ILP Models for the Synthesis of Asynchronous Control Circuits

Josep Carmona
Computer Architechture Department
Universitat Politècnica de Catalunya
Avda. Canal Olimpic, s/n
Barcelona (08860), Spain

Jordi Cortadella
Software Department
Universitat Politècnica de Catalunya
Jordi Girona 1-3
Barcelona (08034), Spain

ABSTRACT

A new technique for the logic synthesis of asynchronous circuits is presented. It is based on the structural theory of Petri nets and integer linear programming. The technique is capable of checking implementability conditions, such as, complete state coding, and deriving a gate netlist to implement the specified behavior. This technique can synthesize specifications with few thousands of transitions in the Petri net, providing a speed-up of several orders of magnitude with regard to other existing techniques.

1. INTRODUCTION

It is well known the leadership of the synchronous paradigm in the hardware design industry. However, factors like performance, power efficiency, modularity and clock skew among others invite both theoreticians and designers to look into the asynchronous world in order to improve the quality of their designs [6]. Nowadays, the major drawback for going asynchronous is that they are difficult to design, and opposite to the synchronous counterpart, the level of maturity of existing CAD tools for asynchronous design is still insufficient. However, in the last ten years several research groups around the world have developed CAD tools for the synthesis and verification of asynchronous circuits which represents the first step in filing the gap [1, 17, 20, 10, 5]. This paper presents a novel approach for the synthesis of asynchronous circuits.

The synthesis of asynchronous circuits from a given formalism like an automaton or a Petri net can be split into two steps [5]: (i) checking and (possibly) forcing implementability conditions and (ii) deriving the next-state function for each signal generated by the system. Existing CAD tools for synthesis perform steps (i) and (ii) at the underlying state graph 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.

ICCAD'03, November 11-13, 2003, San Jose, California, USA.

In order to avoid the state explosion problem, structural methods for steps (i) and (ii) have been proposed in the literature [19, 16, 12]. The work proposed in [19, 16] uses graph theoretic-based algorithms while [12] use both graph theoretic (by using causal order partial semantics of the Petri net, called unfoldings [13]) and linear algebraic techniques. A new and promising direction is presented in [11], where the *encoding problem* [4] is faced by adopting the Boolean Satisfiability (SAT) approach.

To the best of our knowledge, the work in [12] is the first one that uses linear algebraic techniques to approach the encoding problem. Although completely characterizing the encoding problem, the techniques presented in [12, 11] need to compute the unfolding of the net, whose size can be exponential on the size of the net. In addition, the checking of the Complete State Coding (CSC) in [12] needs to solve non-linear integer programming problems, which are $\mathcal{NP}$*-hard*. The work presented in this paper proposes linear algebraic methods for deriving sufficient conditions for the encoding problem and novel methods for performing the synthesis in a modular fashion. In our approach, the computation of the unfolding is not performed, at the expense of checking only sufficient conditions for synthesis. However, the experimental results indicate that this approach is highly accurate and provides a speed-up of several orders of magnitude with regard to [12, 11].

Moreover, a novel algorithm for computing the set of signals needed to synthesize a given signal is presented, which also uses integer programming techniques. This allows to project the behavior into that set of signals and perform the synthesis on the projection. Experimental results are presented, illustrating the improvement with respect to previous methods for synthesis.

In summary, the work presented in this paper aims at facing the two important steps (i) and (ii) in the synthesis of asynchronous circuits: it proposes powerful methods for checking CSC/USC and a novel method for decomposing the specification into smaller ones *while preserving the implementability conditions.* The methods presented here in combination with the ones presented in [3] provide a complete design flow for the synthesis of controllers. The overall design flow can synthesize large, highly concurrent specifications that cannot be handled by state-based methods. Moreover, the quality of the circuits is comparable to the one obtained by state-based methods. The results show that the approach is specially suited for the synthesis of well-structured specifications, that can be generated automatically from Hardware Description Languages (HDL).

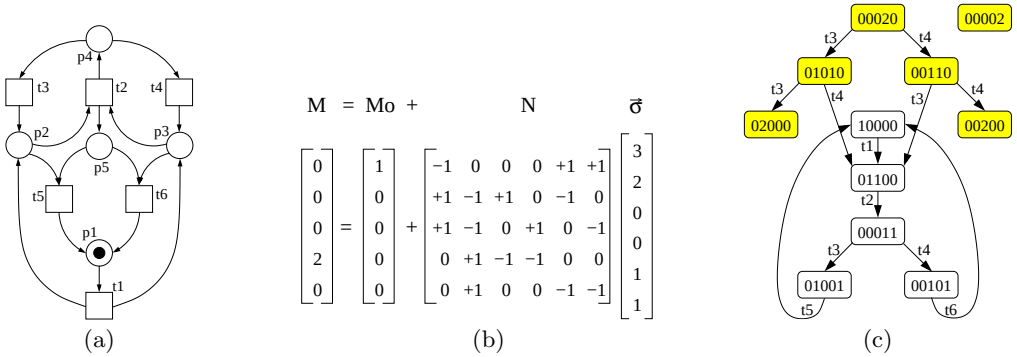

Figure 1: (a) Petri net, (b) Spurious solution $M = (00020)^T$, (c) Potential reachability graph.

The paper is organized as follows. Section 2 presents definitions and basic background needed in this paper. Section 3 describes the methods to check the correct encoding. Section 4 describes the framework for synthesis of asynchronous circuits. Sections 5 and 6 present a case study and give experimental results, respectively.

