Inferring Assertion for Complementary Synthesis

ShengYu Shen
School of Computer Science, National University of Defense Technology
410073,DeYa Avenue
ChangSha,China
syshen@nudt.edu.cn

Ying Qin
School of Computer Science, National University of Defense Technology
410073,DeYa Avenue
ChangSha,China
yingqin@nudt.edu.cn

JianMin Zhang
School of Computer Science, National University of Defense Technology
410073,DeYa Avenue
ChangSha,China
jmzhang@nudt.edu.cn

ABSTRACT

Complementary synthesis can automatically synthesize the decoder circuit of an encoder. However, its user needs to manually specify an assertion on some configuration pins to prevent the encoder from reaching the non-working states. To avoid this tedious job, we propose an automatic approach to infer this assertion. **First**, we propose a halting algorithm that can decide the existence of the decoder for every particular assertion. **Second**, for every invalid value of configuration pins that leads to the non-existence of the decoder, we use cofactoring and Craig interpolation to infer a new formula, which covers a larger set of such invalid values. This second step is repeated until all invalid values are covered by these inferred formulas. **Finally**, we obtain the final assertion by anding the inverses of all these inferred formulas. The decoder exists if and only if this final assertion is still satisfiable. To illustrate its usefulness, we have run our algorithm on several complex encoder circuits, including PCI-E and Ethernet. Experimental results show that our algorithm can always infer assertions for them.

Categories and Subject Descriptors

B.5.2 [**REGISTER-TRANSFER-LEVEL IMPLEMENTATION**]: Design Aids—*Automatic synthesis*; B.4.4 [**INPUT/OUTPUT AND DATA COMMUNICATIONS**]: Performance Analysis and Design Aids—*Formal models*

General Terms

Algorithms, Design, Theory, Verification

Keywords

Complementary Synthesis, Inferring Assertion, Cofactoring, Craig Interpolation

1. INTRODUCTION

One of the most difficult jobs in designing communication and multimedia chips is to design and verify the complex complementary circuit pair (E, E^{-1}), in which the encoder E transforms information into a format suitable for transmission and storage, while its complementary circuit(or decoder) E^{-1} recovers this information.

In order to facilitate this job, complementary synthesis [1, 2] is proposed to automatically synthesize the decoder circuit of an encoder, by checking whether the encoder's input letter can be uniquely determined by its output sequence. Such encoder circuits normally have several modes, each of which corresponds to a non-overlapped state set:

1. One of the most important modes is the working mode, in which the encoder encodes its input letters. So the encoder's input letter can be determined by its output sequence, which leads to the existence of its decoder.
2. On the other hand, the encoder still has many other non-working modes, such as the testing and sleep mode, in which the encoder, respectively, processes test commands, or does nothing. In these modes, the encoder's input letter can't be determined by its output sequence, which leads to the non-existence of its decoder.

Thus, the user of complementary synthesis needs to manually specify an assertion that asserts constant values on some configuration pins, to prevent the encoder from reaching those non-working states. To avoid this tedious job of manually specifying assertion, we propose an automatic approach to infer this assertion:

1. **First**, we propose a halting algorithm that decides the existence of the decoder for every particular assertion.
2. **Second**, for every invalid value of configuration pins that leads to the non-existence of the decoder, we use cofactoring[3] and Craig interpolation[4] to infer a new formula, which covers a larger set of such invalid values. This second step is repeated until all invalid values are covered by these inferred formulas.
3. **Finally**, we obtain the final assertion by anding the inverses of all these inferred formulas. The decoder exists if and only if this final assertion is still satisfiable.

We have implemented our algorithm in the OCaml language, and solved the generated SAT instances with the Minisat

solver[8]. The benchmark set includes several complex encoders from industrial projects (e.g., PCI-E[9] and Ethernet[10]). Experimental results show that our algorithm can always infer assertions for them.

This paper is organized as follows. Section 2 introduces background materials. Section 3 presents the overall framework of our algorithm and the proof of its correctness, while Section 4 and 5 discuss how to infer a new formula covering the invalid value of configuration pins, and how to minimize the parameter value discovered. Section 6 and 7 present experimental results and related work. Finally, Section 8 concludes with a note on future work.

2. PRELIMINARIES

2.1 Propositional satisfiability

The Boolean value set is denoted as $B = \{0, 1\}$. For a Boolean formula F over a variable set V, the propositional satisfiability problem(abbreviated as SAT) is to find a satisfying assignment $A : V \to B$, so that F can be evaluated to 1. If such a satisfying assignment exists, then F is satisfiable; otherwise, it is unsatisfiable. A computer program that decides the existence of such a satisfying assignment is called a SAT solver, such as Zchaff[5], Grasp[6], Berkmin[7], and MiniSAT[8]. Normally, a SAT solver requires the formula to be represented in the conjunctive normal form(CNF), in which a formula is a conjunction of its clause set, and a clause is a disjunction of its literal set, and a literal is a variable or its negation. A formula in the CNF format is also called a SAT instance,

For a Boolean function $f : B^n \to B$, we use $supp(f)$ to denote its support set $\{v_1 \dots v_n\}$. According to [3], the positive and negative cofactors of $f(v_1 \dots v \dots v_n)$ with respect to variable v are $f_v = f(v_1 \dots 1 \dots v_n)$ and $f_{\overline{v}} = f(v_1 \dots 0 \dots v_n)$, respectively. **Cofactoring** is the action that applies 0 or 1 to v to get f_v or $f_{\overline{v}}$.

