State Encoding of Asynchronous Controllers using Pseudo-Boolean Optimization

Alberto Moreno and Jordi Cortadella
Department of Computer Science, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain.

***Abstract*—State encoding of asynchronous controllers is a challenging problem that faces a vast space of solutions. Subtle differences in the insertion of signals may result in significant variations in the complexity of the logic. This paper proposes a novel approach that models the encoding problem as Pseudo-Boolean formula. A cost function that estimates the complexity of the logic is incorporated, where the estimator of essential literals becomes one of the most important terms of the function. The new approach has been tested in 175 benchmarks with encoding conflicts, including 127 four-phase latch controllers. The presence of logic estimators in the formula contributes to an average reduction of 43% in literals when compared to a plain SAT version of the problem, at the expense of a longer runtime. When comparing to the region-based approach in petrify, an average reduction of 14% in literals is obtained.**

I. Introduction

State encoding is one of the critical problems during the synthesis of asynchronous control circuits. Several methods have been proposed in the past, either for circuits working in fundamental mode [1] or input/output mode [2], among others. In the latter case, the concurrency between input and output events imposes more severe constraints on the insertion of internal signals to disambiguate encoding conflicts. What makes encoding difficult is the preservation of the implementability properties of the specification (e.g., consistency and persistence) after the insertion of new events.

In this paper we will face the encoding problem in its most generic form, i.e., using state-based models (state graphs) in which all possible interleavings of concurrent events are explicitly represented. State graphs (SGs) can be derived from higher level formalisms such as Signal Transition Graphs (STGs) or Burst-Mode (BM) machines.

The space of configurations for state encoding is huge and similar solutions may result in significantly different logic complexity. One of the challenges in solving the problem is finding low-complexity correct solutions.

This paper proposes an approach based on satisfiability (SAT) with two main features: (1) all possible solutions for the encoding problem are represented by one Boolean formula and (2) simple estimators of logic complexity are added to the formula in such a way that high-quality solutions can be obtained by Pseudo-Boolean optimization.

The work goes beyond a previous SAT-based approach presented in [3], both in the space of explored solutions and in the estimation of logic complexity. The results obtained by our method shows that still a tangible margin for improvement was left by the best previous approaches implemented in petrify [2] or MPSAT [4].

II. Overview

Let us consider the following sequence of events that models the behavior of a controller with $\{r_0, a_1, a_2\}$ and $\{a_0, r_1, r_2\}$ as input and output signals, respectively:

$$(r_0^+ \bullet r_1^+\ a_1^+\ r_1^-\ a_1^- \bullet r_2^+\ a_2^+\ a_0^+\ r_0^-\ r_2^-\ a_2^-\ a_0^-)*$$

The bullets $\bullet$ represent a pair of states with the same encoding separated by a complementary subsequence of events ($r_1^+\ a_1^+\ r_1^-\ a_1^-$). Solving the encoding problem requires the insertion of a signal x with an event that breaks this subsequence.

Given that a_1 is an input signal, a new event (e.g., x^+) can only be inserted between a_1^+ and r_1^- in order to maintain the handshaking protocol with the environment. Still, there is some freedom for the insertion of the complementary event x^-. Let us consider three different solutions:

$$(r_0^+\ r_1^+\ a_1^+\ \mathbf{x^+}\ r_1^-\ a_1^-\ r_2^+\ a_2^+\ a_0^+\ r_0^-\ r_2^-\ a_2^-\ \mathbf{x^-}\ a_0^-)* \quad (1)$$

$$(r_0^+\ r_1^+\ a_1^+\ \mathbf{x^+}\ r_1^-\ a_1^-\ r_2^+\ a_2^+\ a_0^+\ r_0^-\ \mathbf{x^-}\ r_2^-\ a_2^-\ a_0^-)* \quad (2)$$

$$(r_0^+\ r_1^+\ a_1^+\ \mathbf{x^+}\ r_1^-\ a_1^-\ r_2^+\ a_2^+\ \mathbf{x^-}\ a_0^+\ r_0^-\ r_2^-\ a_2^-\ a_0^-)* \quad (3)$$

A well-established estimator of the complexity of a logic circuit is the number of literals of the Boolean equations after logic minimization. We use the same criterion in this paper.

The state encoding problem faces a vast space of solutions. The challenge is to find the ones that lead to simpler circuits without resorting to logic minimization during the exploration.

This paper proposes a SAT-based approach. in which the main contribution is the incorporation of logic complexity estimators in the same formula. The most important estimator used in this paper is the number of *essential literals*. Informally, if the encoding of two states, s_1 and s_2, only differs in one signal value (e.g., $z = 1$ in s_1, $z = 0$ in s_2), and s_1 and s_2 belong to the on- and off-set of the next-state function for signal x, respectively, then z is essential for x, i.e., z must be in the support of x. The important aspect is that the presence of essential literals is a local property (between pairs of states) that can be efficiently encoded in a Boolean formula. Moreover, the number of essential literals can be minimized by using Pseudo-Boolean optimization [5].

We have observed that there is a very high correlation between the number of essential literals and the final literals of a function represented as a factored form. Fig. 1 depicts a plot comparing essential vs. actual literals for a large number of controllers. The solid line represents the ideal prediction (essential = actual). The red dashed line represents a linear regression ($R^2 = 0.91$), that indicates that the number of essential literals is a good estimator.

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/ASYNC.2018.00013

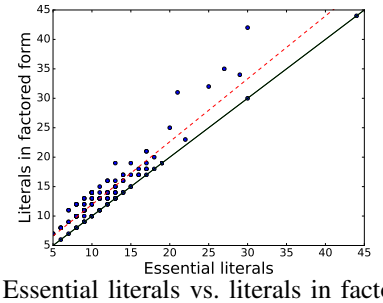


Fig. 1: Essential literals vs. literals in factored form.

The following table reports the logic equations for the previous solutions of the example. The number of essential literals is represented in brackets and is a lower bound (and a good estimator) of the number of literals of the equations.