2. BASIC DEFINITIONS

The theory presented in this paper holds for the class of consistent Signal Transition Graphs. The necessary definitions to support the theory are presented next.

2.1 Petri Nets

A *Petri Net* (PN) is a 4-tuple $N = \langle \mathcal{P}, \mathcal{T}, \mathcal{F}, M_0 \rangle$, where $\mathcal{P}$ is the set of places, $\mathcal{T}$ is the set of transitions, $\mathcal{F} : (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}) \rightarrow \{0, 1\}$ is the flow relation, and M_0 is the initial marking. A marking of a PN is an assignment of a non-negative integer to each place. If k is assigned to place p by marking M, denoted $M(p) = k$, we will say that p is marked with k tokens. Given a node $x \in \mathcal{P} \cup \mathcal{T}$, its post-set and pre-set are denoted by ${}^\bullet x$ and $x^\bullet$ respectively. An example of Petri net is shown in Figure 1(a).

A transition t is *enabled* in a marking M when all places in ${}^\bullet t$ are marked. When a transition t is enabled, it can *fire* by removing a token from each place in ${}^\bullet t$ and putting a token to each place in $t^\bullet$. A marking M' is *reachable* from M if there is a sequence of firings $t_1 t_2 \dots t_n$ that transforms M into M', denoted by $M[t_1 t_2 \dots t_n \rangle M'$. A sequence of transitions $t_1 t_2 \dots t_n$ is a *feasible sequence* if it is firable from M_0. The set of reachable markings from M_0 is denoted by $[M_0\rangle$. A PN is *k-bounded* if no marking in $[M_0\rangle$ assigns more than k tokens to any place. The *language* of N, denoted by $L(N)$, is the set of feasible sequences of N.

2.2 Linear Algebra

A *linear programming problem* (LP) is a system $A \cdot X \leq B$ of linear inequalities called constraints, and optionally a linear function $C^T \cdot X$ called the *objective function* [14]. A solution of the problem is a vector of rational numbers that satisfies the constraints. A solution is *optimal* if it maximizes the value of the objective function over the set of all solutions. An *integer linear programming problem* (ILP) is an LP where every element in the solution is in $\mathcal{Z}$. An (integer) linear problem is *feasible* if it has a solution.

Given an occurrence sequence $M_0 \xrightarrow{\sigma} M$ of a system 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 σ minus the tokens removed by the output transitions of p appearing in σ. If we denote $\#(\sigma, t)$ the number of times that a transition t occurs in σ, we can write the *token conservation* equation for p as:

$$M(p) = M_0(p) + \sum_{t \in {}^\bullet p} \#(\sigma, t)\mathcal{F}(t, p) - \sum_{t \in p^\bullet} \#(\sigma, t)\mathcal{F}(p, t)$$

The token conservation equations for all the places in the net can be written in the following matrix form (see Figure 1):

$$M = M_0 + \mathbf{N} \cdot \vec{\sigma}$$

where $\vec{\sigma} = (\#(\sigma, t_1), ..., \#(\sigma, t_n))$ is called the *Parikh vector* of σ, and $\mathbf{N} \in \mathcal{Z}^{P \times T}$ is the *incidence matrix* of N defined by:

$$\mathbf{N}(p, t) = \mathcal{F}(p, t) - \mathcal{F}(t, p)$$

If a marking M is reachable from M_0, then there exists a sequence σ such that $M_0 \xrightarrow{\sigma} M$, and the following problem has at least the solution $X = \vec{\sigma}$

$$M = M_0 + \mathbf{N} \cdot X \tag{1}$$

The equation $M = M_0 + \mathbf{N} \cdot X$ is called the *marking equation*. If the marking equation is infeasible, then M is not reachable from M_0. The inverse does not hold in general: there are markings satisfying the marking equation which are not reachable. Those markings are said to be *spurious* [18]. Figure 1(a)-(c) presents an example of spurious marking: the Parikh vector $\vec{\sigma} = (3, 2, 0, 0, 1, 1)$ and the marking $M = (00020)$ are a solution to the marking equation of the Petri net of Figure 1(a), as is shown in Figure 1(b). However, M can not be reachable by any feasible sequence: only traces visiting *negative* markings, i.e. markings where some place is negatively marked, can lead to M. Figure 1(c) depicts the graph containing the reachable markings and the spurious markings (shadowed). This graph is called the *potential reachability graph*. The initial marking is represented by the state (10000).

Therefore the marking equation provides only a necessary condition for reachability of a marking. However, if the net is *Free-choice* [8], *live*, bounded and *reversible*, the marking equation, together with the set of *traps* of the system, completely characterizes reachability [7].

2.3 Signal Transition Graphs

A Signal Transition Graph (STG) [4] is a triple $\langle N, \Sigma, \Lambda \rangle$, where N is a Petri net, Σ is a set of signals, partitioned into input signals (Σ_I), output signals (Σ_O), and internal signals (Σ_{INT}), and Λ is the labeling function $\Lambda : \mathcal{T} \rightarrow (\Sigma \times \{+, -\}) \cup \{\varepsilon\}$, where all transitions not labeled with

the silent event (ε) are interpreted as signal changes. Rising and falling transitions of a signal $a \in \Sigma$ are denoted by $a+$ and $a-$, respectively, while $a*$ denotes a generic rising or falling transition[1]. A signal is said to be *enabled* in a marking M if there is a transition of the signal enabled in M. Function Λ can be defined for transition sequences: $\Lambda(t_1 t_2 \dots t_n) = \Lambda(t_1)\Lambda(t_2)\dots\Lambda(t_n)$. We define $L(S) = \{\sigma | t_1 t_2 \dots t_n \in L(N) \wedge \Lambda(t_1 t_2 \dots t_n) = \sigma\}$.

