Cover letter

The following paper is accepted by ICCAD'11: “Inferring Assertion for Complementary Synthesis”.

This TCAD paper extends that ICCAD11 paper in the following ways:
1. Explaining our idea with an example in Section II.
2. Proposing an automatic algorithm to discover all decoders' Boolean relations in Subsection V-A and V-B.
3. Proposing an automatic algorithm to characterize precondition for each discovered decoder in Subsection V-D.
4. Presenting the experimental results on dealing with multiple decoders in Subsection VI-D.

Section III, Section IV, Subsection VI-A, VI-B VI-C and Appendix is the same as that of ICCAD’11 paper.

Inferring Assertion for Complementary Synthesis

ShengYu Shen, *Member, IEEE,* Ying Qin, KeFei Wang, ZhengBin Pang, JianMin Zhang, and SiKun Li

***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. This algorithm iteratively discovers the encoder's configuration value that leads to the nonexistence of the decoder, and then use Craig interpolation to infer a new formula that covers a larger set of such invalid configuration values. These inferred formulas will be ruled out until no more invalid configuration values can be found. The final assertion is obtained by anding the inverses of all these inferred formulas.

However, under this inferred assertion, multiple decoders may exist simultaneously. So we propose a second algorithm to discover them, by iteratively testing whether $\mathbb{R}$, the Boolean relation that reversely computes the encoder's input, functionally depends on those discovered decoders. A new decoder will be discovered for every failed test. This step is repeated until all decoders are discovered.

To help the user selects the correct decoder, a third algorithm is proposed to characterize a formula that represents each decoder's precondition, by solving the functional dependency problem between $\mathbb{R}$ and the discovered decoders.

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. Moreover, when there exists multiple decoders, the user can easily select the correct one by inspecting the precondition formulas.

***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 the complex complementary circuit pair (E, E^{-1}), in which the encoder E transforms information into a particular format, while its complementary circuit(or decoder) E^{-1} recovers this information.

Thus, complementary synthesis [1] was proposed to automatically synthesize an encoder's decoder. As shown in Fig. 1a), it includes three steps: **First**, manually specifying an assertion that assigns constant value on some configuration pins of the encoder, to prevent it from reaching the non-working states without a decoder. **Second**, determining if the decoder exists for this assertion. **Finally**, building the decoder's Boolean function.

Manually specifying an assertion needs the user to read lots of documentations and try many combinations. Even with all these efforts, the user may still specify an improper assertion that will lead to incorrect result. To avoid this tedious job, we propose in this paper the following three algorithms:

The authors are with the School of Computer, National University of Defense Technology, ChangSha, HuNan, 410073 CHINA. e-mail: {syshen, yingqin, kfwang, zbpang, jmzhang, skli}@nudt.edu.cn, (see http://www.ssypub.org/).

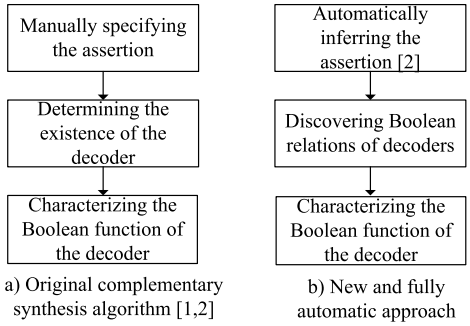

Fig. 1. The original and new flows of complementary synthesis

First, we propose an algorithm that automatically infers this assertion [2], which corresponds to the first step of the new approach in Fig. 1b). This algorithm iteratively discovers the encoder's configuration value that leads to the nonexistence of the decoder, and then use Craig interpolation to infer a new formula that covers a larger set of such invalid configuration values. These inferred formulas will be ruled out until no more invalid configuration values can be found. The final assertion is obtained by anding the inverses of all these inferred formulas.

