightarrow v_1 = v_1'$.

We also denote the sets of input and output *interconnections* for a component v as: $I^H(v) = \{(v', v) \in H\}$, and $O^H(v) = \{(v, v') \in H\}$, respectively. The sets of input and output *signals*, or simply inputs and outputs, for a component are denoted: $I^Y(v) = \{y : \rho^{-1}(y) \in I^H\}$, and $O^Y(v) = \{y : \rho^{-1}(y) \in O^H\}$, respectively.

latching of some data path action or symolises a request/acknowledgement sent to some other part of the logic. Similarly, $x = 0$ may be interpreted as the control for a release of a data path operation or a handshake action.

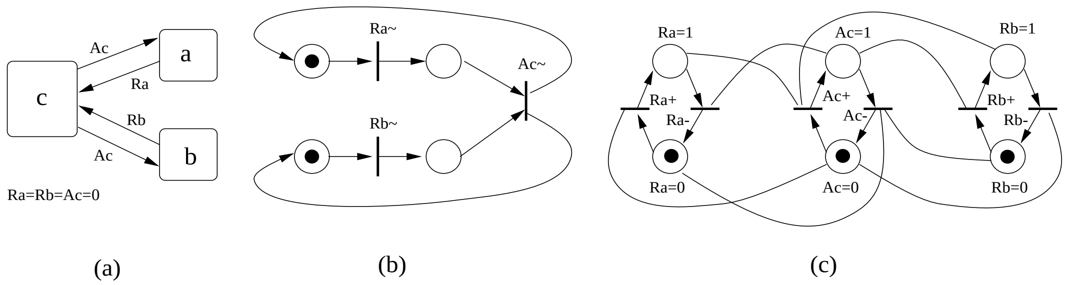


Figure 7: An asynchronous control structure (a) and its contextual signal transition graphs: event-based (a) and level-based (b).

Note that signal are always associated with some interconnections and the interconnections always have a source and destination node. Interconnections with explicit "dangling" ends are therefore not allowed. The ACS model is therefore *autonomous*. It should be obvious we may need to be able to model non-autonomous systems, i.e. the systems which have interconnections with the environments. Such interconnections are primary inputs and primary outputs for the system. In order to model such a system as an ACS, where the system's components are associated with nodes, we must represent the environment as a set of nodes (in particular, single node) connected to the system's part of the ACSs by the arcs standing for the primary inputs and the primary outputs. The modelling of the overall "world" as an autonomous system is a convenient abstraction, useful for its subsequent behavioural representation with a contextual net.

In general, for every variable $y \in Y$, the set $S(y) = \{y^0, y^1, ..., y^k\}$ is called the *set of variable values*, or states. An ACS $\langle V, H, Y, \rho \rangle$ is called a *Binary Asynchronous Control Structure* if $\forall y : S(y) = \{0, 1\}$. Hence, for a BACS, the set of allowed changes can be denoted as $Y \times \{+, -\}$, where "+" stands for a signal change from 0 to 1, and "-" for a signal change from 1 to 0. An ACS is called initialised if each variable $y \in Y$ is provided with an initial state. For binary ACSs such a state is a mapping $s_0 : Y \to \{0, 1\}^{|Y|}$.

In this paper we shall restrict ourselves with the discussion of only binary ACSs, but the general CN-based modelling approach is applicable for a more general class of ACSs.

A simple example of an ACS is described in Figure 7.(a). Here $V = \{a, b, c\}$, $H = \{(c, a),$ (a, c), (c, b), $(b, c)\}$, $Y = \{R_a, R_b, A_c\}$, and $\rho(a, c) = R_a$, $\rho(b, c) = R_b$, $\rho(c, b) = \rho(c, a) = A_c$. Furthermore, $I^H(c) = \{(a, c), (b, c)\}$, $I^Y(c) = \{R_a, R_b\}$, and so on. This ACS is initialised in state s_0 where $R_a = R_b = A_c = 0$.

The above interconnection model of a ACS hides the functionality of its nodes. The latter can be described in various ways. If the ACS is a logic circuit, then each node is associated with boolean functions defining the relationship between the input signals and the output signals of the node. Another possibility is that some nodes can be described behaviourally, e.g. by means of a PN or a CN. In the next subsection we shall introduce such a behavioural model, which is based on a CN and can be used for verification of the ACS.