An example of STG is shown in Figure 3(a). For simplicity, those places that only have one predecessor and one successor transition are not depicted. In that case, the tokens are held on the corresponding arcs.

Let $R \subseteq [M_0\rangle$ be the set of markings where transition t_i is enabled. Transition t_j *triggers* transition t_i ($t_j \rightarrow t_i$) if there exists a reachable marking M such that $M[t_j\rangle M'$, $M \notin R$ and $M' \in R$. Given two signals a and b, we say that b triggers a if there are two transitions a_i* and b_j* such that $b_j* \rightarrow a_i*$. We will call $Trig(a)$ the set of trigger signals of a. The *projection* of an STG S onto a set of signals Σ' is another STG, denoted by $S_{\Sigma'}$, whose behavior is observationally equivalent to S after hiding the signals in $\Sigma \setminus \Sigma'$ [2]. The STG in Figure 3(d) is a projection of the one in Figure 3(c) onto the set of signals $\{d, ldtack, csc\}$.

Finally, one of the basics ingredients for Sections 3.1, 3.2 and 4 is introduced: the concept of *complementary sequence*. Given a set of signals $\Sigma = \{a_1, \dots, a_n\}$, and a transition sequence σ, we define $\texttt{code\_change}(\Sigma,\sigma)$ as a vector of length $|\Sigma|$ where component i corresponds to the result of the equation $\sum_{a_i+} \#(\sigma, a_i+) - \sum_{a_i-} \#(\sigma, a_i-)$. When the i-th component of the vector is 0, we say that the signal a_i is *balanced*. A balanced signal means that the number of positive and negative transitions of the signal in the sequence is the same. A *complementary sequence* σ is a feasible transition sequence such that $\texttt{code\_change}(\Sigma,\sigma) = \mathbf{0}$ (i.e. all the signals are balanced).

2.4 Encoding

Each marking of an STG is encoded with a *binary vector* of signal values by means of a labeling function $\lambda : [M_0\rangle \rightarrow \{0,1\}^{|A|}$. All markings must be *consistently encoded* by λ, i.e. no marking M can have an enabled rising (falling) transition $a+$ ($a-$) if $\lambda(M)_a = 1$ ($\lambda(M)_a = 0$). Intuitively consistency ensures that the rising and falling transitions of the same signals alternate in any feasible sequence of transitions.

Figure 3(b) depicts the set of reachable states derived from the STG in Figure 3(a), with the corresponding encoding. The derived graph is called *state graph* (SG).

An STG is said to satisfy the *complete state coding* (CSC) property if, when the same binary code is assigned to two different markings, the set of internal and output signals enabled at each marking is the same. The STG in Figure 3(a) does not satisfy the CSC property, since there are two different markings with the code 10101, and two output transitions, $d+$ and $lds-$, only enabled in one of them.

A more restrictive property, called *unique state coding* (USC), holds if all reachable markings are assigned a unique binary code, i.e., $\forall M_1, M_2 \in [M_0\rangle : M_1 \neq M_2 \Rightarrow \lambda(M_1) \neq \lambda(M_2)$.

The CSC property is a necessary condition for the correct implementation of an STG specification (see Section 2.5 below). When the CSC condition holds, the events that the circuit must produce at each reachable state are uniquely determined by the binary code of the state itself.

[1]Along this paper, we will often use the label of a transition to denote the transition itself.

2.5 Synthesis of speed-independent circuits

Here, we briefly sketch how a circuit can be derived from an STG. This theory is valid for the class of *speed-independent* circuits, which are correct when assuming that all components of the circuit can have any delay [15].

For an STG S to be correctly implemented by an speed-independent circuit, four conditions must hold [5]:

1. the set of reachable states of S must be finite (*boundedness*),
2. function λ must consistently encode the reachable markings of S (*consistency*),
3. S must fulfill the CSC property,
4. for any pair of signals x and y such that x disables y it implies that x and y are input signals (*output persistency*).

If we call $a_1, \dots, a_n$ the signals of the circuit, each non-input signal x can be implemented by a gate that realizes a logic function f_x. The logic function is defined for each binary vector $v \in \{0,1\}^n$ as follows:

$$f_x(v) = \begin{cases} 1 & \text{if } \exists M : \lambda(M) = v \ \wedge \ (\text{some } x+ \text{ enabled in } M \\ & \quad \vee \ (\lambda(M)_x = 1 \ \wedge \ \text{no } x- \text{ enabled in } M)) \\ 0 & \text{if } \exists M : \lambda(M) = v \ \wedge \ (\text{some } x- \text{ enabled in } M \\ & \quad \vee \ (\lambda(M)_x = 0 \ \wedge \ \text{no } x+ \text{ enabled in } M)) \\ - & \text{if } \nexists M : \lambda(M) = v \end{cases}$$

In case the CSC property does not hold, the previous definition is ambiguous, since a binary vector could be found for which there are two different markings that would make f_x equal to 0 and 1 simultaneously.

3. ILP FOR CHECKING STATE ENCODING

In this section it is shown how to formulate an ILP problem in order to verify if a given specification is correctly encoded. The techniques presented provide a significant speed-up to the ones presented recently ([12, 11]).

