Synthesis of Asynchronous Controllers Using Integer Linear Programming

Josep Carmona, José-Manuel Colom, Member, IEEE, Jordi Cortadella, Member, IEEE, and Fernando García-Vallés

Abstract—A novel strategy for the logic synthesis of asynchronous control circuits is presented. It is based on the structural theory of Petri nets and integer linear programming. Techniques that are capable of checking implementability conditions, such as complete state coding, and deriving a gate netlist to implement the specified behavior are presented. These techniques can handle Petri net specifications consisting of several thousands of transitions and provide a significant speed-up compared with techniques that have previously been proposed.

Index Terms—Asynchronous circuits, logic synthesis, Petri nets, structural methods.

I. INTRODUCTION

O NE OF THE main reasons why many designers are reluctant to use asynchronous circuits is that these circuits are difficult to design and the level of maturity of existing computer-aided design (CAD) tools is still insufficient. However, in the last decade, several research groups have developed CAD tools for the synthesis of asynchronous circuits [1]–[5]. This represents the first step in bridging the gap between designers and design automation. This paper presents a novel approach for the synthesis of asynchronous control circuits.

The synthesis of asynchronous circuits from a given formalism, such as an automaton or a Petri net, can be split into two main steps [5]: 1) checking and (possibly) forcing implementability conditions and 2) deriving the Boolean equation for each signal generated by the system. Most existing synthesis tools perform steps 1) and 2) at the underlying state graph (SG) level, thus suffering from the well-known state explosion problem. These tools, although using symbolic techniques for alleviating the cost of representing the state space, can only synthesize specifications with moderate size.

In order to avoid the state explosion problem, various structural methods have been proposed [6]–[10]. The work proposed in [6] and [8] uses graph theoretic-based algorithms, whereas [9] and [10] combine partial order methods (Petri net unfoldings [11]) with either integer linear programming (ILP) [9] or Boolean satisfiability [10].

This paper is inspired by previous works [7], [12] and proposes a complete design flow for asynchronous controllers based on structural techniques. The contributions of this paper are, first, to show the effectiveness of structural methods when used with large specifications, second, to develop a novel algorithm for synthesis based on linear algebraic methods, and, third, to integrate the algorithm in a complete synthesis flow for asynchronous controllers.

This paper aims at facing the two important steps in the synthesis of asynchronous circuits: it proposes powerful methods for checking complete state coding (CSC)/unique state coding (USC) [13] and a novel method for decomposing the specification into smaller ones while preserving implementability conditions. The methods presented in this paper can be combined with structural or direct translation techniques for encoding (e.g., [14] and [15]) to provide a complete design flow for the synthesis of controllers. This flow would be able to synthesize large highly concurrent specifications that cannot be handled by state-based methods.

A. Synthesis Example: VME Bus Controller

The contributions presented in the paper are briefly summarized by synthesizing a simple VME bus controller. It consists of three entities (the bus, the device, and the controller) that interact through a bidirectional buffer according to a given protocol. Fig. 1(a) shows the protocol and the signals involved. The protocol for the read cycle is shown in the timing diagram, depicted in Fig. 1(b). The arcs denote the causality relations between events.

The protocol can be formally specified with a signal transition graph (STG). Fig. 1(c) shows the STG specifying the read cycle. The goal is, starting from this STG, to derive the implementation of each output signal of the controller under a given delay model, e.g., to derive Boolean equations for signals \( dtack \), \( lds \), and \( d \) under the speed-independent delay model [5].

The first step is to check whether the initial STG fulfills the implementability conditions for speed independence [5]. This checking is traditionally done on the state space of the system and therefore suffers from the state explosion problem. In Section IV, efficient ILP methods for performing this task are presented. For the example, this checking reveals encoding problems that must be solved to guarantee a speed-independent implementation.

A typical way of solving the encoding problem is by inserting new signals in the specification, which helps in disambiguating the conflicting states. For the example, one could resort to some of the structural methods that avoid the computation of the state
space, such as the ones presented in [14] and [15]. This could
derive the STG in the right part of Fig. 2, with several new
signals inserted to solve the encoding problem. In this case,
the encoding proposed in [14] has been applied. Intuitively,
it consists of adding a new layer of state signals (called $s_{pi}$
in the figure) that encode the value of the places in the Petri
net. In this way, the activity of the signals mimics the token
flow in the net.

Typically, structural encoding techniques are conservative
and produce a new specification with several redundant signals.
To check whether one state signal is redundant, it is enough
to check that there are no encoding conflicts after hiding the
signal. This checking can be performed by using the ILP-based

Besides hiding redundant signals, one could also modify the
specification to make it more efficient (e.g., by increasing the
concurrency of internal signals, as proposed in [15]). Other
possible transformations are also presented in [14], [16], and
[17]. Given that the transformations preserve the behavior of the
system, their legality is reduced to checking that no encoding
conflicts appear when applied.

Finally, synthesis must be performed. Here, the key point is
to tackle the synthesis of each output signal $x_i$ individually
by projecting the whole behavior only onto those signals that
are necessary for $x_i$ to be correctly implemented, i.e., the
support of $x_i$. Fig. 3 presents the projection after computing
the corresponding support for each output signal.

The last step is to synthesize a circuit for each output signal.
Provided that the support of each signal is typically small, the
projections have a small state space and, thus, conventional
state-based methods for synthesis can be used. Fig. 4 shows the
final synthesis for the signals $d$ and $dtack$.

**B. Synthesis Flow for Large Asynchronous Controllers**

The previous example illustrates how a design flow can be
devised in such a way that state-based methods can be relegated
to small specifications. A possible framework for synthesis is
the one depicted in Fig. 5. The ILP models are used to check
the legality of the transformations and to calculate the support
for each output signal. State-based methods for synthesis are
only used when the projection for the support of each output
signal has been calculated.

This paper presents ILP-based methods to check the exis-
tence of encoding conflicts and to calculate the support required
to implement each signal. Within the framework depicted in
Fig. 5, we show that the path from an implementable STG down
to the circuit is feasible and can be performed efficiently.

**II. Basic Notions**

We assume the reader to be familiar with Petri nets. A survey
is presented in [17]. Some basic concepts and notation are
reviewed as follows.
A. Petri Nets

A Petri Net (PN) is a four-tuple \( N = (P, T, F, m_0) \), where \( P \) is the set of places, \( T \) is the set of transitions, \( F \subseteq (P \times T) \cup (T \times P) \) is the flow relation, and \( m_0 \in \mathbb{N}^{|P|} \) is the initial marking. A Petri net is usually represented as a bipartite graph in which \( P \) and \( T \) are the nodes. For any two nodes \( x \) and \( y \), if \((x, y) \in F\), then there is an arc from \( x \) to \( y \).

A marking \( m \) of a PN is a \(|P|\) vector where the component \( p \) of the vector is a natural number. If \( k \) is assigned to place \( p \) by marking \( m \) [denoted by \( m(p) = k \)], we will say that \( p \) is marked with \( k \) tokens at \( m \).

Given a node \( x \in P \cup T \), the set \( x^\bullet = \{ y \mid (y, x) \in F \} \) is the pre-set of \( x \) and the set \( x^\star = \{ y \mid (x, y) \in F \} \) is the post-set of \( x \). A transition \( t \) is enabled at marking \( m \) if each place \( p \in t^\bullet \) is marked. When a transition \( t \) is enabled, it can fire by removing one token from each place \( p \in t^\bullet \) and adding one token to each place \( q \in t^\star \).

