Synthesizing Complementary Circuits without Manually Specifying Assertions

ShengYu Shen, Ying Qin, JianMin Zhang, and SiKun Li
School of Computer Science, National University of Defense Technology
410073,ChangSha,China
Email: {syshen,qy123,jmzhang,skli}@nudt.edu.cn

***Abstract*—Complementary synthesis can automatically synthesize the decoder circuit of an encoder. But 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. For every invalid value of configuration pins that leads to the nonexistence of the decoder, we use Craig interpolation to infer a new formula that covers a larger set of such invalid values. This step is repeated until all invalid values are covered by these inferred formulas. The final assertion is obtained by anding the inverses of all these inferred formulas.

However, multiple decoders may exist simultaneously under this inferred assertion. So we propose another algorithm to discover them, by iteratively testing whether R, the Boolean relation that uniquely determines the input letter, can be expressed as a combination of those discovered decoders. For every value of configuration pins that fails this test, a new decoder's Boolean relation is discovered by asserting this value into R. This step is repeated until all decoders are discovered.

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 and generate decoders for them.

***Index Terms*—Complementary Synthesis, Inferring Assertion, Cofactoring, Craig Interpolation**

I. 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, a complementary synthesis algorithm [1], [2] is proposed to automatically synthesize the decoder circuit of an encoder, by checking whether the encoder's input can be uniquely determined by its output.

However, 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 modes. For example, in the testing and sleep mode, the encoder either processes test commands or just does nothing respectively. In these modes, the encoder's input letter cannot be determined by its output sequence, which leads to the nonexistence of its decoder.

To avoid this tedious job of specifying assertion manually, we propose an automatic approach to infer this assertion: **First**, we use the halting algorithm proposed by Shen et al. [3] to find an invalid value of configuration pins that leads to the nonexistence of the decoder. **Second**, we use cofactoring [4] and Craig interpolation [5] to infer a new formula, which covers a larger set of such invalid values. These two steps are 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.

However, multiple decoders may exist simultaneously under this assertion. For the encoder in Fig. 1, no matter what the value of c, in can be uniquely determined by out. But there are two decoders, one is $in \stackrel{def}{=} out-1$, the other is $in \stackrel{def}{=} out-2$. Such inconvenience is the price that must be paid to avoid the trouble of specifying assertions manually. This problem is not as serious as it appears: according to experimental results, only two of our five benchmarks have two decoders with the other three having only one.

So we propose another algorithm to discover all these decoders, by iteratively testing whether R, the Boolean relation that uniquely determines the input letter, can be expressed as a combination of the discovered decoders. For every value of configuration pins that fails this test, a new decoder's Boolean relation is discovered by asserting this value into R. This step is repeated until all decoders are discovered.

We have run this algorithm on several complex encoders from industrial projects (e.g., PCI-E [6] and Ethernet [7]). Experimental results show that our algorithm can always infer assertions and generate decoders for them. All programs and results can be downloaded from http://www.ssypub.org.

The remainder of this paper is organized as follows. Section II introduces background materials. Section III presents the algorithm that infers assertions, and the proof of its correctness. Section IV reduces the parameters' values discovered by Section III, while Section V discusses how to discover all decoders. Section VI and VII present experimental results and

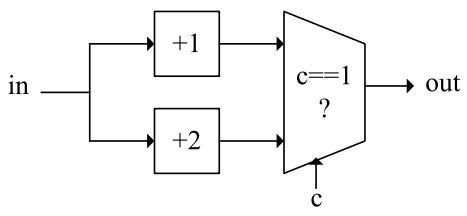

Fig. 1. The simultaneous existence of multiple decoders

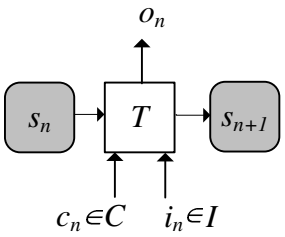

Fig. 2. Mealy finite state machine with configuration

related works. Finally, Section VIII concludes this paper.