3.1 ILP for USC Checking

A USC conflict appears in the SG of a system when there are two reachable markings M_1, M_2 such that M_2 is reachable from M_1 by firing a complementary sequence z, i.e. $M_0 \xrightarrow{x} M_1 \xrightarrow{z} M_2$. Using the marking equation, a sufficient condition for USC can be obtained:

THEOREM 3.1. *Let $S = \langle\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, M_0\rangle, \Sigma, \Lambda\rangle$ be a consistent STG. S has USC if the following ILP problem is infeasible:*

ILP model for USC checking:

Reachability conditions:
$$M_1 = M_0 + \mathbf{N}x$$
$$M_2 = M_1 + \mathbf{N}z$$
$$M_1, M_2, x, z \geq 0, x, z \in \mathcal{Z}^{|T|}$$

$$\texttt{code\_change}(\Sigma, z) = \mathbf{0}$$
$$M_1 \neq M_2 \qquad (2)$$

PROOF. If no solution exists for (2) then no possible complementary sequence exists between any pair of reachable markings $M_1, M_2 \in [M_0\rangle$ such that $M_1 \neq M2$. □

In fact the constraint $M_1 \neq M_2$ is not linear, but it can be replaced by testing instead if at least one place has different amount of tokens in M_1 and M_2. Therefore the initial non-linear problem can be transformed to $|P|$ linear problems. However, if the system is k-bounded, any reachable marking can be encoded with a $|P|$ k-ary vector. This allows us to express the inequality between M_1 and M_2 as the inequality of two k-ary numbers [12].

3.2 ILP for CSC Checking

A CSC conflict exists when there exist two reachable markings M_1, M_2 such that M_2 is reachable from M_1 though a complementary sequence z and the set of non-input signals enabled in M_1 is different from the one in M_2. Note that the definition of CSC allows to check individually for each non-input signal a whether a has a CSC violation. When every non-input signal fulfills the CSC conditions, the entire system has CSC. The check of CSC for each non-input signal can be performed in the following way: let a_i* be a transition of signal a. Then, a CSC conflict exists if: (i) M_2 is reachable from M_1 by firing a complementary sequence, (ii) M_1 and M_2 have the same code, (iii) a_i* is enabled in M_1 and (iv) for every transition a_j* of signal a, a_j* is not enabled in M_2. If the net is 1-bounded (also called *safe*), the enabledness of a transition x at a marking M can be characterized by the sum of tokens of the places in ${}^\bullet x$ at M: x is enabled at M if and only if the sum of tokens of the places in ${}^\bullet x$ is equal to the number of places in ${}^\bullet x$.

Now we can present a sufficient condition for CSC for each non-input signal a:

THEOREM 3.2. *Let $S = \langle\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, M_0\rangle, \Sigma, \Lambda\rangle$ be a consistent safe STG and non-input signal $a \in \Sigma$. S has CSC for a if the following problem is infeasible for each transition a_i*:*

ILP model for CSC checking:

$$\begin{array}{ll} (i) & \boxed{\textit{Reachability conditions (same as in (2))}} \\ (ii) & \texttt{code\_change}(\Sigma, z) = \mathbf{0} \\ (iii) & \sum_{p \in {}^\bullet a_i*} M_1(p) = |{}^\bullet a_i * | \\ (iv) & \forall a_j*: \sum_{p \in {}^\bullet a_j*} M_2(p) < |{}^\bullet a_j * | \end{array} \quad (3)$$

PROOF. If (3) has no solutions, no complementary sequence exists between any pair of reachable markings M_1 and M_2, with only M_1 enabling signal a. □

Note that the constraint $M_1 \neq M_2$ is not needed in (3).

4. ILP FOR SYNTHESIS

In this section a new design flow for the synthesis of asynchronous circuits is presented. The major gains with respect to previous methods are:

- The synthesis of a speed-independent circuit implementing non-input signals is always guaranteed.
- The use of structural methods allows to deal with very large specifications.

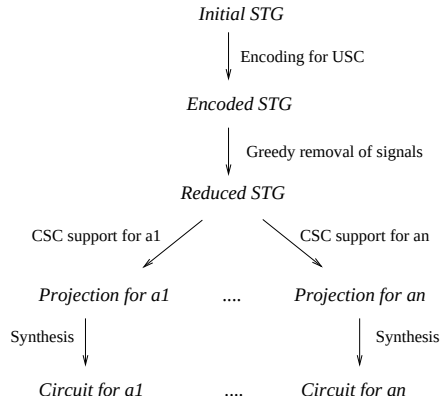

Figure 2: Synthesis of asynchronous circuits.

- Optimization techniques can be applied in some stages of the process.

The synthesis flow is depicted in Figure 2. Given a consistent STG, structural methods to ensure USC are applied (see [3]). These methods produce an over-encoding of the STG that conservatively ensure the USC property. Since many of these signals may be unnecessary to guarantee either USC or even CSC, they are iteratively removed using greedy heuristics until no more signals can be removed without violating the USC/CSC property. This greedy process makes use of the ILP methods presented in Section 3. The reduced STG is next projected onto different sets of signals to implement each individual output signal.

In this section we propose a novel method to calculate the subset of signals onto which the STG must be projected to implement each signal. The support of the next-state function of each signal will be a subset of this support. Since the cardinality of the support is usually small (3-5 signals and in very rare cases is more than 10), state-based algorithms can be used for synthesis after the projection (e.g. petrify [5]).