A marking \( m' \) is reachable from \( m \) if there is a sequence of firings \( t_1 t_2 \ldots t_n \) that transforms \( m \) into \( m' \), denoted by \( m[t_1 t_2 \ldots t_n] m' \). A sequence of transitions \( t_1 t_2 \ldots t_n \) is a feasible sequence if it is firable from \( m_0 \). The set of reachable markings from \( m_0 \) is denoted by \([m_0] \). By considering the set of reachable markings as the set of states of the system, and the transitions among these markings as the transitions between the states, a reachability graph can be obtained representing the underlying behavior of the PN.

A PN is \( k \)-bounded if no marking in \([m_0]\) assigns more than \( k \) tokens to any place of the net. It is safe if it is 1-bounded. For simplicity, the Petri nets considered in this paper are assumed to be safe. The extension of the results to bounded Petri nets is discussed in Section VI-C.

B. Linear Programming

A linear inequality is defined by a vector \( a \in \mathbb{R}^n \) and a constant \( b \in \mathbb{R} \). It is represented by \( a \cdot x \leq b \) and is feasible over a set \( A \) if there exists some assignment \( k \in A \) to \( x \) satisfying \( a \cdot k \leq b \).

A linear programming problem (LP) is a finite set of linear inequalities. It can be represented in matrix notation as \( A \cdot x \leq B \), where each row of \( A \) corresponds to a linear inequality and \( B \) contains the constant terms of the inequalities. Optionally, it can have a linear optimization function \( c^T \cdot x \) called the objective function. A solution of the problem is a vector that satisfies the linear inequalities. A solution is optimal if it maximizes the objective function (over the set of all solutions). An LP is feasible if it has a solution.

As a particular case, LP can also handle equality constraints, e.g., \( a \cdot x = b \), by splitting them into inequality constraints: \( a \cdot x \leq b \) and \( a \cdot x \geq b \).

LP can be solved in polynomial time [18]. The most popular algorithm for LP is the simplex [19] that performs very efficiently in practice, although it is exponential in the worst case [20].

An ILP is an LP in which the variables have integrality constraints, i.e., the solutions can only have integer values. ILP is NP-complete [21], and different methods have been proposed.
to solve it. They are usually based on solving the LP version of the problem and iteratively adding constraints to enforce the integrality around the optimal solution [22].

ILP has been successfully used to solve different problems in CAD, such as retiming [23], scheduling in high-level synthesis [24], or global routing [25].

C. ILP and Petri Nets

Given a firing sequence \( m_0(\sigma)m \) of a PN \( N \), the number of tokens for each place \( p \) in \( m \) is equal to the number of tokens of \( p \) in \( m_0 \) plus the number of tokens added by the input transitions of \( p \) appearing in \( \sigma \) minus the tokens removed by the output transitions of \( p \) appearing in \( \sigma \), i.e.,

\[
m(p) = m_0(p) + \sum_{t \in \cdot p} \#(\sigma, t) - \sum_{t \in \cdot p} \#(\sigma, t)
\]

where \#(\( \sigma, t \)) denotes the number of occurrences of transition \( t \) in the sequence \( \sigma \). The matrix \( N \in \mathbb{Z}^{|P| \times |T|} \) defined by

\[
N(p, t) = F(t, p) - F(p, t)
\]

is called the incidence matrix of \( N \), where \( F(x, y) = 1 \) indicates that \((x, y) \in F \) and \( F(x, y) = 0 \) indicates that \((x, y) \notin F \).

Let \( \sigma \) be a feasible sequence of \( N \), and \( T = \{t_1, \ldots, t_n\} \). The vector \( \vec{\sigma} = (\#(\sigma, t_1), \ldots, \#(\sigma, t_n)) \) is called the Parikh vector of \( \sigma \).

Using the previous definitions, the token conservation equations for all the places in the net can be written in matrix form as

\[
m = m_0 + N \cdot \vec{\sigma}
\]

leading to the characterization of the reachability set by means of an ILP model.

Theorem 2.1 (Marking Equation [26]): If a marking \( m \) is reachable from \( m_0 \), then there exists a sequence \( \sigma \) such that \( m_0(\sigma)m \) and the following problem has at least the solution \( X = \vec{\sigma} \), i.e.:

\[
m = m_0 + N \cdot X.
\]

This is called the marking equation.

Special attention must be paid to the previous theorem: the marking equation only provides a necessary condition for reachability. If the marking equation is infeasible, then \( m \) is not reachable from \( m_0 \). But the inverse does not hold, in general; there can be unreachable markings satisfying the marking equation. Those markings are said to be spurious [27].

Fig. 6 presents an example of a Petri net with initial marking (1,0,0,0,0). Fig. 6(b) depicts the graph containing the reachable markings and the spurious markings (shadowed). This graph is called the potential reachability graph.\(^1\)

The Parikh vector \( \vec{\sigma} = (2, 1, 0, 0, 1, 0) \) and the marking \( m = (0, 0, 1, 1, 0) \) are a solution of the marking equation. However, \( m \) cannot be reached by any feasible sequence; only sequences visiting negative markings can lead to \( m \). For example, the following sequence would satisfy the marking equation, i.e.:

\[
10000 \to t_1 \to 01100 \to t_2 \to 00011 \to t_3 \to [1[-]1010 \to t_1 \to 00110.
\]

Note that a negative marking must be visited \((1[-]1010)\) for \( 00110 \) to be reachable.

D. SGs

Asynchronous circuits can be modeled with a reachability graph, where the events represent changes in the value of the system signals.

A transition labeled as \( x_i + (x_i-) \) denotes a rising (falling) of signal \( x_i \). It switches from 0 to 1 (1 to 0), while \( x_i* \) denotes a generic transition, rising or falling. Each state of an asynchronous circuit can be encoded with a binary vector, representing the signal values on that state. The encoded reachability graph is called SG. Formally, an SG is a five-tuple \( \Lambda = (S, \Sigma, T, s_0, \lambda) \), where \( S \) is the set of states and \( s_0 \) is the initial state. \( \Sigma \) is the set of signals partitioned into three subsets: inputs, outputs, and internal. \( T \subseteq S \times \Sigma \times \{+,-\} \times S \) is the labeled transition relation. A transition is represented by an arc notation (e.g., \( s_1 \xrightarrow{1[0]} s_2 \)). Finally, \( \Lambda : S \to \mathbb{B}[^2] \) is the encoding function. We denote by \( \lambda_c(s) \) the value of signal \( x \) in state \( s \). Fig. 7 shows the SG specifying the behavior of the bus controller for the read cycle.

E. STGs and Trigger Signals

Transitions in PN can represent signal changes of an asynchronous circuit. This model is called STG [13], [28]. An STG is a triple \((N, \Sigma, \Lambda)\), where \( N = (P, \cdot, F, m_0) \) is a PN, \( \Sigma \) is the set of signals, and \( \Lambda : T \to \Sigma \times \{+,-\} \) is the labeling function.

An example STG specifying the bus controller is shown in Fig. 1. By convention, places with only one predecessor and one successor transition are not shown graphically. The reachability graph associated to an STG is an SG. The SG associated to the STG of Fig. 1 is shown in Fig. 7.
III. SYNTHESIS OF SPEED-INDEPENDENT CIRCUITS

Speed-independent (SI) circuits are the class of circuits that work correctly, regardless of the delay of their components (gates). In this section, some basic concepts on the synthesis of SI circuits are presented. We refer the reader to [5] for a deeper background on this subject.