II. Preliminaries

A. Propositional satisfiability and related topics

The Boolean value set is denoted as $\mathbb{B} = \{0, 1\}$. For a Boolean formula F over a variable set V, the propositional satisfiability problem(SAT) is to find a satisfying assignment $A : V \to \mathbb{B}$, so that F can be evaluated to 1. If A exists, then F is satisfiable; otherwise, it is unsatisfiable. A SAT solver is a program that decides the existence of A. Normally, the formula F is represented in the conjunctive normal form(CNF), where 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 CNF formula is also called a SAT instance.

For a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$, we use $supp(f)$ to denote its support set $\{v_1 \dots v_n\}$. According to Ganai et al. [4], 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 $f_{\overline{v}}$.

Craig [5] had proved the following theorem:

***Theorem** 1 (Craig Interpolation Theorem):* Given two Boolean formulas ϕ_A and ϕ_B, with $\phi_A \wedge \phi_B$ unsatisfiable, there exists a Boolean 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. ϕ_I is referred to as the **interpolant** of ϕ_A with respect to ϕ_B.

B. Determining the existence of the decoder

Complementary synthesis [1] includes two steps: determining the existence of the decoder and characterizing its Boolean function. We will only introduce the first step here. To model the encoder with configuration pins, we extend the traditional definition of Mealy finite state machine [8]:

***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.

As shown in Fig. 2, 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 $\exists i_j o_j c : (s_{j+1}, o_j) \equiv T(s_j, i_j, c)$ for all $n \le j < m$. A **loop** is a path s_n^m with $s_n \equiv s_m$.

The **uninitialized recurrence diameter** of a Mealy machine M is defined as the longest path without loop:

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

The only difference between this definition and the state variables recurrence diameter [9] is that our $uirrd$ does not consider the initial state. We only used this definition to prove our theorems. Our algorithm does not compute it.

An assertion(or formula) on configuration pins is defined as 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 Fig. 3, the decoder exists if there are three parameters p, d and l, so that i_n of the encoder 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 can be formally defined as:

***Definition** 2:* **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 i_n can be uniquely determined by o_{n+d-l}^{n+d-1}, and R covers the constant configuration letter c. This equals the unsatisfiability of $F_{PC}(p, d, l, R)$ in Equation (2). We further define $E \vDash PC(R)$ as $\exists p, d, l : E \vDash PC(p, d, l, R)$.

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

Line 2 and 3 of Equation (2) correspond respectively to two paths of E. Line 4 forces the output sequences of these two paths to be the same, while Line 5 forces their input letters to be different. The last line constrains c to be covered by R.

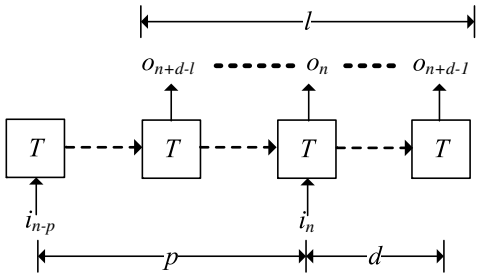

Fig. 3. The parameterized complementary condition

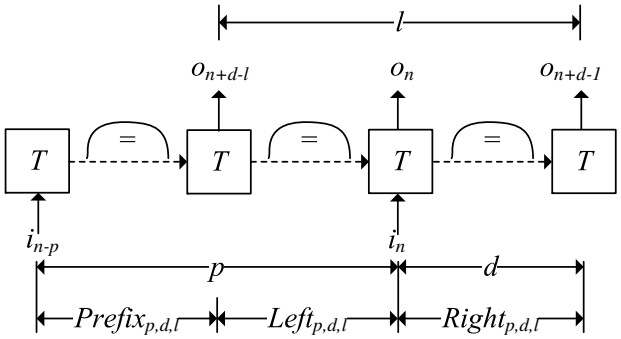

Fig. 4. The loop-like non-complementary condition

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 exists.

However, if the decoder does not exist, this algorithm will not halt. To find a halting algorithm, Shen et al. [3] defined how to determine the nonexistence of the decoder.