Theorem 1 (Craig Interpolation Theorem[4]). *Given two first order logic formulas ϕ_A and ϕ_B, with $\phi_A \wedge \phi_B$ unsatisfiable, there exists a formula ϕ_I referring only to the common variables of ϕ_A and ϕ_B such that $\phi_A \to \phi_I$ and $\phi_I \wedge \phi_B$ is unsatisfiable. The formula ϕ_I is referred to as the interpolant of ϕ_A with respect to ϕ_B.*

In the remainder of this paper, we will focus on the Boolean propositional logic only, and use the algorithm proposed by McMillan[11] to generate interpolants from the proof of unsatisfiability, which is generated by MiniSAT[8].

2.2 Recurrence diameter

To model the encoder with configuration pins, we extend the traditional definition of Mealy finite state machine [12]:

Definition 1. **Mealy finite state machine with configuration** is a 6-tuple $M = (S, s_0, I, C, O, T)$, consisting of a finite state set S, an initial state $s_0 \in S$, a finite set of input letters I, a finite set of configuration letters C, a finite set of output letters O, and a transition function $T : S \times I \times C \to S \times O$ that computes the next state and the output letter from the current state, the input letter and the configuration letter.

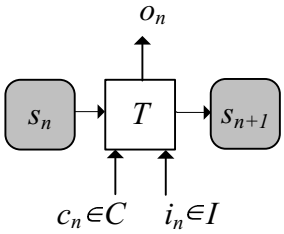

Figure 1: Mealy finite state machine with configuration

As shown in Figure 1, as well as in the remainder of this paper, the state is represented as a gray round corner box, and the transition function T is represented as a white rectangle. We denote the state, the input letter, the output letter and the configuration letter at the n-th cycle respectively as s_n, i_n, o_n and c_n. We further denote the sequence of state, input letter, output letter and configuration letter from the n-th to the m-th cycle respectively as s_n^m, i_n^m, o_n^m and c_n^m. A **path** is a state sequence s_n^m with $\exists i_j o_j (s_{j+1}, o_j) \equiv T(s_j, i_j, c_j)$ for all $n \leq j < m$. A **loop** is a path s_n^m with $s_n \equiv s_m$.

Kroening et al. [13] defined the **state variables recurrence diameter** with respect to M, denoted by $rrd(M)$, as the longest path without loop in M starting from an initial state. In this paper, we define a similar concept: the **uninitialized state variables recurrence diameter** with respect to M, denoted by $uirrd(M)$, as the longest path without loop in M.

$$uirrd(M) \stackrel{def}{=} \max\{i | \exists s_0 \dots s_i i_0 \dots i_i o_0 \dots o_i c : \bigwedge_{j=0}^{i-1} (s_{j+1}, o_j) \equiv T(s_j, i_j, c) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} s_j \neq s_k\} \quad (1)$$

The only difference between these two definitions is that our $uirrd$ does not consider the initial state. These definitions are only used in proving our theorems below. Our algorithm does not need to compute these diameters.

2.3 The original algorithm to determine the existence of the decoder

The complementary synthesis algorithm[1] includes two steps: determining the existence of the decoder and characterizing its Boolean function. We only introduce the first step here.

Definition 2. **An assertion(or a formula) on configuration pins** is a configuration letter set R. For a configuration letter c, $R(c)$ means $c \in R$. If $R(c)$ holds, we also say that R covers c.

As shown in Figure 2, a sufficient condition for the existence of E^{-1} is, there exist three parameters p, d and l, so that

i_n of E can be uniquely determined by the output sequence o_{n+d-l}^{n+d-1}. d is the relative delay between o_{n+d-l}^{n+d-1} and i_n, while l is the length of o_{n+d-l}^{n+d-1}, and p is the length of the prefix path used to rule out some unreachable states. This condition is formally defined below:

Definition 3. **Parameterized Complementary Condition (PC)**: For encoder E, assertion R, and three integers p,d and l, $E \vDash PC(p,d,l,R)$ holds if

1. i_n can be uniquely determined by o_{n+d-l}^{n+d-1} on s_{n-p}^{n+d-1}.
2. R covers all c_x, where $n-p \le x \le n+d-1$.

This equals the unsatisfiability of $F_{PC}(p,d,l,R)$ in Equation (2). We further define $E \vDash PC(R)$ as $\exists p,d,l : E \vDash PC(p,d,l,R)$.

$$F_{PC}(p,d,l,R) \stackrel{def}{=} \left\{ \begin{array}{ll} & \bigwedge_{m=n-p}^{n+d-1}\{(s_{m+1},o_m) \equiv T(s_m,i_m,c_m)\} \\ \wedge & \bigwedge_{m=n-p}^{n+d-1}\{(s'_{m+1},o'_m) \equiv T(s'_m,i'_m,c'_m)\} \\ \wedge & \bigwedge_{m=n+d-l}^{n+d-1} o_m \equiv o'_m \\ \wedge & i_n \neq i'_n \\ \wedge & \bigwedge_{x=n-p}^{n+d-1} c_x \equiv c \\ \wedge & \bigwedge_{x=n-p}^{n+d-1} c'_x \equiv c \\ \wedge & R(c) \end{array} \right\} \quad (2)$$

The 2nd to 5th lines of Equation (2) correspond to Condition 1 of Definition 3. The 2nd and the 3rd lines of Equation (2) correspond respectively to two paths of E. The only difference between them is that a prime is appended to every variable in the 3rd line. The 4th line forces the output sequences of these two paths to be the same, while the 5th line forces their input letters to be different.

At the same time, the last three lines of Equation (2) correspond to Condition 2 of Definition 3. The 6th and the 7th lines constrain that all configuration letters are equal to c, while the last line constrains c to be covered by R.