4.2 Contextual Signal Transition Graphs

In order to move towards the behavioral modelling of ACSs we need to make the following assumption. We assume that every binary signal $y \in Y$ is classified into one of the two groups, depending on whether there is any semantical distinction between $y+$ (rising event) and $y-$ (falling event):

- *Level-based signals* $Y^=$, where the rising and falling events are semantically different, and therefore there is a need to explicitly use $y+$ and $y-$ to annotate transitions in the CN model.
- *Event-based signals* $Y^\sim$, where the rising and falling events are semantically equivalent, and therefore there will be no reason to explicitly use $y+$ and $y-$ in the CN model. Instead, we use the name $y^\sim$ (a toggle transition).

Thus, for a given ACS, a semantic partitioning into level- and event-based signals must be made: $Y = Y^= \cup Y^\sim$, such that $Y^= \cap Y^\sim = \emptyset$. It is, of course, always possible to *expand* an event-based interpretation into a level-based one. Such an expansion is essentially a trivial refinement and should be easily reflected in the CN model. The opposite conversion, *contraction* of a level-based to event-based model is less intuitive, and in practice requires the knowledge of global information about the classification of other signals. We can say about an ACS for which the above classification has been made that it is an level-based (event-based) ACS if $Y = Y^=$ ($Y = Y^\sim$). Otherwise, the ACS is called *mixed*.

For transition naming in CN models of ACSs, we will also allow the use of "dummies", or "silent transitions", which are used for modelling convenience and need not be associated with a specific signal in the ACS.

We can now define an ACS interpretation of a CN as follows.

Definition 4.1 *A* **Contextual Signal Transition Graph** *(*CSTG*) is a tuple* $\langle P, T, F, C, m_0, Y, \lambda\rangle$*, where* $\langle P, T, F, C, m_0\rangle$ *is a* CN*,* Y *is a set of signals in the modelled* ACS *such that* $Y = Y^= \cup Y^\sim, Y^= \cap Y^\sim = \emptyset$*, and* $\lambda : T \to (Y \times \{+, -, \sim\})$ *is the* **signal labelling (partial) function***.*

Similar classification, level-based, event-based or mixed, is extended to the CSTG models of their respective ACSs.

Example. A simple example of a CN model of an ACS in Figure 7.(a) is shown in Figure 7.(b). This is an event-based CSTG. We can observe that this model is essentially an ordinary PN – it does not involve contextual arcs. This model describes a circuit consisting of a C-element and two inverter. Another, more complex CN model of this ACS, with contextual arcs is shown in Figure 7.(c). This model explicitly represents the boolean conditions at which each gate switches from logical state 0 to logical state 1 and vice versa. For example, we can observe that the C-element switches its output A_c from 0 to 1 when both its inputs R_a and R_b are at 1. Similarly, it switches A_c from 1 to 0 when both R_a and R_b are at 0.

In the following subsection we explain a bit more the process of constructing a CSTG model for ACSs whose nodes are defined as logical gates.

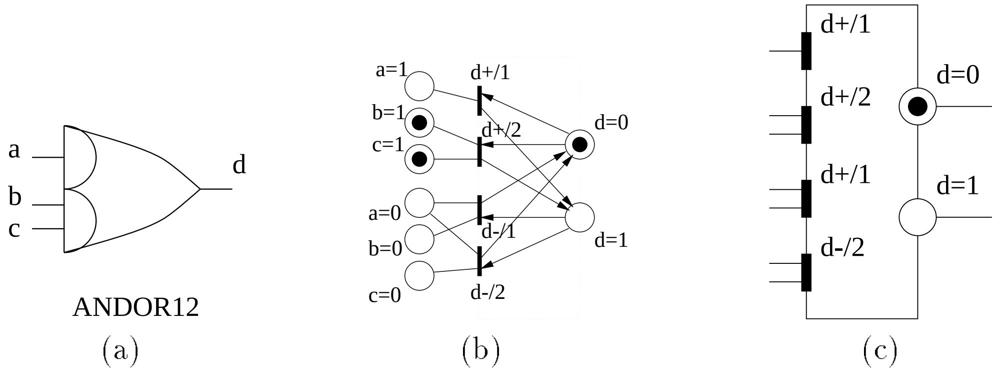


Figure 8: Logic gate and corresponding contextual STG

4.3 Constructing contextual STGs for asynchronous designs

We cannot provide in this paper a comprehensive "library" of fragments of CSTGs for various asynchronous system elements. We just show the basic idea of modelling boolean gates with CSTGs and then show several fragments of CSTGs for the modules used in our main application example.