	Solution (1)	Solution (2)	Solution (3)
$r_1 =$	[2] $r_0\bar{x}$	[2] $r_0\bar{x}$	[2] $r_0\bar{r}_2\bar{x}$
$r_2 =$	[3] $r_0\bar{a}_1x$	[2] $\bar{a}_1x$	[3] $\bar{a}_1x + r_0a_2$
$a_0 =$	[2] $a_2 + a_0x$	[1] a_2	[2] $a_2\bar{x}$
$x =$	[3] $a_1 + a_2 + \bar{a}_0x$	[3] $a_1 + r_0x$	[3] $a_1 + a_2x$

Besides essential literals, there are other estimators that also have some correlation with the complexity of the logic: size of the don't care set and number of entry points of the excitation regions. These estimators will be discussed later in the paper.

III. Background

This section reviews some known concepts on Boolean functions, state graphs and speed-independent circuits.

A. Boolean Functions

An incompletely specified function (ISF) is a functional mapping $F : \mathbb{B} \rightarrow \{0, 1, -\}$, 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. State graphs

The work presented in this paper uses State Graphs (SG), that can be derived from higher-level formalisms such as STGs or BM machines. An SG is a 4-tuple (S, T, Σ, δ) such that S is a finite set of states, $T \subseteq S \times S$ is the set of transitions and $\Sigma = \text{In} \cup \text{Out} \cup \text{Int}$ is a set representing the input, output and internal signals. Finally, $\delta : T \rightarrow \Sigma \times \{+, -\}$ is a labelling function that associates a signal transition to each transition.

Rising and falling transitions of signal a between states s_1 and s_2 are represented by $s_1 \xrightarrow{a^+} s_2$ and $s_1 \xrightarrow{a^-} s_2$, respectively. A generic transition of signal a is represented by $s_1 \xrightarrow{a} s_2$. A signal a is said to be *enabled* at state s_1 if there exists a transition of the form $s_1 \xrightarrow{a} s_2$ for some $s_2 \in S$. Henceforth, we will assume that the SGs used for synthesis are *deterministic*.

Signal transitions implicitly induce a *state encoding*. Thus, $s(a) = 1$ or $s(a) = 0$ represent the fact that a has value 1 or 0 in state s, respectively. In particular, $s_1 \xrightarrow{a^+} s_2$ implies $s_1(a) = 0$ and $s_2(a) = 1$. Similarly, $s_1 \xrightarrow{a^-} s_2$ implies $s_1(a) = 1$ and $s_2(a) = 0$. If $s_1 \xrightarrow{b} s_2$, for any $b \neq a$, then $s_1(a) = s_2(a)$. An SG is said to be *consistent* if these rules can be applied to every signal and state without any contradiction. $code(s) = (s(a_1), s(a_2), ..., s(a_n))$, with $\Sigma = \{a_1, a_2, ..., a_n\}$ defines the encoding of state s.

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 $ON_a = ER_a^+ \cup QR_a^+$ 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:

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

C. Speed independence

A signal a triggers a signal b if there is a transition $s_1 \xrightarrow{a} s_2$ such that b is enabled in s_2 and not enabled in s_1. Conversely, a disables b if b is enabled in s_1 and not in s_2. An SG 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 SG satisfies the *Unique State Coding* (USC) condition if every state in S is assigned a unique binary code, i.e.,

$$\forall s_1, s_2 \in S : s_1 \neq s_2 \implies code(s_1) \neq code(s_2).$$

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

$$\forall s_1, s_2 \in S, \forall a \in \text{Out} \cup \text{Int} :$$
$$(s_1 \in ON_a \wedge s_2 \in OFF_a) \implies code(s_1) \neq code(s_2)$$

An SG is said to be *commutative* if for any state s_1 in which the traces ab and ba are enabled, the firing of the traces leads to the same state, i.e., if $s_1 \xrightarrow{a} s_2$, $s_2 \xrightarrow{b} s_4$, $s_1 \xrightarrow{b} s_3$ and $s_3 \xrightarrow{a} s_5$, then $s_4 = s_5$.

An important result on speed independence is as follows [2]:

> *a deterministic SG with CSC that satisfies persistency and commutativity is implementable as a speed-independent circuit.*

An additional important property is *input-properness*. An SG 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.

Solving the state encoding problem is based on inserting new signals to disambiguate CSC violations. The insertion of new signals proposed in this paper preserves the conditions for speed-independence and input-properness.

D. Signal Insertion

The insertion of a new *internal signal* into an existing SG is now described. This transformation preserves *trace equivalence*. An in-depth explanation signal insertion can be found in [2], [3].

Henceforth, the new inserted signal will be named $x \notin \Sigma$, whereas the signals from the original SG will be named $a, b \in \Sigma$. The signal insertion process requires all states in S to be partitioned into four sets[1]: ER^+, ER^-, QR^+ and QR^-. These sets will determine the future ERs and QRs of x.

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

$$\begin{aligned} EXIT = \{ & s_1 \to s_2 \mid \\ & (ER^+(s_1) \wedge \neg ER^+(s_2)) \vee (ER^-(s_1) \wedge \neg ER^-(s_2))\} \end{aligned}$$

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

$$\begin{aligned} CONC = \{ & s_1 \to s_2 \mid \\ & (ER^+(s_1) \wedge ER^+(s_2)) \vee (ER^-(s_1) \wedge ER^-(s_2))\} \end{aligned}$$

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:

$$\begin{aligned} T_x &= \{s \to \hat{s} : s \in ER\} \\ T_d &= \{\hat{s_1} \to s_2 : s_1 \to s_2 \in EXIT\} \\ T_c &= \{\hat{s_1} \to \hat{s_2} : s_1 \to s_2 \in CONC\} \end{aligned}$$

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

The new SG $(S', T', \Sigma', \delta')$, obtained after the insertion of x in the original SG $(S, T, \lambda_S, \lambda_E)$ is defined as:

- $S' = S \cup \hat{S}$
- $T' = (T \cup T_x \cup T_d \cup T_c) \setminus EXIT$
- $\Sigma' = \Sigma \cup \{x\}$
- The labeling function defined as:
  - $\delta'(s \to \hat{s}) = x^+$ if $s \in ER^+$
  - $\delta'(s \to \hat{s}) = x^-$ if $s \in ER^-$
  - $\delta'(\hat{s}_1 \to \hat{s}_2) = \delta(s_1 \to s_2)$
  - $\delta'(\hat{s}_1 \to s_2) = \delta(s_1 \to s_2)$
  - $\delta'(s_1 \to s_2) = \delta(s_1 \to s_2)$

Fig. 2 shows an example of signal insertion on a fragment of an SG. On the left, the figure shows the SG before signal insertion in which every state has been tagged with one of the *ER*s or *QR*s 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. 3. 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.