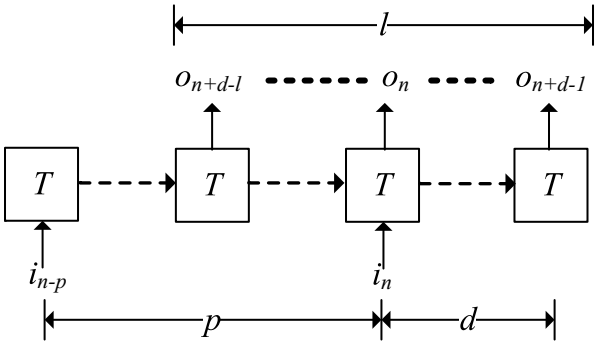

Figure 2: The parameterized complementary condition

The algorithm based on checking $E \vDash PC(R)$[1, 2] just enumerates all combinations of p,d and l, from small to large, until $F_{PC}(p,d,l,R)$ becomes unsatisfiable, which means that the decoder E^{-1} exists.

3. A HALTING ALGORITHM TO INFER ASSERTION

PC in Definition 3 only defines how to determine the existence of decoder E^{-1}. So when the decoder does not exist, the old algorithm[1, 2] will never halt. To find a halting algorithm, we must define how to determine the non-existence of E^{-1}. This will be discussed in subsection 3.1, while the proof of its correctness is presented in subsection 3.2. Based on this result, the overall framework of our algorithm will be presented in subsection 3.3,

3.1 Determining the non-existence of the decoder

According to Definition 3 and Figure 2, E^{-1} exists if there is a parameter value tuple $< p,d,l >$ that makes $E \vDash PC(p,d,l,R)$ holds. So, intuitively, E^{-1} does not exist if for every parameter value tuple $< p,d,l >$, we can always find another tuple $< p',d',l' >$ with $p' > p$,$l' > l$ and $d' > d$, such that $E \vDash PC(p',d',l',R)$ does not hold.

This case can be detected by the SAT instance in Figure 3, which is similar to Figure 2, except three new constraints are inserted to detect loops on s_{n-p}^{n+d-l},$s_{n+d-l+1}^{n}$ and s_{n+1}^{n+d}.

If this SAT instance is satisfiable, for any parameter value $< p,d,l >$, we can unfold these three loops until we find $< p',d',l' >$ that is larger than $< p,d,l >$. We will prove in next subsection that this unfolded instance is still satisfiable, which means $E \vDash PC(p',d',l',R)$ does not hold. So the decoder does not exist.

According to Line 2 and 3 of Equation (2), there are actually two paths, so we need to detect these loops on both of them, i.e., on the product machine M^2 defined below:

Definition 4. **Product machine:** For Mealy machine $M = (S, s_0, I, C, O, T)$, its product machine is $M^2 = (S^2, s_0^2, I^2, C^2, O^2, T^2)$, where T^2 is defined as $(< s_{m+1}, s'_{m+1} >, < o_m, o'_m >$

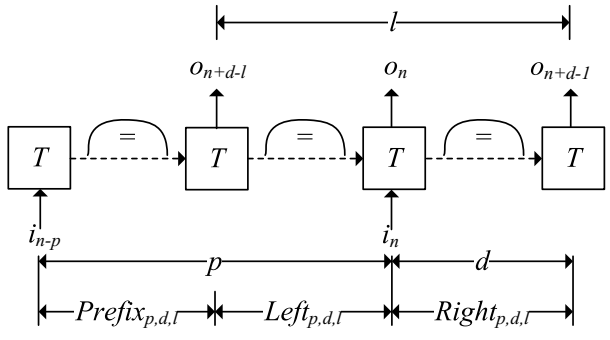

Figure 3: The loop-like non-complementary condition

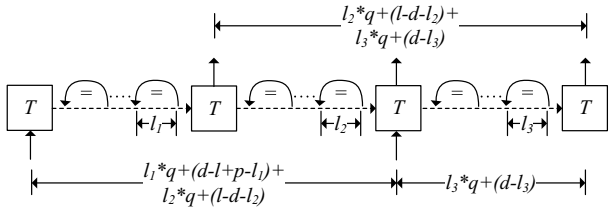

Figure 4: The loop-like non-complementary condition unfolded for q times

$) = T^2(< s_m, s'_m >, < i_m, i'_m >, < c_m, c'_m >)$ with $(s_{m+1}, o_m) = T(s_m, i_m, c_m)$ and $(s'_{m+1}, o'_m) = T(s'_m, i'_m, c'_m)$.

Thus, we define the loop-like non-complementary condition below to determine the non-existence of E^{-1}:

Definition 5. **Loop-like Non-complementary Condition (LN):** For encoder E and its Mealy machine $M = (S, s_0, I, C, O, T)$, assume its product machine is $M^2 = (S^2, s_0^2, I^2, C^2, O^2, T^2)$, then $E \vDash LN(p, d, l, R)$ holds if i_n can not be uniquely determined by o_{n+d-l}^{n+d-1} on the path s_{n-p}^{n+d-1}, and there are loops on $(s^2)_{n-p}^{n+d-l}$, $(s^2)_{n+d-l+1}^{n}$ and $(s^2)_{n+1}^{n+d}$. This equals the satisfiability of $F_{LN}(p, d, l, R)$ in Equation (3). We further define $E \vDash LN(R)$ as $\exists p, d, l : E \vDash LN(p, d, l, R)$.