A. Implementability Conditions

A set of properties that guarantee the existence of an SI circuit is introduced below. They are defined at the level of SG, but can easily be extended to STGs. The properties are the following: boundedness, consistency, CSC, and output persistency.

Boundedness: The set of states must be finite. Although this seems to be an obvious assumption at the level of an SG, it is not so obvious at the level of an STG, since an STG with a finite structure may have an infinite number of reachable markings.

Consistency: Consistency holds when the events $x_1^+$ and $x_1^-$ alternate in any trace of the behavior. This is formally defined as follows. A consistent SG satisfies the following conditions for each transition $s \xrightarrow{t} s'$:

1) if $e = x_1^+$, then $\lambda_i(s) = 0$ and $\lambda_i(s') = 1$;
2) if $e = x_1^-$, then $\lambda_i(s) = 1$ and $\lambda_i(s') = 0$;
3) in all other cases, $\lambda_i(s) = \lambda_i(s')$.

CSC: This property is illustrated in Fig. 7, in which there are two states with the same binary encoding (10101) that are behaviorally different. This fact implies that the system does not have enough information to determine how to react by only looking at the value of its signals.

An SG satisfies the USC condition if every state in $S$ is assigned a unique binary code. An SG satisfies the CSC condition if for every pair of states having the same binary code the sets of enabled noninput signals at each state are the same.

Both properties are sufficient to derive the Boolean equations for the synthesized circuit. However, given that only the behavior of the noninput signals must be implemented, encoding ambiguities for input signals are acceptable.

Output Persistency: This property is required to ensure that the synthesized circuit is hazard free. An event $x$ is said to disable another event $y$ if there is a transition $s \xrightarrow{t} s'$ such that $y$ is enabled in $s$ but not in $s'$. An SG is said to be output persistent if for any pair of events $x$ and $y$ such that $x$ disables $y$, both $x$ and $y$ are input events. Nonpersistency may produce a nondeterministic behavior (e.g., hazards) when the system visits a state in which an event may be temporarily enabled without actually firing.

B. Deriving Boolean Equations

The procedure to derive the next-state functions for output signals from an SG $A = (S, \Sigma, T, s_i, \lambda)$ is introduced. The procedure defines an incompletely specified function from which a minimized gate implementation can be obtained.

The positive and negative excitation regions (ERs) of a signal $x \in \Sigma$, denoted by $ER(x^+)$ and $ER(x^-)$, are the sets of states in which $x^+$ and $x^-$ are enabled, respectively, i.e.,

\[
ER(x^+) = \{ s \in S | \exists s' \xrightarrow{t} s' : x^+ \}
\]
\[
ER(x^-) = \{ s \in S | \exists s' \xrightarrow{t} s' : x^- \}.
\]

The positive and negative quiescent regions (QRs) of a signal $x \in \Sigma$, denoted by $QR(x^+)$ and $QR(x^-)$, are the sets of states in which $x$ has the same value, 1 or 0, and is stable, i.e.,

\[
QR(x^+) = \{ s \in S | \lambda_x(s) = 1 \land s \notin ER(x^-) \}
\]
\[
QR(x^-) = \{ s \in S | \lambda_x(s) = 0 \land s \notin ER(x^+) \}.
\]

Given a specification of $n$ signals, the derivation of an incompletely specified function $F^x$ for each output signal $x$ and for each $v \in \mathbb{B}^n$ can be formalized as

\[
F^x(v) = \begin{cases} 
1, & \text{if } \exists s \in ER(x^+) \cup QR(x^+) : \lambda(s) = v \\
0, & \text{if } \exists s \in ER(x^-) \cup QR(x^-) : \lambda(s) = v \\
-, & \text{if } \not\exists s \in S : \lambda(s) = v
\end{cases}.
\]

The previous definition is ambiguous when there are two states, $s_1$ and $s_2$, for which $\lambda(s_1) = \lambda(s_2) = v$, $s_1 \in ER(x^+) \cup QR(x^+)$, and $s_2 \in ER(x^-) \cup QR(x^-)$. This ambiguity is precisely what the CSC property avoids, and this is why CSC is a necessary condition for implementability.

\footnote{Output persistency is also known as output semimodularity [29].}
Fig. 8. Example abcd. (a) STG. (b) SG.

Fig. 9. Complex gate implementation for abcd example.

IV. ILP FOR STATE ENCODING VERIFICATION

This section shows how to formulate an ILP problem to verify that a given specification fulfills the USC or CSC properties.

Definition 4.1: Given a signal \( a \) with initial value \( \lambda_0(m_0) = v \), the value of \( a \) after firing a sequence \( \sigma \) is equal to \( \lambda_0(m_0) + C_a \vec{\sigma} \), where \( C_a \) is a \([|T|] \)-element integer vector such that

\[
C_a(t) = \begin{cases} 
-1, & \text{if } \Lambda(t) = a- \\
1, & \text{if } \Lambda(t) = a+ \\
0, & \text{otherwise}
\end{cases}
\]

The extension to all the signals is straightforward. The value of all signals after firing a sequence \( \vec{\sigma} \) is represented by a \(|\Sigma| \)-element vector \( x \), where

\[
x = \lambda(m_0) + C \vec{\sigma}
\]

and \( C \) is a matrix of dimension \(|\Sigma| \times |T|\), with each row representing the vector \( C_a \) of one of the signals.

In particular, if there are two sequences, \( \sigma_1 \) and \( \sigma_2 \), such that \( C\vec{\sigma}_1 = C\vec{\sigma}_2 \), then both sequences produce the same transformation on the value of the signals. Therefore, if \( m_0(\sigma_1) m_1 \) and \( m_0(\sigma_2) m_2 \), then \( m_1 \) and \( m_2 \) have the same encoding \( \lambda(m_1) = \lambda(m_2) \).

A. ILP for Checking USC

A USC conflict appears in the SG of a system when there are two feasible sequences, \( \sigma_1 \) and \( \sigma_2 \), leading to different reachable markings, \( m_1 \) and \( m_2 \), such that the value of the signals in both markings is the same. Using the marking equation (see Section II-C), a sufficient condition for USC can be obtained.

Theorem 4.1: Let \( S = ((P,T,F,m_0), \Sigma, \Lambda) \) be a consistent STG, and \( N \) be the incidence matrix of the PN. \( S \) has USC if the following ILP problem is infeasible.

ILP model for checking USC:

Reachability conditions:

\[
\begin{align*}
& m_1 = m_0 + N\vec{\sigma}_1 \\
& m_2 = m_0 + N\vec{\sigma}_2 \\
& m_1, m_2, \vec{\sigma}_1, \vec{\sigma}_2 \geq 0, \quad \vec{\sigma}_1, \vec{\sigma}_2 \in \mathbb{Z}^{|T|} \\
& C\vec{\sigma}_1 = C\vec{\sigma}_2 \\
& m_1 \neq m_2
\end{align*}
\]

Proof: A solution of the model would describe a pair of different markings, \( m_1 \) and \( m_2 \), and two firing sequences represented by the vectors \( \vec{\sigma}_1 \) and \( \vec{\sigma}_2 \), respectively. The constraint \( C\vec{\sigma}_1 = C\vec{\sigma}_2 \) enforces both markings to have the same encoding (see Definition 4.1). The model being infeasible implies that there are no two reachable markings with the same encoding.