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

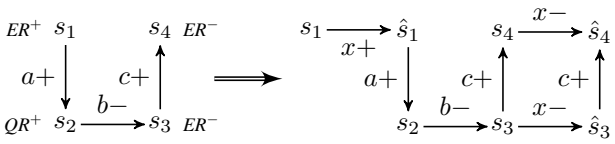


Fig. 2: SG before and after signal insertion.

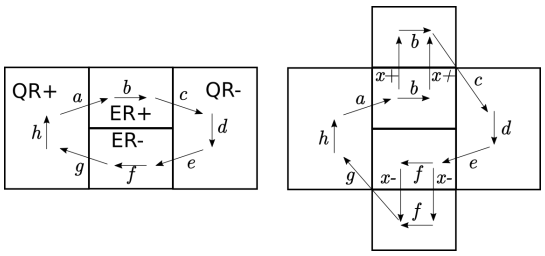


Fig. 3: Partitioning of the state space into the ER and QR regions of x before (left) and after (right) the insertion.

IV. SAT formula for the signal insertion problem

The SAT formulation is inspired by the work in [3]. The main difference of this paper is that the CSC problem is solved by inserting signals sequentially rather than inserting all signals at once. This strategy explores a larger space of solutions, since it allows one internal signal to trigger another internal signal. This enables the generation of solutions that cannot be found by the approach in [3].

Signal insertion is based on partitioning the set of states into four subsets as described in the previous section. The SAT formula encodes this partitioning. Additionally, it also encodes the properties for speed-independent implementability: consistency, persistence and input-properness.

A. Boolean variables

Two variables are defined for every state s: $v_1(s)$ and $v_2(s)$. They encode the membership of s to one of the ER/QR regions of x. The encoding used in this work is:

$$\begin{aligned} ER^+(s) &= v_1(s) \wedge v_2(s) & \quad ER^-(s) &= v_1(s) \wedge \neg v_2(s) \\ QR^+(s) &= \neg v_1(s) \wedge v_2(s) & \quad QR^-(s) &= \neg v_1(s) \wedge \neg v_2(s) \end{aligned}$$

The total number of variables is $2 \times |S|$. Additional variables, will be required for optimization purposes (see Section V).

B. Consistency

Constraints to ensure the consistency of x (i.e., x^+ and x^- alternate) must be included in the SAT formula. That means that all paths across the SG must visit the insertion regions in the order[2] $ER^+ \to QR^+ \to ER^- \to QR^- \to ER^+ \to \cdots$.

[2]Transitions $ER^+ \to ER^-$ and $ER^- \to ER^+$ are also possible.

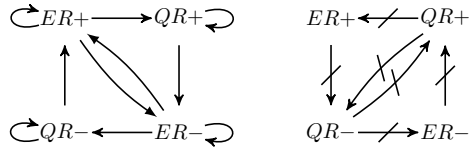


Fig. 4: Consistent (left) and inconsistent (right) transitions.

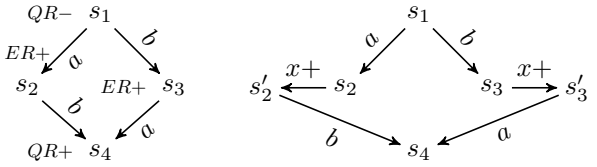


Fig. 5: Non-persistent signal insertion.

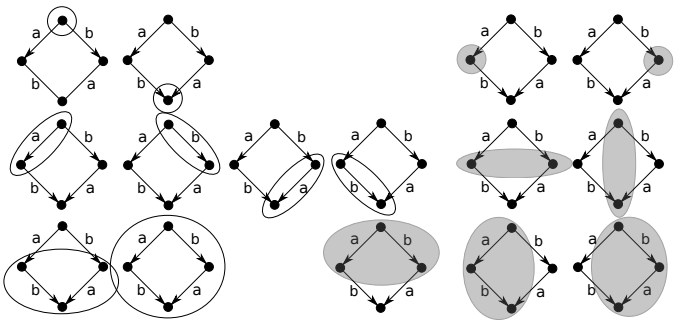


Fig. 6: Persistent insertions are circled on the left. Non-persistent insertions are shadowed in gray.

Fig. 4 shows the legal transitions between sets (left) and the illegal transitions (right). The constraint can be formulated as:

$$\begin{aligned}&\forall s_1 \rightarrow s_2 \in T: \\ &\quad \neg(QR^-(s_1) \wedge QR^+(s_2)) \wedge \neg(QR^-(s_1) \wedge ER^-(s_2)) \wedge \\ &\quad \neg(QR^+(s_1) \wedge QR^-(s_2)) \wedge \neg(QR^+(s_1) \wedge ER^+(s_2)) \wedge \\ &\quad \neg(ER^+(s_1) \wedge QR^-(s_2)) \wedge \neg(ER^-(s_1) \wedge QR^+(s_2))\end{aligned}$$

C. Persistence

The insertion of a new signal must guarantee that no new non-persistence is introduced. For that, it suffices to look at *diamonds* of concurrent transitions [2], [3]. Fig. 5 (left) depicts a diamond with a possible assignment of ER/QR regions for signal insertion. On the right, the result after signal insertion is shown. It can be noticed that this insertion does not maintain persistence, e.g., a is enabled in s_1 but not in s_3.

Fig. 6 shows all possible allocations of ERs in a diamond. The circled states identify the ER for insertion. Circled regions preserve persistence, whereas shadowed regions do not.