Second, under this inferred assertion, however, there may exist multiple decoders. Thus, as shown in the second step of Fig. 1b), we propose in this paper a new algorithm to discover all these decoders by iteratively testing whether $\mathbb{R}$, the Boolean relation that reversely computes the encoder's input, functionally depends on those discovered decoders. For every configuration value of the encoder that fails this test, a new decoder's Boolean relation is discovered by asserting this configuration value into $\mathbb{R}$. This step is repeated until all decoders are discovered.

Third, to help the user select the correct decoder, we propose a third algorithm to characterize a precondition formula for each decoder, which represents the encoder's configuration value set that corresponds to this decoder. The user can easily select the correct decoder by inspecting these preconditions.

We have run this algorithm on several complex encoders from industrial projects (e.g., PCI-E [3] and Ethernet [4]). For most benchmarks, there exists only one decoder. For other benchmarks with multiple decoders, the user can easily select the correct one, by inspecting the precondition formulas and finding out the meaning of no more than one pin, instead of up to 120 pins like the original approach [1].

The remainder of this paper is organized as follows. Section II explains our ideas with a simple example. Section III introduces background materials. Section IV presents the algorithm that infers assertions. Section V discusses how to discover all decoders and characterize their precondition formulas on configuration pins. Section VI and VII present

experimental results and related works. Finally, Section VIII concludes this paper.

II. Case Study

In this section, we will explain our ideas with a simple encoder in Fig. 2. This encoder has two constant configuration pins : c_1 and c_2. The functionality of this encoder is:

1) When $c_2 \equiv 0$, 20 will be assigned to o.
2) Otherwise, if $c_1 \equiv 0$, $i+1$ will be assigned to o.
3) Otherwise, $i+2$ will be assigned to o.

The **first** step of our algorithm finds out that, when $c_2 \equiv 0 \wedge c_1 \equiv 0$, it can not determine the value of i from o, because the latter one is always 20. So it tries to enlarge this formula with Craig interpolation, and finds out that the resulted formula $c_2 \equiv 0$ covers more configuration values that can lead to the nonexistence of the decoder. By inverting this formula, we obtain the final assertion $c_2 \equiv 1$ that can ensure the existence of the decoder.

However, under this inferred assertion, there actually exists two decoders. One is $i \stackrel{def}{=} o-1$, the other is $i \stackrel{def}{=} o-2$. To discover these two decoders, the **second** step of our algorithm constructs a SAT instance to test whether $\mathbb{R}$, the Boolean relation between i and o, can be expressed as a combination of $\mathbb{D}$, the set of already discovered decoders. The encoding of this SAT instance follows the functional dependency approach proposed by Lee et al. [5].

$\mathbb{D}$ is empty at this time, so $\mathbb{R}$ can not be expressed as a combination of $\mathbb{D}$, and this SAT instance will be satisfiable. Lets assume that its satisfying assignment is $c_2 \equiv 1 \wedge c_1 \equiv 0$. By asserting this assignment into $\mathbb{R}$, we get the Boolean relation of the decoder $i \stackrel{def}{=} o-1$, and insert it into $\mathbb{D}$.

Now the **second** step of our algorithm constructs the second SAT instance to test whether $\mathbb{R}$ can be expressed as a combination of the already discovered decoder, that is, $i \stackrel{def}{=} o-1$. This new SAT instance will be satisfiable again, and the new satisfying assignment is $c_2 \equiv 1 \wedge c_1 \equiv 1$. By asserting this assignment into $\mathbb{R}$, we get the Boolean relation of the other decoder $i \stackrel{def}{=} o-2$,

Again, the **second** step of our algorithm constructs a third SAT instance to test whether $\mathbb{R}$ can be expressed as a combination of the two discovered decoders. At this time, this SAT instance will be unsatisfiable, which means no more decoder can be found.

With the functional dependency approach proposed by Lee et al. [5], we can characterize each decoder's precondition formula, which represents the set of encoder's configuration values that correspond to this decoder. The precondition formula of the decoder $i \stackrel{def}{=} o-1$ is $c_1 \equiv 0$, and the precondition formula of the decoder $i \stackrel{def}{=} o-2$ is $c_1 \equiv 1$.