The constraint \( m_1 \neq m_2 \) is not linear, but it can be replaced by testing if at least one place has a different amount of token in \( m_1 \) and \( m_2 \). Therefore, the initial nonlinear problem can be transformed into \(|P|\) linear problems, each one checking that \( m_1(p_i) > m_2(p_i) \). Given the symmetry of \( m_1 \) and \( m_2 \) in the model, there is no need to check that \( m_2(p_i) > m_1(p_i) \).

However, if the Petri net is safe (LP-based sufficient checks for safeness can be used [27]), any reachable marking can be encoded with a binary number of \(|P|\) digits. This allows us to express the inequality \( m_1 \neq m_2 \) as an inequality of two binary numbers, \( n_1 < n_2 \), where each \( n \) is represented by

\[
n = \sum_{i=1}^{|P|} 2^{i-1} m(p_i).
\]

This technique can be easily extended to \( k \)-bounded nets by expressing it as an inequality of two radix-\( k \) numbers [9].

Note that the marking equation provides only a necessary condition for reachability and, thus, Theorem 4.1 is a sufficient condition for USC.

B. ILP for Checking CSC

A CSC conflict between two reachable markings, \( m_1 \) and \( m_2 \), is a USC conflict that additionally fulfills the following condition: the set of noninput signals enabled in \( m_1 \) is different from the one in \( m_2 \). Note that the definition of CSC enables checking for CSC violations individually for each noninput signal. Checking CSC for a signal \( a \) can be performed in the following way: let \( a_1* \) be a transition of signal \( a \). Then, a CSC conflict exists if: 1) \( m_1 \) and \( m_2 \) are reachable markings; 2) \( m_1 \) and \( m_2 \) have the same code; 3) \( a_1* \) is enabled in \( m_1 \);
and 4) for every transition \(a_j^*\) of signal \(a\), \(a_j^*\) is not enabled in \(m_2\). Provided that the STG is safe, the enabledness of a transition \(t\) can be characterized by the sum of tokens in the preceding places. \(t\) is enabled at \(m\) if the sum of tokens of the places in \(\star t\) is equal to \(|\star t|\).

Now we can present a sufficient condition for CSC for each noninput signal \(a\).

Theorem 3: Let \(S = ((P, T, F, m_0), \Sigma, \Lambda)\) be a consistent STG and let \(a \in \Sigma\) be a noninput signal. \(S\) has CSC for \(a\) if the following problem is infeasible for each transition \(a_j^*\)

\[
\begin{align*}
(i) & \quad \text{Reachability conditions (same as in (2))} \\
(ii) & \quad C\tilde{d}_1 = C\tilde{d}_2 \\
(iii) & \quad \sum_{p \in \star t} a_j^*(p) = |a_j^*| \\
(iv) & \quad \forall a_j^* : \sum_{p \in \star t} a_j^*(p) < |a_j^*|
\end{align*}
\]

(3)

Proof: Conditions (i) and (ii) characterize the reachability of two markings with the same encoding (as in Theorem 4.1). Condition (iii) enforces a solution in which \(a_j^*\) is enabled in a safe net. Finally, condition (iv) models the fact that no transition of signal \(a\) is enabled in \(m_2\). If the model is infeasible for all transitions \(a_j^*\), then no pair of reachable markings can violate the CSC property for signal \(a\).

Note that the constraint \(m_1 \neq m_2\) is implied by (iii) and (iv) in this model and that Theorem 4.2 provides a sufficient check for the CSC property.

Let us use the VME Bus Controller [Fig. 1(c)] to illustrate how to detect that there exists a CSC conflict for signal \(d\). The assignments\(^3\) \(\tilde{d}_1 = (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)\) and \(\tilde{d}_2 = (1, 2, 1, 0, 1, 1, 0, 1, 1, 1)\) satisfy the first two constraints of problem (3). The reachable markings are

\[
m_1 = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0) \\
m_2 = (0, 0, 0, 0, 0, 0, 0, 1, 1, 0).
\]

We now analyze how the constraints (iii) and (iv) hold for \(m_1\) and \(m_2\). The former constraint is satisfied because

\[
\sum_{p \in \star t} m_1(p) = m_1(p_2) = 1 = |d| + \]

and constraint (iv) is also satisfied because

\[
\sum_{p \in \star t} m_2(p) = m_2(p_2) = 0 < 1 = |d| + \]

C. Experimental Results for USC/CSC Checking

The methods presented in this section have been implemented in moebius, a tool for the synthesis of speed-independent circuits. The experiments have been executed on a Pentium 4/2.53 GHz and 512 MB RAM.

\(^3\)The vectors of transitions \(\tilde{d}_1\) and \(\tilde{d}_2\) follow the order \((lds^+, dsr^+, ldtack^+, ldtack^-, d^-, dtack^+, dtack^-, d^+, lds^-, dsr^-, d^-)\).

Several parameterized examples have been used to compare with other existing approaches and to evaluate the impact of the size of specification on the efficiency of the method. The following examples have been used.

1) \(\text{PpWk}(m, n)\) and \(\text{PpArb}(m, n)\): examples modeling an \(m\)-stage pipeline. In these examples, each pipeline stage has little interaction with the other stages, thus showing a high degree of concurrency. In addition, \(\text{PpArb}(m, n)\) also includes arbitration, which makes some signals non-persistent. To make the example implementable, all the nonpersistent signals have been assumed to be inputs. Every benchmark in this set has CSC conflicts. These examples were obtained from [10].

2) \(\text{PpWkCsc}(m, n)\) and \(\text{PpArbCsc}(m, n)\): a modification of the previous benchmarks to fulfill the CSC property.

3) \(\text{TangramCsc}(m, n)\): examples obtained by translating a synthetic Tangram program into a netlist of handshake components (shown in Fig. 10). Each handshake component is specified as an STG and the final controller is obtained as the composition of all STGs. Each \(n\)-way component is implemented as a tree of two-way components. This is a parameterized benchmark that could represent a typical controller obtained from the direct translation of languages like Tangram [30] or Balsa [31].

4) \(\text{Art}(m, n)\): examples modeling a different way of synchronizing \(m\) pipelines. STG is depicted in Fig. 11. Every benchmark in this set has CSC conflicts.

5) \(\text{ArtCsc}(m, n)\): transformation of \(\text{Art}(m, n)\) by inserting internal signals to fulfill the CSC property [14]. The nets in this class of benchmarks are much larger compared to the corresponding benchmark without CSC. Therefore, checking CSC/USC for these benchmarks is a hard task.

The experiments for CSC/USC detection are presented in Tables I and II. The ILP-based approach never reported any...
Moreover, let us assume that \( b \) is the only trigger signal of \( a \) and that \( a^+ \) is not enabled in state 01001. To calculate the support for signal \( a \), we might try the set \( \{a, b\} \), which includes the trigger signal of \( a \). If the behavior is projected onto \( \{a, b\} \), the following trace is obtained:

\[
\begin{align*}
  b^+ &\rightarrow 01101 \quad a^+ \rightarrow b^- \rightarrow c^- \rightarrow b^+ \rightarrow a^- \rightarrow 01001 
\end{align*}
\]

Moreover, \( a^- \) is the only trigger signal of \( b \) and that \( b^- \) is not enabled in state 01001. To calculate the support for signal \( b \), we might try the set \( \{b, c\} \), which includes the trigger signal of \( b \). If the behavior is projected onto \( \{b, c\} \), the following trace is obtained:

\[
\begin{align*}
  b^+ &\rightarrow 01101 \quad a^+ \rightarrow b^- \rightarrow c^- \rightarrow b^+ \rightarrow a^- \rightarrow 01001 
\end{align*}
\]

This projection shows a CSC conflict. Hence, it is not possible to synthesize \( a \) with only \( \{a, b\} \) in the support. When considering the other signals, we can classify them into two sets:

1) balanced signals, which have the same number of rising and falling transitions between two conflicting states (e.g., signal \( c \));
2) unbalanced signals, the remaining signals (e.g., signals \( d \) and \( e \)).