For each diamond $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_4$ and $s_1 \xrightarrow{b} s_3 \xrightarrow{a} s_4$, persistence can be formulated with the following three constraints:

$$\begin{aligned} ER^+(s_1) \wedge ER^+(s_4) &\Rightarrow ER^+(s_2) \wedge ER^+(s_3) \\ \neg ER^+(s_1) \wedge \neg ER^+(s_4) &\Rightarrow \neg ER^+(s_2) \wedge \neg ER^+(s_3) \\ ER^+(s_2) \wedge ER^+(s_3) &\Rightarrow ER^+(s_4) \end{aligned}$$

Similar clauses apply for ER^-.

D. Input-properness

Input-properness is guaranteed by forbidding x to trigger an input signal, i.e., not allowing any input transitions to exit ER. Formally, for each $s_1 \xrightarrow{a} s_2$, such that a is an input event:

$$(ER^+(s_1) \Rightarrow ER^+(s_2)) \ \wedge \ (ER^-(s_1) \Rightarrow ER^-(s_2))$$

V. Pseudo-Boolean formula for optimization

This section introduces the optimization part of the SAT formula for generating high-quality solutions. Optimization is performed by defining a cost function as a linear combination of Boolean variables. This function biases the explored solutions towards disambiguating CSC conflicts with low logic cost. Methods for Pseudo-Boolean optimization can be used to formulate the problem with a linear cost function and still using SAT solving engines [5].

A. Reduction of CSC Conflicts

After the insertion of a signal x, some of the CSC conflicts will be solved and some will not. We next propose a formulation to quantify the remaining conflicts after the insertion.

Let us call *CSCpairs* the sets of pairs of states with CSC conflicts. For each pair (s_i, s_j) in the previous set we define a new variable $c_{i,j}$ that denotes whether a CSC conflict remains after signal insertion.

A CSC conflict is solved for (s_i, s_j) if the two states have a different value for x. This requires both states to be in different QRs of x. Notice that the presence in some ER means that sibling states would be created that would inherit the original CSC conflict. Thus, for any $(s_i, s_j) \in$ *CSCpairs*:

$$\neg c_{i,j} \ \Leftrightarrow \ [QR^-(s_i) \wedge QR^+(s_j)] \vee [QR^+(s_i) \wedge QR^-(s_j)]$$

USC conflicts may also become CSC conflicts after signal insertion (they are called *secondary conflicts*). They occur when two states still have the same code and x becomes enabled in one of them but not in the other one. While this can be easily modelled as a Boolean formula, these conflicts have a very minor impact and can be ignored in practice.

The total number of conflicts (minus secondary conflicts) that will remain after inserting x can be easily computed as:

$$\text{Conf} = \sum_{(s_i,s_j) \in CSCpairs} c_{i,j}.$$

B. Estimation of logic: essential literals

The encoding for essential literals is the most elaborate of the ones presented here. For clarity, let us define $\widehat{d}(s_1, s_2)$ as the Hamming distance between the binary encodings of s_1 and s_2 after the insertion of x. Additionally, the predicate $\widehat{d}_1(s_1, s_2)$ is true if $\widehat{d}(s_1, s_2) = 1$. This predicate can also be extended and used for sibling states, e.g., $\widehat{d}_1(s_1, \hat{s_2})$. We will use the predicates $\widehat{ON}_y(s)$ and $\widehat{OFF}_y(s)$ to denote the fact that state s belongs to the on- and off-set of signal y, respectively, after the insertion of signal x. Further details about the encoding of these predicates are given in the appendix.

The basic condition for a signal z becoming an essential literal for signal y is as follows: there must be a pair of states $s_1 \in \widehat{ON}_y(s)$ and $s_2 \in \widehat{OFF}_y(s)$, such that $\widehat{d}_1(s_1, s_2)$ and $s_1(z) \neq s_2(z)$. We can also distinguish between positive and negative essential literals depending on the polarity of the essential literal z with regard to y.

We can now define the basic predicate that represents the fact that two states (or their siblings) with Hamming distance one can be at the on/off-set of y after the signal insertion:

$$\begin{aligned} D1(s_1,s_2,y) \equiv\ & (\widehat{d_1}(s_1,s_2) \wedge \widehat{ON_y}(s_1) \wedge \widehat{OFF_y}(s_2)) \vee \\ & (\widehat{d_1}(\hat{s_1},s_2) \wedge \widehat{ON_y}(\hat{s_1}) \wedge \widehat{OFF_y}(s_2)) \vee \\ & (\widehat{d_1}(s_1,\hat{s_2}) \wedge \widehat{ON_y}(s_1) \wedge \widehat{OFF_y}(\hat{s_2})) \vee \\ & (\widehat{d_1}(\hat{s_1},\hat{s_2}) \wedge \widehat{ON_y}(\hat{s_1}) \wedge \widehat{OFF_y}(\hat{s_2})) \end{aligned}$$

Next, the constraint for essential literals is defined, where $E^+_{z\to y}$ and $E^-_{z\to y}$ are new boolean variables that represent the fact that z is a positive and negative essential literal for y, respectively.

$$\begin{aligned} &\forall s_1, s_2 \in S: \\ &\quad \big(D1(s_1,s_2,y) \wedge s_1(z)=1 \wedge s_2(z)=0 \Rightarrow E^+_{z\to y}\big)\ \wedge \\ &\quad \big(D1(s_1,s_2,y) \wedge s_1(z)=0 \wedge s_2(z)=1 \Rightarrow E^-_{z\to y}\big) \end{aligned}$$

The number of essential literals after the insertion of x can now be computed as:

$$\text{EssLit} = \sum_{y,z\in\Sigma\cup\{x\}} E^+_{z\to y} + E^-_{z\to y}$$

C. Don't Care set

A large DC-set increases the opportunities for logic minimization. After the insertion of the new signal, the size of the DC-set depends on the amount of new sibling states, which is determined by the size of the ERs for signal x. A simple way for estimating their size is to count the signals that are concurrent with x after the insertion.

