Digital Object Identifier

State-based encoding of large asynchronous controllers

ALBERTO MORENO[1], **JORDI CORTADELLA**[1], **(Fellow, IEEE)**
[1]Department of Computer Science, Universitat Politècnica de Catalunya, Barcelona 08034, Spain

This work has been partially supported by funds from the Spanish Ministry for Economy and Competitiveness and the European Union (FEDER funds) under grant TIN2017-86727-C2-1-R, the Generalitat de Catalunya (2017 SGR 786 and FI-DGR 2015).

ABSTRACT State encoding is one of the fundamental problems in the synthesis of asynchronous controllers. The requirement for a correct hazard-free implementation imposes severe constraints on the way encoding signals can be inserted in the specification of a controller. Even though some specification formalisms, such as Burst-mode machines or Signal Transition Graphs, enable to specify behaviors at the event level, the state encoding methods that provide the best good-quality solutions work at the state level. This imposes a severe limitation on the size of the controllers that can be handled by these methods. This paper proposes a method to solve the encoding problem for large asynchronous controllers using state-based methods. It is based on an iterative process of projection and re-composition that reduces the size specification by hiding signals, partially solves the encoding problem at the state level and re-composes the original specification using a synchronous product. The process iterates until all encoding conflicts have been solved. The method is proved to preserve the behavior of the specification (branching bisimilarity) and shown to be capable of providing good-quality solutions for controllers of more than 100 signals and 10^6 states.

INDEX TERMS Asynchronous circuits, Circuit Synthesis, Logic Circuits, State Encoding

I. INTRODUCTION

Asynchronous controllers dictate the sequencing of data computations and transfers between computational units in asynchronous dataflow circuits [1]. They can also be used as pure control devices in some application domains, such as interfacing between digital and analog components [2].

Asynchronous controllers are typically specified using state-based formalisms, such as Burst-Mode machines [3], or event-based formalisms, such as Signal Transition Graphs (STGs) [4]. A typical specification describes the interaction of a set of control signals by defining the relationship between their rising and falling events. Such specifications must fulfill certain implementability properties to allow a correct circuit (e.g. hazard-free) to be derived.

When pursuing automatic logic synthesis, the most conventional algorithmic strategies resort to a low-level state-based representation of the behavior to derive the Boolean equations of the circuit [5]. This representation is often called State Graph and can be derived by explicitly enumerating the reachable states from higher-level representations.

One of the fundamental problems in the synthesis of asynchronous controllers is state encoding. In general, this requires the insertion of new internal signals to disambiguate encoding conflicts. What makes encoding difficult is the preservation of the implementability properties that ensure the absence of hazards after the insertion of new events. The problem is even more challenging when the controllers work under the input-output mode of operation targeting at the synthesis of speed-independent circuits [5]. Informally, the input-output mode allows the signals at the interface of the circuit to change concurrently even when the circuit has not reached a stable state. In this paper we will face the problem of state encoding for the input-output mode of operation.

A. PREVIOUS WORK

The existing methods to solve the state encoding problem can be divided into two categories: structural and state-based.

Structural methods have been proposed for STGs and exploit the properties of the underlying Petri nets to avoid an explicit enumeration of the state space [6], [7]. In state-base methods, the state space is enumerated explicitly by representing all possible interleavings of concurrent events [8], [9]. State-based methods enable a more accurate exploration of the space of solutions and can potentially lead to better circuits. However, they may suffer from the state explosion problem when the specification is highly concurrent.

The structural methods work directly on the graph representation of the specification (e.g., a Petri net) or some unfolded version. They have limitations about the type of acceptable representations, e.g., safe or free-choice Petri nets, and the locations where the new signals can be inserted. Even for structural methods, some controllers may be too large. For this reason, some techniques have been proposed to decompose a large controller into smaller ones that can be synthesized separately [10].

One of the main problems for decomposition techniques is the appearance of *irreducible conflicts* that cannot be solved while preserving implementability. Solutions for that problem are suggested in [10] by introducing a structure called *gyroscope* that inserts new signals with a high degree of concurrency. However, this structure aims at solving conflicts without paying attention at the cost of implementing the circuit, e.g., the complexity of the Boolean equations.

B. CONTRIBUTIONS OF THIS WORK

After many years of experience with asynchronous controllers we have observed the following facts:

- State-based methods can be superior to structural methods for the state encoding problem. The main reason is that the exploration space for signal insertion is larger and better estimators for good-quality solutions can be used [9].
- Most of the controllers are designed manually by humans and the largest specifications usually have no more than 10^7 states.
- The best-quality state-based methods for encoding can manage up to 10^3 states with an affordable runtime.

Therefore, there is a gap of roughly 4 orders of magnitude between what is computationally affordable for state encoding and the size of large controllers.

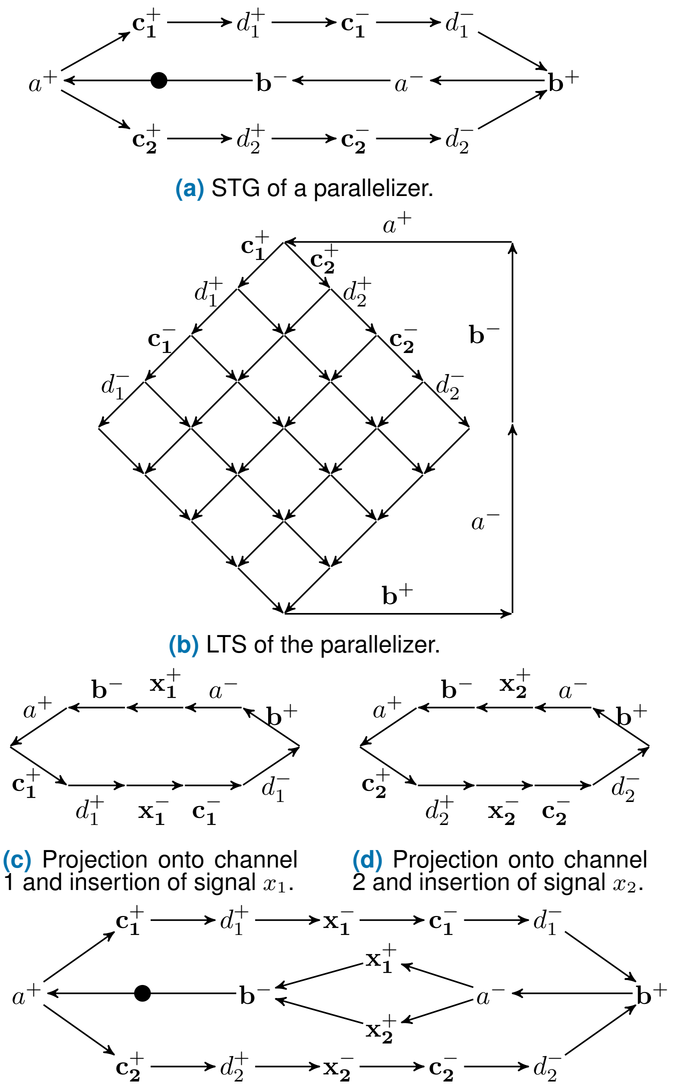
In previous work, the decomposition into smaller sub-controllers has been proposed [10]. Besides the requirement to insert the *gyroscope* structures to avoid irreducible conflicts, the resolution of conflicts at each sub-controller is agnostic on the behavior of the other sub-controllers. This may have a negative effect in the quality of the solutions.

This paper proposes a new approach that explicitly keeps track of the complete state space. The approach iteratively projects the behavior of the controller into subsets of relevant signals and partially solves the encoding problem on the projections. The new signals are incorporated into the original specification and the process is re-executed until all encoding conflicts have been solved.

Unlike other decomposition techniques, irreducible conflicts do not pose any hurdle for the proposed method. While a projection might cause these kind of conflicts, the iterative nature of the projection and re-composition allows for these conflicts to be solved in subsequent iterations.

An important aspect of the method is that the projections can be calculated efficiently. Algorithms with complexity $O(m \log n)$[1] to minimize labelled transitions systems up to

[1] n and m are the number of states and transitions, respectively.



c_1^+ d_1^+ c_1^- d_1^- a^+ b^- a^- b^+ c_2^+ d_2^+ c_2^- d_2^-
x_1^- x_1^+ x_2^- x_2^+

(a) STG of a parallelizer.

(b) LTS of the parallelizer.

(c) Projection onto channel 1 and insertion of signal x_1.

(d) Projection onto channel 2 and insertion of signal x_2.

(e) STG after state encoding.

FIGURE 1: State encoding for a parallelizer.

some criterion of behavioral equivalence (branching bisimilarity) can be used [11]. Thus, the large controllers can still be manipulated and the state encoding problem solved in small controllers using SAT-based methods [9].

The re-composition of the system with the new inserted signals can be done via synchronous products, which can have a quadratic runtime in the worst-case, but typically run in linear time due to the high similarity of the two components.

II. OVERVIEW

This section sketches the main features of the method proposed in this paper. The example shown in Fig. 1 will be used to illustrate the method. Fig. 1a shows an STG specifying the behavior of a parallelizer, which is a controller used in handshake circuits to fork the execution of two asynchronous processes.

Signals a (input) and b (output) are the handshake signals of the channel that triggers the activity of the parallel processes represented by the handshake signals c_i (output) and d_i (input).

The controller can be represented by a Petri net in a very succinct way. Yet, due to the high level of concurrency, it suffers from the state explosion problem. This means that the number of valid markings, corresponding to states in a labeled transition system (LTS), grows exponentially with the number of channels. Fig. 1b shows the LTS representation of the same parallelizer. To quantify the state explosion, the following table shows the number of states needed to represent a parallelizer with n processes:

Processes	2	3	4	5	6	7	8	n
Signals	6	8	10	12	14	16	18	$2n+2$
States	28	128	628	3K	16K	78K	391K	5^n+3

Let us now consider one of the channels, which follows the sequence $\langle c_i+, d_i+, c_i-, d_i-\rangle$. The reader will notice that the states before c_i+ and after d_i- have the same encoding. This conflict occurs at every channel. In this particular example, it is sufficient to focus on each channel individually to solve the corresponding encoding conflict. Thus, a channel can be freed from conflicts if a signal is inserted between d_i+ and c_i-. This example suggests that not all the information is relevant to find a solution for certain encoding conflicts.