According to Definition 2 and Fig. 3, the decoder exists if there are three parameter values p, d and l that make $E \vDash PC(p,d,l,R)$ hold. So, intuitively, the decoder does not exist if for every combination of parameter values p, d and l, we can always find another combination p', d' and 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 Fig. 4, which is similar to Fig. 3, except that three new constraints are inserted to detect loops on paths 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 values p, d and l, we can unfold these three loops until we find p' ,d' and l' that are larger than p, d and l. It is obvious 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. Thus, the nonexistence of the decoder can be determined by:

***Definition** 3:* **Loop-like Non-complementary Condition (LN):**For encoder E and its Mealy machine $M = (S, s_0, I, C, O, T)$, $E \vDash LN(p,d,l,R)$ 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_{n-p}^{n+d-l}, $s_{n+d-l+1}^{n}$ and s_{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 $\exists p,d,l : E \vDash LN(p,d,l,R)$.

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

The only difference between Equation (2) and (3) lies in the last three lines of (3), which will be used to detect loops.

According to Shen et al. [3], we have the following theorem:

***Theorem** 2 ():* $E \vDash LN \leftrightarrow \neg\{E \vDash PC\}$

So, the halting algorithm [3] just enumerates all combinations of p, d and l, and checks $E \vDash PC(p,d,l,R)$ and $E \vDash LN(p,d,l,R)$ in every iteration. According to Theorem 2, it will eventually halt with one and only one answer between $E \vDash PC(R)$ and $E \vDash LN(R)$.