Firstly, to model level-based circuits, consisting of logic gates, we adopt the same idea as in Circuit PNs. The main idea of this type of modelling is illustrated in Figure 8, where a logic gate of the type ANDOR12 is represented by a fragment of a CSTG. In this representation each wire y is associated with two *complementary* (in any reachable marking one and only one place out of the pair contains a token) places $y0$ and $y1$, standing for the logical 0 and 1 on the wire, respectively. For the output wire (the output of the gate) the net contains a number of transitions labelled with $y+$ and $y-$, standing respectively for the rising and falling edges. The conditions for firing these transitions are 'collected' through the combinations of input places in accordance with the Set and Reset conditions of the corresponding boolean function of the gate. For instance, the output of the ANDOR12 gate can be set to logical 1 in one of the two possible cases: either input a is at 1 or *both* b and c are at 1.

It is easy to observe that the input signals should be connected to the output transitions with *contextual* arcs thus preventing the change of the input as a result of the output signal transition. Indeed, any change of the marking of the input places should be caused by the environment, external to the gate. At the same time, if the output wire of the gate is connected to some other gate's input, the same contextual arcs will link the places storing the output's state to the transitions of the other gate's output.

To illustrate the above interconnection, refer to Figure 9, where a simple circuit (a simplified version of a circuit to control read operations in a VME bus Slave interface [21]) is converted into the corresponding CSTG.

The above examples use two forms of representing a CSTG. One form is *structural* or *modular*. It hides the internals of the fragments of the net corresponding to each logic gate. This form reflects the structure of the circuit. The other form is *behavioural*, that is a standard Petri net form, with all gate fragments being refined.

We have developed an automatic conversion tool, which takes as its input the specification

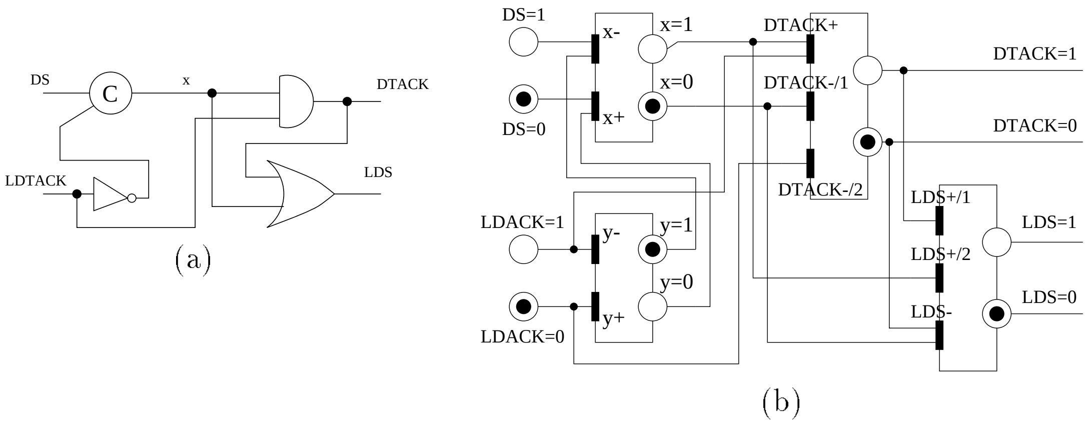


Figure 9: Logic circuit and corresponding contextual STG

of the logic circuit in format blif and produces as its output the CSTG description in the astg format [15]. The latter is a fairly common form of representation for PNs and STGs. It is used as input by most verification and synthesis tools, including our tool PUNT, which implements the CN unfolding algorithm described in Section 3.

4.4 Modelling an Asynchronous Communication Mechanism

A good example for the application of our CSTG unfolding technique can be found in [16]. This paper describes an asynchronous communication mechanism, which is used in designing real-time systems for safety-critical applications. The mechanism is defined at the high abstraction level as a concurrent system consisting of a reader process, a writer process and a four-slot data array and a set of shared control variables:

Writing		*Reading*	
$wr:$	$d[n, \overline{s[n]}] := input$	$r0:$	$r := l$
$w0:$	$s[n] := \overline{s[n]}$	$r1:$	$v := s$
$w1:$	$l := n \| n := \overline{r}$	$rd:$	$output := d[r, v[r]]$

The four slot data array is denoted by variable $d[i, j]$, which has a two-dimensional structure. The first subscript i of d defines one of the two pairs of slots (numbered as pair 0 and pair 1), while the second subscript j defines one of the two slots in each pair (numbered as slot 0 and slot 1). The values of subscripts are obtained through the auxiliary shared control variables. Variables n (for "next"), l (for "latest") and r (for "reading") are binary. The variables $s[i]$ and $v[i]$ are single dimensional arrays, each consisting of two elements; these arrays give pointers to the slots which are currently being written and read, respectively.