Only unbalanced signals between pairs of conflicting states may help to disambiguate encoding conflicts. In this case, by adding one of them in the support (e.g., signal \( d \)), the conflict disappears, thus resulting in the following projection that has no CSC conflicts:

\[
\begin{align*}
  b^+ &\rightarrow 01101 \quad a^+ \rightarrow b^- \rightarrow d^- \rightarrow b^+ \rightarrow a^- \rightarrow 01001 
\end{align*}
\]

Section V-B explains how to use ILP models to compute supports. Section V-C presents an optimization to do this calculation more efficiently.

A. Projections and Observational Equivalence

To formalize the projection of the behavior onto a set of signals, we resort to the concept of observational equivalence [32]. Those signals not participating in the support of another signal are considered to be silent. 

Definition 5.1 (Projection): The projection of an STG \( S \) into a set of signals \( X \) is the STG \( S_X \) that results from substituting every transition of a signal not in \( X \) by the silent event \( r \).

A projected STG can be simplified by hiding the silent events while preserving observational equivalence (see the formal definition in Section V-D). The simplifications can be applied at the level of PN (e.g., as in [33]) and at the level of SG (by calculating the equivalence relation of states and taking only one representative state for each class [32]).

According to Chu [13], two conditions are required for \( \Sigma' \subseteq \Sigma \) to be sufficient to implement a signal \( a \). These conditions are captured by the following definition.

Definition 5.2 (CSC Support): Let \( S \) be an STG and let \( a \in \Sigma \) be a noninput signal. A set \( \Sigma' \subseteq \Sigma \) is a CSC support of \( a \) if

1) \( S_{\Sigma'} \) has no CSC conflicts for signal \( a \);
2) \( \text{Trig}(a) \subseteq \Sigma' \).

For example, a possible CSC support for signal \( d \) from the STG in Fig. 2 (bottom-left picture) is \( \{d, ld t a c k, sp9, sp10\} \). Fig. 3 shows the projection induced by this CSC support and also for the rest of noninput signals. In the figure, signals not appearing in the support have been removed from the net.

Next, an algorithm to calculate a CSC support for a noninput signal is presented.

B. Computing Support for Synthesis

The problem faced in this section is the following: given an STG \( S = (\langle P, T, F, m_0 \rangle, \Sigma, \Lambda) \) and a noninput signal \( a \in \Sigma \), compute a subset \( \Sigma' \subseteq \Sigma \) that suffices to implement \( f_a \).
Deciding whether $\Sigma' \subseteq \Sigma$ is a CSC support for signal $a$ can be calculated by solving the ILP model

$$\text{ILP model for checking CSC support:}$$

$$C^s\hat{\sigma}_1 = C^s\hat{\sigma}_2$$

where $C'$ is a reduced matrix obtained from $C$ containing only the rows corresponding to the signals in $\Sigma'$.

If (4) is infeasible, then $\Sigma'$ is enough to guarantee that the projection $S_{\Sigma'}$ has the CSC property for signal $a$. This will be formally proved in Section V-D (Theorem 5.1).

If (4) is feasible, $\Sigma'$ must be augmented until it becomes infeasible. But, how to augment $\Sigma'$?

Let us assume $\hat{\sigma}_1$ and $\hat{\sigma}_2$ are a solution for (4), leading to the markings $m_1$ and $m_2$, respectively. The signals $s \in \Sigma$ can be classified into two categories:

1) balanced, if $C_s\hat{\sigma}_1 = C_s\hat{\sigma}_2$;
2) unbalanced, if $C_s\hat{\sigma}_1 \neq C_s\hat{\sigma}_2$.

Balanced signals have the same value in $m_1$ and $m_2$. In particular, all signals in $\Sigma'$ are balanced. To disambiguate the conflict between $m_1$ and $m_2$, some unbalanced signal must be included in $\Sigma'$. 

### Table I

| benchmark     | $|P|$ | $|T|$ | $|\Sigma|$ | CLP | SAT | ILP |
|---------------|------|------|----------|-----|-----|-----|
| PrWk(2,9)     | 71   | 38   | 19       | 0.09| 0.04| 0.15|
| PrWk(2,12)    | 95   | 50   | 25       | 0.42| 0.37| 0.20|
| PrWk(3,6)     | 70   | 38   | 19       | 0.12| 0.04| 0.11|
| PrWk(3,9)     | 106  | 56   | 28       | 10.89| 0.10| 0.25|
| PrWk(3,12)    | 142  | 74   | 37       | 933.37| 0.04| 0.35|
| PrWkCsc(2,9)  | 72   | 38   | 19       | 3.14| 0.18| 0.16|
| PrWkCsc(2,12) | 96   | 50   | 25       | 246.26| 1.02| 0.31|
| PrWkCsc(3,6)  | 72   | 38   | 19       | 2.97| 0.04| 0.19|
| PrWkCsc(3,9)  | 108  | 56   | 28       | 2075.48| 0.31| 0.41|
| PrWkCsc(3,12) | 144  | 74   | 37       | 933.37| 0.41| 0.80|
| PrArr(2,9)    | 86   | 48   | 23       | 0.01| 0.02| 0.05|
| PrArr(2,12)   | 110  | 60   | 29       | 0.00| 0.05| 0.11|
| PrArr(3,6)    | 92   | 54   | 25       | 0.00| 0.06| 0.12|
| PrArr(3,9)    | 128  | 72   | 34       | 0.01| 0.08| 0.08|
| PrArr(3,12)   | 164  | 90   | 43       | 0.00| 0.33| 0.19|
| PrArrCsc(2,9) | 88   | 48   | 23       | 40.89| 0.61| 0.28|
| PrArrCsc(2,12)| 112  | 60   | 29       | 1021.51| 16.24| 0.45|
| PrArrCsc(3,6) | 95   | 54   | 25       | 61.30| 0.56| 0.34|
| PrArrCsc(3,9) | 131  | 72   | 34       | 138.04| 264.57| 0.92|
| PrArrCsc(3,12)| 167  | 90   | 43       | 138.04| 264.57| 1.30|
| TangramCsc(3,2)| 142 | 92 | 38 | 0.01 | 0.01 | 1.08 |
| TangramCsc(4,3)| 321 | 202 | 83 | 0.06 | 0.04 | 9.00 |
| Art(10,9)     | 216  | 198  | 99       | 0.00| 0.42| 0.06|
| Art(20,9)     | 436  | 398  | 199      | 5.00| 10.35| 0.24|
| Art(30,9)     | 656  | 598  | 299      | 38.02| 81.82| 0.56|
| Art(40,9)     | 876  | 798  | 399      | 138.04| 264.57| 0.92|
| Art(50,9)     | 1096 | 998  | 499      | 377.00| 630.41| 1.46|
| ArtCsc(10,9)  | 752  | 630  | 315      | mem | 14 m | 3 m |
| ArtCsc(20,9)  | 1532 | 1270 | 635      | time | mem | 27 m |
| ArtCsc(30,9)  | 2312 | 1910 | 955      | time | mem | 27 m |
| ArtCsc(40,9)  | 3092 | 2550 | 1275     | time | mem | 3.5 h |
| ArtCsc(50,9)  | 3872 | 3190 | 1595     | time | mem | 7 h |