At this point, by inspecting these two precondition formulas, we know that their essential difference is the assignment to c_1. So we only need to find out the meaning of c_1 by reading the documentation or source code, instead of both c_1 and c_2 like the original approach [1].

Some larger benchmark, such as XFI in Subsection VI-A, has more than 100 configuration pins. Our approach can significantly save the user's efforts by reducing the number of pins whose meaning need to be manually find out.

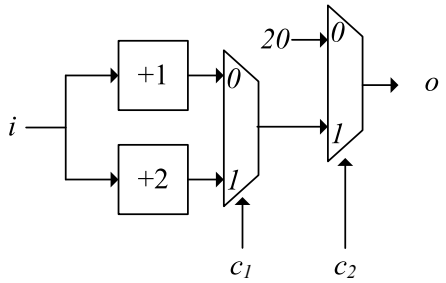

Fig. 2. The encoder used to explain our ideas

III. 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 $A : V \rightarrow \mathbb{B}$, 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, such as Zchaff [6], Grasp [7], Berkmin [8], and MiniSAT [9]. 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 $f : \mathbb{B}^n \rightarrow \mathbb{B}$, we use $supp(f)$ to denote its support set $\{v_1 \dots v_n\}$. According to Ganai et al. [10], 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 1 or 0 to v to get f_v or $f_{\overline{v}}$.

Craig [11] 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 $\phi_A \rightarrow \phi_I$ and $\phi_I \wedge \phi_B$ is unsatisfiable. ϕ_I is referred to as the **interpolant** of ϕ_A with respect to ϕ_B.

In the remainder of this paper, we will use the algorithm proposed by McMillan [12] to generate interpolant from the proof of unsatisfiability, which is generated by MiniSAT [9].

Given a Boolean function $f : \mathbb{B}^l \rightarrow \mathbb{B}$ and a vector of Boolean functions $G = (g_1, ..., g_n)$ with $g_i : \mathbb{B}^l \rightarrow \mathbb{B}$ for $i = 1, \dots, n$, **functional dependency** [5] is the problem that finds a third Boolean function $h : \mathbb{B}^n \rightarrow \mathbb{B}$, such that $f(X) = h(g_1(X), \dots, g_n(X))$. Lee et al. [5] constructed a SAT instance to test whether h existed, and used Craig interpolation [12] to characterize it.

B. Determining the existence of the decoder for complementary synthesis

To model the encoder with configuration pins, we extend the traditional definition of Mealy finite state machine [13]:

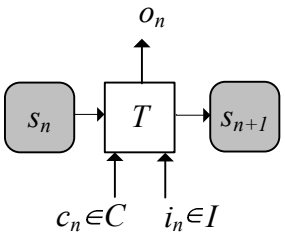

Fig. 3. Mealy finite state machine with configuration

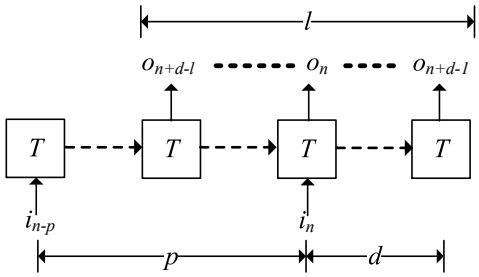

Fig. 4. The parameterized complementary condition

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 $T : S \times I \times C \rightarrow S \times O$ that computes the next state and the output letter from the current state, the input letter and the configuration letter.

As shown in Fig. 3, 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 $n \leq j < m$. 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:

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

The only difference between this definition and Kroening's state variables recurrence diameter [14] 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. 4, 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 amounts to 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.

The algorithm based on checking $E \vDash PC(R)$ [1], [15] 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.