The primary aim of this communication mechanism is to organise the process of writing to and reading from data slots by the writer and reader in such a way that:

- (1) they can proceed from one operation to another, cyclically executing their respective sequences $wr, w0, w1$ and $r0, r1, rd$, in their own pace – that is *completely independently* of each other; and

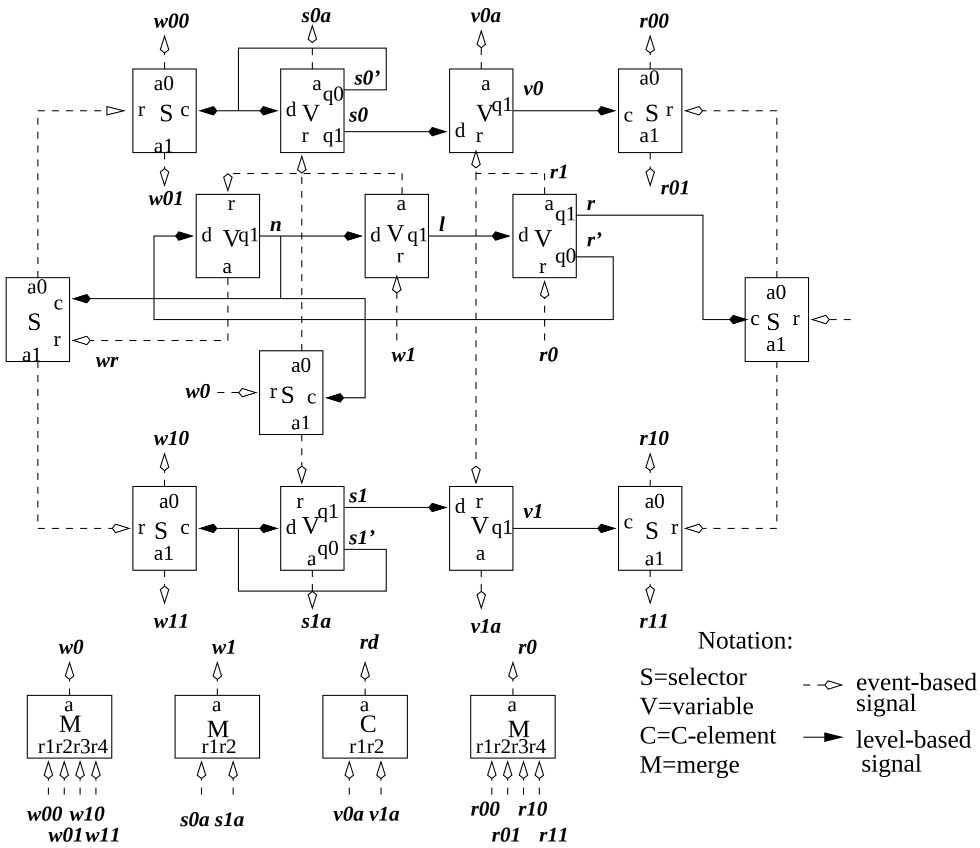


Figure 10: Asynchronous control structure for Simpson's communication mechanism

- (2) they never "clash" on writing to and reading from the same slot.

The author of the mechanism claimed (and proved in a fairly laborious way [17], analogous to manual reachability analysis) that the system satisfies the above stated requirements. For the first condition, he assumed that the writer may write data into a slot which has been written to but those data have not been read by the reader. Similarly, the reader is allowed to read from the same slot twice or more without a new data being written into the slot. With this assumption [5] both the writer and the reader may access data slots directly, without checking any sort of semaphore or other blocking (mutual exclusion) mechanisms. This would guarantee that both processes can act in a *truly* asynchronous and mutually independent fashion. This is critical from the viewpoint of *real-time* applications.

The second requirement is important from the *functional correctness* point of view. Any reading or writing operation applied to a slot of data is not atomic because the structure of data can be complex. Hence we cannot allow the reading and the writing of the same slot to interleave in time. Thus the system must provide mutual exclusion only at the level of data slots, without using such methods on control variables.

The repetitive access of the writer and the reader to the pairs and the slots with those pairs is controlled entirely by means of the shared variables, which can be implemented in hardware as bistables (i.e. flip-flops). The order in which these variables are assigned

[5] In this paper, we do not consider Simpson's "freshness" conditions[17].

new values and probed by the processes is outlined in the above procedures. Note that the notation $\|$ in operation $w1$ stands for parallel execution of its left and right hand side subaction.