### Table II

| benchmark     | $|P|$ | $|T|$ | $|\Sigma|$ | CLP | ILP |
|---------------|------|------|----------|-----|-----|
| PrWk(3,9)     | 106  | 56   | 28       | 10.53| 0.03|
| PrWk(3,12)    | 142  | 74   | 37       | 876.63| 0.05|
| PrWkCsc(3,9)  | 108  | 56   | 28       | 2002.29| 0.67|
| PrWkCsc(3,12) | 144  | 74   | 37       | time | 1.07|
| PrArr(3,9)    | 128  | 72   | 34       | 0.01| 0.06|
| PrArr(3,12)   | 164  | 90   | 43       | 0.00| 0.08|
| PrArrCsc(3,9) | 131  | 72   | 34       | time | 1.05|
| PrArrCsc(3,12)| 167  | 90   | 43       | time | 1.69|
| TangramCsc(3,2)| 142 | 92 | 38 | 0.01 | 1.07 |
| TangramCsc(4,3)| 321 | 202 | 83 | 0.06 | 6.52 |
| Art(40,9)     | 876  | 798  | 399      | 146.02| 1.26|
| Art(50,9)     | 1096 | 998  | 499      | 328.04| 1.95|
| ArtCsc(40,9)  | 3092 | 2550 | 1275     | time | 14 m |
| ArtCsc(50,9)  | 3872 | 3190 | 1575     | time | 23 m |
The algorithm for finding a CSC support is based on the previous observation. The algorithm is shown in Fig. 12. It iteratively adds unbalanced signals to $\Sigma'$ until the ILP model becomes infeasible. Initially, $\Sigma'$ contains the trigger signals of the signal under synthesis. To be precise, the trigger signals are calculated structurally (see Section II-E) by including all the predecessor events of some $a \in \alpha$ in the STG. This structural calculation gives an overestimation of $\text{Trig}(a)$, still guaranteeing the correctness of the approach.

Let us illustrate how the algorithm works with the synthesis of signal $x_5$ in the STG depicted in Fig. 15, originally presented in [10]. Initially

$$\Sigma' = \text{Trig}(x_5) \cup \{x_5\} = \{z, x_5\}$$

which makes the model (4) feasible with the traces

$$\sigma_1 = x_1^+ x_2^+ x_3^+ z^+ [x_5^+ \text{ enabled}]$$
$$\sigma_2 = y_1^+ y_2^+ y_3^+ z^+ [x_5^\prime \text{ not enabled}]$$

All $x_i$ and $y_i$ signals in $\sigma_1$ and $\sigma_2$ are unbalanced. The algorithm picks one of them arbitrarily. In this case, it adds signal $x_1$ to $\Sigma'$, thus having

$$\Sigma' = \Sigma' \cup \{x_1\} = \{z, x_1, x_5\}.$$ 

The algorithm keeps finding solutions to model (4) and adding signals until the CSC support is derived as

$$\Sigma' = \{z, x_1, x_2, x_3, x_4, x_5\}.$$ 

It can be observed that most of the new added signals are not essential for a CSC support of signal $x_5$. We next describe a strategy to calculate the CSC support more efficiently.

### C. Optimized Calculation of CSC Support

The calculation can be highly improved by adding a cost function to the ILP model (4). The cost function tries to minimize the number of unbalanced signals between the two markings with a CSC conflict. The existence of a pair of conflicting states $m_1$ and $m_2$ can be characterized by two traces $\sigma_1$ and $\sigma_2$ such that

$$m_0 \overset{\sigma_1}{\rightarrow} m_1 \overset{\sigma_2}{\rightarrow} m_2.$$ 

(5)

This new characterization requires a slight modification of the reachability conditions in the ILP model, substituting the condition $m_2 = m_0 + N \bar{\sigma}_2$ by $m_2 = m_1 + N \bar{\sigma}_2$. In fact, this new characterization does not cover all possible cases of CSC conflicts. There might be two conflicting states such that there are not mutually reachable (in the case SG is not strongly connected).4 In such a case, the new characterization of the model using (5) will be infeasible and therefore a final checking with the conventional model (4) should be performed.

With the new characterization using (5), the objective function added to the ILP model is

$$\min_{a \in \Sigma' \Sigma'} \sum \bar{\sigma}_2(a)$$

which minimizes the silent activity of the trace $\sigma_2$, i.e., the number of events not in $\Sigma'$. From the signals appearing in $\sigma_2$, we are interested in those that produce some change in the encoding function, i.e., signals $a$ such that $C_{\sigma_2} a \neq 0$, since these are the signals that can disambiguate the encoding conflict between $m_1$ and $m_2$. Intuitively, this minimization guides the search toward signals that are almost essential to solve CSC conflicts. In particular, if there is only one unbalanced signal $a$ in $\sigma_2$, then signal $a$ is essential to solve CSC, since it is the only thing that can disambiguate the conflict for the trace $\sigma_2$.

When using the objective function in the previous example (Fig. 15), and starting with the support $\Sigma' = \{z, x_5\}$ for the synthesis of $x_5$, the traces found for the ILP model are

$$\text{min} \ \sum_{a \in \Sigma' \Sigma'} \bar{\sigma}_2(a)$$

The only unbalanced signals in the trace $m_1 \overset{\sigma_2}{\rightarrow} m_2$ are $x_4$ and $y_4$. By picking one of them, e.g., $x_4$, we obtain the CSC support $\Sigma' = \{z, x_4, x_5\}$ that makes the model (4) infeasible (a similar result would have been obtained by including $y_4$). This strategy drastically reduces the inclusion of irrelevant signals for CSC support. Fig. 13 shows the projection for the synthesis of $x_5$.

4Asynchronous specifications without strongly connected behaviors rarely appear in practice. A typical cause for disconnected behavior is the existence of initialization sequences. Two possible solutions can be used to overcome this situation: 1) insert a fake resetting sequence that drives the system to its initial state, which is not always easy if persistency and consistency must be preserved, or 2) use the nonoptimized calculation of the CSC support.
D. Projection Onto the CSC Support

Given an STG and the CSC support \( \Sigma' \subseteq \Sigma \) for a noninput signal \( a \), the projection of the behavior to obtain an observationally equivalent SG without silent events is done as follows.\(^5\)

1) PN transformations preserving observational equivalence are applied to the STG (e.g., fusion of series/parallel places/ transitions [17]). Most of the asynchronous specifications are well structured, and these transformations reduce the size of the STG drastically.

2) The SG of the reduced STG is derived.

3) The remaining silent events are hidden by collapsing observational equivalent states in the SG [32].

For the implementation of signal \( a \), every signal in \( \Sigma' \setminus \{a\} \) is considered to be an input signal [13]. We next prove that the projection preserves the implementability conditions. Boundedness and consistency are trivially preserved. We will now focus on CSC and output persistency.