III. Inferring Assertion

A. The overall algorithm framework

The algorithm that infers assertion is 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 if F_LN(p,d,l, ⋀_{na∈NA} ¬na) is satisfiable then
12:         let A be the satisfying assignment, and A(c) be the
            configuration letter leading to the nonexistence of
            decoder
13:         na ← InferCoveringFormula(A(c))
14:         NA ← NA ∪ {na}
15:     end if
16: end for
```

In Line 1, NA is used to record all inferred formulas that can lead to the nonexistence of the decoder. They are all inferred by the procedure $InferCoveringFormula$ in Line 13, whose functionality is to infer a formula that can cover not only the current configuration letter, but also many other configuration letters leading to the nonexistence of the decoder. More details of this procedure will be presented in the next subsection.

Line 3 ensures that the lengths of s_{n-p}^{n+d-l}, $s_{n+d-l+1}^{n}$ and s_{n+1}^{n+d} are all set to x, whose value is enumerated in Line 2. In this way, many redundant combinations of p,d and l no longer need to be tested.

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 $\bigwedge_{na\in NA}\neg na$.

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 $A(c)$ in Line 12. The procedure $InferCoveringFormula$ in Line 13 is used to infer a formula na

that covers not only $A(c)$, but also a large set of invalid configuration letters. They will be ruled out in Line 14.

We can prove that Algorithm 1 is a halting one.

***Theorem** 3 ():* Algorithm 1 is a halting algorithm.

Proof: According to Theorem 2, 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 $A(c)$. Since the number of such $A(c)$ 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. ■

B. Inferring new formula covering invalid configuration letter

This section will introduce the implementation of $InferCoveringFormula$ in Line 13 of Algorithm 1. It will be used to infer a Boolean formula na that covers not only $A(c)$, but also many other configuration letters leading to the nonexistence of the decoder. This job will be accomplished in the following three steps:

First, deriving an equivalent form of F_{LN} with an object variable, such that it can be used to define a Boolean function f. **Second**, reducing the support set of f with cofactoring [4], until only c remains. **Third**, characterizing f with Craig Interpolation. This f is also the formula na in Line 13 of Algorithm 1.

These three steps will be presented below:

*1) **Deriving an equivalent form of** F_{LN} **with an object variable**:* 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:

$$G(p,d,l)\stackrel{def}{=}\left\{\begin{array}{cc} & \bigwedge_{m=n+d-l}^{n+d-1} o_m\equiv o'_m \\ \wedge & \bigvee_{x=n-p}^{n+d-l-1}\bigvee_{y=x+1}^{n+d-l}\{s_x\equiv s_y\wedge s'_x\equiv s'_y\} \\ \wedge & \bigvee_{x=n+d-l+1}^{n-1}\bigvee_{y=x+1}^{n}\{s_x\equiv s_y\wedge s'_x\equiv s'_y\} \\ \wedge & \bigvee_{x=n+1}^{n+d-1}\bigvee_{y=x+1}^{n+d}\{s_x\equiv s_y\wedge s'_x\equiv s'_y\} \end{array}\right\} \quad (4)$$

Then, F_{LN} can be transformed into :

$$F'_{LN}(p,d,l,R)\stackrel{def}{=}\left\{\begin{array}{cc} & \bigwedge_{m=n-p}^{n+d-1}\{(s_{m+1},o_m)\equiv T(s_m,i_m,c)\} \\ \wedge & \bigwedge_{m=n-p}^{n+d-1}\{(s'_{m+1},o'_m)\equiv T(s'_m,i'_m,c)\} \\ \wedge & i_n\neq i'_n \\ \wedge & R(c) \\ \wedge & t\equiv G(p,d,l) \end{array}\right\} \quad (5)$$

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

At the same time, according to Fig. 4, F'_{LN} actually defines a function $f':S^2\times I^{(d+p)*2}\times C\rightarrow\mathbb{B}$, 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 (5).

*2) **Reducing the support set of** f **with cofactoring**:* According to Line 11 of Algorithm 1, F_{LN} is satisfiable, and A is its satisfying assignment. 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 :

$$F"_{LN}(c,t)\stackrel{def}{=}\left\{\begin{array}{cc} & F'_{LN} \\ \wedge & i_{n-p}^{n+d-1}\equiv A(i_{n-p}^{n+d-1}) \\ \wedge & (i')_{n-p}^{n+d-1}\equiv A((i')_{n-p}^{n+d-1}) \\ \wedge & s_{n-p}\equiv A(s_{n-p}) \\ \wedge & (s')_{n-p}\equiv A((s')_{n-p}) \end{array}\right\} \quad (6)$$

Now, $F"_{LN}$ defines another function $f"$, whose support set is reduced to c. It is obvious that $F"_{LN}(c,t)\wedge t\equiv 1$ 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 step.