Let us assume that the initial state of all binary variables n, r, l is 0 and that $s = v = (0, 0)$. The writing process is repeated as follows. The writer process first puts data into the data slot pointed by the current value of n (next pair to write) and the inverse of the $s[n]$. Then the writer toggles $s[n]$ to its inverse state. Finally, the writer assigns l (latest) to the value of n (next) and at the same time puts into n (next) the inverse value of r (reading).

The reading process begins with the assignment of r (pair for reading) to l (latest pair to have been written to). Then the reader copies the value of the writer's slot pointer s to its slot pointer v. Finally, the reader reads a value from the data slot pointed by its current value of r and $v[r]$, after which the reading process is repeated.

The above control mechanism was implemented [16] in the form of a hardware structure which can be modelled as an ACS shown in Figure 10. The nodes of this structure are components of four main types: "Selector", "Variable", "Merge" and "C-element". This ACS is of a mixed type; it has both event-based (shown with dashed lines) and level-based (shown with solid lines) signals. Note that in this implementation, we slightly simplified the system by serialising the parts of the $w1$ operation – first, $l := n$, and then $n := \overline{r}$. This does not affect the basic idea of independent interaction between the writer and the reader.

Selector (can also be called Switch) is a node which has one event-based input r (Request) and two event-based outputs $a0$ and $a1$ (Acknowledgements); it also has a level based input c (Control). The latter controls on which of the two Acknowledge outputs the node has to produce an event when it receives an event on Request. This behaviour is reflected in the CSTG model of Selector shown in Figure 11.(a)

Variable (can also be called Flip-flop) is a node which stores a level-based value on its output q (its complement q' is also available if needed). This module can copy the value of its level-based input d into q upon the arrival of a Request on the event-based input r. The result of copying is the acknowledged on the event-based output a. If the previous value of q matched the new value of d, then the node retains its value, responding only with its acknowledgement on a. This behaviour is reflected in the CSTG model of Variable shown in Figure 11.(b).

Merge (can also be called XOR from exclusive-OR) is a node which can in general have n event-based Request inputs $r_1, \ldots r_n$ and one event-based Acknowledge output a. Whenever it receives a request event on *one* of its inputs it produces an output acknowledgement. It is essential that the environment never generates two requests concurrently. This behaviour, corresponding to an XOR-gate, is defined in the CSTG model of Merge shown in Figure 11.(c).

C-element (can also be called Join) is a node which, likewise Merge, can generally have n event-based Request inputs $r_1, \ldots r_n$ and one event-based Acknowledge output a. However, unlike Merge, in order to generate an acknowledgement event on a, it requires requests to be generated on *all* its inputs. This behaviour is defined in the CSTG model of Merge shown in Figure 11.(d).

Finally, our ACS has a few arcs which lead from the same node to several other nodes which are labelled with the same signal (we call those connections **Fan-outs**). Those can be easily modelled by corresponding fragments of CSTGs, shown in Figure 11.(e), depending on the type of the signal.