In order to exploit this feature, this paper proposes the following method:

1) Find a group of signals to be hidden and project the behavior onto the remaining signals. For the example, a good strategy is to hide every signal except a, b, and one of the channels (c_i and d_i). Initially, the signals c_2 and d_2 are hidden while c_1 and d_1 are maintained.
2) Insert new signals to solve the encoding conflicts of the simplified controller. Fig. 1c shows the projected LTS after the insertion of signal x_1.
3) Recompose the full controller by doing a synchronous product between the original controller and the simplified one with the new inserted signals.
4) If not all conflicts have been solved, go to step 1 and repeat the process using the full controller with the new inserted signals.

In this example, the second iteration would generate the projection shown in Fig. 1d, after hiding c_1, d_1 and x_1. After recomposing the original LTS, the behavior shown in the STG of Fig. 1e would be obtained.

By *hiding* a well selected set of signals, an asynchronous controller can be simplified enough so that it is possible to use state encoding techniques that can handle the full state space.

In general, asynchronous controllers do not show behaviors as simple as the one of the parallelizer and the automation of the process requires smart strategies to calculate projections. This paper presents a method to simplify arbitrary asynchronous controllers and obtain small projections that can be manageable by encoding tools working at state level.

III. BACKGROUND

This section reviews some known concepts on Boolean functions, asynchronous LTSs and speed-independent circuits. Additionally, it revisits the notion of branching bisimilarity to characterize systems that are behaviorally equivalent.

A. BOOLEAN FUNCTIONS

An incompletely specified function (ISF) is a functional mapping $F : \mathbb{B} \to \{0, 1, -\}$, where $\mathbb{B} = \{0, 1\}$ and '$-$' represents the *don't care* (DC) value. The subsets of $\mathbb{B}^n$ in which F has the 0, 1 and DC values are called the OFF-, ON- and DC-set, respectively.

Let $F(x_1, x_2, \ldots, x_n)$ be a Boolean function of n Boolean variables. The set $X = \{x_1, x_2, ..., x_n\}$ is the *support* of the function F. A variable $x_i \in X$ is *essential* for function F if there exist at least two elements of $\mathbb{B}^n$, v_1 and v_2, that only differ on the value of x_i, such that $F(v_1) = 0$ and $F(v_2) = 1$.

B. ASYNCHRONOUS LABELED TRANSITION SYSTEM

An *Asynchronous Labeled Transition Systems* (ALTS) is a 4-tuple $A = (S, \Sigma, T, s_0)$ where:

- S is a finite non-empty set of states.
- $\Sigma = \text{In} \cup \text{Out} \cup \text{Int}$ is the set of signals, with In, Out and Int being disjoint sets of input, output and internal signals, respectively.
- $T \subset S \times \mathcal{L}_\tau(\Sigma) \times S$ is the set of transitions, with
  - $\mathcal{L}(\Sigma) = \Sigma \times \{+, -\}$
  - $\mathcal{L}_\tau(\Sigma) = \mathcal{L}(\Sigma) \cup \{\tau\}$
  - For every $(s, a, s') \in T$, $s \neq s'$.
  - At most one transition $(s, a, s') \in T$ exists between s and s'.
- s_0 is the initial state.

Henceforth, we will also assume that all states in S are reachable from s_0. The label τ is used to represent a silent (non-observable) event. A τ-free ALTS is an ALTS in which there is no transition with label τ. This is an important property for state-based encoding tools. These tools require either a τ-free ALTS or all τ transitions to be *inert*, i.e., hidable while preserving the behavior of the specification.

We denote $(s, a, s') \in T$ by $s \xrightarrow{a} s'$, where $a \in \mathcal{L}_\tau(\Sigma)$ is an event (possibly silent). Rising and falling transitions of signal $a \in \Sigma$ between states s and s' are represented by $s \xrightarrow{a^+} s'$ and $s \xrightarrow{a^-} s'$, respectively. We will sometimes refer to $s \xrightarrow{a^\pm} s'$ as a generic transition of signal a.

We will refer to events that possibly have arbitrarily many τ events interleaved. We use $s \overset{a}{\Rightarrow} s'$ as a possibly empty (ϵ) sequence of transitions with the trace $\tau^* a$. In particular, if $s \overset{\epsilon}{\Rightarrow} s'$ (empty transition) then $s = s'$. Additionally, $\alpha \in \mathcal{L}_\tau(\Sigma)^*$ denotes a sequence of (possibly empty) events, with $\alpha = a_1 a_2 \ldots a_n$ and $s \xrightarrow{\alpha} s'$ the sequence of transitions that leads from s to s' by following the events of α. If $s \overset{\alpha}{\Rightarrow} s'$, then τ events may be interleaved between events in the form $\tau^* a_1 \tau^* a_2 \tau^* \ldots a_n$.

An event a is *enabled* in state s if there is a transition $s \xrightarrow{a} s'$ for some s'. Furthermore a signal a is enabled in s

if $s \xrightarrow{a^\pm} s'$ for some s'. A sequence of events $\alpha \in \mathcal{L}_\tau(\Sigma)^*$ is enabled in state s if $s \xrightarrow{\alpha} s'$ for some s'.

C. BRANCHING BISIMILARITY

Milner proposed *observational equivalence* [12] (or weak bisimilarity) as a branching time semantics to classify systems according to their capability of being distinguishable by an external observer under the presence of unobservable events. *Branching bisimilarity* was later introduced as a stronger equivalence that preserves the branching structure of processes [13]. The difference between both equivalences is very subtle and irrelevant in most practical cases.

Given an ALTS $A = (S, \Sigma, T, s_0)$ we call a relation $R \subseteq S \times S$ a branching bisimulation relation if for all $s, t \in S$ such that sRt, the following conditions hold for all $a \in \mathcal{L}_\tau(\Sigma)$ [14]:

- If $s \xrightarrow{a} s'$, then
  - either $a = \tau$ and sRt', or
  - there is a sequence $t \overset{\tau^*}{\Longrightarrow} t'$ such that sRt' and $t' \xrightarrow{a} t''$ with $s'Rt''$.
- Symmetrically, if $t \xrightarrow{a} t'$, then
  - either $a = \tau$ and sRt', or
  - there is a sequence $s \overset{\tau^*}{\Longrightarrow} s'$ such that $s'Rt$ and $s' \xrightarrow{a} s''$ with $s''Rt'$.

Two states s and t are branching bisimilar, denoted by $s \approx t$, if there is a branching bisimulation R such that sRt. Two ALTSs A_1 and A_2 are branching bisimilar, denoted by $A_1 \approx A_2$ if their initial states are branching bisimilar.

D. STATE ENCODING

Signals in an ALTS implicitly assign binary codes to the state. Thus, $s(a) = 1$ or $s(a) = 0$ represent the fact that a has value 1 or 0 in state s, respectively. In particular, $s \xrightarrow{a^+} s'$ implies $s(a) = 0$ and $s'(a) = 1$. Similarly, $s \xrightarrow{a^-} s'$ implies $s(a) = 1$ and $s'(a) = 0$. If $s \xrightarrow{b} s'$, with $b \in \Sigma \cup \{\tau\}$, for any $b \neq a$, then $s(a) = s'(a)$. An ALTS is said to be *consistent* if these rules can be applied to every signal and state without any contradiction. In a consistent ALTS with $\Sigma = \{a_1, a_2, ..., a_n\}$, a code can be assigned to every state: $code(s) = (s(a_1), s(a_2), ..., s(a_n))$.

The positive and negative *excitation regions* of signal a, denoted ER_a^+ and ER_a^- respectively, are the sets of states in which a^+ (for ER_a^+) and a^- (for ER_a^-) are enabled. The positive and negative *quiescent regions* of signal a, denoted QR_a^+ and QR_a^- respectively, are the sets of states in which a is not enabled and has value 1 (for QR_a^+) and 0 (for QR_a^-). For convenience we also define $ER_a = ER_a^+ \cup ER_a^-$ and $QR_a = QR_a^+ \cup QR_a^-$. When referring to individual states, $ER_a^+(s)$, $ER_a^-(s)$, $QR_a^+(s)$ and $QR_a^-(s)$ denote that s belongs to ER_a^+, ER_a^-, QR_a^+ and QR_a^- respectively.

We define the on- and off-regions for signal a as $ON_a = ER_a^+ \cup QR_a^+$ and $OFF_a = ER_a^- \cup QR_a^-$. The next-state function of a signal defines its future value in the next stable state. Thus, an enabled signal toggles its value, whereas a stable signal maintains its value. The next-state function for signal a is an ISF defined as follows:

$$\begin{aligned} ONset(a) &= \cup_{s \in ON_a} code(s) \\ OFFset(a) &= \cup_{s \in OFF_a} code(s) \\ DCset(a) &= \mathbb{B}^n \setminus (ONset(a) \cup OFFset(a)) \end{aligned}$$

An ALTS satisfies the *Complete State Coding* (CSC) property if the next-state function for any non-input signal is well defined, i.e.,

$$\begin{aligned} &\forall s, s' \in S, \forall a \in \text{Out} \cup \text{Int} : \\ &\quad (s \in ON_a \wedge s' \in OFF_a) \implies code(s) \neq code(s') \end{aligned}$$

The CSC property is a necessary condition for a specification to be implementable as a circuit. If the previous condition does not apply for the states s, s' and signal a, we say that there is a CSC conflict between s and s'. Furthermore, we say that a has a CSC conflict in s, s' when:

$$\begin{aligned} &CSC_a(s, s') \implies \\ &\qquad code(s) = code(s') \wedge (s \in ON_a \wedge s' \in OFF_a) \end{aligned}$$

Finally, the number of CSC conflicts for signal a is defined as the number of pairs of states s, s' such that a is in CSC conflict.

E. SPEED INDEPENDENCE AND CONFLICTS

From [12], an ALTS $A = (S, \Sigma, T, s_0)$ is weakly deterministic if, for every state $s \in S$ and for every sequence of events $\alpha \in \mathcal{L}_\tau(\Sigma)^*$, whenever $s_1 \overset{\alpha}{\Rightarrow} s_2$ and $s_1 \overset{\alpha}{\Rightarrow} s_3$ then $s_2 \approx s_3$. For the rest of the paper, the term *determinism* will refer to weak determinism.