IV. A Halting Algorithm to Infer Assertion

To rule out all those configuration letters that can lead to the nonexistence of the decoder, we first need to define how to determine the nonexistence of the decoder for a particular configuration letter. This will be discussed in Subsection IV-A. Based on this result, the overall framework of our algorithm will be presented in Subsection IV-B,

A. Determining the nonexistence of the decoder

According to Definition 2, the decoder exists for a set of configuration letters R if there is a parameter value tuple $< p, d, l >$ that makes $E \vDash PC(p, d, l, R)$ holds. So, intuitively, the decoder 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 5, which is similar to Figure 4, 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:

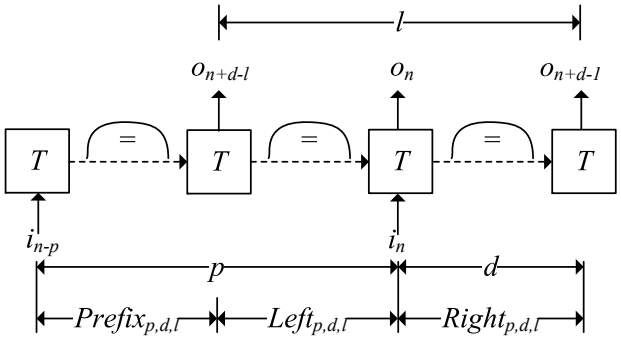

Fig. 5. The loop-like non-complementary condition

***Definition** 3:* **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 $(< s_{m+1}, s'_{m+1} >, < o_m, o'_m >) = T^2(< s_m, s'_m >, < i_m, i'_m >, < \ c_m, c'_m \ >)$ 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 nonexistence of the decoder:

***Definition** 4:* **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 amounts to 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 5:

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

Thus, we have the following theorem:

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

The proof of its correctness will be given in the appendix section at the end of this paper.

B. Algorithm implementation

Algorithm 1 is used to infer assertion on configuration pins that can lead to the existence of the decoder.