$$F_{LN}(p,d,l,R) \stackrel{def}{=} \left\{ \begin{array}{rc} & \bigwedge_{m=n-p}^{n+d-1}\{(s_{m+1},o_m) \equiv T(s_m,i_m,c_m)\} \\ \wedge & \bigwedge_{m=n-p}^{n+d-1}\{(s'_{m+1},o'_m) \equiv T(s'_m,i'_m,c'_m)\} \\ \wedge & \bigwedge_{m=n+d-l}^{n+d-1} o_m \equiv o'_m \\ \wedge & i_n \neq i'_n \\ \wedge & \bigwedge_{x=n-p}^{n+d-1} c_x \equiv c \\ \wedge & \bigwedge_{x=n-p}^{n+d-1} c'_x \equiv c \\ \wedge & R(c) \\ \wedge & \bigvee_{x=n-p}^{n+d-l-1} \bigvee_{y=x+1}^{n+d-l}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+d-l+1}^{n-1} \bigvee_{y=x+1}^{n}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+1}^{n+d-1} \bigvee_{y=x+1}^{n+d}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \end{array} \right\} \quad (3)$$

By comparing Equation (2) and (3), it is obvious that their only difference lies in the last three newly inserted lines in (3), which will be used to detect loops on the following three paths shown in Figure 3:

$$\begin{array}{c} Prefix_{p,d,l} = (s^2)_{n-p}^{n+d-l} \\ Left_{p,d,l} = (s^2)_{n+d-l+1}^{n} \\ Right_{p,d,l} = (s^2)_{n+1}^{n+d} \end{array} \quad (4)$$

In the next subsection, we will prove that $E \vDash LN(R) \leftrightarrow \neg\{E \vDash PC(R)\}$, that is, LN and PC are exclusive to each other.

Before proceeding to next subsection, we need to define how to unfold these loops in above three paths. Assume the length of loops in $Prefix_{p,d,l}$, $Left_{p,d,l}$ and $Right_{p,d,l}$ are l_1, l_2 and l_3 respectively. Further assume they are unfolded for q times. Then, the SAT instance generated from this unfolding is shown in Figure 4. It is obvious that, the unfolded SAT instance corresponds to $F_{LN}(p", d", l", R)$, where:

$$\begin{array}{rcc} p" & = & l_1 * q + (d - l + p - l_1) + l_2 * q + (l - d - l_2) \\ d" & = & l_3 * q + (d - l_3) \\ l" & = & l_2 * q + (l - d - l_2) + l_3 * q + (d - l_3) \end{array} \quad (5)$$

Obviously, for every particular $< p, d, l >$ and $< p', d', l' >$, there always exists a q, such that $Prefix_{p",d",l"}$, $Left_{p",d",l"}$ and $Right_{p",d",l"}$ resulted from this unfolding are **not shorter** than $Prefix_{p',d',l'}$,$Left_{p',d',l'}$ and $Right_{p',d',l'}$ respectively.

3.2 Proving correctness

We first need to prove some lemmas.

Lemma 1 (). *For $F_{LN}(p", d", l", R)$ in Figure 4, we have $F_{LN}(p, d, l, R) \rightarrow F_{LN}(p", d", l", R)$*

Proof. The formula $F_{LN}(p", d", l", R)$ is:

$$F_{LN}(p",d",l",R) \stackrel{def}{=} \left\{ \begin{array}{rc} & \bigwedge_{m=n-p"}^{n+d"-1}\{(s_{m+1},o_m) \equiv T(s_m,i_m,c_m)\} \\ \wedge & \bigwedge_{m=n-p"}^{n+d"-1}\{(s'_{m+1},o'_m) \equiv T(s'_m,i'_m,c'_m)\} \\ \wedge & \bigwedge_{m=n+d"-l"}^{n+d"-1} o_m \equiv o'_m \\ \wedge & i_n \neq i'_n \\ \wedge & \bigwedge_{x=n-p"}^{n+d"-1} c_x \equiv c \\ \wedge & \bigwedge_{x=n-p"}^{n+d"-1} c'_x \equiv c \\ \wedge & R(c) \\ \wedge & \bigvee_{x=n-p"}^{n+d"-l"-1} \bigvee_{y=x+1}^{n+d"-l"}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+d"-l"+1}^{n-1} \bigvee_{y=x+1}^{n}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+1}^{n+d"-1} \bigvee_{y=x+1}^{n+d"}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \end{array} \right\} \quad (6)$$

Assume that $F_{LN}(p, d, l, R)$ is satisfied, and its satisfying assignment is A. We need to prove that $F_{LN}(p", d", l", R)$ is also satisfied with A.

The directed arcs numbered from 1 to 12 in Figure 5, show the correspondence between $F_{LN}(p, d, l, R)$ and the $F_{LN}(p", d", l", R)$.

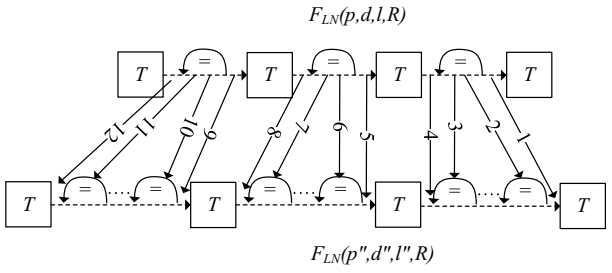

Figure 5: Correspondance between $F_{LN}(p, d, l, R)$ and $F_{LN}(p", d", l", R)$

The arcs 2 and 3 mean that we can apply the satisfying assignment of the loop in $Right_{p,d,l}$ to the unfolded loops in $Right_{p",d",l"}$. The arcs 1 and 4 mean that we can apply the the satisfying assignments of the two paths not in the loop, to $Right_{p",d",l"}$. With the arcs from 1 to 4, we can make the path $Right_{p",d",l"}$ satisfiable.