For the rest of the section, the following nomenclature will be used: \( S \) and \( S_{\Sigma'} \) represent SG before and after the projection onto \( \Sigma' \). The symbols \( s \) and \( s' \) denote states from \( S \) and \( S_{\Sigma'} \), respectively, and \( \lambda(s)|_{\Sigma} \) denotes the encoding of state \( s \) projected onto \( \Sigma' \). Finally, \( s_1 \overset{\tau}{\Rightarrow} s_2 \) denotes a trace \( s_1 \overset{\tau}{\Rightarrow} s_2 \) in which \( \tau \) represents silent events (not in \( \Sigma' \)), and \( s \approx s' \) denotes the observational equivalence between one state of \( S \) and another state of \( S_{\Sigma'} \).

The following lemma establishes a partial encoding equality between observationally equivalent states.

**Lemma 5.1:** If \( s \approx s' \), then \( \lambda(s)|_{\Sigma'} = \lambda(s') \).

**Proof:** By contradiction, assume there exists a signal \( x \in \Sigma \) such that \( \lambda_x(s) = 0 \) and \( \lambda_x(s') = 1 \). Since \( S \) and \( S_{\Sigma'} \) are consistent, the first firable event of \( x \) from \( s \) would be \( x^+ \), whereas the first firable event of \( x \) from \( s' \) would be \( x^- \), which contradicts the fact that \( s \approx s' \).\(\square\)

The CSC support calculated by the algorithm in Fig. 12 guarantees the infeasibility of the ILP model (4) on the original STG. The next theorem proves that the projected STG has the CSC property.

**Theorem 5.1:** Let \( S \) be an STG and \( \Sigma' \) a support found by algorithm CSC support for the noninput signal \( a \). Then \( S_{\Sigma'} \) has CSC for signal \( a \).

**Proof:** By contradiction, let us assume that \( s_1' \) and \( s_2' \) are states from \( S_{\Sigma'} \) that have a CSC conflict, i.e., \( \lambda(s_1') = \lambda(s_2') \), but a signal transition \( a^+ \) is enabled only in \( s_1' \), Given that \( S \) is observationally equivalent to \( S_{\Sigma'} \), there exist states \( s_1 \) and \( s_2 \) in \( S \) such that \( s_1 \approx s_1' \) and \( s_2 \approx s_2' \). Moreover, Lemma 5.1 guarantees that

\[
\lambda|_{\Sigma'}(s_1) = \lambda(s_1') = \lambda(s_2') = \lambda|_{\Sigma'}(s_2).
\]

Since \( s_1 \approx s_1' \) and \( a^+ \) is enabled in \( s_1' \), then a trace \( s_1 \overset{a^+}{\Rightarrow} \) is also firable. In particular, there is a sequence

\[
s_1 \overset{\tau^n}{\Rightarrow} s_1 \overset{a^+}{\Rightarrow} \]

for some \( n \) in which \( a^+ \) is enabled in the state \( s_1' \). Moreover, since only \( \tau^n \) separates \( s_1 \) from \( s_1' \), we also have that \( \lambda|_{\Sigma'}(s_1) = \lambda|_{\Sigma'}(s_1') \). Finally, \( s_2 \approx s_2' \) guarantees that \( a^+ \) is not enabled in \( s_2 \). Under the previous conditions, the ILP model (4) of \( S \) would be feasible, with \( s_1' \) and \( s_2' \) having a CSC conflict. This would contradict the fact that \( \Sigma' \) has been calculated by the algorithm CSC support.\(\square\)

**Theorem 5.2:** Let \( S \) be an output-persistent STG and \( \Sigma' \) be a CSC support of signal \( a \). Then, \( S_{\Sigma'} \) is output persistent.

**Proof:** Given that \( a \) is the only output signal, the only possible violation of output persistency would involve the disabling of an event of \( a \). Assume that an event \( a^+ \) is disabled by some other event \( x^+ \) in \( S_{\Sigma'} \). Since \( S \) is output persistent, the disabling of \( a^+ \) in \( S_{\Sigma'} \) is created when hiding some event not in \( \Sigma' \). This situation is depicted in Fig. 14, in which a fragment of SG is shown. Fig. 14(a) shows SG before the projection and Fig. 14(b) the same graph after hiding the event \( \tau \). Since \( a \) is output persistent in \( S \), \( a^+ \) is not enabled in \( s_1 \) [from Fig. 14(a)]. This implies that \( \tau \) is a trigger event of \( a \). However, \( \tau \) cannot be hidden since \( \text{Trig}(a) \subseteq \Sigma' \) and the signals in \( \Sigma' \) are not hidden.\(\square\)

As it was expected, the presence of all trigger signals in the projection for the synthesis of \( a \) preserves output persistency.

E. Example: Synthesis of PrARB\(\text{C}(2,3)\)

We illustrate the method with the synthesis of PrARB\(\text{C}(2,3)\), shown in Fig. 15 and described in Section IV-C. We focus on the synthesis of the signals \( x_1 \), \( x_2 \), and \( x_3 \) (the calculation of the support for \( x_6 \) was shown in Sections V-B and V-C). In [35], the reader can find a complete description of the example.

The results of applying CSC support computation, projection, and speed-independent synthesis for signals \( x_1 \), \( x_2 \), and \( x_3 \) are shown in Fig. 15. For each projection, synthesis is performed by using the tool petrify.

F. Experimental Results

Experiments have been performed on some of the benchmarks described in Section IV. The results are reported in Table III. The column \(|S|\) indicates the number of states of the SGs. Given the complexity of the benchmark, it was not possible to calculate the number of states for ARTCSC.

The table also reports the number of output signals of the circuit and the size of the SGs (in states and signals) after the projection onto the support. It is important to observe that the SGs generated after the projections are manageable by conventional state-based synthesis tools.

The column “Literals” reports the number of literals of the netlist in factored form. The results are compared with the
circuits obtained by petrify. The CPU time includes the time for calculating the support and the time for synthesis.

Table III shows that the quality of the circuits obtained by the ILP-based technique is comparable to that of the circuits obtained by petrify. Moreover, it is clear that the ILP-based approach can deal with much larger specifications.

The TANGRAMCSC(4,3) example, shown in Fig. 11, illustrates the suitability of our approach for the synthesis of specifications generated from an HDL. According to [30], the cost of implementing the handshake components is as that shown in Table IV.\(^6\)

The circuit in Fig. 11 has three sequencers, eight parallelizers, and nine mixers: 319 literals. This would be the cost obtained by a syntax-directed translation. The cost obtained by logic synthesis methods is significantly smaller (247 literals).

\(^6\)A C-element is assumed to cost five literals: \(c = ab + c(a + b)\).