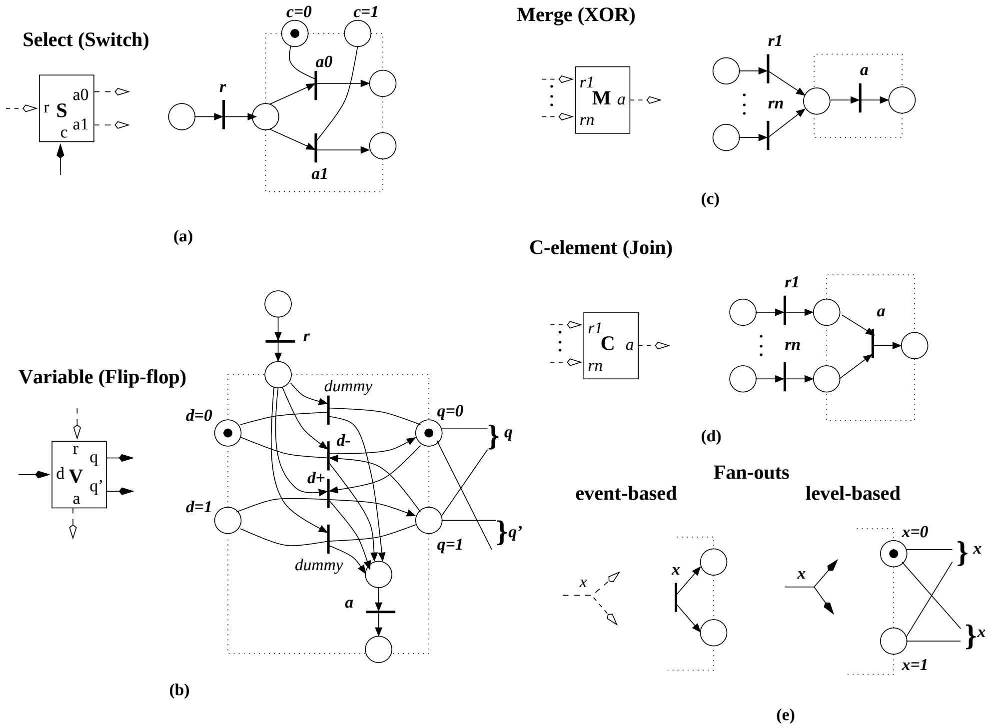


Figure 11: Nodes of the asynchronous control structure and their contextual STG models

We can now interconnect the CSTG fragments using contextual or ordinary flow relations, according to the ACS (Figure 10) and the type of the signals; level-based or event-based, respectively. We do not show such a net due to its fairly large size. This net has fourteen places with contextual arcs from them – those places correspond to seven level-based signals (variables $n, l, r, s0, s1, v0$ and $v1$.)

Due to the relatively large number of contextual arcs and a high degree of concurrency in the CSTG, its corresponding ordinary STG model produces a very large size unfolding segment. We were unable to build such a segment due to memory blow-up. However, the application of the new CN unfolding algorithm allowed us to build the segment (up to cut-off transition instances), and its size is rather modest.

In the process of building the unfolding we were verifying both of the above correctness conditions. The first condition, mutual independence and non-blocking, was checked in terms of persistence of transitions of the CN (e.g., the act of disabling a transition involved in the writing process by a transition involved in the reading process and vice versa). We found that the unfolding was free from ordinary conflicts. The only remaining source of non-persistence is due to the presence of semi-conflicts. For example, when the reader changes the value of variable r in operation $r0$, the writer may concurrently 'listen to' the value of r in its operation $w1$. Such a listening is realised via the contextual arcs between the places $r = 0$ and $r = 1$ and the transitions in its fragment corresponding to variable n (transitions labelled $n+$, $n-$ and two dummies – see Figure 11.(b)). It is thus possible to imagine that when the writer enables e.g. transition labelled $n+$ (under the state $n = 0$ and $r = 1$), the reader switches r from 1 to 0, and thus disables $n+$. But this disabling of $n+$ will always be simultaneously with the enabling of a dummy transition in the same CN fragment (for n),

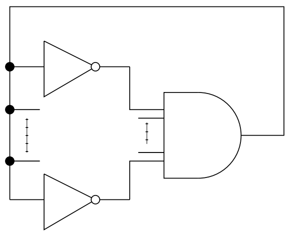


Figure 12: Illustration of the benchmark circuit.

and this the progress of the writer is not affected. We were able to check in the unfolding that **only** such types of non-persistence are possible, thus proving that the non-blocking (real-time) requirement is satisfied.

Let us now consider the “no-clash” condition. This condition, in terms of the properties of the CSTG model can be stated as follows:

- for all four pairs of the read and write signals $(w00, r00), (w01, r01), (w10, r10)$ and $(w11, r11)$, referring to the same data slot, it is not possible to reach a marking in the CSTG in which a corresponding pair of transitions is simultaneously enabled and is not in conflict.

In terms of the unfolding, we simply had to verify whether any pair of transition instances in the CSTG unfolding segment which map onto their transitions in the CSTG in the above mentioned pairs, belong to the independence relation. The result of our check proved that the mechanism satisfies its “no clash” condition.

5 Circuit verification results

The CN unfolding method described earlier was implemented within the PN/STG analysis tool “PUNT” which uses the PN unfolding approach. We also used an enhanced cutoff condition [14] which allows us to avoid possible redundancy of the McMillan's unfolding and to construct PN unfolding segment for larger PNs.

We constructed the CN unfolding segment for the CN model of the Simpson's communication mechanism. The CN model had 47 places, 56 transitions, 142 ordinary arcs and 44 contextual arcs, which in ordinary PN would correspond to a total of 230 ordinary arcs. The constructed CN unfolding segment had 1299 transition instances and showed that there are no independent instances of transitions for read and write operations on the same data slot. We failed to build the PN unfolding segment for the ordinary PN model due to the size ($>$3000 transition instances).