4.1 Computing a Support for Synthesis

The problem faced in this section is the following: given an STG $S = \langle\langle \mathcal{P}, \mathcal{T}, \mathcal{F}, M_0\rangle, \Sigma, \Lambda\rangle$ and a non-input signal $a \in \Sigma$, can we compute a subset Σ' of Σ such that it is enough for implementing f_a? Two conditions must be satisfied by Σ' [4]:

1. $Trig(a) \subseteq \Sigma'$.
2. $S_{\Sigma'}$ must have CSC for signal a.

Let such Σ' be called *a CSC support of signal a in S*:

DEFINITION 4.1 (CSC SUPPORT). *Let S be an STG with set of events Σ, and a non-input signal $a \in \Sigma$. A set $\Sigma' \subseteq \Sigma$ is a CSC support of a in S if $S_{\Sigma'}$ has no CSC conflicts for signal a and $Trig(a) \subseteq \Sigma'$.*

For example, a possible CSC support for signal d from the STG shown in Figure 3(c) is $\{ldtack, csc\}$. Figure 3(d) shows the projection induced by this CSC support. The rest of this section is devoted to explain how to compute efficiently a CSC support for a given signal a.

The computation of a CSC support can be performed iteratively: starting from an initial assignment, ILP techniques can be used to guide the search. Imagine we have an initial set of signals $\Sigma' \subseteq \Sigma$, candidate to be the CSC support of a

given signal a. A way of determining whether Σ' is a CSC support for signal a is by solving the following ILP problem:

ILP model for checking CSC support:

$$\frac{\boxed{(i),\ (iii) \text{ and } (iv) \text{ from } (3)}}{\texttt{code\_change}(\Sigma', z) = \mathbf{0}} \quad (4)$$

If (4) is infeasible, then Σ' is enough for implementing a. Otherwise the set Σ' must be augmented from signals in $\Sigma \setminus \Sigma'$ until (4) is infeasible. Moreover if (4) is feasible, adding a balanced signal b from $\Sigma \setminus \Sigma'$ will not turn the problem infeasible because z is still balanced for $\Sigma' \cup \{b\}$. On the contrary, adding an unbalanced signal will assign a different code to markings M_1 and M_2 of (4). Therefore, the unbalanced signals in z will be the candidates to be added to Σ'. The algorithm for finding a CSC support set for a non-input signal a is the following:

Algorithm for the calculation of CSC support:

CSC_Support (STG S, Signal a) **returns** CSC support of a

$\Sigma' := \text{Trig}(a) \cup \{a\}$
while (4) is infeasible **do**
  Let b be an unbalanced signal in z
  $\Sigma' := \Sigma' \cup \{b\}$
endwhile
return Σ'

Implementation note

In order to avoid, as much as possible, unnecessary inclusions of signals in the CSC support of a signal, we add an objective function to the problem (4). Assume E_0 and E_1 are the set of signals in Σ with initial value 0 and 1, respectively. We want to search for solutions of (4) such that the number of unbalanced signals is minimal. The minimization of the objective function

$$min\,[\texttt{code\_change}(E_0, z) - \texttt{code\_change}(E_1, z)]$$

avoids any vector z as a solution if there is another vector z' such that a signal with initial value 0 (1) is not balanced in $\texttt{code\_change}(E_0, z)$ ($\texttt{code\_change}(E_1, z)$) but is balanced in $\texttt{code\_change}(E_0, z')$ ($\texttt{code\_change}(E_1, z')$). Therefore, this function reduces the number of unbalanced signals and, thus, less choices are possible when the model is feasible and a new unbalanced signal must be added to Σ'.

4.2 Projection into the CSC Support

Assume that for a non-input signal a its CSC support set CSC(a) has been computed by Algorithm CSC_Support. The next step is to derive the projection of the STG S into CSC(a) ($S_{\mathsf{CSC}(a)}$). The projection is computed by the transformations described in [3]. It is assumed that the projections preserve trace equivalence on the set of traces with respect to the signals in CSC(a) (i.e.$L(S)|_{CSC(a)} = L(S_{CSC(a)})$).

Although $S_{\mathsf{CSC}(a)}$ and S are different STGs, next theorem shows that projection preserves CSC:

THEOREM 4.1. *Let S be an STG with CSC for the non-input signal a, and $S_{\mathsf{CSC}(a)}$ be the projection preserving trace equivalence with S on the set CSC(a). Then $S_{CSC(a)}$ has CSC.*

PROOF. By contradiction. Let us assume there are two reachable markings s'_1, s'_2 in $S_{CSC(a)}$ such that $s'_0 \stackrel{\sigma'_1}{\rightarrow} s'_1$, $s'_0 \stackrel{\sigma'_2}{\rightarrow} s'_2$, $\lambda(s'_1) = \lambda(s'_2)$ and only s'_1 enables some transition a_i* of signal a. Let $s_1, s_2, \sigma_1, \sigma_2$ such that $s_0 \stackrel{\sigma_1}{\rightarrow} s_1$, $s_0 \stackrel{\sigma_2}{\rightarrow} s_2$, $\sigma_1|_{\mathsf{CSC}(a)} = \sigma'_1$ and $\sigma_2|_{\mathsf{CSC}(a)} = \sigma'_2$. Finally, let $Trig(a_i*)$ denote the set of triggering transitions of a_i*. Two cases arise:

$\lambda(s_1) = \lambda(s_2)$: then given that we have CSC for signal a in S either both s_1 and s_2 enable a or none of them enables a. If both enable a then given that the set of trigger transitions of a_i* are in the CSC support of a and $s_2 \stackrel{a_i*}{\rightarrow}$ implies that $Trig(a_i*) \subseteq \sigma'_2$. But then if every trigger transition is fired in σ'_2, there is a state s reachable by firing some prefix of σ'_2 at s'_0 which enables a_i*. If $s \neq s'_2$ we have that $a_i* \in \sigma'_2$, and therefore $a_i* \in \sigma_2$ because $\sigma'_2 \subseteq \sigma_2$. At this point, the fact that $s_2 \stackrel{a_i*}{\rightarrow}$ implies that $\sigma_2 = \delta a_i * \gamma$, with $Trig(a_i*) \subseteq \gamma$. We can iterate this process again with γ, and using the finiteness of σ_2 we can conclude that $s'_2 \stackrel{a_i*}{\rightarrow}$, contradicting the assumption that a_i* is only enabled in s'_1. If neither s_1 nor s_2 enables a then the trace $\sigma'_1 a_i*$ belongs to $L(S_{\mathsf{CSC}(a)})$ but does not belong to $L(S)|_{\mathsf{CSC}(a)}$, contradicting the assumption $L(S)|_{CSC(a)} = L(S_{CSC(a)})$.

$\lambda(s_1) \neq \lambda(s_2)$: if both s_1 and s_2 enable a or none of them enables a, then the same reasoning of the previous case can be applied. If only one of them enables a, then CSC(a) is not a valid CSC support because some signal from $\Sigma \setminus$ CSC(a) which makes $\lambda(s_1) \neq \lambda(s_2)$ hold is not added to CSC(a) in order to make $\lambda(s'_1)$ and $\lambda(s'_2)$ different. □

5. AN EXAMPLE: VME BUS

This section presents an example illustrating the theory presented in previous sections. The STG of Figure 3(a) will be used. It is representing a device controlling data transfers between a VME bus and a device. The STG has a USC conflict: it can be detected in the ILP (2) by the assignment $\{dsr+, lds+, ldtack+\}$ to vector x and the assignment $\{d+, dtack+, dsr-, d-, dtack-, dsr+\}$ to vector z. The conflict is depicted in the state graph of the system (two different states labeled as 10101), shown in Figure 3(b). This conflict is also a CSC conflict, because the output signals d and lds are both enabled in only one of the conflicting states.

The STG of Figure 3(c) is obtained after solving the conflict, where the new internal signal *csc* has been inserted for solving the conflict. From this STG, Algorithm CSC_Support has been invoked for signal d. The resulting CSC support for d is $\{ldtack, csc\}$, which induces the projection shown in Figure 3(d). From the projection, the logic equation implementing d is $d = ldtack \cdot csc$, depicted in Figure 3(e). It is worth mentioning that the implementation for every non-input signal of the example is identical to the one obtained by using petrify. This also happens with other examples that we have synthesized with both approaches.

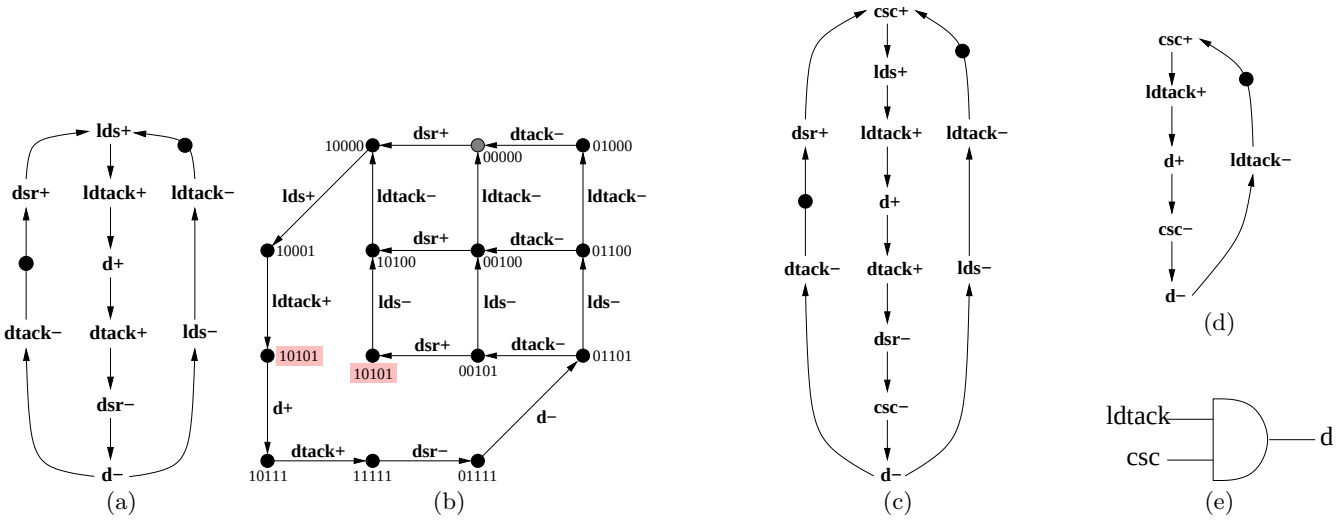

Figure 3: (a) STG, (b) State graph (<dsr,dtack,ldtack,d,lds>), (c) STG with CSC, (d) Projection for signal d, (e) Circuit implementing d.

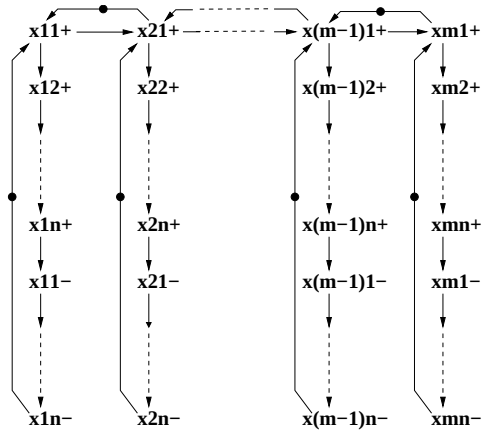

Figure 4: ART(m, n).