A signal a triggers another signal b if there is a transition $s \xrightarrow{a^\pm} s'$ such that b is enabled in s' and not enabled in s. Conversely, a disables b if b is enabled in s and not in s'. An $ALTS$ is said to be *output persistent* if for any pair of signals a and b such that a disables b, then both a and b are input signals.

An $ALTS$ is said to be *commutative* if for any state s in which $s \xrightarrow{ab} s'$ and $s \xrightarrow{ba} s''$, then $s' = s''$.

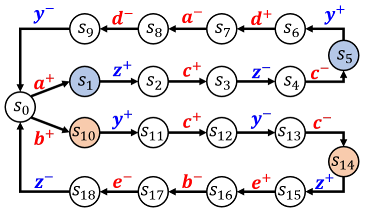
A *Well-Formed* ALTS (WF-ALTS) is an ALTS such that is deterministic, commutative and output persistent. An important result on speed independence is the following [8]:

> A WF-ALTS that satisfies the CSC property is implementable as a speed-independent circuit.

An additional important property is *input-properness*. An ALTS is input-proper if no internal signal triggers any input signal. This guarantees that the behavior of the environment does not depend on any unobservable signal of the circuit.

A signal a is said to be *in conflict* if there is another signal b such that either a disables b or b disables a. We say that σ is a conflict-free set of signals if every signal $a \in \sigma$ is not in conflict.

Solving the state encoding problem is based on inserting new signals to disambiguate CSC violations. The insertion of



y^-
d^-
a^-
d^+
y^+
a^+
z^+
c^+
z^-
c^-
b^+
y^+
c^+
y^-
c^-
z^-
e^-
b^-
e^+
z^+
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18}

FIGURE 2: ALTS with CSC conflicts.

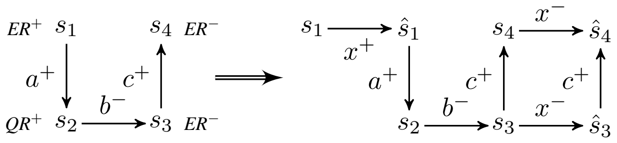


FIGURE 3: $ALTS$ before and after signal insertion.

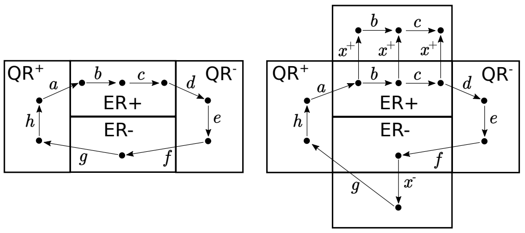


FIGURE 4: Partitioning of the state space into the ER and QR regions of x before (left) and after (right) the insertion.

new signals proposed in this paper preserves the conditions for speed-independence and input-properness.

F. EXAMPLE

Fig. 2 depicts an ALTS with five input signals ($a, \dots, e$) and two output signals (y, z). The pairs of states (s_1, s_5) with code $abcdeyz = 1000000$ and (s_{10}, s_{14}) with code 0100000 are in CSC conflict, since states in each pair share code but differ in onset for y and z.

There are two signals in conflict, a and b, since they disable each other at state s_0. The ALTS is a WF-ALTS since it is deterministic, output persistent and commutative.

IV. ALTS TRANSFORMATIONS

This section describes a collection of transformations over WF-ALTSs. The purpose of these transformations is to provide an infrastructure to insert/hide signals and recompose the original specification with new signals that solve the CSC conflicts. All these transformations must satisfy two properties:

- The behavior of the system must be preserved (branching bisimilarity).
- The implementatibility conditions must hold.

The most important result of this section indicates that the projections of the specification should never hide signals in conflict. This strategy allows to work with τ-free WF-ALTSs when inserting new signals to solve CSC conflicts.

A. SIGNAL INSERTION

The insertion of a new *internal signal* is now described. This transformation is always applied to a τ-free WF-ALTS. Signal insertion was proposed in [8], [15] and proved to preserve trace equivalence when the new inserted signal is silent. Since WF-ALTS are also deterministic, signal insertion also preserves branching bisimilarity [16].

Henceforth, the new inserted signal will be named $x \notin \Sigma$, whereas the signals from the original $ALTS$ will be named $a, b \in \Sigma$. The signal insertion process requires all states in S to be partitioned into four sets[2]: ER^+, ER^-, QR^+ and QR^-. These sets will determine the future ERs and QRs of x.

[2]When no subscript is specified in the sets, they are assumed to refer to the new inserted signal.

After inserting signal x, some transitions will be delayed (triggered) by x. These are the transitions that *exit* ER:

$$EXIT = \{s \xrightarrow{a} s' \mid (ER^+(s) \wedge \neg ER^+(s')) \vee (ER^-(s) \wedge \neg ER^-(s'))\}$$

Some other transitions will become concurrent with x. These are transitions that will remain inside ER:

$$CONC = \{s \xrightarrow{a} s' \mid (ER^+(s) \wedge ER^+(s')) \vee (ER^-(s) \wedge ER^-(s'))\}$$

The set of new states created by the insertion of x is called $\hat{S}$. For every state $s \in ER$ a new *sibling* state $\hat{s} \in \hat{S}$ is added. New transitions are also added with the new states. In particular, the new sets of transitions are:

$$\begin{aligned} T_x &= \{s \xrightarrow{x^+} \hat{s} : s \in ER^+\} \cup \{s \xrightarrow{x^-} \hat{s} : s \in ER^-\} \\ T_d &= \{\hat{s} \xrightarrow{a} s' : s \xrightarrow{a} s' \in EXIT\} \\ T_c &= \{\hat{s} \xrightarrow{a} \hat{s}' : s \xrightarrow{a} s' \in CONC\} \end{aligned}$$

with T_x referring to the transitions between siblings, T_d to the delayed transitions and T_c to the concurrent transitions.

The new $ALTS$ (S', Σ', T', s'_0), obtained after the insertion of x in the original $ALTS$ (S, Σ, T, s_0) is defined as:

- $s'_0 = s_0$
- $S' = S \cup \hat{S}$
- $T' = (T \cup T_x \cup T_d \cup T_c) \setminus EXIT$
- $\Sigma' = \Sigma \cup \{x\}$

Fig. 3 shows an example of signal insertion on a fragment of an ALTS. On the left, the figure shows the ALTS before signal insertion in which every state has been tagged with one of the ERs or QRs of x. On the right, states in the ER of x have been duplicated and the new transitions defined accordingly.

A generic view of signal insertion is depicted in Fig. 4. On the left, the partition of S into the four ER/QR regions of x is shown. On the right, the state space after adding the sibling states is shown.

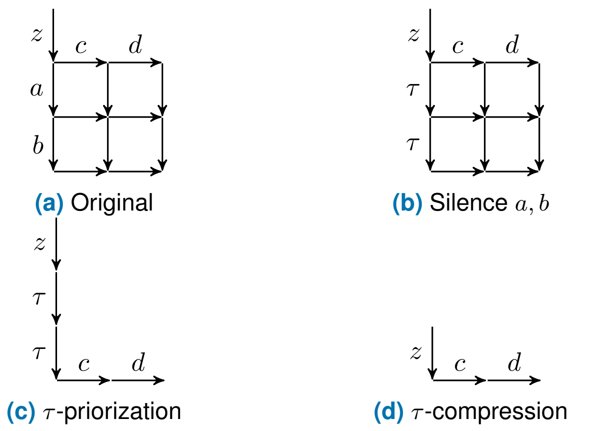


FIGURE 5: Process to hide signals a and b.

B. HIDING SIGNALS

Given an LTS $A = (S, \Sigma, T, s_0)$, a set of signals σ can be *silenced*, denoted $silence(A, \sigma)$, if every event of every signal $a \in \sigma$ is substituted by τ. Fig. 5b shows an example of the silence operation on Fig. 5a for signals a and b.

We are now interested in removing the new τ transitions that appear after a *silence* operation. This will yield a smaller ALTS. One of the ways of achieving this is by using τ-priorization and τ-compression operations described in [17].

The τ-priorization operation consists on the following: if there is a transition $s \xrightarrow{\tau} s'$, then any other transition $s \xrightarrow{a} s'$ is removed.

Intuitively, this operation assumes *zero-delay* τ-transitions and non-zero delay for the other transitions. This makes the model *prioritize* τ's over other transitions. Fig. 5c shows the τ-priorization for the ALTS 5b.

The τ-compression is an operation aimed at removing sequences of τ transitions. If a state s has only one transition, and this transition is $s \xrightarrow{\tau} s'$, then s is *merged* with s'. An example for this operation can be seen in Fig. 5d.

Important results about these operations can be found in [17]. In particular, these operations preserve *branch bisimilarity*. If the τ transitions are persistent, then applying both operations yields a τ-free ALTS.

Finally, given an ALTS $A_1 = (S_1, \Sigma_1, T_1, s_0^1)$, a set of signals σ is said to be *hidden* in $A_2 = (S_2, \Sigma_2, T_2, s_0^2)$, denoted $A_2 = hide(A_1, \sigma)$, if A_2 is the τ-compression of the τ-priorization of $silence(A_1, \sigma)$.

We can now define the concept of *branch bisimilarity with respect to a set of signals*. Let $A_1 = (S_1, \Sigma_1, T_1, s_0^1)$, $A_2 = (S_2, \Sigma_2, T_2, s_0^2)$ be two ALTS, then A_1 is branching bisimilar with respect to σ, denoted $A_1 \approx_\sigma A_2$, iff $silence(A_1, \sigma) \approx silence(A_2, \sigma)$.

With these definitions and results we can now state that, given the τ-free WF-ALTS A_1, and $A_2 = hide(A_1, \sigma)$, then $A_1 \approx_\sigma A_2$. Furthermore, if σ is conflict-free, then A_2 is τ-free.

On the other hand, hiding a signal in conflict does not yield a τ-free ALTS. Fig. 6a shows an ALTS with a non-persistent τ transition. In this case, τ-priorization cannot be applied without breaking branching bisimilarity. The τ transition can still be removed, as shown in Fig. 6b, but this ALTS only preserves *trace equivalence*, which is a weaker equivalence class.

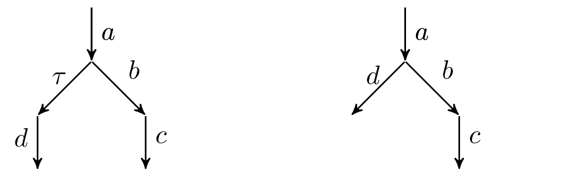

(a) Non-persistent τ transition.
(b) Removed τ transition.