Similarly, we can also make $Prefix_{p",d",l"}$ and $Left_{p",d",l"}$ satisfiable. Thus the 2nd line of Equation (6) is satisfied with the assignment A. Similarly, the 3rd to 8th lines of Equation (6) are also satisfied with the assignment A.

At the same time, there are q loops in $Prefix_{p",d",l"}$, $Left_{p",d",l"}$ and $Right_{p",d",l"}$, which will make the last three lines in Equation (6) satisfied.

Thus,the satisfying assignment A of $F_{LN}(p,d,l,R)$ can also make $F_{LN}(p",d",l",R)$ satisfied. This concludes the proof. □

Lemma 2 (). *For two tuples $< p,d,l >$ and $< p',d',l' >$, if $Prefix_{p',d',l'}$,$Left_{p',d',l'}$ and $Right_{p',d',l'}$ are **not shorter** than $Prefix_{p,d,l}$,$Left_{p,d,l}$ and $Right_{p,d,l}$ respectively, then $E \vDash PC(p,d,l,R) \rightarrow E \vDash PC(p',d',l',R)$.*

PROOF. It is obvious that $F_{PC}(p,d,l,R)$ is a sub-formula of $F_{PC}(p',d',l',R)$, so the unsatisfiability of the former implies the unsatisfiability of the latter. Thus, $E \vDash PC(p,d,l,R) \rightarrow E \vDash PC(p',d',l',R)$ holds. □

The following two theorems will prove that $E \vDash LN(R) \leftrightarrow \neg\{E \vDash PC(R)\}$.

Theorem 2 (). $E \vDash LN(R) \rightarrow \neg\{E \vDash PC(R)\}$

PROOF. We can prove it by contradiction. Assume that $E \vDash LN(R)$ and $E \vDash PC(R)$ both hold. This means there exist $< p,d,l >$ and $< p',d',l' >$, such that $E \vDash PC(p,d,l,R)$ and $E \vDash LN(p',d',l',R)$.

On one hand, $E \vDash LN(p',d',l',R)$ means there are loops in $Prefix_{p',d',l'}$,$Left_{p',d',l'}$ and $Right_{p',d',l'}$. By unfolding these loops, we get another tuple $< p",d",l" >$, so that :

1. $Prefix_{p",d",l"}$,$Left_{p",d",l"}$ and $Right_{p",d",l"}$ are longer than $Prefix_{p,d,l}$,$Left_{p,d,l}$ and $Right_{p,d,l}$ respectively
2. According to Lemma 1, $F_{LN}(p",d",l",R)$ is satisfiable.

$F_{PC}(p",d",l",R)$ is a sub-formula of $F_{LN}(p",d",l",R)$, so $F_{PC}(p",d",l",R)$ is also satisfiable, which means that $E \vDash PC(p",d",l",R)$ does not hold.

On the other hand, according to Lemma 2, $E \vDash PC(p",d",l",R)$ holds. This contradiction concludes the proof. □

Theorem 3 (). $E \vDash LN(R) \leftarrow \neg\{E \vDash PC(R)\}$

PROOF. We can also prove it by contradiction. Assume that neither $E \vDash LN(R)$ nor $E \vDash PC(R)$ holds. Then for every $< p,d,l >$ and $< p',d',l' >$, $F_{PC}(p,d,l,R)$ is satisfiable, while $F_{LN}(p',d',l',R)$ is unsatisfiable.

Thus, assume $uirrd(M^2)$ is the uninitialized state variables recurrence diameter of E's product machine. Let's define $< p,d,l >$ as:

$$\begin{aligned} p &= uirrd(M^2) * 2 + 2 \\ d &= uirrd(M^2) + 1 \\ l &= uirrd(M^2) * 2 + 2 \end{aligned} \tag{7}$$

With this definition, it is obvious that $Prefix_{p,d,l}$,$Left_{p,d,l}$ and $Right_{p,d,l}$ are all longer than $uirrd(M^2)$. This means there are loops in all these three paths, which will make $F_{LN}(p,d,l,R)$ satisfiable. This contradicts with the fact that $F_{LN}(p',d',l',R)$ is unsatisfiable for every $< p',d',l' >$. This contradiction concludes the proof. □

3.3 Algorithm implementation

Theorems 2 and 3 show that, we can enumerate all combinations of $< p,d,l >$ from small to large, and check $E \vDash PC(p,d,l,R)$ and $E \vDash LN(p,d,l,R)$ in every iteration. This process will eventually terminate with one and only one answer between $E \vDash PC(R)$ and $E \vDash LN(R)$. The implementation of this algorithm will be presented below.

Algorithm 1 InferAssertion

```
1: NA = {}
2: for x = 0 → ∞ do
3:    < p,d,l >=< 2x,x,2x >
4:    if F_PC(p,d,l,⋀_{na∈NA} ¬na) is unsatisfiable then
5:       if ⋀_{na∈NA} ¬na is satisfiable then
6:          decoder exists with final assertion ⋀_{na∈NA} ¬na
7:       else
8:          decoder does not exist
9:       end if
10:      halt
11:   else
12:      while F_LN(p,d,l,⋀_{na∈NA} ¬na) is satisfiable do
13:         let c be the configuration letter leading to the
            non-existence of decoder
14:         na ← InferCoveringFormula(c)
15:         NA ← NA ∪ {na}
16:      end while
17:   end if
18: end for
```

In Line 1 of Algorithm 1, NA will be used to record all inferred formulas that can lead to the non-existence of the decoder. They are all inferred by the procedure $InferCovering\ Formula$ in Line 14, whose functionality is to infer a formula that can cover not only c, but also many other configuration letters leading to the non-existence of the decoder. More details of this procedure will be presented in Section 4.