6. EXPERIMENTAL RESULTS

The method presented in this paper has been implemented in moebius, a tool for the synthesis of speed-independent circuits. The experiments have been performed on a $Pentium^{TM}$ 4/2.53 Ghz and 512M RAM.

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

- PPWK(m, n) and PPARB(m, n): examples modeling m pipelines weakly synchronized. In addition PPARB(m, n) also includes arbitration. Every benchmark in this set has CSC conflicts. These examples were obtained from [12].

- PPWKCSC(m, n) and PPARBCSC(m, n): a modification of the previous benchmarks to fulfill the CSC property.

- TANGRAMCSC(m, n): examples obtained by translating a synthetic Tangram program into a netlist of handshake components, shown in Fig. 5. Each handshake component is specified as a Petri net and the final controller is obtained as the composition of all Petri nets. The symbols "`;`", "`||`" and "`M`" represent sequencers, parallelizers and mixers, respectively. Each n-way component is implemented as a tree of 2-way components. This is a parametrizable benchmark that represents a typical controller obtained from the direct translation of languages like Tangram [1] or Balsa [9].

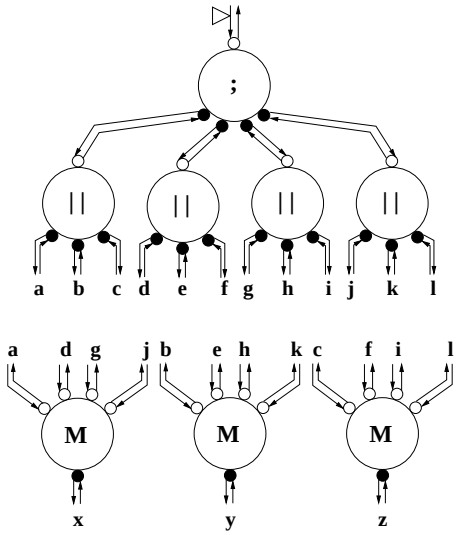

Figure 5: Netlist of handshake components from a Tangram program.

- ART(m, n): examples modeling a different way of synchronizing m pipelines. The STG is depicted in Figure 4. Every benchmark in this set has CSC conflicts.

- ARTCSC(m, n): transformation of the corresponding benchmark by means of the insertion of a new set of signals in order to fulfill the CSC property [3]. The nets in this class of benchmarks are extremely large compared to the corresponding benchmarks (for instance, ART$(30, 9)$ has 636 places while ARTCSC$(30, 9)$

benchmark	$\|P\|$	$\|T\|$	$\|\Sigma\|$	CLP	SAT	ILP
PpWk(2,9)	71	38	19	0.09	0.04	0.15
PpWk(2,12)	95	50	25	0.42	0.37	0.20
PpWk(3,6)	70	38	19	0.12	0.04	0.11
PpWk(3,9)	106	56	28	10.89	0.10	0.25
PpWk(3,12)	142	74	37	933.37	0.04	0.35
PpWkCsc(2,9)	72	38	19	3.14	0.18	0.16
PpWkCsc(2,12)	96	50	25	246.26	1.02	0.31
PpWkCsc(3,6)	72	38	19	2.97	0.04	0.19
PpWkCsc(3,9)	108	56	28	2075.48	0.31	0.41
PpWkCsc(3,12)	144	74	37	time	1.41	0.80
PpArb(2,9)	86	48	23	0.01	0.02	0.05
PpArb(2,12)	110	60	29	0.00	0.05	0.11
PpArb(3,6)	92	54	25	0.00	0.06	0.12
PpArb(3,9)	128	72	34	0.01	0.08	0.08
PpArb(3,12)	164	90	43	0.00	0.33	0.19
PpArbCsc(2,9)	88	48	23	40.89	0.61	0.28
PpArbCsc(2,12)	112	60	29	1021.51	16.24	0.45
PpArbCsc(3,6)	95	54	25	61.30	0.56	0.34
PpArbCsc(3,9)	131	72	34	time	1.52	0.69
PpArbCsc(3,12)	167	90	43	time	16.39	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	time	14 m	3 m
ArtCsc(20,9)	1532	1270	635	time	mem	27 m
ArtCsc(30,9)	2312	1910	955	time	mem	1.5 h
ArtCsc(40,9)	3092	2550	1275	time	mem	3.5 h
ArtCsc(50,9)	3872	3190	1595	time	mem	7 h

Table 1: CSC detection for well-structured STGs.

has 2312) implying an exponential growth of the underlying state space. Therefore the check of CSC/USC for this benchmarks is a hard task.

The experiments for CSC/USC detection are presented in Tables 1 and 2. Each table reports the CPU time of each approach in seconds. We use 'time' and 'mem' to indicate that the algorithm did not complete in less than 10 hours or produced memory overflow, respectively. The tools for comparing the experimental results are:

- CLP: the approach presented in [12] for the verification of USC/CSC. It uses non-linear integer programming methods and the unfolding of the net.
- SAT: the approach presented in [11] for the verification of CSC[2]. It uses a satisfiability solver and the unfolding of the net.
- ILP: the approach presented in this paper.