FIGURE 6: Removing a non-persistent τ transition does not preserve branching bisimilarity.

C. SYNCHRONOUS PRODUCT

The synchronous product of two LTSs can be defined as follows. Let $A_1 = (S_1, \Sigma_1, T_1, s_0^1)$, $A_2 = (S_2, \Sigma_2, T_2, s_0^2)$ be two ALTS. The synchronous product of A_1 and A_2, denoted by $A_1 \times A_2$ is another LTS (S, Σ, T, s_0) defined by:

- $s_0 = \langle s_0^1, s_0^2 \rangle \in S$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $S \subseteq S_1 \times S_2$ is the set of states reachable from s_0 according to the following definition of T':
- Let $\langle s_1, s_2 \rangle \in S$:
  - If $a \in \Sigma_1 \cap \Sigma_2$, $s_1 \xrightarrow{a} s_1'$ in T_1 and $s_2 \xrightarrow{a} s_2'$ in T_2, then $\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1', s_2' \rangle$ in T'
  - If $a \in \Sigma_1 \setminus \Sigma_2$ and $s_1 \xrightarrow{a} s_1'$ in T_1, then $\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1', s_2 \rangle$ in T'
  - If $a \in \Sigma_2 \setminus \Sigma_1$ and $s_2 \xrightarrow{a} s_2'$ in T_2, then $\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1, s_2' \rangle$ in T'
  - No other transitions belong to T'
- $T \subseteq T'$ is the set of transitions between states in S that belong to T'.

Theorem 1. *Let $A_1 = (S_1, \Sigma_1, T_1, s_0^1)$ and $A_2 = (S_2, \Sigma_2, T_2, s_0^2)$ be two τ-free WF-ALTS, with $\sigma_1 = \Sigma_1 \setminus \Sigma_2$ and $\sigma_2 = \Sigma_2 \setminus \Sigma_1$. Let $A_3 = A_1 \times A_2$. If $A_1 \approx_{\sigma_1 \cup \sigma_2} A_2$ then $A_3 \approx_{\sigma_2} A_1$ and $A_3 \approx_{\sigma_1} A_2$.*

See the proof in the appendix.

Informally, the synchronous product of two ALTS that are branching bisimilar with respect to their common signals will also be branching bisimilar to the original ALTSs with respect to their common signals. This becomes important later to ensure that the approach presented in this paper preserves branching bisimilarity at all steps.

V. CSC RESOLUTION ALGORITHM

Intuitively, solving CSC conflicts is an iterative process with the following steps:

- Hide a subset of signals to reduce the size of the ALTS.
- Insert a new signal to solve some of the CSC conflicts of the remaining signals.

- Re-compose the ALTS by recovering the previously hidden signals.
- Reduce concurrency of the newly inserted signal.

This process is repeated until all CSC conflicts have been solved. A high-level description of the algorithm is shown in Algorithm 1.

The projection step (line 1) is necessary to reduce the ALTS to a size that is manageable by CSC solving algorithms. Preserving the relevant signals of the CSC conflicts is essential to derive good-quality solutions. The insertion of a new signal x_i (line 2) will solve only conflicts of the remaining signals. The details of the projection step are discussed in Section V-A.

The hidden signals are recovered by re-composing the original ALTS with the new inserted signal. This is achieved by computing a synchronous product (line 3) between the new ALTS (with the new inserted signal) and the original one.

Re-composition implicitly creates a high-degree of concurrency of the new inserted signal with the signals that were hidden by the projection. In particular, the CSC conflicts for the hidden signals are not solved by the new signal. To mitigate this effect, the concurrency of the new signal is reduced (line 4). As a side-effect, new CSC conflicts are solved and the size of the ALTS is also reduced.

An accurate description of concurrency reduction is out of the scope of this paper. An in-depth discussion can be found in [18].

```
Algorithm 1: SOLVECSC($A$)
  input  : An ALTS with CSC conflicts.
  output: An ALTS without CSC conflicts.
  begin
      while $A$ has CSC conflicts do
1         $B$ = Project($A$)         /* Hiding signals    */
2         $x_i$ =insertSignal($B$)   /* Solving CSC       */
3         $A = A \times B$           /* Re-composition    */
4         reduceConcurrency($A$, $x_i$)
      return $A$
```

The following subsections describe few more details about projection and re-composition. Afterwards a discussion about some of the properties of this algorithm is presented.

A. PROJECTION

The main objective of the projection step is to reduce the size of the ALTS by means of hiding signals. The only constraint is that none of the hidden signals can be *in conflict*. The reason is that τ events become *inert* if they are not in conflict [17] and, thus, they can be completely removed during state minimization. That means that any ALTS can become τ-free if no signals in conflict are hidden.

The set of signals to hide has an impact on the quality of the solution. Next, a set of concepts useful to define the criteria to select the signals are discussed:

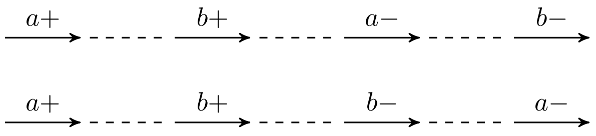


FIGURE 7: Top: lock relation between a and b. Bottom: a and b are not in lock relation.

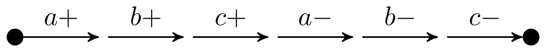


FIGURE 8: Hiding signals a, b and c causes the CSC conflict represented by the dots to collapse.

- **Concurrency:** Hiding signals with high concurrency has a bigger impact on the size of the ALTS. It is thus convenient to hide signals with a large ER.
- **Lock relation:** Two signals a and b are in lock relation when, for every possible trace in the ALTS, there must be a transition b between two transitions for signal a and vice versa. Fig. 7 shows an example of lock relation. Signals in lock relation are helpful to solve conflicts [19].
- **Signals with CSC conflicts:** Hiding signals with many CSC conflicts generate ALTSs with fewer conflicts. This gives less information to the signal insertion process. In general, preserving signals with many conflicts leads to more informed decisions and better solutions.
- **Conflict collapse:** A CSC conflict between the states s and s' *collapses* when all the signals present in a path between s and s' are hidden. Fig. 8 shows an example of a collapsing conflict. A collapsed conflict is not observed and cannot be solved in the projected ALTS. It is thus convenient not to hide signals that collapse conflicts, whenever possible.

Algorithm 2 shows a high level description of the projection algorithm. The first step is to choose the signal a with the largest amount of CSC conflicts. This signal will be the anchor of the new ALTS and will **not** be hidden. The objective is to solve as many conflicts as possible for a. Next, signals are iteratively hidden until the size of the ALTS is below a threshold. At each iteration, the best candidate signal for hiding is obtained.

The criteria (in priority order) to select the best candidates is as follows:

- From the set of signals that do not collapse conflicts, and are not in lock relation with a, the best candidate is the one with the largest ER (highest concurrency).
- In case of a tie, the signal with the smallest number of CSC conflicts is selected.
- If no such signal exists, the lock relation constraint is dropped and the best candidate is selected.
- In case of a tie, the signal that collapses the smallest number of conflicts is selected.

Algorithm 2: PROJECT(A)

```
input  : An ALTS with CSC conflicts.
output: An ALTS with a size under thresholdSize
begin
    $a$ = signalWithLargestCSCconflictNumber($A$)
    while size($A$) $>$ thresholdSize do
        $b$ = findSignalToHide($A$, $a$)
        hideSignal($A$, $b$)
    return $A$
```

B. RE-COMPOSITION

Re-composition aims to re-introduce the hidden signals after signal insertion. This is achieved by calculating the synchronous product of the original ALTS with the projection after solving CSC.

The conflicts solved in the projected ALTS will also be solved in the re-composed ALTS, whereas the other ones will remain. If some conflicts were collapsed during signal hiding, they will also remain after re-composition.

Re-composition greatly increases concurrency, specially when a high number of signals were hidden during the projection. This may have the undesired effect of increasing the total number of CSC conflicts, thus precluding convergence of the algorithm. For this reason, concurrency reduction is an effective way of avoiding this effect.

C. CONCURRENCY REDUCTION

We resort to the concurrency reduction transformation proposed in [18]. A concurrency reduction operation over a signal a reduces the size of the ER for that signal and preserves commutativity, determinism and persistency. In particular, it also preserves branching bisimilarity with respect to a.

Concurrency reduction has two positive effects:

- The size of the ALTS is reduced.
- Additional CSC conflicts are solved.

This operation makes some states unreachable and, as a by-product, CSC conflicts are reduced if some of these states are involved in the conflicts. Furthermore, the reduction of states also increases the DC-set of the logic functions and the opportunities to simplify the Boolean equations.

There are multiple ways of performing concurrency reduction, each one deriving a different solution. Fig. 9 shows an example with different valid reductions.

In this work, a greedy approach has been used to decide how concurrency must be reduced. At every state, the number of possible reductions may be potentially of the order of 2^k, with k being the number of enabled signals at the state. To avoid a worst-case exponential cost in the exploration of highly-concurrent controllers, a limit is defined for the maximum number of solutions that are evaluated.

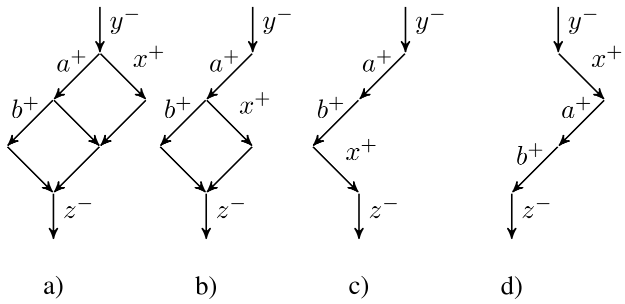


FIGURE 9: Concurrency reduction of x: a) full concurrency; b) no concurrency with a; c) no concurrency with a and b, b triggers x; d) no concurrency with a and b, x triggers a.

The following criteria are taken into account to estimate the quality of each solution:

- The number of CSC conflicts that disappear.
- The number of states that become unreachable.
- The number of new trigger signals that appear/disappear.