*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 only, while all other variables are encoded independently. Thus, we can construct ϕ_A and ϕ_B as:

$$\phi_A\stackrel{def}{=}CNF(F"_{LN}(c,t))\wedge t\equiv 1 \quad (7)$$

$$\phi_B\stackrel{def}{=}CNF'(F"_{LN}(c,t'))\wedge t'\equiv 0 \quad (8)$$

Obviously, $\phi_A\wedge\phi_B$ is unsatisfiable. With McMillan's interpolant generating algorithm [10], we can generate an interpolant circuit with Boolean function $ITP:C\rightarrow\mathbb{B}$.

According to Theorem 1, ITP is inconsistent with ϕ_B in Equation (8). So it characterizes a set $C'\subseteq C$ that can make ϕ_A in Equation (7) satisfiable. According to Equations (5) and (6), it is obvious that:

1) The $A(c)$ in Line 12 of Algorithm 1 is in C'. According to Theorem 3, this ensures that Algorithm 1 is halting.
2) All $c'\in C'$ can also lead to the nonexistence of the decoder. This will speedup Algorithm 1 significantly.

IV. Removing Redundancy

The p, d and l found by Algorithm 1 contain some redundancy, which can cause unnecessarily large overheads on the circuit area and on the runtime of characterizing Boolean function of the decoder. So, Algorithm 2 is used to minimize p, d and l before passing it to the next algorithm.

This algorithm reduces the value of p, d and l iteratively, and tests whether the reduced values can still make $E\vDash PC(R)$ hold. We will not go into the details here.

V. Discovering Multiple Decoders' Boolean Relations

Subsection V-A introduces how to discover decoders and its correctness proof, while Subsection V-B introduces the implementation of this algorithm.

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

for $p' = p \rightarrow 0$ **do**
  if $F_{PC}(p'-1, d, l, R)$ is satisfiable **then**
    break
  end if
end for
for $d' = d \rightarrow 0$ **do**
  if $F_{PC}(p', d'-1, l, R)$ is satisfiable **then**
    break
  end if
end for
for $l' = 1 \rightarrow l - (d - d')$ **do**
  if $F_{PC}(p', d', l', R)$ is unsatisfiable **then**
    break
  end if
end for
print "final result is $< p', d', l' >$"

A. Constructing SAT instance to discover decoders

Assume the assertion inferred by Algorithm 1 is:

$$IA \stackrel{def}{=} \bigwedge_{na \in NA} \neg na \tag{9}$$

Simultaneously, we also use IA to denote the set of configuration letters covered by it. So the actual meaning of IA depends on its context. We further assume the parameter value tuple reduced by Algorithm 2 is $< p, d, l >$, and the Boolean relation that uniquely determines i_n from o_{n+d-l}^{n+d-1} and the configuration letter c is $F_{PC}(p, d, l, IA)$. To simplify the presentation, we denote i_n and o_{n+d-l}^{n+d-1} respectively as :

$$X \stackrel{def}{=} o_{n+d-l}^{n+d-1} \tag{10}$$

$$Y \stackrel{def}{=} i_n \tag{11}$$

Thus, the Boolean relation that uniquely determines i_n from o_{n+d-l}^{n+d-1} and c can be denoted as:

$$R(c, X, Y) \stackrel{def}{=} F_{PC}(p, d, l, IA) \tag{12}$$

Assume f is the function defined by R, which computes Y from X and c:

$$Y = f(c, X) \tag{13}$$

Simultaneously, for every configuration letter $c_i \in IA$, there is a particular R_{c_i} defined below:

$$R_{c_i}(X, Y) \stackrel{def}{=} c \equiv c_i \wedge R(c, X, Y) \tag{14}$$

Many of these c_i share the same R_{c_i}, so IA can be partitioned into $\{IA_1, \ldots, IA_n\}$, such that:

1) All c in the same IA_i share the same R_i.
2) Two c and c' in two different IA_i and $IA_{i'}$ do not share the same R_i.

For the set $\{IA_1, \ldots, IA_n\}$, assume the Boolean relation shared by all $c \in IA_i$ is R_i, and the function defined by R_i is f_i, then f can be rewritten as:

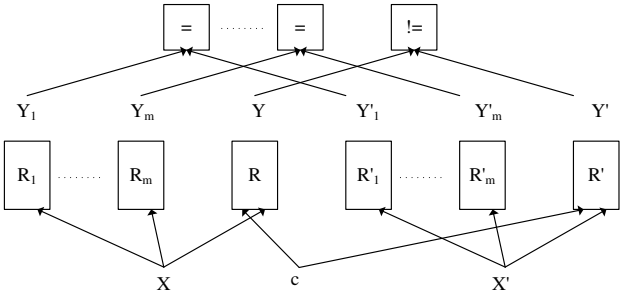

Fig. 5. The SAT instance that discovers decoders

$$f(c, X) = \bigvee_{i=1}^{n} \{IA_i(c) \wedge f_i(X)\} \tag{15}$$

Thus, our job here is to find out the set $\{R_1, \ldots, R_n\}$ step by step. Assume that we have already obtained a set of Boolean relations $\{R_1, \ldots, R_m\}$. To test whether it contains $\{R_1, \ldots, R_n\}$, that is, whether all decoders have already been discovered, we construct the following SAT instance, which is also shown in Fig. 5:

$$\left\{ \begin{array}{ll} & R(c, X, Y) \wedge \bigwedge_{i=1}^{m} R_i(X, Y_i) \\ \wedge & R'(c, X', Y') \wedge \bigwedge_{i=1}^{m} R'_i(X', Y'_i) \\ \wedge & \bigwedge_{i=1}^{m} Y_i \equiv Y'_i \\ \wedge & Y \neq Y' \end{array} \right\} \tag{16}$$