The variables $conc_{a+}$ and $conc_{a-}$ indicate whether there is a transition $a+$ or $a-$, concurrent with x. The following predicates represent the concurrent events with x:

$$\begin{aligned} &\forall\ s_1 \xrightarrow{a+} s_2: \\ &\quad (ER^+(s1) \wedge ER^+(s2)) \vee (ER^-(s1) \wedge ER^-(s2)) \Rightarrow conc_{a+} \\ &\forall\ s_1 \xrightarrow{a-} s_2: \\ &\quad (ER^+(s1) \wedge ER^+(s2)) \vee (ER^-(s1) \wedge ER^-(s2)) \Rightarrow conc_{a-} \end{aligned}$$

The number of concurrent signals, highly correlated with the size of the ER_x, is thus computed as:

$$\text{ERsize} = \sum_{a\in\Sigma} (conc_{a+} + conc_{a-})$$

D. Entry points

We say that s is an entry point (EP) for ER^+_x if $s \in ER^+_x$ and all its predecessor states are outside ER^+_x (similarly for ER^-_x). The events leading to EPs determine the trigger signals of x. Thus, reducing the number of EPs also contributes to reduce the causality relations with the remaining signals of the circuit. We have observed that penalizing the amount of EPs helps to find solutions with simpler logic.

For each state s, we define the variable $ep(s)$ that determines whether s is an EP for x:

$$\begin{aligned} &\forall s_i \to s_j: \\ &\big(\neg ER^+(s_i) \wedge ER^+(s_j)\big) \vee \big(\neg ER^-(s_i) \wedge ER^-(s_j)\big) \implies ep(s_j) \end{aligned}$$

The number of entry points can now be computed by:

$$\text{numEP} = \sum_{s\in S} ep(s)$$

E. Cost function

The multiobjective cost function used to estimate the quality of a solution is defined as:

$$\text{Cost} = \alpha\cdot\text{Conf} + \beta\cdot\text{EssLit} + \gamma\cdot\text{numEP} + \delta\cdot\text{ERsize} \quad (4)$$

with $\alpha, \beta, \gamma, \delta$ being adjustable coefficients.

This function needs to be encoded as a SAT formula. The larger the coefficients, the more complex the formula. This affects the runtime dramatically and limits the range of values that can be used in practice. We found that weights ≤ 3 produce good results with reasonable execution times.

Having a diversity of cost functions with different coefficients also contributes to a wider exploration of solutions. In our experiments we have also generated results by exercising a small set of cost functions and selecting the best solution. This strategy will be further discussed in Section VIII.

VI. SAT-based optimization algorithm

The optimization algorithm iteratively tries to insert new signals (one at a time) into the SG until CSC is solved or no satisfiable is found. The core of the algorithm is the function *findModelForOneSignal*, which returns a model that encodes the definition of the $ER^{\pm}/QR^{\pm}$ regions for the insertion of a new signal.

Algorithm 1 sketches the procedure to find a solution for signal insertion using pseudo-Boolean optimization. The cost function (4) is encoded as a set of SAT clauses [6]. The function is minimized by iteratively constraining the formula until it becomes unsatisfiable. If a model with $Cost = k$ is found in one iteration, the constraint $Cost < k$ is encoded and added for the next iteration. This strategy speeds-up the optimization by taking advantage of the clauses learned by the SAT solver from the previous iterations [6].

A binary search on the value of k could also be possible, but it cannot take advantage of the learned clauses. We have not observed a clear benefit when using binary search.

Algorithm 1: FINDMODELFORONESIGNAL(G)

```
input : An SG with CSC conflicts.
output: A SAT model for signal insertion.
begin
    CNF = encodeCSCconstraints(G)
    model = SATsolver(CNF)
    bestModel = model
    while isSatisfiable(model) do
        k = getCost(model)
        addClausesForCost(CNF, Cost < k)
        model = SATsolver(CNF)
        if isSatisfiable(model) then bestModel = model
    return bestModel
```

The PBLib [7] toolkit was used for the encoding of Pseudo-Boolean constraints and solving the SAT formulas. Internally, PBLib uses Minisat [8] as SAT solver.

VII. Comparison with 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 [4] (based on unfoldings) and structural methods using integer-linear programming [9].
- State-based methods, such as petrify [2] and a previous SAT-based approach [3].

We will use the example of Fig. 7, depicting one of the 4-phase latch controllers presented in [10], to discuss the differences among tools. This figure includes the approach presented here, that will from now on be referred to as PBASE. The logic equations for each solution are the following:

	MPSAT	Petrify	PBASE
$la =$	$x_2(\overline{rr}+\bar{x}_1)+x_3$	$\bar{x}_1$	$\bar{x}_1$
$rr =$	$x_2\ lr\ \bar{x}_1+x_3$	$x_2+rr\ \bar{x}_1$	$(\bar{x}_1\ \overline{ra})+rr\ \bar{x}_1$
$x_1 =$	$\bar{x}_2\ lr+x_1(\overline{ra}+\bar{x}_3)$	$(\bar{x_2}\ rr)+x_1+\bar{lr}$	$(ra\ rr\ \overline{lr})+x_1(rr\ \overline{lr})$
$x_2 =$	$(x_3+x_1)+x_2\ lr$	$\overline{ra}(\bar{x_1}+lr)+x_2\ lr$	
$x_3 =$	$(x_2x_1\overline{ra})+x_3\bar{x}_1$		

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 SG 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. In Fig. 7, the MPSAT solution requires three new signals and 22 literals. The reader can intuitively perceive that the new events have simple causality relations. This phenomenon also occurs for the ILP-based method proposed in [9].

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. It requires two signals and 13 literals.

PBASE provides the most efficient solution, with only one signal and 11 literals. Notice that the two new events have multiple causality relations (two input and two output arcs). Although the figure shows a Petri net, these relations are naturally found at state level ignoring the model of the original specification. In this particular case, the solution was representable as a nice Petri net.

With regard to the estimation of logic, structural methods are mostly based on finding trigger relations between events. This gives a lower bound on the number of literals, although it is less accurate than the estimation given by essential literals.

The SAT-based approach presented in [3] has two main limitations. First, all new signals are inserted simultaneously and cannot have mutual trigger relations between them. Second, the approach is simply based on finding valid solutions without any estimation of the logic cost. The solutions provided by this approach are significantly worse than the ones generated by the other tools discussed in this section.

VIII. Results

This section shows the experimantal results for PBASE and a comparison with Petrify and MPSAT. Additionally, we have re-implemented the approach from [3] (referred to as SAT), and included it as a baseline.