The number of CSC conflicts that disappear after the reduction is used to maximize the utility of the inserted signal. Furthermore, reducing the number of states as much as possible is important to prevent the size of the ALTS from growing excessively. Finally, the number of triggers is highly correlated with the number of essential literals. This is, at the same time, correlated with the number of literals after logic synthesis [9]. Ideally, the concurrency reduction operation would minimize the number of essential literals. This work proposes to use the trigger events as proxy for essential literals for the sake of performance.

D. PROPERTIES OF THE ALGORITHM

There are two main properties that this algorithm must have to be an effective and valid technique:

- The computational complexity must be affordable.
- It must preserve the behavior of the specification (branching bisimilarity).

Complexity. For projection, hiding a signal is done by a step of *silencing*, followed by τ-priorization and τ-compression. Silencing can be trivially done in O($|T|$), whereas τ-priorization and τ-compression can also be solved in O($|T|$) [17].

The other important operation for the projection is selecting the signal to be hidden. The worst-case cost is dominated by the detection of pairs of conflicts that collapse. Theoretically, this operation is O($|C| + |S|$), with $|C|$ representing the number of CSC conflicts and $|S|$ the number of states. Although a theoretical upper bound for $|C|$ is $|S|^2$, in practice $|C| < |S|$ for realistic controllers. Since the projection step is repeated on the order of O($|\Sigma|$), the average complexity for projection is O($|S| \times |\Sigma|$).

The complexity of the re-composition step is the one of the synchronous product. However this is a singular synchronous product $A_1 \times A_2$ in which A_2 is a projection of A_1 with a newly inserted signal. If A_1 has $|S|$ states, the product will have at most $2|S|$ states, under the assumption that the new signal can be highly concurrent with the original specification. Thus, the synchronous product can run in linear time.

The cost of concurrency reduction is maintained as $O(|S| \times |\Sigma|)$ and guaranteed by the heuristic that explores a small amount of options at each state in which the new signal is enabled (discussed in Section V-C).

In general, the average runtime for solving CSC of a large controller can be modeled as:

$$\text{Runtime(CSC)} = O\left(|X| \times (|S| \times |\Sigma| + \text{Runtime}(\text{CSC}_{proj}))\right)$$

where $|X|$ is the number of inserted signals. For every signal, the cost might be dominated by the projection/re-composition steps ($O(|S| \times |\Sigma|)$) or the runtime for solving CSC of the projected controllers. The dominating term will depend on the size of the projected controllers. If they are small, the effort will be dominated by the projection/re-composition. Conversely, a little effort in projection (hiding few signals) will result in larger controllers and a major effort in solving CSC. Defining the appropriate size of the projected controllers is a tuning parameter of the method.

Branching bisimilarity. We need to show that the insertion of a new signal x through the following transformations preserves branching bisimilarity:

$$A_1 \xrightarrow{hide(A_1,\sigma)} A_2 \xrightarrow{insert(A_2,x)} A_3 \xrightarrow{A_3 \times A_1} A_4 \approx_x A_1$$

Theorem 2. *Let $A_1 = (S_1, \Sigma_1, T_1, s_0^1)$ be a τ-free WF-ALTS, $A_2 = (S_2, \Sigma_2, T_2, s_0^2)$ such that $A_2 = hide(A_1, \sigma)$, with σ being a conflict-free set of signals, $A_3 = (S_3, \Sigma_3, T_3, s_0^3)$ such that $A_3 = insert(A_2, x)$, with $x \notin \Sigma_1$, and $A_4 = (S_4, \Sigma_4, T_4, s_0^4)$ such that $A_4 = A_3 \times A_1$. Then $A_1 \approx_x A_4$.*

Proof. First note that the hiding operation and the insertion operation preserves branching bisimilarity, so $A_1 \approx_\sigma A_2$ and $A_2 \approx_x A_3$. Since branching bisimilarity is transitive, $A_1 \approx_{\sigma\cup\{x\}} A_3$. Then, by Theorem 1, $A_4 \approx_\sigma A_3$ and $A_4 \approx_x A_1$. □

From [18] it can be easily observed that the basic transformation for concurrency reduction (forward reduction) is equivalent to a τ-priorization assuming the new inserted signal x is silent and confluent. Thus, branching similarity is also preserved when applying concurrency reduction.

VI. RIP-OFF AND RE-ENCODE

Besides the basic algorithm previously described, there is an improvement that can be done as a post-processing step. The idea is very simple: hide one of the inserted signals (rip-off) and find a different solution (re-encode). This process is repeated until no further improvements are observed. This step improves the quality of the results, at the expense of a cost in execution time.

This technique exploits the fact that signals are inserted sequentially and some CSC conflicts might be resolved by more than one signal. Typically, the first inserted signals are eager to resolve a large amount of conflicts. But some of the conflicts may also be resolved later by new inserted signals. By ripping-off some of the first signals and re-encoding, the constraints of are relaxed, i.e., they number of CSC conflicts is smaller, and better solutions can be found. In some rare cases, it may even occur that ripping-off some signal does not introduce any CSC conflict, thus detecting that the signal was redundant.

For a fast estimation of the quality of the solutions, this work uses a cost function based on the one proposed in [9], which accounts for the number of essential literals, entry points and size of the excitation regions.

Algorithm 3: RIPOFFREENCODE(A)

```
input  : An ALTS with CSC property.
output: A re-encoded ALTS.
begin
    $C$ = costSolution($A$)
    do
        $improved$ = False
        $X$ = insertedSignals($A$)
        sortByEssentialLiterals($X$)
        /* in descending order                */
        foreach $x \in X$ do
            $B$ = hideSignal($A$, $x$)
            solveCSC($B$)
            $newC$ = costSolution($B$)
            if $newC < C$ then
                $improved$ = True
                $C$ = $newC$
                $A$ = $B$
                break
    while $improved$
    return $A$
```

Algorithm 3 shows the strategy proposed in this work. The algorithm consists of two nested loops. The external loop repeats the process until no further improvements are found. The set of inserted signals (X) is visited in descending order of essential literals, which is an estimation of the logic complexity of the signal. The rationale behind this order is that signals with more literals offer more opportunities for improvement after logic synthesis.

The inner loop stops when some improvement has been detected. After that, the cost of the signals is re-evaluated and the outer loop starts again. An algorithm for calculating essential literals is described at the appendix.

VII. PREVIOUS ART

We next discuss the main differences with the most relevant approaches proposed for asynchronous controllers working in input/output mode. We can distinguish two main categories:

- Structural methods working at Petri net level, such as MPSAT [7] and structural methods using integer-linear programming [6].
- State-based methods, such as Petrify [8] and PBASE [9].

Regarding the exploration of insertion points for the new signals, the main limitation of the structural methods is that the original specification acts as a *corset*. The new events must be *anchored* in existing nodes of the Petri net (or its unfolding). If two different Petri nets have the same reachability graph, the space of solutions is also different and a subset of the solutions available at ALTS level. Moreover, the insertion must be done in such a way that the causality relations can be expressed with the semantics of a Petri net. This has the disadvantage that many valid solutions are not explored. On the other hand, structural methods do not require to enumerate all possible markings, thus dealing effectively with the state explosion problem.

Petrify is a special case. The insertion of signals is done at state level, however the sets of states for insertion are built based on combinations of regions (that correspond to Petri net places). Petrify only uses simple combinations of regions that prevent the exploration of intricate solutions that could potentially be better.

PBASE, on the other hand, works exclusively at the state level. It uses a combination of SAT and pseudo-Boolean optimization in order to find high quality solutions. It requires to explicitly enumerate all states and encode a SAT formula that is satisfiable if and only if a solution is valid. A posterior pseudo-Boolean optimization step ensures that the solution returned is the optimum according to an heuristic. Since pseudo-Boolean optimization is NP-hard, this approach has the risk of falling into a worst-case scenario. This makes PBASE the most sensitive approach to the state explosion problem. On the other hand, its ability to encode any possible solution and the high quality heuristics used to find solutions allows it to often find solutions with the fewer number of literals than other approaches.

VIII. EXPERIMENTAL RESULTS

This section presents experimental results for the projection and recomposition technique, henceforth called SEPR (State Encoding using Projection and Re-composition). The signal insertion step is performed with PBASE, even though it is possible to use any other state graph based approach (like Petrify). Experimental results include a comparison of the method against PBASE as a baseline and MPSAT for large controllers. Versions with the rip-off and re-encode technique (SEPR-R) are also included.

In every case, the projection steps of the algorithm are performed until the ALTS satisfies all the following conditions:

- $|S| \times |\Sigma| < 500$.
- At least one signal has been hidden.

The last condition is included for the smallest controllers. Some of them are small enough to already satisfy the first condition. Hiding at least one signal guarantees that the technique is used in every instance.

The benchmarks are divided into three groups: small, medium and large. The following subsections present and discuss the results for the different groups, as well as experiments testing the scalability of the approach.

A. SMALL CONTROLLERS

Controllers in this group have been taken from [9] and have less than 1000 states (with the exception of c10). This experiment is performed to give a baseline comparison with PBASE, since it cannot be used for larger controllers due to the execution time. Additionally, a comparison with the rip-off and re-encode technique is included.

Table I shows the results, with column *Signals/Literals* reporting the number of state signals that were inserted and the number of literals of the Boolean equations (in factored form) after logic synthesis. *CPU(sec)* reports the CPU time required to solve CSC. The number of states ($|S|$) and input/output signals (I/O) are also reported. The table compares PBASE (PB), SEPR (SP) and SEPR-R (RR). In some cases, PBASE was not able to solve CSC in less than 10 hours. This is denoted as *Time* in the table.

A summary of the results for Table I can be found in Table II, which presents a pairwise comparison between different techniques. Row *Solved* reports the number of solved instances. The remaining data in the table only reports the total results for the benchmarks that were solved by both techniques under comparison (i.e. ignoring controllers not solved by both). *Ratio* reports the average ratio of literals between every pair of techniques.

The comparison between PBASE and SEPR shows a significant difference in execution time, without hardly sacrificing quality: the number of literals only increases by 1%. The addition of the rip-off technique has a very minor impact on quality, while increasing execution time. The main reason is because the number of inserted signals is small (less than 3 in most cases), giving few opportunities to explore different re-encodings. Thus, the rip-off technique is not well suited for small controllers. Nonetheless, all controllers were solvable with SEPR-R.

Although the work of this paper was not originally meant to be used for small controllers, the experiments show that the technique contributes to reduce runtime without having a significant impact on quality.