**TABLE IV**

<table>
<thead>
<tr>
<th>Component</th>
<th>C-elements</th>
<th>2-input gates</th>
<th>literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way sequencer (L)</td>
<td>1</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>2-way parallelizer (P)</td>
<td>3</td>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td>2-way mixer (M)</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

**TABLE V**

<table>
<thead>
<tr>
<th>benchmark</th>
<th>number of nodes</th>
<th>unfolding</th>
<th>net</th>
<th>unfolding net</th>
</tr>
</thead>
<tbody>
<tr>
<td>TANGRAMCSC(4,3)</td>
<td>523</td>
<td>555</td>
<td>1,06</td>
<td></td>
</tr>
<tr>
<td>PpWxCSC(3,12)</td>
<td>216</td>
<td>815</td>
<td>3.77</td>
<td></td>
</tr>
<tr>
<td>PeArbCSC(3,12)</td>
<td>257</td>
<td>1307</td>
<td>5.09</td>
<td></td>
</tr>
<tr>
<td>ArtCSC(10,9)</td>
<td>1382</td>
<td>7661</td>
<td>5.54</td>
<td></td>
</tr>
<tr>
<td>ArtCSC(20,9)</td>
<td>2802</td>
<td>29541</td>
<td>10.54</td>
<td></td>
</tr>
<tr>
<td>ArtCSC(30,9)</td>
<td>4222</td>
<td>65021</td>
<td>15.54</td>
<td></td>
</tr>
<tr>
<td>ArtCSC(40,9)</td>
<td>5642</td>
<td>115901</td>
<td>20.54</td>
<td></td>
</tr>
<tr>
<td>ArtCSC(50,9)</td>
<td>7062</td>
<td>180381</td>
<td>25.54</td>
<td></td>
</tr>
</tbody>
</table>

**VI. RELATED WORK AND DISCUSSION**

A. **CSC Checking**

The closest work to the techniques presented in this paper has been published in [9], [10], and [36].

In all cases, the synthesis problem is reduced to the problem of checking the CSC property. The underlying model to solve the problem is integer programming (in [9] or this paper) or SAT [10]. All these models require algorithms with exponential complexity on the size of the model.

Conceptually, there are two main differences between the work in [9] and [10] and the one presented in this paper.

1) The methods in [9] and [10] are exact, whereas the ones in this paper are approximate due to the potential existence of spurious markings.

2) The methods in [9] and [10] work with net unfoldings, whereas the ones in this paper work with the net itself.

The second aspect is crucial when efficiency is an issue. The computational complexities of ILP and SAT depend significantly on the size of the model. In the worst case, the unfolding of a net can be exponential in the size of the net [11]. In practice, unfoldings rarely suffer an explosion; however, they often generate a graph much larger than the original net, as shown in the examples presented in Table V. Some specific examples have been selected to illustrate the relationship between the size of the net and the unfolding. Several instances of ArtCSC have been selected to show, in this particular case, the quadratic growth of the unfolding.

We also observe that TANGRAMCSC has an unfolding with size similar to the net. This is the reason why SAT-based methods [10] can perform better than ILP for this case when checking CSC.

B. **Impact of Spurious Markings**

One can conclude that working with the original net is clearly more efficient. However, what is the price one has to pay for being approximate? Being more specific, what is the impact of spurious markings on the characterization of CSC conflicts?

The theory says that the marking equation characterizes the reachability set exactly for certain classes of Petri nets such as live state machines (live PN s where every transition has one input and one output place), or live marked graphs (live PN s where every place has one input and one output transition), or acyclic PN s (nets without cycles). Other subclasses enjoy weaker but still useful properties, e.g., in live, bounded, and reversible (extended) free choice nets (PNs where all conflicts are free because each maximal set of conflicting transitions share the same set of input places), the reachable markings are the solutions of the marking equation that mark every trap [37]. For the rest of cases, we can palliate the problem of spurious solutions using marking equations with additional linear constraints coming from trap invariants [38] or by the addition of some special places named cutting implicit places [27] to the original STG that remove spurious solutions from the original marking equation. Unfortunately, in the general case, it is not possible to remove all spurious solutions from the marking equation [27].

Still, the existence of spurious markings would be harmful only if they would collide with other markings with the same encoding. Such situation would only have a negative impact in the case that spurious conflicts would be detected in the original net.

For the calculation of the support of each signal, the existence of spurious markings would not affect the quality of the result. In the worst case, it might imply some slight overestimation of the support that would be swept away when doing synthesis with a state-based algorithm (e.g., petrify).

For the above reason, it is believed that the approximate techniques presented in this paper will rarely have any impact on the synthesis results but will always contribute to reduce the computational complexity significantly (the results of Table III support this claim). Still, if the approximate techniques would fail, the user could always resort to some of the exact techniques at the expense of a higher CPU time.

C. **Applicability to Bounded Petri Nets**

The ILP-based methods proposed in this paper can be applied to safe Petri nets. There is only one reason why this constraint is required: conditions (iii) and (iv) of model (3).

In practice, most of the specifications of asynchronous controllers are safe by construction. Still, the ILP-based methods presented in this paper can be easily extended to structurally bounded Petri nets with weights on the arcs. Roughly speaking, a Petri net is structurally bounded when the bounds on the places can be calculated by using LP models. This calculation has polynomial complexity. We refer the interested reader to [39], where a method to model all enabling conditions with linear inequalities is described. Briefly, any -bounded place is substituted by a set of safe places that mimic the behavior of the original place. This technique also requires the duplication of some transitions and makes the ILP model more complex.

The methods based on unfoldings [10] impose the constraints of the algorithm to calculate the unfolding. The most popular algorithms work on safe Petri nets [40]. There are more
sophisticated and complex strategies that can handle bounded Petri nets [41]. To the best of our knowledge, no algorithm has been proposed to calculate the unfolding of a Petri net with weighted arcs.

D. Calculation of CSC Support

The seminal work by Chu [13] introduced the idea of net contraction (projection) of an STG onto the support of a signal. Later on, Puri and Gu [42] presented a method to calculate the support for each signal based on the satisfiability of a Boolean formula. However, it was an algorithm that required the explicit enumeration of the reachable states of the system.

Recently, Yoneda et al. [43] presented an alternative method that avoids the enumeration of the state space and computes the support from an initial partition of the STG. This approach resembles our technique with the difference that a guided simulation is used to find out a trace connecting states with CSC violations. In our approach, the trace is found simultaneously with the CSC checking using an ILP model.

The method presented in [43] cannot check the CSC property. This prevents the method from removing redundant internal signals in the specification that might be generated from conservative encoding techniques. Another aspect to be taken into account is that the calculation of support must be done by projecting STG onto the potential subset of signals each time the support must be augmented. This strategy may be faster than ILP for examples with moderate size. However, it may suffer from an excess of projections when the specification is large.

Finally, Khomenko et al. have extended the work in [10] to derive a method for computing support sets [36]. The method uses incremental SAT techniques, a variation of the SAT problem, where also partial satisfying assignments can be obtained. The approach is opposite to ours: for computing the support from an initial partition of the STG, this approach resembles our technique with the difference that a guided simulation is used to find out a trace connecting states with CSC violations. In our approach, the trace is found simultaneously with the CSC checking using an ILP model.

As a conclusion, all methods discussed in this section have similar goals while using different approaches. Their application is not mutually exclusive and, possibly, the combination of different features from each approach could lead to a hybrid scheme in which different algorithms could cooperate in the same synthesis framework. More investigation into this direction is required in the future.

VII. CONCLUSION

This paper has shown how to use linear algebraic techniques to verify the implementability of asynchronous specifications and synthesize asynchronous controllers. The experiments show that good quality results can be obtained efficiently by using approximate techniques that avoid the explicit enumeration of state space.

These techniques open the possibility of building a general framework for the synthesis of asynchronous controllers. Such a framework can start by an HDL specification, from which a Petri net can be derived by syntax-directed translation.

ACKNOWLEDGMENT

The authors would like to thank V. Khomenko and T. Yoneda for providing examples, tools, and discussion on different approaches for the synthesis of large asynchronous circuits. They would also like to thank the reviewers for their constructive comments and suggestions to improve the manuscript.

REFERENCES