We have used a large diversity of benchmarks from the literature and all the 4-phase latch controllers presented in [10] (127 out of 137 had CSC conflicts). The solutions for all benchmarks can be found in [11].

Table I shows the results for a variety of heterogeneous controllers. The column *Signals/Literals* reports 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 of the SG is in column $|S|$. The *I/O* column contains the number of input/output signals of the SG. This table compares results between Petrify, MPSAT and two versions of PBASE, *single* and *multi*, using different versions of the $(\alpha, \beta, \gamma, \delta)$ coefficients for optimization function (4):

- PBASE(single): using the coefficients (2,1,3,2).
- PBASE(multi): using multiple different values for the coefficients and choosing the best solution. The set of coefficients were (0,1,1,1), (3,2,2,0), (3,1,1,0) and (1,0,1,1), besides the one used for PBASE(single).

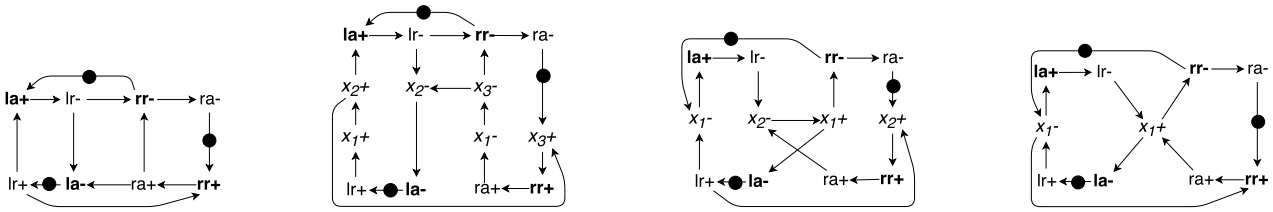
PBASE(multi) explores a larger variety of solutions at the expense of computational time. It also uses a fast heuristic in the first iteration to be able to solve larger problems. A 10-minute timeout is set up and the best solution found when the timeout expires is returned. The combination of the fast heuristic with the timeout allows to solve problems that could not be solved with the simpler version.

In some cases, the tools were not able to complete the task. These cases are reported with one of the following codes:

- Unsf: Unsafe Petri nets. MPSAT is unable to solve them.
- Fail: The tool was unable to find a solution.
- Time: No solution found in less than 1 hour.

A summary of the results for Table I can be found in Table II, including the results for SAT [3]. This table presents a comparison between PBASE(multi) and the other tools. Row *Solved* reports the number of solved instances. The remaining data in the table only report the total results for the benchmarks that were solved by both tools under comparison. Results for those not solved by both were ignored in the summary. The CPU time is divided into 3 groups as a function of problem size (see Table III for the group division). This puts into scale the amount of time used for the largest problems. The final row reports the ratio of literals obtained by any pair of tools taking the other tools as a reference.

SAT gives the lowest-quality solutions, as it does not include any quality estimator in the model, while PBASE outperforms the other methods, with an average improvement of 13% in



(a) Original STG (b) MPSAT (c) Petrify (d) PBASE

Fig. 7: 4-phase latch controller L220oR2242 (from [10]). State encoding solutions obtained by different tools.

TABLE I: Experimental results for Petrify (Pfy), MPSAT (MP), PBASE(single) (PB(s)) and PBASE(multi) (PB(m)).