B. MEDIUM CONTROLLERS

In this experiment, the controllers have up to 14,000 states. This size is already out of the scope of the controllers manageable by PBASE. For this reason, MPSAT is used as reference.

The controllers come from different sources. Some of them (master-read versions) come from [9]. The *art(m,n)* are

TABLE I: Experimental results for small controllers. Comparing PBASE (PB), SEPR (SP) and SEPR-R (RR).

			CPU(sec)			Signals/Literals						CPU(sec)			Signals/Literals		
Example	I/O	$\|S\|$	PB	SP	RR	PB	SP	RR	Example	I/O	$\|S\|$	PB	SP	RR	PB	SP	RR
adfast	3/3	44	5.6	**1.2**	1.3	**2/14**	**2/14**	**2/14**	pla	0/3	12	0.2	**0.1**	0.1	**1/14**	2/16	2/16
alloc-outbound	4/3	17	0.8	**0.2**	0.3	**2/16**	**2/16**	**2/16**	ram-read-sbuf	5/5	36	1.2	0.3	**0.3**	**1/22**	**1/22**	**1/22**
c10	0/10	2046	32.7	**4.1**	15.4	**1/31**	2/32	**1/31**	sbuf-ram-write	5/5	58	9.1	**1.9**	2.1	**2/23**	**2**/24	**2**/24
c6	0/6	126	1.1	**0.2**	0.3	**1/19**	**1/19**	**1/19**	sbuf-read-ctl	2/4	14	0.2	**0.1**	**0.1**	**1/15**	**1/15**	**1/15**
duplicator	2/2	20	0.5	**0.2**	0.2	**2/13**	**2/13**	**2/13**	seq2	3/3	12	**0.1**	**0.1**	0.1	**1/8**	**1/8**	**1/8**
future	4/4	36	0.4	**0.1**	**0.1**	**1/18**	**1/18**	**1/18**	seq3	4/4	16	0.6	**0.2**	0.3	**2/14**	**2/14**	**2/14**
glc	2/1	17	**0.1**	0.1	0.2	**1/10**	**1**/11	**1**/11	seq4	5/5	20	1.4	**0.4**	0.6	**2/19**	**2/19**	**2/19**
lazy_ring.noncsc	5/3	160	27.6	**0.9**	1.0	**1/22**	**1/22**	**1/22**	seq8	9/9	36	108.7	**41.0**	56.6	**3**/44	5/**43**	5/**43**
mmu0	4/4	174	89.1	**1.8**	2.9	**3/28**	**3**/29	**3**/29	seq-mix	4/4	20	2.0	**0.5**	0.9	**3/18**	**3**/20	**3**/20
mmu1	4/4	82	8.1	**1.0**	2.1	2/25	**2/24**	**2/24**	vbe4a.nousc	3/3	58	5.1	**1.4**	2.2	**3/18**	**3/18**	**3/18**
mod4_counter	1/2	16	0.3	**0.1**	0.2	**2/26**	**2/26**	**2/26**	vbe5a	3/3	44	4.2	**0.7**	0.9	**2/14**	**2/14**	**2/14**
mr0	5/6	302	Time	**1.2**	2.7	-/-	4/31	**3/30**	vbe6a.nousc	4/4	128	41.0	**1.2**	1.7	**2/30**	**2/30**	**2/30**
mr1	4/5	190	91.6	**7.5**	11.1	**3/26**	**3/26**	**3/26**	vbe6x.nousc	3/3	48	4.4	**0.3**	0.3	**2**/23	**2/22**	**2/22**
nak-pa	4/5	56	0.7	0.2	**0.2**	**1/18**	**1/18**	**1/18**	vme_read	8/6	251	16.2	**0.7**	0.7	**1/30**	**1**/31	**1**/31
nowick	3/2	18	0.2	0.1	**0.1**	**1/13**	**1/13**	**1/13**	vme_read_write	3/3	28	1.0	0.5	**0.3**	**1/22**	**1/22**	**1/22**
par2	3/3	28	4.4	**0.5**	0.8	**2/16**	**2/16**	**2/16**	vme_write	8/6	817	Time	**1.0**	1.0	-/-	**1/36**	**1/36**
par4	5/5	628	Time	**3.6**	10.7	-/-	**4/32**	**4/32**	vmebus	3/3	24	0.5	**0.2**	**0.2**	**1/19**	**1/19**	**1/19**

TABLE II: Summary for the benchmarks in Table I.

	PB	SP	PB	RR	SP	RR
Solved	31	34	31	34	34	34
CPU (sec)	459	68	459	104	74	118
Signals	53	57	53	56	66	64
Literals	628	634	628	633	733	731
Ratio	**1.00**	**1.01**	**1.00**	**1.01**	**1.00**	**1.00**

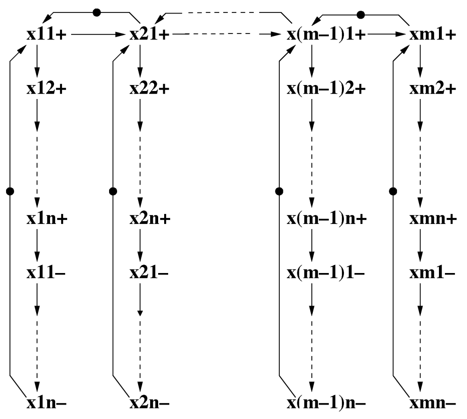


FIGURE 10: Art(m,n). Source: [20].

parameterized controllers from [20]. They model a synchronization of m pipelines, as shown by the STG depicted in Figure 10. These controllers have a high number of states and require a moderately high number of signals to guarantee CSC.

Another set of parameterized controllers is *PpArb(m,n)*, obtained from [21]. They model m pipelines synchronized with arbitration. Figure 11 shows an example for *PpArb(2,3)*. These controllers are highly concurrent and have a large set of states, but a comparatively small number of signals. They can be solved with few signal insertions.

The *ParMix(m,n)* controllers are based on the ones presented in [20]. These controllers show a handshake of sequencers, parallelizers and mixers, as represented by Figure 12. The original controllers in [20] did not have any CSC conflict. The ones presented here have been modified (by hiding internal signals) such that the sequencer and every parallelizer have conflicts. The result is a controller with a high number of signals and CSC conflicts.

Finally, the *SeqPar(n)* controllers are introduced in this work. Like the *ParMix(m,n)*, they represent a handshake of smaller controllers. A *SeqPar(n)* controller represents an n-level tree of alternating handshakes of sequencers and controllers. Fig. 13 shows an example with three levels. Since every parallelizer and sequencer has CSC conflicts, this class of controllers also contains a high number of signals and CSC conflicts.

Table III shows results for this experiment. The codeword *Time* is used when a controller could not be solved in less than 10 hours. The codeword *Fail* marks an instance in which a solution could not be found. A summary for Table III can be found in Table IV.

The results show that SEPR generates slightly better results than MPSAT, even before the rip-off technique. In general, the execution time is slightly higher than MPSAT, with the exception of the controller *ParMix(2,4)*. This controller biases the total execution time for MPSAT in Table IV. Nonetheless, this result is important because it hints at a trend in the *ParMix* and *SeqPar* controllers: MPSAT takes too long to solve these classes of problems and hits the 10-hour timeout for most of them. A possible explanation is later discussed in Section VIII-D.

Another singularity is the *master-read* controller. This controller is the original specification of *master-read2*, in-

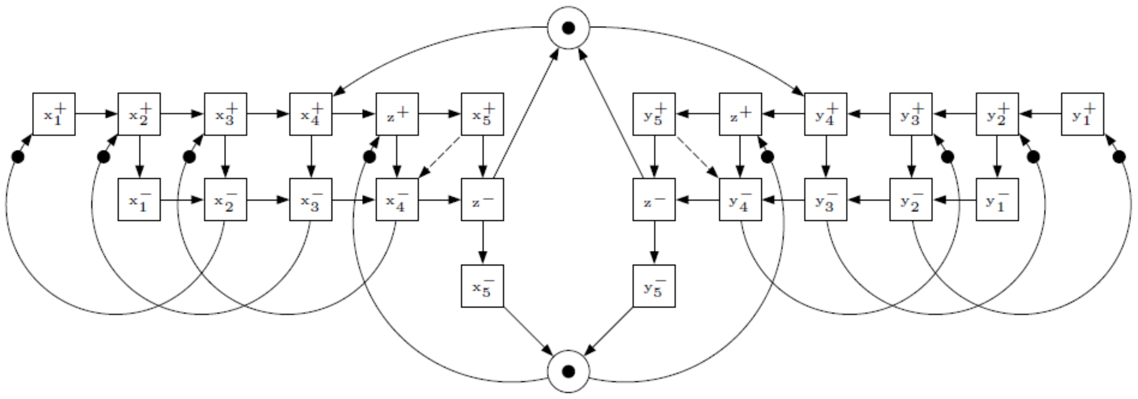


FIGURE 11: PpArb(2,3). Source [21].

TABLE III: Experimental results for medium controllers. Comparing MPSAT (MP), SEPR (SP) and SEPR with Rip-off (RR).