Line 3 ensures that the length of $Prefix_{p,d,l}$,$Left_{p,d,l}$ and $Right_{p,d,l}$ are all set to x, whose value is enumerated in Line 2. In this way, many redundant combinations of p,d and l are no longer need to be tested. Thus, the performance of this algorithm can be significantly boosted.

Line 4 means the input letter can be uniquely determined by the output sequence with the assertion $\bigwedge_{na \in NA} \neg na$. Line 5 means that there is at least one configuration letter that can lead to the existence of the decoder, and the final assertion is $\bigwedge_{na \in NA} \neg na$ in the 6th line.

Line 7 means that the inferred assertion $\bigwedge_{na \in NA} \neg na$ has ruled out all configuration letters, that is, no configuration letter can lead to the existence of the decoder. There must be some bugs in the encoder.

Line 11 means that the decoder does not exist with the configuration letter c in Line 12. We need to rule out c such that Algorithm 1 can continue searching for other configuration letters that may lead to the existence of the decoder. The procedure $InferCoveringFormula$ in Line 14 will be used to infer a formula na that covers not only c, but also a large set of invalid configuration letters. They will be ruled out in Line 15.

We can prove that Algorithm 1 is a halting one.

Theorem 4 (). *Algorithm 1 is a halting algorithm.*

PROOF. According to Theorems 2 and 3, Algorithm 1 will eventually reach Line 4 or 11.

In the former case, this algorithm will halt at Line 10.

In the latter case, a new formula na will be inferred, which will cover the configuration letter c. Because the number of such c is finite, all of them will eventually be ruled out by $\bigwedge_{na \in NA} \neg na$. Then Algorithm 1 will eventually reach Line 4, and halt at Line 10. □

4. INFERRING FORMULA THAT COVERS INVALID CONFIGURATION LETTER

This section will introduce the implementation of $InferCoveringFormula$ in Line 14 of Algorithm 1. It will be used to infer a Boolean formula na that covers not only c, but also many other configuration letters leading to the non-existence of the decoder. This job will be accomplished in the following three steps:

1. Transforming F_{LN} into an equivalent form with an object variable, such that it can be used to defined a Boolean function f.
2. Eliminating some variables from the support set of f with cofactoring[3], until only c remains.
3. Characterizing f with Craig Interpolation. This f is also the formula na in Line 14 of Algorithm 1.

These steps will be presented in the following subsections.

4.1 An equivalent form of F_{LN}

As mentioned above, we need to transform F_{LN} into another equivalent form with an object variable.

First, we need to move the 4th line and the last three lines of Equation (3) into a new subformula:

$$G(p,d,l) \stackrel{def}{=} \left\{ \begin{array}{cc} & \bigwedge_{m=n+d-l}^{n+d-1} o_m \equiv o'_m \\ \wedge & \bigvee_{x=n-p}^{n+d-l-1} \bigvee_{y=x+1}^{n+d-l} \{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+d-l+1}^{n-1} \bigvee_{y=x+1}^{n} \{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+1}^{n+d-1} \bigvee_{y=x+1}^{n+d} \{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \end{array} \right\} \quad (8)$$

And then, F_{LN} can be transformed into :

$$F'_{LN}(p,d,l,R) \stackrel{def}{=} \left\{ \begin{array}{cc} & \bigwedge_{m=n-p}^{n+d-1} \{(s_{m+1}, o_m) \equiv T(s_m, i_m, c_m)\} \\ \wedge & \bigwedge_{m=n-p}^{n+d-1} \{(s'_{m+1}, o'_m) \equiv T(s'_m, i'_m, c'_m)\} \\ \wedge & i_n \neq i'_n \\ \wedge & \bigwedge_{x=n-p}^{n+d-1} c_x \equiv c \\ \wedge & \bigwedge_{x=n-p}^{n+d-1} c'_x \equiv c \\ \wedge & R(c) \\ \wedge & t \equiv G(p,d,l) \end{array} \right\} \quad (9)$$

It is obvious that F_{LN} and $F'_{LN} \wedge t \equiv 1$ is equisatisfiable.

At the same time, according to Figure 3, F'_{LN} actually defines a function $f' : S^2 \times I^{(d+p)*2} \times C \to B$, whose support set $supp(f')$ is $\{s_{n-p}, s'_{n-p}, i_{n-p}^{n+d-1}, (i')_{n-p}^{n+d-1}, c\}$, and its output is the object variable t in the last line of Equation (9).

4.2 Cofactoring

According to Line 11 of Algorithm 1, F_{LN} is satisfiable. We further assume that A is the satisfying assignment of F_{LN}. We can just assert the value of i_{n-p}^{n+d-1}, $(i')_{n-p}^{n+d-1}$, s_{n-p} and $(s')_{n-p}$ into formula F'_{LN}, and get :

$$F"_{LN}(c,t) \stackrel{def}{=} \left\{ \begin{array}{cc} & F'_{LN} \\ \wedge & i_{n-p}^{n+d-1} \equiv A(i_{n-p}^{n+d-1}) \\ \wedge & (i')_{n-p}^{n+d-1} \equiv A((i')_{n-p}^{n+d-1}) \\ \wedge & s_{n-p} \equiv A(s_{n-p}) \\ \wedge & (s')_{n-p} \equiv A((s')_{n-p}) \end{array} \right\} \quad (10)$$

Now, $F"_{LN}$ defines another function $f"$, whose support set is reduced to c. $F"_{LN}(c,t) \wedge t \equiv 1$ is the formula that covers a set of invalid configuration letters. But it is still a large complicated CNF clause set. To reduce its size, we need the characterizing algorithm in the next subsection.