			CPU(sec)				Signals/Literals							CPU(sec)				Signals/Literals			
Example	I/O	$\|S\|$	Pfy	MP	PB(s)	PB(m)	Pfy	MP	PB(s)	PB(m)	Example	I/O	$\|S\|$	Pfy	MP	PB(s)	PB(m)	Pfy	MP	PB(s)	PB(m)
adc.buff1	0/2	6	0.4	0.9	**0.3**	5.4	**2/9**	**2/9**	**2**/11	**2/9**	nowick	3/2	18	0.2	**0.1**	0.2	5.4	**1/13**	**1/13**	**1/13**	**1/13**
adfast	3/3	44	1.2	**0.1**	5.6	16.0	**2/14**	**2**/21	**2/14**	**2/14**	par2	3/3	28	0.2	**0.1**	4.4	12.9	**2/16**	**2/16**	**2/16**	**2/16**
alloc-outbound	4/3	17	0.2	**0.1**	0.8	7.3	**2/16**	**2**/17	**2/16**	**2/16**	par4	5/5	628	3.9	**0.2**	Time	544.6	**4/32**	**4/32**	-/-	**4/32**
buf2	0/2	8	**0.1**	Unsf	0.8	7.9	**3**/14	-/-	**3**/15	**3/13**	pla	0/3	12	**0.1**	0.1	0.2	4.5	**1/14**	2/16	**1/14**	**1/14**
buf_dum.1	0/2	8	**0.1**	0.1	0.8	6.8	**3**/14	**3**/15	**3**/15	**3/13**	ram-read-sbuf	5/5	36	1.8	**0.1**	1.2	10.3	**1/18**	**1**/19	**1**/22	**1/18**
buf_unsafe.1	0/2	12	Fail	Unsf	**5.3**	23.3	-/-	-/-	**5/26**	**5/26**	read_write	7/4	322	2.0	**0.2**	79.1	164.8	**1/24**	**1**/26	**1/24**	**1/24**
c10	0/10	2046	Time	Fail	**32.7**	136.7	-/-	-/-	**1/31**	**1/31**	sbuf-ram-write	5/5	58	5.0	**0.2**	9.1	24.1	**2**/22	**2**/31	**2**/23	**2/21**
c6	0/6	126	4.2	**0.8**	1.1	6.9	**1/19**	**1/19**	**1/19**	**1/19**	sbuf-read-ctl	2/4	14	**0.1**	0.2	0.2	4.5	**1/15**	**1/15**	**1/15**	**1/15**
csc-div1	0/2	8	**0.0**	0.1	0.1	4.7	**1/16**	**1/16**	**1/16**	**1/16**	seq2	3/3	12	0.1	0.1	**0.1**	3.0	**1/8**	**1/8**	**1/8**	**1/8**
duplicator	2/2	20	0.4	**0.1**	0.5	5.9	**2**/18	**2/13**	**2/13**	**2/13**	seq3	4/4	16	0.5	**0.1**	0.6	6.9	**2/14**	**2/14**	**2/14**	**2/14**
future	4/4	36	1.0	**0.2**	0.4	6.0	**1/18**	3/33	**1/18**	**1/18**	seq4	5/5	20	1.3	**0.2**	1.4	8.3	3/20	3/20	**2/19**	**2/19**
glc	2/1	17	0.1	**0.1**	0.1	4.9	**1/10**	**1**/11	**1/10**	**1/10**	seq8	9/9	36	4.7	**1.1**	108.7	302.6	4/47	7/44	**3**/44	**3/37**
ircv-bm	5/4	44	5.8	**0.4**	9.8	39.9	**2**/37	**2**/31	**2**/35	**2/28**	seq-mix	4/4	20	1.1	**0.2**	2.2	10.8	3/20	3/20	3/**18**	**2/18**
isend	4/3	36	4.2	**0.4**	4.4	23.8	3/48	3/34	3/**29**	**2/29**	sis-master-read	6/7	1882	3.3	**0.3**	Time	309.3	**1/38**	**1**/40	-/-	**1**/39
lazy_ring.noncsc	5/3	160	1.7	**0.4**	27.6	53.9	**1**/24	2/29	**1**/22	**1/20**	trcv-bm	5/4	44	8.7	**0.3**	7.6	35.3	**2**/37	**2**/32	**2/31**	**2/31**
master-read	6/7	8932	**54.6**	Fail	Time	Time	**8/68**	-/-	-/-	-/-	tsend-bm	5/4	40	4.6	**0.2**	7.9	21.0	2/39	2/**27**	3/34	**1**/28
master-read2	0/13	8932	26.3	**15.5**	Time	Time	6/**70**	**5**/75	-/-	-/-	vbe4a.nousc	3/3	58	1.4	**0.2**	5.1	21.1	**3**/26	4/23	**3**/18	**3/16**
master-read.1098	6/7	1098	9.7	**3.3**	Time	537.3	**4**/57	6/43	-/-	5/**41**	vbe5a	3/3	44	0.9	**0.1**	4.2	15.0	**2/14**	**2**/21	**2/14**	**2/14**
mmu0	4/4	174	2.7	**0.1**	89.1	198.3	**3**/29	**3**/28	**3**/28	**3/26**	vbe6a.nousc	4/4	128	1.1	**0.2**	41.0	114.4	3/31	**2/30**	**2/30**	**2/30**
mmu1	4/4	82	1.1	**0.2**	8.1	27.1	**2**/32	**2**/25	**2**/25	**2/23**	vbe6x.nousc	3/3	48	0.4	**0.2**	4.4	17.3	**2/22**	**2/22**	**2**/23	**2/22**
mod4_counter	1/2	16	**0.1**	**0.1**	0.3	8.0	**2**/26	**2/25**	**2**/26	**2**/26	vme_read	8/6	251	4.0	**0.1**	16.2	39.0	**1**/32	**1**/33	**1/30**	**1/30**
mr0	5/6	302	4.4	**0.4**	Time	600.2	**3**/45	**4/29**	-/-	4/33	vme_read_write	3/3	28	0.3	**0.3**	1.0	8.6	**1**/23	2/27	**1/22**	**1/22**
mr1	4/5	190	3.4	**0.6**	91.6	201.7	4/35	4/31	**3**/26	**3/25**	vme_write	8/6	817	7.8	**0.2**	Time	602.1	**1**/38	**1**/38	-/-	**1/35**
nak-pa	4/5	56	0.7	**0.1**	0.7	9.0	**1**/18	**1**/18	**1**/18	**1/16**	vmebus	3/3	24	0.8	**0.2**	0.5	7.2	**1/19**	2/28	**1/19**	**1/19**

TABLE II: Summary for the benchmarks in Table I.

	Pfy	PB(m)	MP	PB(m)	SAT	PB(m)	PB(s)	PB(m)
Solved	46	46	44	46	43	46	41	46
CPU (small)	13	163	4	155	0	148	26	186
CPU (medium)	30	226	2	226	0	226	62	226
CPU (large)	53	3675	8	3675	73	3812	487	1218
Signals	88	83	97	80	72	78	78	74
Literals	1081	943	1042	930	1938	948	864	820
Ratio	**1.00**	**0.87**	**1.00**	**0.89**	**1.00**	**0.49**	**1.00**	**0.95**

TABLE III: Average CPU time for different SG sizes.

			Avg. CPU (s)	
Size	**Condition**	n	**PB(s)**	**PB(m)**
small	$\|S\| < 40 \ \wedge \ \|\Sigma\| \leq 15$	22	1.2	8.5
medium	$40 \leq \|S\| < 100 \ \wedge \ \|\Sigma\| \leq 15$	10	6.2	22.6
large	$\|S\| \geq 100 \ \vee \ \|\Sigma\| > 15$	9(s)/14(m)	54.1	272.3

TABLE IV: Summary for the 127 4-phase latch controllers.

	Pfy	PB(m)	MP	PB(m)	SAT	PB(m)	PB(s)	PB(m)
Solved	127	127	72	127	85	127	127	127
CPU (sec)	41	928	11	347	1	430	118	928
Signals	231	207	111	84	110	110	207	207
Literals	1818	1550	952	778	1386	943	1593	1550
Ratio	**1.00**	**0.85**	**1.00**	**0.82**	**1.00**	**0.68**	**1.00**	**0.97**

the number of literals with regard to petrify. PBASE(multi) offers a tangible improvements with regard to PBASE(single). However, this comes at the expense of a higher computational cost. Section VIII-A discusses this problem.

Interestingly, one of the tiniest and most difficult problems for state encoding (`buf_unsafe.1`), was only solved by PBASE. It required 5 state signals and the SG was expanded from 12 states to 69 after signal insertion.

Table IV reports the summary of results for the 127 4-phase latch controllers [10] without CSC. Even though all benchmarks were small, only Petrify and PBASE could solve all of them. Since many of them were specified as unsafe Petri nets, MPSAT could not handle them. SAT also failed in many examples due to the impossibility of inserting state signals with causality relations among them. This feature was essential for the other methods to solve some of the examples. The results for both tables show an average reduction of 14% in literals when compared to petrify.