			CPU(sec)			Signals/Literals		
Example	I/O	$\lvert S\rvert$	MP	SP	RR	MP	SP	RR
art(3,4)	0/12	2048	**4.1**	16.5	21.6	6/54	**4/49**	**4/49**
art(3,5)	0/15	4000	**10.2**	12.9	16.8	6/57	**4/53**	**4/53**
art(3,6)	0/18	6912	38.7	**23.5**	28.0	6/60	**4/56**	**4/56**
art(4,3)	0/12	10368	**8.7**	39.5	81.8	10/70	**5/69**	**5/69**
master-read	6/7	8932	Fail	**42.9**	160.7	-/-	9/74	**6/59**
master-read2	0/13	8932	**15.6**	147.6	201.1	**5**/75	7/**69**	7/**69**
master-read.1098	6/7	1098	**3.6**	4.4	12.4	6/43	6/44	**4/39**
PpArb(2,3)	2/9	1088	0.3	**0.1**	0.2	**1/39**	**1**/42	**1**/42
PpArb(3,3)	3/13	14336	**0.3**	2.9	4.5	**2/61**	**2**/69	**2**/69
sis-master-read	6/7	1882	0.4	**0.4**	0.5	**1**/39	**1/37**	**1/37**
ParMix(2,4)	0/38	13852	766.8	**53.4**	99.2	**5**/123	6/**121**	6/**121**
ParMix(3,3)	0/46	3796	Time	**76.8**	134.3	-/-	**6/157**	**6/157**
SeqPar(4)	0/72	7452	Time	**194.5**	817.2	-/-	11/210	**9/195**

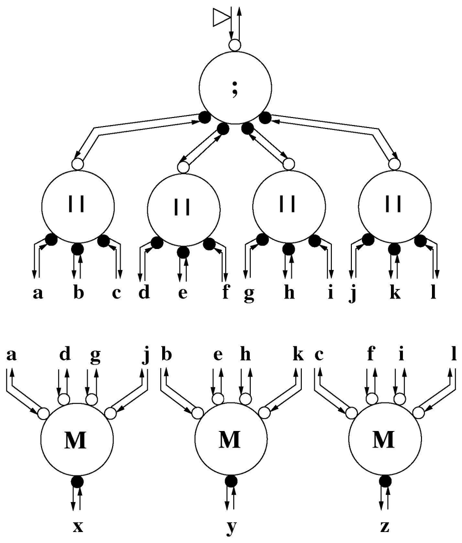


FIGURE 12: ParMix(4,3). Source [20].

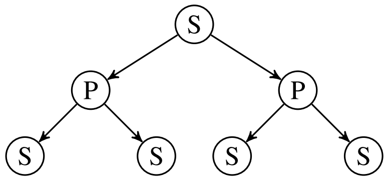


FIGURE 13: SeqPar(3).

TABLE IV: Summary for the benchmarks in Table III.

	MP	SP	MP	RR	SP	RR
Solved	10	13	10	13	13	13
CPU (sec)	849	301	849	466	615	1578
Signals	48	40	48	38	66	59
Literals	621	609	621	604	1050	1015
Ratio	**1.00**	**0.98**	**1.00**	**0.97**	**1.00**	**0.97**

cluding the inputs and outputs (all signals in *master-read2* are artificially declared as outputs). The presence of inputs reduces the space of valid solutions since the input properness property prevents the insertion of a signals triggering inputs. While MPSAT can solve *master-read2*, it fails to find a solution for *master-read*. This highlights the increased power of the state-based techniques to find intricate solutions in highly restrictive specifications.

Finally, the rip-off technique shows an overall reduction of 2% in the number of literals with respect to the base approach, at the cost of a higher execution time. The higher number of inserted signals with respect to the small controllers allows this technique to improve the SEPR solutions.

C. LARGE CONTROLLERS

The last experimental results are for large controllers, which was the main motivation for this work. These controllers can have up to several million of states. The aim of this experiment is to show the scalability of the approach presented in this paper.

The controllers in this test come from the same sources as the ones in the previous results. Table V reports the results for large controllers, which are summarized in Table VI.

SEPR can solve problems up to 4.5 million states in a reasonable time. The rip-off technique significantly increases the execution time, even more than in previous results. This is because there are more candidates to rip-off, which also increases the opportunities to generate better results. This last approach allows for solutions with higher quality than those of MPSAT. Every instance can be solved with the SEPR and SEPR-R.

Even though MPSAT uses structural methods, it solves CSC using a SAT formulation of the problem [7]. The runtime highly depends on the size of the SAT formula, which is mainly determined by the size of the unfolding and the number of signals. Although the unfolding can grow exponentially under the presence of multiple choices in the specification, in practice the number of signals is the one that has the largest impact on MPSAT runtime. The following section discusses the scalability of different approaches.

It is also important to note that only examples suitable for MPSAT have been selected, which need to have an underlying safe Petri net. These constraints do not apply for state-based methods.

D. SCALABILITY

This section studies the scalability of SEPR with regard to MPSAT, with the goal of comparing a state-based method with a structural one. The experiments are performed with three suites of benchmarks: *Sequencer(n)*, *Art(m,n)* and *Parallelizer(n)*. The circuits have been scaled with the parameter n. In the case of *Art(m,n)*, m has been set at 3. The following table shows how these circuits grow with n:

	Seq(n)	*Art*(3,n)	*Par*(n)
Signals	$2n+2$	$3n$	$2n+2$
States	$4n+4$	$32n^3$	5^n+3

Signals grow linearly with n in all cases. The main difference is in the size of the set of states. For *Seq*, it grows linearly, whereas for *Art* and *Par* the growth is cubic and exponential, respectively.

Fig. 14 reports the execution time of these benchmarks for MPSAT, SEPR and SEPR-R. The x-axis represents n and the y-axis represents the execution time in seconds (log scale). Table VII reports the total sum of literals after logic synthesis for all controllers of every class.

Fig. 14a depicts the results for *Seq*. The dashed line represents a linear regression of SEPR, with $R^2 = 0.946$. SEPR and SEPR-R manifest a linear asymptotic behavior, whereas MPSAT hits a computational wall around $n = 20$. The main reason is that MPSAT does not scale well with the number of signals.

The results for *Art* are reported in Fig. 14b. In this case, the dashed line is a cubic polynomial regression of SEPR, with $R^2 = 0.988$. This is consistent with the cubic polynomial growth of the number of states. MPSAT shows an exponential behavior, mostly dominated by the number of signals.

Finally, Fig. 14c shows results for *Par*. In this case, the complexity of the ALTS is dominated by the number of states, rather than the number of signals. The dashed line represents an exponential regresion of SEPR[3]. Clearly, MPSAT overtakes the state-based methods since the number of states grows much faster than the number of signals. Working with the unfolding of a Petri net, rather than its reachability set, is a clear advantage in this case.

MPSAT is more scalable for large state spaces that can be succinctly represented by a Petri net. However, the runtime grows exponentially with the number of signals. The main reason is the way that MPSAT estimates the logic complexity of the circuit, using a quadratic number of SAT variables to encode the trigger relations between pairs of signals [7].

E. FINAL REMARKS

The results show a good picture of how SEPR scales. For small controllers, it reduces runtime while maintaining quality. For medium controllers, the quality of the solution and runtime are slightly better than the structural methods.

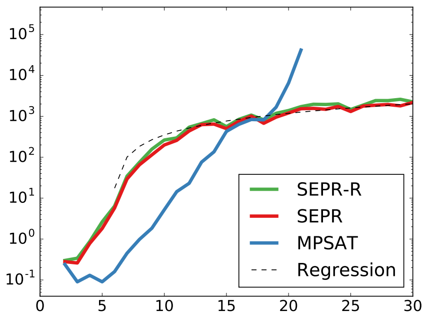
For controllers with a large number of signals, SEPR can go much beyond the complexity wall hit by other tools (e.g., MPSAT or petrify). The base version of the tool, SEPR, sometimes provides solutions with slightly lower quality than MPSAT, but the re-encoding strategy using rip-off gives an opportunity to improve the results, specially in those controllers that require a larger number of encoding signals. In fact, it generates the best results for most cases, with the exceptions of the *PpArb* class of controllers, which are solved with just one signal.

Finally, the strongest advantages of SEPR are in the number of problems solved and the scalability of the approach. Structural methods depend on the Petri net structure, which limits the solutions that can be found. In the case of MPSAT, for example, it cannot solve unsafe nets. But even when safe Petri nets are used (as in the case of the benchmarks

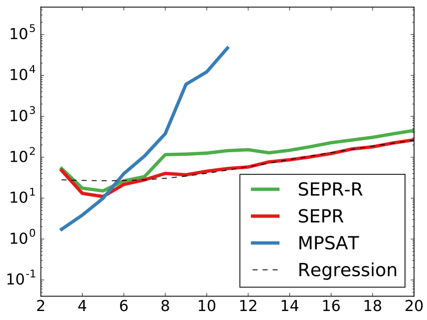
[3]The regression is on the order of 4.5^n (states grow on the order ofÂň5^n). Given the small number of points and the dominance of the large values, the regression may not be sufficiently meaningful. However it helps to hypotesize the exponential relationship with the state space.

TABLE V: Experimental results for large controllers. Comparing MPSAT (MP) and SEPR (SP).

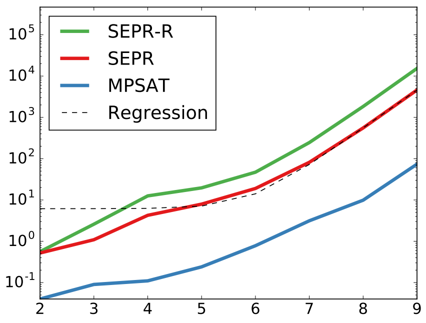
			CPU(sec)			Signals/Literals		
Example	I/O	$\|S\|$	MP	SP	RR	MP	SP	RR
art(4,4)	0/16	$0.3 \cdot 10^5$	**17.0**	38.3	129.7	9/76	**6/74**	**6/72**
art(5,4)	0/20	$5.2 \cdot 10^5$	**25.4**	677.7	1676.2	10/104	**7/100**	**7/100**
art(5,5)	0/25	$16 \cdot 10^5$	**225.8**	2210.3	6422.0	12/105	**8**/115	**8/102**
par8	9/9	$3.9 \cdot 10^5$	**9.9**	554.7	1833.7	**8/64**	**8/64**	**8/64**
PpArb(2,6)	2/15	$0.7 \cdot 10^5$	**1.6**	7.5	8.6	**1/69**	**1**/72	**1**/72
PpArb(2,9)	2/21	$44.6 \cdot 10^5$	**6.5**	808.0	826.8	**1/99**	**1**/102	**1**/102
ParMix(4,4)	0/86	$1.1 \cdot 10^5$	Time	**480.2**	1911.6	-/-	**11**/313	**11/298**
ParMix(5,4)	0/110	$2.2 \cdot 10^5$	Time	**812.9**	7126.0	-/-	16/411	**14/387**
SeqPar(5)	0/126	$2.4 \cdot 10^5$	Time	**892.2**	4505.0	-/-	12/396	**10/394**



(a) Seq(n)



(b) Art (3,n)



(c) Par(n)

FIGURE 14: Runtime growth, in seconds (y-axis) with the size of the ALTS, defined by n (x-axis).

TABLE VI: Summary for large controllers.