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:             Assume A is the satisfying assignment, and A(c) is
                the configuration letter leading to the nonexistence
                of decoder
14:             na ← InferCoveringFormula(A(c))
15:             NA ← NA ∪ {na}
16:         end while
17:     end if
18: end for
```

Intuitively, this algorithm just iteratively test all combinations of $< p, d, l >$ in Line 3. For every configuration letter $A(c)$ found in Line 13 that can lead to the nonexistence of the decoder, a larger set of such configuration letters are discovered by the procedure $InferCoveringFormula$ in Line 14, and ruled out in Line 15. This loop is repeated until the formula F_{PC} in Line 4 is unsatisfiable, which means the final inferred assertion is $\bigwedge_{na \in NA} \neg na$. At this point, if this assertion is satisfiable, then the decoder exists. Otherwise, the decoder does not exist.

The implementation of the procedure $InferCoveringFormula$ will be presented in Subsection IV-C.

We can prove that Algorithm 1 is a halting one.

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

Proof: According to Theorems 2, Algorithm 1 will eventually reach Line 4 or 12.

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 $A(c)$. Because 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. ■

C. 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 $A(c)$, but also many other configuration letters leading to the nonexistence of the decoder. This job will be accomplished in the following three steps:

*1) **Transforming F_{LN} into an equivalent form with an object variable, such that it can be used to defined a Boolean function f.**:* First, we need to move the 4th line and the last three lines of Equation (3) into a new subformula:

$$G(p,d,l) \stackrel{def}{=} \left\{ \begin{array}{cc} & \bigwedge_{m=n+d-l}^{n+d-1} o_m \equiv o'_m \\ \wedge & \bigvee_{x=n-p}^{n+d-l-1} \bigvee_{y=x+1}^{n+d-l} \{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+d-l+1}^{n-1} \bigvee_{y=x+1}^{n} \{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+1}^{n+d-1} \bigvee_{y=x+1}^{n+d} \{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \end{array} \right\} \quad (5)$$

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

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

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

At the same time, according to Figure 5, 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 (6).

*2) **Eliminating some variables from the support set of f' with cofactoring [10], until only $A(c)$ remains.**:* According to Line 12 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 :

$$F"_{LN}(c,t) \stackrel{def}{=} \left\{ \begin{array}{cc} & F'_{LN} \\ \wedge & i_{n-p}^{n+d-1} \equiv A(i_{n-p}^{n+d-1}) \\ \wedge & (i')_{n-p}^{n+d-1} \equiv A((i')_{n-p}^{n+d-1}) \\ \wedge & s_{n-p} \equiv A(s_{n-p}) \\ \wedge & (s')_{n-p} \equiv A((s')_{n-p}) \end{array} \right\} \quad (7)$$

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.

*3) **Characterizing f with Craig Interpolation. This f is also the formula na in Line 14 of Algorithm 1.**:* 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:

$$\phi_A \stackrel{def}{=} CNF(F"_{LN}(c,t)) \wedge t \equiv 1 \quad (8)$$

$$\phi_B \stackrel{def}{=} CNF'(F"_{LN}(c,t')) \wedge t' \equiv 0 \quad (9)$$

It is obvious that $\phi_A \wedge \phi_B$ is unsatisfiable. With McMillan's algorithm [12], 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 (9). So it characterizes a set $C' \subseteq C$ that can make ϕ_A in Equation (8) satisfiable. According to Equations (6) and (7), it is obvious that:

1) The $A(c)$ in Line 13 of Algorithm 1 is in C'. According to Theorem 3, this will ensure 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.

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. And Subsection V-D presents how to characterize the precondition formula of each discovered decoder, such that the user can select the correct decoder.

A. Constructing SAT instance to discover decoders

Assume the assertion inferred by Algorithm 1 is:

$$IA \stackrel{def}{=} \bigwedge_{na \in NA} \neg na \quad (10)$$

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 discovered by Algorithm 1 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 :

$$X \stackrel{def}{=} o_{n+d-l}^{n+d-1} \quad (11)$$

$$Y \stackrel{def}{=} i_n \quad (12)$$

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

$$\mathbb{R}(c,X,Y) \stackrel{def}{=} F_{PC}(p,d,l,IA) \quad (13)$$

Assume the function defined by $\mathbb{R}$ is f, which computes Y from X and c:

$$Y = f(c,X) \quad (14)$$

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