A. Scalability

A major concern is scalability with the size of the SG. The main reason for the increase of the CPU time is the size of the SAT formula, which is mainly dominated by the clauses representing the cost function (Pseudo-boolean constraints).

Table III reports the average execution times for the benchmarks classified in three categories according to the number of states ($|S|$) and signals ($|\Sigma|$) of the SG (n reports the number of instances in each class). While the runtime is low for small examples, it drastically increases for large SGs.

Our current work is focused on resorting to decomposition techniques that can handle SGs with 10^6 states and beyond, inspired by approaches previously proposed at the level of STG [12]. The existing algorithms to calculate projections on subsets of events using algorithms with $O(m \log n)$ complexity[3] [13] may play a crucial role to achieve that goal.

IX. Conclusions

This paper has introduced a novel approach for state encoding based on Pseudo-Boolean optimization. The approach allows to encode any valid solution as well as estimators of logic complexity. The results show a significant reduction in the number of literals with respect to the existing tools.

Scalability for large controllers is an aspect that must be considered in the near future, possibly using divide-and-conquer strategies to decompose the specification into a set of smaller communicating controllers.

The exploration of solutions trading-off performance and complexity is also another aspect that can be addressed to reduce the input/output response time of the controllers.

Appendix

This section defines some Boolean predicates that were used in Section V for the cost function. Henceforth, the suffix $\pm$ is used to indistinctly refer to the positive and negative regions.

The next predicate indicates that a transition in the original SG is delayed by x after its insertion:

$$\mathrm{Del}_x(s,a) \;\equiv\; \exists s \xrightarrow{a} s' : s \rightarrow s' \in \mathit{EXIT}$$

It can also be interpreted as "x is a trigger of a in s". The following predicates encode the ERs and QRs in the new SG based on the original one. The $\widehat{\ }$ symbol indicates that the region refers to the new SG after inserting signal x:

$$\widehat{QR}_a^+(s) = QR_a^+(s) \vee (ER_a^-(s) \wedge \mathrm{Del}_x(s,a))$$
$$\widehat{QR}_a^-(s) = QR_a^-(s) \vee (ER_a^+(s) \wedge \mathrm{Del}_x(s,a))$$
$$\widehat{ER}_a^\pm(s) = ER_a^\pm(s) \wedge \neg\mathrm{Del}_x(s,a)$$
$$\widehat{ER}_a^\pm(\hat{s}) = ER_a^\pm(s) \wedge ER_x^\pm(s) \qquad \widehat{QR}_a^\pm(\hat{s}) = QR_a^\pm(s)$$
$$\widehat{ER}_x^\pm(s) = ER_x^\pm(s) \qquad \widehat{QR}_x^\pm(s) = QR_x^\pm(s)$$
$$\widehat{ER}_x^\pm(\hat{s}) = \mathrm{False} \qquad \widehat{QR}_x^\pm(\hat{s}) = ER_x^\pm(s)$$

Predicates denoting the membership of a state to the onset or offset of a signal after the insertion of x are defined as:

$$\widehat{ON_y}(s) \equiv \widehat{ER}_y^+(s) \vee \widehat{QR}_y^+(s); \quad \widehat{OFF_y}(s) \equiv \widehat{ER}_y^-(s) \vee \widehat{QR}_y^-(s)$$

The following predicates define the encoding of x in a state s after signal insertion:

$$\widehat{ONE_y}(s) = \widehat{ER}_y^-(s) \vee \widehat{QR}_y^+(s); \quad \widehat{ZERO_y}(s) = \widehat{ER}_y^+(s) \vee \widehat{QR}_y^-(s)$$
$$\widehat{EQ_y}(s_1,s_2) = \widehat{ZERO_y}(s_1) \wedge \widehat{ZERO_y}(s_2) \vee \widehat{ONE_y}(s_1) \wedge \widehat{ONE_y}(s_2)$$

[3] n is the number of states and m is the number of arcs.

The Hamming distance of the encoding of two states, s_1 and s_2, before the insertion of signal x is defined as:

$$d(s_1,s_2) = \sum_{a \in \Sigma} (s_1(a) \neq s_2(a)).$$

Finally, we are interested in the Hamming distance being one after the insertion of x. This is defined by $\widehat{d}_1(s_1,s_2)$:

$$\widehat{d}_1(s_1,s_2) \equiv \begin{cases} \mathit{False} & \text{if } d(s_1,s_2) > 1 \\ \widehat{EQ_Y}(s_1,s_2) & \text{if } d(s_1,s_2) = 1 \\ \neg\widehat{EQ_Y}(s_1,s_2) & \text{if } d(s_1,s_2) = 0 \end{cases}$$

Acknowledgment

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).

References

[1] R. M. Fuhrer, B. Lin, and S. M. Nowick, "Symbolic hazard-free minimization and encoding of asynchronous finite state machines," in *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1995.

[2] 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.

[3] 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*, vol. 7, no. 1/2, pp. 101–115, Feb. 1994.

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

[5] P. Barth, "A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization," Max Planck Institut für Informatik, Saarbrücken, Germany, Tech. Rep. MPI-I-95-2-003, 1995.

[6] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–25, 2006.

[7] T. Philipp and P. Steinke, "PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF," in *Theory and Applications of Satisfiability Testing – SAT 2015*, ser. LNCS, M. Heule and S. Weaver, Eds. Springer, 2015, vol. 9340, pp. 9–16.

[8] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *6th Int. Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[9] 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.

[10] G. Birtwistle and K. S. Stevens, "The family of 4-phase latch protocols," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 2008, pp. 71–82.

[11] "Benchmark repository," http://www.cs.upc.edu/~jordicf/petrify/benchmarks.

[12] D. Wist, M. Schaefer, W. Vogler, and R. Wollowski, "Signal Transition Graph decomposition: internal communication for speed independent circuit implementation," *IET Computers Digital Techniques*, vol. 5, no. 6, pp. 440–451, 2011.

[13] 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.