From the results one can conclude, as it was expected, that checking USC is simpler than checking CSC, given the different nature of the two problems. Moreover, when some encoding conflict exists, the ILP solver can find it in short time. This is explained by the fact that proving the absence

[2]Checking for USC is not implemented in SAT

benchmark	$\|P\|$	$\|T\|$	$\|\Sigma\|$	CLP	ILP
PpWk(3,9)	106	56	28	10.53	0.03
PpWk(3,12)	142	74	37	876.63	0.05
PpWkCsc(3,9)	108	56	28	2002.29	0.67
PpWkCsc(3,12)	144	74	37	time	1.17
PpArb(3,9)	128	72	34	0.01	0.06
PpArb(3,12)	164	90	43	0.00	0.08
PpArbCsc(3,9)	131	72	34	time	1.05
PpArbCsc(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

Table 2: USC detection for well-structured STGs.

of encoding conflicts requires an exhaustive exploration of the *branch-and-bound* tree visited by ILP solvers. The superiority of ILP with respect to CLP and SAT is evident.

Table 3 shows experiments on synthesis to check the quality of the generated circuits. The column 'Lit' reports the number of literals, in factored form, of the netlist. The results are compared with the circuits obtained by `petrify` [5], a state-based synthesis tool, on the same controllers. From the reported CPU time, the time needed for computing a support and for projection was negligible when compared to the time needed for deriving logic equations.

The TangramCsc$(4,3)$ example, shown in Figure 5, illustrates the suitability of our approach for the synthesis of specifications generated from a HDL. According to [1], the cost of implementing the handshake components is the following[3]:

Component	C-elements	2-input gates	literals
2-way sequencer (;)	1	2	9
2-way parallelizer (||)	3	4	23
2-way mixer (M)	2	1	12

The circuit in Fig. 5 has 3 sequencers, 8 parallelizers and 9 mixers: 319 literals. This would be the cost obtained by a *syntax-directed translation*. The cost obtained by logic synthesis methods is significantly smaller. Table 3 also shows that the quality of the circuits obtained by the ILP-based technique is comparable to that of the circuits obtained by `petrify`.

7. CONCLUSIONS

As asynchronous design matures, the role of high-level and logic synthesis becomes more relevant. As the complexity increases, the level of abstraction at which behavior is specified also increases. Therefore, the behavioral structures generated by high-level synthesis tools tend to be well-structured.

Logic synthesis from event-based models can take advantage of structured specifications by using algebraic techniques, such as the one presented in this paper. Though in some cases, these techniques can be conservative, they can always be used as a pre-process to more sophisticated methods that provide more accuracy at the expense of more computational cost. We foresee to see new efforts in this direction as asynchronous circuits have more widespread use.

[3]A C-element is assumed to cost 5 literals: $c = ab + c(a+b)$.

benchmark	states	$\|P\|$	$\|T\|$	$\|\Sigma\|$	Lit.		CPU	
					Pfy	ILP	Pfy	ILP
PpWkCsc(2,6)	8192	47	26	19	57	57	5	1
PpWkCsc(2,9)	524.288	71	38	19	87	87	49	2
PpWkCsc(3,9)	2.7×10^7	106	56	28	–	130	mem	3
PpWkCsc(3,12)	2.2×10^{11}	142	74	37	–	117	time	3
PpArbCsc(2,6)	61440	62	36	17	77	77	21	83
PpArbCsc(2,9)	3.9×10^6	110	60	29	107	107	185	59
PpArbCsc(3,9)	3.3×10^9	131	72	34	163	165	10336	289
PpArbCsc(3,12)	1.7×10^{12}	167	90	43	–	210	time	608
TangramCsc(3,2)	426	142	92	38	97	103	56	146
TangramCsc(4,3)	9258	321	202	83	–	247	mem	2 h

Table 3: Support computation, projection and synthesis compared to state-based approach.

Acknowledgments

This research was supported by the SEGRAVIS (Syntactic and Semantic Integration of Visual Modeling Techniques) contract HPRN-CT-2002-00275 and the Ministry of Science and Technology of Spain, contract TIC2001-2476-C03-02.

8. REFERENCES

[1] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[2] G. Berthelot. Checking Properties of Nets Using Transformations. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, 1986.

[3] J. Carmona, J. Cortadella, and E. Pastor. A structural encoding technique for the synthesis of asynchronous circuits. *Fundamenta Informaticae*, 50(2):135–154, March 2002.

[4] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[6] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, volume 38. Marcel Dekker, New York, February 1998.

[7] J. Desel and J. Esparza. Reachability in cyclic extended free-choice systems. *TCS 114, Elsevier Science Publishers B.V.*, 1993.

[8] J. Desel and J. Esparza. *Free-choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

[9] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[10] Robert M. Fuhrer. *Sequential optimization of asynchronous and synchronous finite-state machines*. PhD thesis, Columbia University, NY, 1999.

[11] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting state coding conflicts in STG unfoldings using SAT. In *Int. Conf. on Application of Concurrency to System Design*, June 2003.

[12] Victor Khomenko, Maciej Koutny, and Alex Yakovlev. Detecting state coding conflicts in stgs using integer programming. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 338–345, 2002.

[13] Kenneth McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.

[14] Stephan Melzer and Javier Esparza. Checking system properties via integer programming. In *European Symposium on Programming*, pages 250–264, 1996.

[15] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[16] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Structural methods for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998.

[17] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[18] Manuel Silva, Enrique Teruel, and José Manuel Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:309–373, 1998.

[19] Peter Vanbekbergen. *Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications*. PhD thesis, Catholic University of Leuven, 1993.

[20] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.