Line 1 of Equation (16) represents the Boolean relations in $\{R_1, \ldots, R_m\}$ and R. Line 2 is a copy of Line 1. The only common variable shared by them is c. Line 3 forces all Y_i and Y'_i to take on the same values, while the last line forces Y and Y' to be different.

The following theorem proves that, if Equation (16) is unsatisfiable, then all decoders have been discovered.

***Theorem** 4 ():* If Equation (16) is unsatisfiable, then $\{R_1, \ldots, R_m\}$ contains $\{R_1, \ldots, R_n\}$.

Proof: The proof is by contradiction. Assume $R_n \notin \{R_1, \ldots, R_m\}$, and IA_n is its corresponding set of configuration letters, and $c_n \in IA_n$.

We can construct an assignment A such that $A(c) \equiv c_n$. Thus we have $\{R(c, X, Y) \wedge A(c) \equiv c_n\} \equiv R_n(X, Y)$, that is, we can change the R and R' in Fig. 5 into R_n with A.

Because $R_n \notin \{R_1, \ldots, R_m\}$, there must exist an assignment A', such that when we assign $A'(X)$ to X and $A'(X')$ to X', we can make both $\bigwedge_{i=1}^{m} Y_i \equiv Y'_i$ and $Y \neq Y'$ hold.

So by combining A and A', Equation (16) becomes satisfied. This contradiction concludes the proof. ■

On the other hand, if Equation (16) is satisfiable, we need to prove that:

***Theorem** 5 ():* If Equation (16) is satisfiable, then there must be at least one decoder that has not been discovered.

Proof: The proof is by contradiction. Assume that all decoders have been discovered, that is, $\{R_1, \ldots, R_m\}$ contains $\{R_1, \ldots, R_n\}$.

This means that the function f can be rewritten as:

$$f(c, X) = \bigvee_{i=1}^{m} \{IA_i(c) \wedge f_i(X)\} \tag{17}$$

Thus, for any assignment A that makes the first three lines of Equation (16) satisfied, the function f can be further rewritten as:

$$\begin{aligned} Y = f(c, X) &= \bigvee_{i=1}^{m} \{IA_i(c) \wedge Y_i\} \\ &= \bigvee_{i=1}^{m} \{IA_i(c) \wedge Y_i'\} \\ &= Y' \end{aligned} \tag{18}$$

Thus, the last line of Equation (16) will never be satisfied. So the Equation (16) is unsatisfiable. This contradiction concludes the proof. ■

With the satisfying assignment A, R_{m+1} defined below is a newly discovered decoder's Boolean relation.

$$R_{m+1} \stackrel{def}{=} \{c \equiv A(c) \wedge R(c, X, Y)\} \tag{19}$$

To prove that our approach does not do redundant work, we need to prove that R_{m+1} has not been discovered before:

***Theorem** 6 ():* $R_{m+1} \notin \{R_1, \ldots, R_m\}$

Proof: The proof is by contradiction. Assume that there is a $0 \leq i \leq m$ such that $R_i \equiv R_{m+1}$, and there is a $c' \in IA_i$.

Since R_i can uniquely determine Y from X, and R can uniquely determine Y from X and c, and $R_{m+1} \equiv R_i$, it is obvious that we can make $Y \equiv Y_i$ by forcing c to be $c' \in IA_i$.

Similarly, we have $Y_i' \equiv Y'$.

According to Line 5 of Equation (16), we have $Y \equiv Y_i \equiv Y_i' \equiv Y'$. This is in contradiction with $Y \neq Y'$ in the last line of Equation (16). This contradiction concludes the proof. ■

B. The implementation of algorithm

Based on all these discussions, Algorithm 3 below describes the overall framework of how to find the Boolean relations of all decoders.

Algorithm 3 $DiscoveringDecoders$

1: **while** Equation (16) is satisfiable **do**
2: Assume A is the satisfying assignment
3: Insert R_{m+1} of Equation (19) into $\{R_1, \ldots, R_m\}$
4: **end while**
5: The set of decoders' Boolean relations is $\{R_1, \ldots, R_m\}$

Line 1 means $\{R_1, \ldots, R_m\}$ does not contain all decoders, there are some decoders not yet discovered.

With the satisfying assignment A returned from Equation (16), R_{m+1} in Line 3 represents the newly discovered decoder's Boolean relation. It will be merged with $\{R_1, \ldots, R_m\}$ to take part in the test in Line 1 again.

The loop in Algorithm 3 monotonically increases the size of $\{R_1, \ldots, R_m\}$. As the number of such decoders is finite, this loop, and therefore, Algorithm 3 will eventually halt.

TABLE I
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

C. Characterizing Boolean Functions of the Discovered Decoders