To illustrate the performance of the CN unfolding we chose a scalable example consisting of an AND gate and invertors in the feedback loop of the AND gate. The circuit is shown in Figure 12. The number of invertors can vary, making this benchmark easily scalable. This circuit is known to have a hazard which exhibits itself as non-persistency of the output of the AND gate. This, of course, can be detected during the construction of the CN unfolding and the construction can be terminated with an error report. Such termination can only be

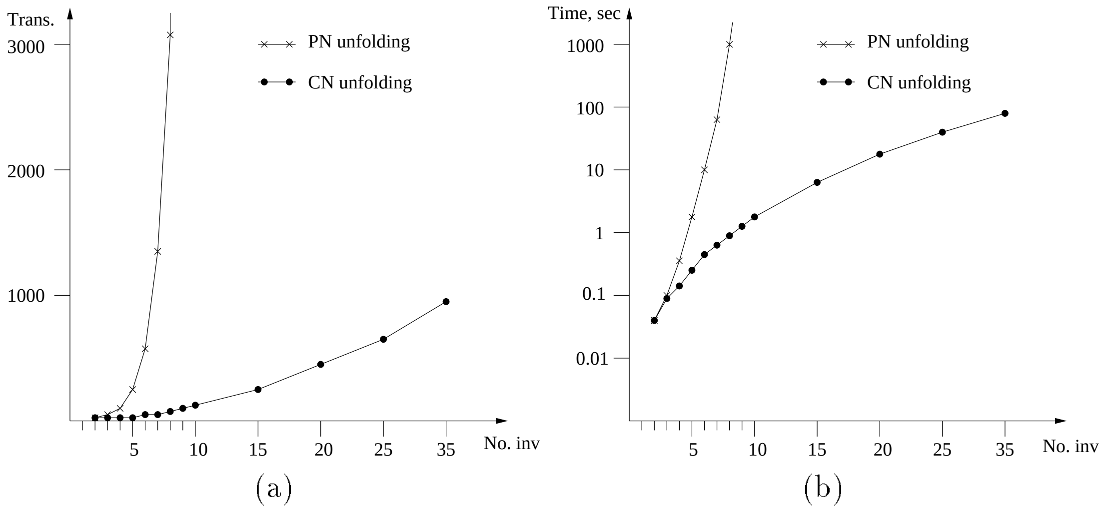


Figure 13: Experimental results: (a) Size of the segment in transition instances; (b) Time taken.

made due to hazard[6] freedom requirements to asynchronous circuits, closely related to signal non-persistency. However, examples such as the communication mechanism, described in the previous section, may require that the unfolding segment is built for the PN (or CN) even though some signals are non-persistent. Therefore, for illustrative purposes, the whole PN (or CN) unfolding segment was built.

The results are presented in Figure 13. The graph in Figure 13(a) illustrates the growth in size of the PN and CN unfolding segments with the increase of the number of invertors. The size of the unfolding includes cutoff transitions. The graph in Figure 13(b) shows the time spent on the construction of the segment. This time was spent on the unfolding of the STG rather than PN, i.e. the algorithm kept track of the binary states of the STG. The superiority of the CN unfolding for this benchmark is obvious. For comparison, the original McMillan's algorithm builds a truncated PN unfolding with 913 transition instances in 13.15sec. for the benchmark with only 4 invertors.

The results demonstrated that the size of and the time to construct the PN unfolding segment explodes double-exponentially. The first exponential dependency is caused by the exponential growth of the number of transition instances due to exponential number of interleavings. The second dependency is caused by the combinatorial complexity during the selection of a set of independent instances for places in $^\bullet t$. When a new instances of t is generated, the algorithm needs to find a set of independent instances of places in $^\bullet t$, i.e. it may try all combinations in the worst case. The number of place instances is large due to exponential number of transitions. Therefore, this selection process greatly affects the performance of the PN unfolding construction algorithm.

Notably, the original McMillan's algorithm becomes triple-exponential for this set of benchmarks. The third exponent is caused by the redundant instances of transitions which cannot be made cutoffs due to the overly strong cutoff condition (see [14] for further discus-

[6]The term "hazard" is used to denote an unspecified and thus not allowed signal change in the circuit.

sion).