4.3 Characterizing $f"$ with Craig interpolation

We then encode $F"_{LN}(c,t)$ into the CNF format, and denote it as $CNF(F"_{LN}(c,t))$. Assume $CNF'(F"_{LN}(c,t'))$ is a copy of $CNF(F"_{LN}(c,t))$. They share the same variable index for c, while all other variables are encoded independently. Thus, we can construct formula ϕ_A and ϕ_B as:

$$\phi_A \stackrel{def}{=} CNF(F"_{LN}(c,t)) \wedge t \equiv 1 \quad (11)$$

$$\phi_B \stackrel{def}{=} CNF'(F"_{LN}(c,t')) \wedge t' \equiv 0 \quad (12)$$

It is obvious that $\phi_A \wedge \phi_B$ is unsatisfiable. With McMillan's algorithm[11], we can generate an interpolant, which is a circuit with Boolean function $ITP : C \rightarrow B$. According to Theorem 1, ITP is inconsistent with ϕ_B in Equation (12). So it characterizes a set $C' \subseteq C$ that can make ϕ_A in Equation (11) satisfiable. According to Equations (9) and (10), it is obvious that:

1. The c in Line 12 of of Algorithm 1 is in C'. According to Theorem 4, this will ensure that Algorithm 1 is halting.
2. All $c' \in C'$ can also lead to the non-existence of the decoder. This will speedup Algorithm 1 significantly.

5. REMOVING REDUNDANCY

Algorithm 2 *RemoveRedundancy*(p, d, l, R)

```
1: for p' = p → 0 do
2:    if F_PC(p' − 1, d, l, R) is satisfiable then
3:       break
4:    end if
5: end for
6: for d' = d → 0 do
7:    if F_PC(p', d' − 1, l, R) is satisfiable then
8:       break
9:    end if
10: end for
11: for l' = 1 → l − (d − d') do
12:    if F_PC(p', d', l', R) is unsatisfiable then
13:       break
14:    end if
15: end for
16: print "final result is < p', d', l' >"
```

The $< p, d, l >$ found by Algorithm 1 contains some redundancy, which will cause unnecessarily large overheads on the circuit area and the run time of the characterizing Boolean function of the decoder. So, Algorithm 2 is used to minimize $< p, d, l >$ before passing it to the characterization algorithm. This algorithm just iteratively reduces the value of p, d and l, and tests whether the reduced values can still make $E \vDash PC(R)$ holds.

6. EXPERIMENTAL RESULTS

We have implemented this algorithm in the OCaml language, and solved the generated SAT instances with Minisat solver[8]. All experiments are run on a PC with a 2.4GHz Intel Core 2 Q6600 processor, 8GB memory and Ubuntu 10.04 linux.

6.1 Benchmarks

Table 1 shows information of the following benchmarks.

1. A XGXS encoder compliant to clause 48 of IEEE-802.3ae 2002 standard [10].
2. A XFI encoder compliant to clause 49 of the same IEEE standard.
3. A 66-bit scrambler used to ensure that a data sequence has sufficiently many 0-1 transitions, so that it can run through a high-speed noisy serial transmission channel.
4. A PCI-E physical coding module [9].
5. The Ethernet module of Sun's OpenSparc T2 processor.

6.2 Results

The 2nd row of Table 2 shows the run time of checking the decoder's existence with the other halting algorithm proposed previously by Shen et.al[14]. And the 3rd row shows the value of d, p and l discovered by that algorithm.

At the same time, the 4th row shows the configuration pin's bit number, the 5th row shows the run time of inferring assertion with our new algorithm, while the last line shows the value of d, p and l discovered.

By comparing the 2nd and the 5th rows, it is obvious that our approach is much slower than that of [14], which is caused by the much more complicated procedure $InferCoveringFormula$ used to infer new formulas.

By comparing the 3rd and the 6th rows, it is obvious that there are some minor differences on those parameter values. This is caused by the different orders in checking various parameter combinations.

6.3 Inferred assertions

We will show here the inferred assertions.

For XGXS: !((bad_disp & !rst & bad_code)) & !((rst & bad_code)) & !((!rst & !bad_disp & bad_code))

For XFI: !((!RESET & !TEST_MODE & !DATA_VALID & DATA _PAT_SEL)) & !((!RESET & !TEST_MODE & !DATA_VA LID & !DATA_PAT_SEL))

For scrambler: True

For PCI-E: !((CNTL_RESETN_P0 & !TXELECIDLE & CNTL_TXEnable_P0 & CNTL_Loopback_P0) | (CNTL_RE SETN_P0 & !TXELECIDLE & !CNTL_TXEnable_P0)) & !((CNTL_RE SETN_P0 & TXELECIDLE) | (CNTL_RES ETN_P0 & !TXELECIDLE & !CNTL_TXEnable_P0) | (!CNTL_RESETN_P0))

For T2 ethernet: !((reset_tx & !txd_sel[0] & !txd_sel[1] & link_up_loc & jitter_study_pci[0] & !jitter_study_pci[1]))