	MP	SP	MP	RR	SP	RR
Solved	6	9	6	9	9	9
CPU (sec)	286	4296	286	10897	6482	24440
Signals	41	31	41	31	70	66
Literals	517	527	517	512	1647	1591
Ratio	**1.00**	**1.02**	**1.00**	**0.99**	**1.00**	**0.97**

TABLE VII: Total number of literals for controller classes.

	Seq(n)	*Art*(3,n)	*Par*(n)
MPSAT	1339	567	352
SEPR	1240	565	352
SEPR-R	1220	521	352

presented here), some other limitations might arise. As for scalability, this approach grows linearly with the number of signals and states. In the case of state explosion typical of high concurrency, this limits the size of the controllers than can be solved (to the order of 10^6 states). But when the controllers have large number of signals, results show that SEPR still manages to grow linearly, as opposed to the exponential growth of MPSAT.

IX. CONCLUSIONS

This work has presented a novel technique to address the problem of state encoding for large asynchronous controllers. The approach allows to project a large specification onto a subset of signals and obtain a smaller one suitable to be handled by state-based encoding techniques. The complete asynchronous controller is recovered by re-composing the original specification with the projected solution.

Results show that asynchronous controllers of several million states are now within reach of state-based encoding techniques. Furthermore, it can speed up the encoding for controllers of smaller sizes. This allows state-based techniques to effectively compete with structural methods and handle controllers that can be generated from different formalisms for which no encoding tools exist yet.

.

APPENDIX A PROOF OF THEOREM 1

Proof.

We will denote the set of states of A_i as S_i and we will use $s_i, s_i', s_i'', \ldots$, to denote different states in S_i. By construction of $A_3 = A_1 \times A_2$, every state in S_3 is a pair $s_3 = \langle s_1, s_2 \rangle$ with $s_1 \in S_1$ and $s_2 \in S_2$.

Let us first prove that $A_3 \approx_{\sigma_2} A_1$. Consider $A_4 = \mathit{silence}(A_3, \sigma_2)$. Then, it suffices to show that $A_1 \approx A_4$.

Since A_4 has the same states as A_3, we can also represent every state in S_4 as a pair $s_4 = \langle s_1, s_2 \rangle$. Let us define a binary relation R between S_1 and S_4 as follows: for every state $s_4 = \langle s_1, s_2 \rangle$, $s_1 R s_4$. It is trivial to see that this relation exists for every $s_4 \in S_4$. Similarly, since $A_1 \approx_{\sigma_1 \cup \sigma_2} A_2$, it follows that by construction of the synchronous product the relation R also exists for every $s_1 \in S_1$. We only need to prove that R is a branching bisimulation, i.e.,

1) Whenever $s_1 R s_4$ and $s_1 \xrightarrow{a} s_1'$, then either $a = \tau$ and $s_1' R s_4$, or there exists a path $s_4 \overset{\tau^*}{\Longrightarrow} s_4'' \xrightarrow{a} s_4'$ such that $s_1 R s_4''$ and $s_1' R s_4'$.
2) Whenever $s_1 R s_4$ and $s_4 \xrightarrow{a} s_4'$, then either $a = \tau$ and $s_1 R s_4'$, or there exists a path $s_1 \overset{\tau^*}{\Longrightarrow} s_1'' \xrightarrow{a} s_1'$ such that $s_1'' R s_4$ and $s_1' R s_4'$.

1) Since A_1 is τ-free, we know that $a \neq \tau$. If $s_1 \xrightarrow{a} s_1'$ and $s_4 = \langle s_1, s_2 \rangle$ then the product also creates a state $s_4' = \langle s_1', s_2' \rangle$ and a path $s_4 \xrightarrow{a} s_4'$. In case $a \notin \Sigma_2$, then $s_2 = s_2'$. Therefore, τ^* is empty and $s_4 = s_4''$. By the definition of R, we have that $s_1 R s_4''$ and $s_1' R s_4'$.

2) Let us assume $s_4 = \langle s_1, s_2 \rangle$. We need to consider two cases: $a = \tau$ and $a \neq \tau$. If $a = \tau$ then τ is hiding a signal in $\Sigma_2 \setminus \Sigma_1$. Therefore, the product generates the state $s_4' = \langle s_1, s_2' \rangle$, since A_1 does not move and, thus, $s_1 R s_4'$. If $a \neq \tau$ then A_1 and A_2 synchronize with a and the product generates the state $s_4' = \langle s_1', s_2' \rangle$. Therefore, the path $s_1 \xrightarrow{a} s_1'$ exists in A_1 and $s_1' R s_4'$. Notice also that $s_1'' = s_1$ and, thus, $s_1'' R s_4$.

By symmetry, $A_3 \approx_{\sigma_1} A_2$ can be proved identically. □

APPENDIX B COMPUTATION OF ESSENTIAL LITERALS

A signal a is said to be an essential literal for a signal b if there exits a pair of states $s, s' \in S$ such that:

- $s \in ON_b$ and $s' \in OFF_b$
- $\forall c \in \Sigma \mid c \neq a : s(c) = s'(c)$
- $s(a) \neq s'(a)$

In this case, if $s(a) = 1$ and $s'(a) = 0$, a is a positive essential literal of b. If $s(a) = 0$ and $s'(a) = 1$, then a is a negative essential literal of b.

Essential literals are important because they are guaranteed to be in the support of the Boolean function. In particular, if a is positive essential literal of b, a will be a literal for the function of b after synthesis. If a is a negative essential literal, then $\neg a$ will be a literal for the function of b after synthesis. A more in-depth discussion about the correlation between essential literals and literals after synthesis can be found in [9].

Given a signal a, it is possible to efficiently compute the set of signals for which a is essential. This can be accomplished by grouping all the states that have the same encoding (minus the code for signal a) and checking, for each non-input signal, which ones meet the condition for a to be essential. If the encodings are stored in a hash table, the worst-case complexity is in the order of $O(|S| \times |\Sigma|)$. Nonetheless, by exploiting bitwise and vectorial instructions in actual hardware, a linear cost $O(|S|)$ can be obtained, as long as $|\Sigma|$ is on the order of the word size.

Finally, to compute the essential literals for all signals, the previously described computation needs to be executed for every signal. The cost of finding all the essential literals is $O(|S| \times |\Sigma|^2)$, or $O(|S| \times |\Sigma|)$ if the size of $|\Sigma|$ is on the order of the word size.

REFERENCES

[1] J. Sparsø and S. Furber, Eds., Principles of Asynchronous Circuit Design: A Systems Perspective. Kluwer Academic Publishers, 2001.

[2] D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, and A. Yakovlev, "Benefits of asynchronous control for analog electronics: Multiphase buck case study," in Proc. Design, Automation and Test in Europe (DATE), Mar. 2017, pp. 1751–1756.

[3] K. Y. Yun and D. L. Dill, "Automatic Synthesis of Extended Burst-Mode Circuits: Part I (Specification and Hazard-Free Implementation)," IEEE Transactions on Computer-Aided Design, vol. 18, no. 2, pp. 101–117, Feb. 1999.

[4] L. Y. Rosenblum and A. V. Yakovlev, "Signal graphs: from self-timed to timed ones," in Proceedings of International Workshop on Timed Petri Nets. Torino, Italy: IEEE Computer Society Press, Jul. 1985, pp. 199–207.

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

[6] J. Carmona and J. Cortadella, "Encoding large asynchronous controllers with ILP techniques," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 1, pp. 20–33, 2008.

[7] V. Khomenko, "Efficient automatic resolution of encoding conflicts using STG unfoldings," IEEE Transactions on VLSI Systems, vol. 17, no. 7, pp. 855–868, 2009.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "A region-based theory for state assignment in speed-independent circuits," IEEE Transactions on Computer-Aided Design, vol. 16, no. 8, pp. 793–812, Aug. 1997.

[9] A. Moreno and J. Cortadella, "State encoding of asynchronous controllers using pseudo-boolean optimization," in Asynchronous Circuits and Systems (ASYNC), 2018 24rd IEEE International Symposium on. IEEE, 2018, pp. 9–16.

[10] D. Wist, R. Wollowski, M. Schaefer, and W. Vogler, "Avoiding irreducible CSC conflicts by internal communication," Fundamenta Informaticae, vol. 95, no. 1, pp. 1–29, 2009.

[11] J. F. Groote, D. N. Jansen, J. J. A. Keiren, and A. J. Wijs, "An $O(m \log n)$ Algorithm for Computing Stuttering Equivalence and Branching Bisimulation," ACM Trans. Comput. Logic, vol. 18, no. 2, pp. 13:1–13:34, Jun. 2017.

[12] R. Milner, Communication and concurrency. Prentice hall New York etc., 1989, vol. 84.

[13] R. J. V. Glabbeed and W. P. Weikland, "Branching time and abstraction in bisimulation semantics," Journal of the ACM, no. 3, pp. 555–600, May 1996.

[14] J. Groote and M. Mousavi, Modeling and Analysis of Communicating Systems. The MIT Press, 2014.

[15] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man, "A generalized state assignment theory for transformations on signal transition graphs," Journal of VLSI signal processing systems for signal, image and video technology, vol. 7, no. 1-2, pp. 101–115, 1994.

[16] R. J. van Glabbeek, "The linear time - branching time spectrum," in CONCUR '90 Theories of Concurrency: Unification and Extension, J. C. M. Baeten and J. W. Klop, Eds. Springer Berlin Heidelberg, 1990, pp. 278–297.

[17] J. F. Groote and J. van de Pol, "State space reduction using partial τ-confluence," in International Symposium on Mathematical Foundations of Computer Science. Springer, 2000, pp. 383–393.

[18] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Automatic handshake expansion and reshuffling using concurrency reduction," in Proc. of HWPN, vol. 98, 1998, pp. 86–110.

[19] P. Vanbekbergen, G. Goossens, F. Catthoor, and H. J. De Man, "Optimized synthesis of asynchronous control circuits from graph-theoretic specifications," IEEE transactions on computer-aided design of integrated circuits and systems, vol. 11, no. 11, pp. 1426–1438, 1992.

[20] J. Carmona, J.-M. Colom, J. Cortadella, and F. García-Vallés, "Synthesis of asynchronous controllers using integer linear programming," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 25, no. 9, pp. 1637–1651, 2006.

[21] V. Khomenko, M. Koutny, and A. Yakovlev, "Detecting state encoding conflicts in STG unfoldings using SAT," Fundamenta Informaticae, vol. 62, no. 2, pp. 221–241, 2004.