With the set $\{R_1, \ldots, R_m\}$ of the discovered decoders' Boolean relations in Algorithm 3, the ALLSAT algorithm proposed by Shen et al. [2] is used to characterizes their Boolean functions. Its details are not presented here.

For readers who are interested in the area and timing character of the generated decoders, please refer to Subsection V.B and V.C of Shen et al. [3].

VI. EXPERIMENTAL RESULTS

We have implemented this algorithm and solved the generated SAT instances with Minisat [11]. All experiments are run on a PC with a 2.4GHz Intel Core 2 Q6600 processor, 8GB memory and Ubuntu 10.04 linux. All experimental results can be downloaded from http://www.ssypub.org.

A. Benchmarks

Table I shows information of the following benchmarks.

1. A XGXS encoder compliant to clause 48 of IEEE-802.3ae 2002 standard [7].
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 sufficient 0-1 transitions, so that it can run through a high-speed noisy serial transmission channel.
4. A PCI-E physical coding module [6].
5. The Ethernet module of Sun's OpenSparc T2 processor.

B. Results

The 2nd row of Table II shows the runtime of the halting algorithm proposed by Shen et al. [12], which checks whether the decoders exist. And the 3rd row shows the value of d, p and l discovered by that algorithm. All these benchmarks have proper embedded assertions.

In contrast, we remove all the assertions and put the benchmarks into this paper's algorithm. The 4th row shows the configuration pin's bit number, the 5th row shows the runtime of inferring assertions and discovering decoders with our new algorithm, the 6th row shows the value of the discovered d, p and l, while the last row shows the number of decoders discovered.

By comparing the 2nd and the 5th rows, it is obvious that our approach is much slower than that of [12], which is caused by the much more complicated procedures $InferCoveringFormula$ and $DiscoveringDecoders$.

However, with reference to the 4th and 5th rows, although XFI and T2 ethernet have 120 and 26 configuration pins respectively, their runtimes are not very long. This is due to the efficient characterization algorithm proposed in Section III.

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

According to the last row, only two out of the five benchmarks have two decoders, while the other three have only one decoder. This means that, in most cases, our algorithm can generate only one decoder, while in other cases, the users need to consult the original designer to find out which one among the resulted decoders is the desired one.

In our case, we just put these two decoders into our simulation environment, and compare their output with the input of the encoder. Within no more than five minutes, we get the correct decoder.

C. Inferred assertions

We will show here the assertions inferred by Algorithm 1.

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_VALID & !DATA_PAT_SEL))

For scrambler:

True

For PCI-E:

!((CNTL_RESETN_P0 & !TXELECIDLE & CNTL_TXEnable_P0 & CNTL_Loopback_P0) | (CNTL_ RESETN_P0 & !TXELECIDLE & !CNTL_TXEnable_P0)) & !((CNTL_RESETN_P0 & TXELECIDLE) | (CNTL_RESETN_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])) & !((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_stud y_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]))

TABLE II
EXPERIMENTAL RESULTS

		XG-XS	XFI	scra-mbler	PCI-E	T2 e-ther
[12]	Runtime checking PC(sec)	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 number	3	120	1	16	26
	Runtime	1.58	372.15	4.61	2.95	104.67
	d, p, l	1,3,1	0,4,2	0,2,2	2,2,1	4,3,1
	Number of decoders	1	2	2	1	1

VII. RELATED WORKS

A. Program inversion

According to Gulwani [13], program inversion is the problem that derives a program P^{-1}, which negates the computation of a given program P. So it is very similar to complementary synthesis.

The initial work on deriving program inversion used proof-based approaches [14], but it could only handle very small programs and very simple syntax structures.

Glück et al. [15] inverted the first-order functional programs by eliminating nondeterminism with LR-based parsing methods. But the use of functional languages in that work is incompatible with our complementary synthesis.

Srivastava et al. [16] inductively ruled out invalid execution paths that could not fulfill the requirement of inversion, until only the valid ones remained. So it can only guarantee the existence of a solution, but not its correctness .