Table 1: Information of Benchmarks

	XGXS	XFI	scrambler	PCIE	T2 ethernet
Line number of Verilog source code	214	466	24	1139	1073
#regs	15	135	58	22	48
Data path width	8	64	66	10	10

Table 2: Experimental Results

		XG-XS	XFI	scrambler	PCI-E	T2 ether
[14]	Run time	0.07	17.84	2.70	0.47	30.59
	d, p, l	1,2,1	0,3,2	0,2,2	2,2,1	4,2,1
ours	#Config pin	3	120	1	16	26
	Run time	3.69	372.15	3.14	30.53	188.56
	d, p, l	1,3,1	0,4,2	0,2,2	2,2,1	4,4,1

& !((reset_tx & !txd_sel[0] & !txd_sel[1] & link_up_loc & jitter_study_pci[0] & jitter_study_pci[1])) & !((reset_tx & !txd_sel[0] & !txd_sel[1] & link_up_loc & !jitter_study_pci[0] & jitter_study_pci[1])) & !((jitter_study_pci[1] & jitter_stu dy_pci[0] & !txd_sel[0] & !txd_sel[1] & !reset_tx & link_up_loc)) & !((jitter_study_pci[1] & !jitter_study_pci[0] & !reset_tx & !txd_sel[0] & !txd_sel[1] & link_up_loc)) & !((jitter_study_pci[0] & !jitter_study_pci[1] & !txd_sel[0] & !txd_sel[1] & !reset_tx & link_up_loc)) & !((reset_tx & !jitter_study_pci[0] & !txd_sel[0] & !txd_sel[1] & link_up_loc & !jitter_study_pci[1]))

7. RELATED WORK

7.1 Complementary synthesis and program inversion

The concept of complementary synthesis was first proposed by Shen et.al[1, 2]. Its major shortcomings is that it is not halting. Shen et.al[14] addressed this problem by building a set of over-approximations that is similar to onion-rings.

According to Gulwani[15], program inversion is the problem that derive a program P^{-1} that negate the computation of a given program P. So the definition of program inversion is very similar to complementary synthesis. Initial work on deriving program inversion used proof-based approaches[16], but it can only handle very small programs and very simple syntax structures. Glück et.al [17] inverts the first-order functional programs by eliminating nondeterminism with LR-based parsing methods. The requirement that the program to be inverted should be expressed in functional language makes it impossible to apply it to our application. Srivastava et.al [18] inductively rules out invalid paths that can't fulfill the requirement of inversion, to narrow down the space of candidate programs until only the valid ones remain. So it can only guarantee the existence of a solution, but not the correctness of this solution if its assumptions do not hold.

7.2 Protocol converter synthesis

The protocol converter synthesis is the problem that automatically generates a translator between two communication protocols. Avnit et.al [19] first defines a general model for describing the different protocols. Then it provides an algorithm to decide whether there are some functionality of a protocol that can't be translated into another. Finally, it synthesizes the translator by computing a greatest fixed point for the update function of buffer's control states.

8. CONCLUSIONS

This paper proposes a fully automatic approach to infer assertion for complementary synthesis. Experimental results show that our approach can infer assertions for many complex encoders, such as PCI-E[9] and Ethernet[10],.

9. ACKNOWLEDGMENTS

This work was funded by projects 60603088 and 61070132 supported by National Natural Science Foundation of China.

10. REFERENCES

[1] S. Shen, J. Zhang, Y. Qin, and S. Li. Synthesizing complementary circuits automatically. In *ICCAD09*, pages 381–388. IEEE, November 2009.

[2] S. Shen, Y. Qin, K. Wang, L. Xiao, J. Zhang, and S. Li. Synthesizing complementary circuits automatically. *IEEE transaction on CAD of Integrated Circuits and Systems*, 29(8):1191–1202, August 2010.

[3] M. K. Ganai, A. Gupta, and P. Ashar. Efficient sat-based unbounded symbolic model checking using circuit cofactoring. In *ICCAD04*, pages 510–517. IEEE, November 2004.

[4] W. Craig. Linear reasoning: A new form of the herbrand-gentzen theorem. *J. Symbolic Logic*, 22(3):250–268, 1957.

[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC01*, pages 530–535. IEEE, June 2001.

[6] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD96*, pages 220–227. IEEE, November 1996.

[7] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *DATE02*, pages 142–149. IEEE, March 2002.

[8] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT03*, pages 502–518. Springer, May 2003.

[9] *en. wikipedia. org/ wiki/ PCI_ Express .*

[10] *en. wikipedia. org/ wiki/ Ethernet .*

[11] K. L. McMillan. Interpolation and sat-based model checking. In *CAV03*, pages 1–13. Springer, July 2003.

[12] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34(5):1045–1079, 1955.

[13] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *VMCAI03*, pages 298–309. Springer, January 2003.

[14] S. Shen, Y. Qin, J. Zhang, and S. Li. A halting algorithm to determine the existence of decoder. In *FMCAD10*, pages 91–100. IEEE, October 2010.

[15] S. Gulwani. Dimensions in program synthesis. In *PPDP10*, pages 13–24. ACM, July 2010.

[16] E. W. Dijkstra. Program inversion. In *Program Construction 1978*, pages 54–57, 1978.

[17] R. Glück and M. Kawabe. A method for automatic program inversion based on lr(0) parsing. *Fundam. Inf.*, 66(4):367–395, November 2005.

[18] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. Foster. Program inversion revisited. *Technical Report MSR-TR-2010-34, Microsoft Research*, 2010.

[19] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, and S. Parameswaran. A formal approach to the protocol converter problem. In *DATE08*, pages 294–299. IEEE, March 2008.