In the CN unfolding the exponential explosion attributed to interleavings is avoided. Since this drastically reduces the number of place instances, the effect of the independent instances selection process is reduced to negligible. Thus the performance of the CN unfolding algorithm on this set of benchmarks is close to polynomial. We also observed that using the McMillan's cutoff condition yields a CN unfolding of the same size as the one obtained using the enhanced cutoff condition. Due to the benefits of exploring the contextual dependency the algorithm did not generate redundant instances of transitions which were causing the blow-up.

6 Conclusions

In this work we suggested a new algorithm for analysis of contextual nets. Our approach is based on previous work and constructs a CN unfolding segment which represents the set of reachable markings (or states for STGs). Our method is aimed to overcome the problems of exponential explosion of the PN unfolding which show up during the analysis of PN model of a CN, i.e. a PN with two-way arcs to represent contextual dependency. This explosion is caused by the difference in the firing semantics of two models. The PN unfolding examines explicitly all possible interleavings of transitions whose firings lead to the same marking. The CN unfolding, suggested here, preserves the step semantics of the transition firings.

CN modelling has wide use in modelling asynchronous circuits. This was the primary application area of the proposed CN unfolding segment algorithm. Our method was implemented and used on a realistic example of a four slot communication mechanism. We also illustrated the behavioural trend of our approach using a scalable benchmark. Experimental results demonstrate the power of the new CN unfolding algorithm.

7 Acknowledgements

The authors would like to thank Ian Clark and Fei Xia from the King's College, London for valuable clarification of the various details of Simpson's communication mechanism.

References

[1] N. Busi and G.M. Pinna. Non-sequential semantics for contextual P/T nets. In *Proc. of 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka*, pages 113–132, June 1996.

[2] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. November 1996. to appear.

[3] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. Technical Report TUM-19599, Institut für Informatik, Technische Universität München, August 1995.

[4] R. Janicki and M. Koutny. On equivalent execution semantics of concurrent systems. In *Lecture Notes in Computer Science, Vol. 266.* Springer-Verlag, 1987.

[5] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavksy. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Wiley and Sons, London, 1993.

[6] A. Kondratyev and A. Taubin. Verification of speed-independent circuits by STG unfoldings. In *Proceedings of the Symposium on Advanced Reserch in Asynchronous Cirsuits and Systems*, pages 64–75, Utah, USA, November 1994.

[7] K.L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of CAV'92*, 1992.

[8] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[9] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 36:545–596, 1995.

[10] T. Murata. Petri nets:properties, analysis and applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.

[11] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In *15th International Conference on Application and Theory of Petri Nets*, Zaragoza, Spain, June 1994.

[12] J. L. Peterson. *Petri Net Theory and the modeling of systems.* Prentice-Hall, Inc., 1981.

[13] W. Reisig. *Petri Nets: an Introduction.* EATCS Monographs in Computer Science. Springer-Verlag, Berlin, 1985.

[14] A. Semenov and A. Yakovlev. Event-based framework for verification of high-level models of asynchronous circuits. Technical Report 487, University of Newcastle upon Tyne, 1994.

[15] E.M Sentovich and et. al. SIS: A system for sequential circuit synthesis. Memorandum No. UCB/ERL M92/41, University of California, Berkeley, 1992.

[16] H.R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings-E*, 137(1):17–30, January 1990.

[17] H.R. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings-E*, 139(1):35–49, January 1992.

[18] V. I. Varshavsky, M. A. Kishinevsky, V. B. Marakhovsky, V. A. Peschansky, L. Y. Rosenblum, A. R. Taubin, and B. S. Tzirlin. *Self-timed Control of Concurrent Processes.* Kluwer AP, 1990. V.I.Varshavsky Ed., (Russian edition: 1986).

[19] V.Varshavksy and V.Marakhovsky. Asynchronous control device design by net model behaviour simulation. In *Proc. of 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka*, June 1996.

[20] A. Yakovlev, M. Kishinevsky, A. Kondratyev, and L. Lavagno. OR causality: modelling and hardware implementation. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lect. Not. Comp. Sci*, pages 568–587, Zaragoza, Spain, June 1994. Springer-Verlag, Berlin.

[21] A. Yakovlev and A. Petrov. Petri nets and parallel bus controller design. In *Proc. of 11th Int. Conf. on Applications and Theory of Petri Nets, Paris*, pages 244–263, June 1990.

[22] A. V. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. *Formal Methods in System Design*, 9(3):139–188, 1996.

[23] Ch. Ykman-Couvreur, B. Lin, and H. DeMan. ASSASSIN: A synthesis system for asynchronous control circuits. Reference manual, IMEC, 1995.