B. Functional dependency

Given a Boolean function $f : \mathbb{B}^l \to \mathbb{B}$ and a vector of Boolean functions $G = (g_1, ..., g_n)$ with $g_i : \mathbb{B}^l \to \mathbb{B}$ for $i = 1, \ldots, n$, functional dependency [17] is the problem that finds a third Boolean function $h : \mathbb{B}^n \to \mathbb{B}$, such that $f(X) = h(g_1(X), \ldots, g_n(X))$.

Similar to Fig. 5, Lee et al. [17] constructed a SAT instance to test whether h existed, and used Craig interpolation to characterize it. But such an approach does not try to discover new g_i if the functional dependency test fails, while our approach does. At the same time, our approach supports multiple bits output, while that of Lee et al. [17] does not.

C. Protocol converter synthesis

The protocol converter synthesis is the problem that automatically generates a translator between two different communication protocols. This is related to our work because both focus on synthesizing communication circuits.

Avnit et al. [18] first defined a general model for describing the different protocols. Then they provided an algorithm to decide whether there is some functionality of a protocol that cannot be translated into another. Finally, they synthesized a translator by computing a greatest fixed point for the update function of the buffer's control states. Avnit et al. [19] improved the algorithm mentioned above with a more efficient design space exploration algorithm.

VIII. Conclusions

This paper proposes a fully automatic approach to infer assertion for complementary synthesis and generate multiple decoders' Boolean relation. Experimental results show that our approach can infer assertions and generate decoders for many complex encoders, such as PCI-E [6] and Ethernet [7].

Acknowledgment

The authors would like to thank the editors and anonymous reviewers for their hard work.

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

References

[1] S. Shen, J. Zhang, Y. Qin, and S. Li, "Synthesizing complementary circuits automatically," in *ICCAD09*. IEEE, Nov. 2009, pp. 381–388.

[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*, vol. 29, no. 8, pp. 1191–1202, Aug. 2010.

[3] S. Shen, Y. Qin, L. Xiao, K. Wang, J. Zhang, and S. Li, "A halting algorithm to determine the existence of the decoder," *accepted by IEEE transaction on CAD of Integrated Circuits and Systems. http://www.ssypub.org*.

[4] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based unbounded symbolic model checking using circuit cofactoring," in *ICCAD04*. IEEE, Nov. 2004, pp. 510–517.

[5] W. Craig, "Linear reasoning: A new form of the Herbrand-Gentzen theorem," *J. Symbolic Logic*, vol. 22, no. 3, pp. 250–268, 1957.

[6] *en.wikipedia.org/wiki/PCI_Express*.

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

[8] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell Systems Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[9] D. Kroening and O. Strichman, "Efficient computation of recurrence diameters," in *VMCAI03*. Springer, January 2003, pp. 298–309.

[10] K. L. McMillan, "Interpolation and SAT-based model checking," in *CAV03*. Springer, July 2003, pp. 1–13.

[11] N. Een and N. Sorensson, "An extensible SAT-solver," in *SAT03*. Springer, May 2003, pp. 502–518.

[12] S. Shen, Y. Qin, J. Zhang, and S. Li, "A halting algorithm to determine the existence of decoder," in *FMCAD10*. IEEE, Oct. 2010, pp. 91–100.

[13] S. Gulwani, "Dimensions in program synthesis," in *PPDP10*. ACM, July 2010, pp. 13–24.

[14] E. W. Dijkstra, "Program inversion," in *Program Construction 1978*, 1978, pp. 54–57.

[15] R. Glück and M. Kawabe, "A method for automatic program inversion based on lr(0) parsing," *Fundam. Inf.*, vol. 66, no. 4, pp. 367–395, Nov. 2005.

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

[17] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," in *ICCAD07*. IEEE, Nov. 2007, pp. 227–233.

[18] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, and S. Parameswaran, "A formal approach to the protocol converter problem," in *DATE08*. IEEE, Mar. 2008, pp. 294–299.

[19] K. Avnit and A. Sowmya, "A formal approach to design space exploration of protocol converters," in *DATE09*. IEEE, Mar. 2009, pp. 129–134.