$$\mathbb{R}_{c_i}(X,Y) \stackrel{def}{=} \mathbb{R}(c,X,Y) \wedge c \equiv c_i \quad (15)$$

Two different c_i and c_j may share the same $\mathbb{R}_{c_i}$, that is, $\mathbb{R}_{c_i} \equiv \mathbb{R}_{c_j}$. So IA can be partitioned into $\{IA_1, \ldots, IA_n\}$, such that:

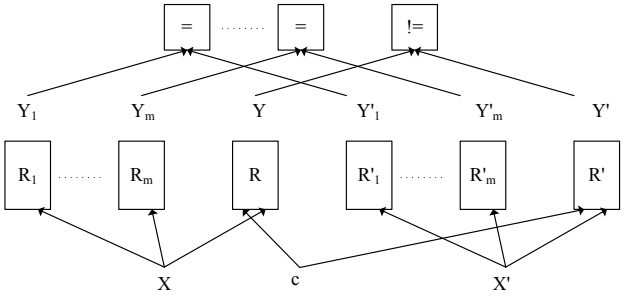

Fig. 6. The SAT instance that discovers decoders

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

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

$$f(c, X) = \bigvee_{i=1}^{n} \{IA_i(c) \wedge f_i(X)\} \tag{16}$$

Thus, our job here is to find out the set $\{\mathbb{R}_1, \ldots, \mathbb{R}_n\}$ step by step. Assume that we have already discovered a set of decoders' Boolean relations $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$. To test whether it contains $\{\mathbb{R}_1, \ldots, \mathbb{R}_n\}$, that is, whether all decoders have already been discovered, we construct the following SAT instance, which is also shown in Fig. 6:

$$\left\{ \begin{array}{ll} & \mathbb{R}(c, X, Y) \wedge \bigwedge_{i=1}^{m} \mathbb{R}_i(X, Y_i) \\ \wedge & \mathbb{R}'(c, X', Y') \wedge \bigwedge_{i=1}^{m} \mathbb{R}'_i(X', Y'_i) \\ \wedge & \bigwedge_{i=1}^{m} Y_i \equiv Y'_i \\ \wedge & Y \neq Y' \end{array} \right\} \tag{17}$$

Line 1 of Equation (17) represents the Boolean relations in $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ and $\mathbb{R}$. 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 (17) is unsatisfiable, then all decoders have been discovered.

***Theorem** 4:* If Equation (17) is unsatisfiable, then $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ contains $\{\mathbb{R}_1, \ldots, \mathbb{R}_n\}$.

Proof: The proof is by contradiction. Assume $\mathbb{R}_n \notin \{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$, 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 $\{\mathbb{R}(c, X, Y) \wedge A(c) \equiv c_n\} \equiv \mathbb{R}_n(X, Y)$, that is, we can change the $\mathbb{R}$ and $\mathbb{R}'$ in Fig. 6 into $\mathbb{R}_n$ with A.

Because $\mathbb{R}_n \notin \{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$, 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 (17) becomes satisfied. This contradiction concludes the proof. ■

On the other hand, if Equation (17) is satisfiable, we need to prove that:

***Theorem** 5:* If Equation (17) 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, $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ contains $\{\mathbb{R}_1, \ldots, \mathbb{R}_n\}$.

This means that the function f can be rewritten as:

$$f(c, X) = \bigvee_{i=1}^{m} \{IA_i(c) \wedge f_i(X)\} \tag{18}$$

Thus, for any assignment A that makes the first three lines of Equation (17) satisfied, the function f can be further rewritten as:

$$\begin{aligned} Y = f(c, X) &= \bigvee_{i=1}^{m} \{IA_i(c) \wedge Y_i\} \\ &= \bigvee_{i=1}^{m} \{IA_i(c) \wedge Y'_i\} \\ &= Y' \end{aligned} \tag{19}$$

Thus, the last line of Equation (17) will never be satisfied. So the Equation (17) is unsatisfiable. This contradiction concludes the proof. ■

With the satisfying assignment A, $\mathbb{R}_{m+1}$ defined below is a newly discovered decoder's Boolean relation.

$$\mathbb{R}_{m+1} \stackrel{def}{=} \{c \equiv A(c) \wedge \mathbb{R}(c, X, Y)\} \tag{20}$$

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

***Theorem** 6:* $\mathbb{R}_{m+1} \notin \{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$

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

Since $\mathbb{R}_i$ can uniquely determine Y from X, and $\mathbb{R}$ can uniquely determine Y from X and c, and $\mathbb{R}_{m+1} \equiv \mathbb{R}_i$, 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 (17), 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 (17). This contradiction concludes the proof. ■

B. The implementation of algorithm

Based on all these discussions, Algorithm 2 below describes the overall framework of how to discover the Boolean relations of all decoders.

Algorithm 2 $DiscoveringDecoders$

```
1: while Equation (17) is satisfiable do
2:     Assume A is the satisfying assignment
3:     Insert R_{m+1} of Equation (20) into {R_1, ..., R_m}
4: end while
5: The set of decoders' Boolean relations is {R_1, ..., R_m}
```

Line 1 means $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ does not contain all decoders, there are some decoders not yet discovered.

With the satisfying assignment A returned from Equation (17), $\mathbb{R}_{m+1}$ in Line 3 is the newly discovered decoder's Boolean relation in Equation (20). It will be inserted into $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ to take part in the test in Line 1 again.

The loop in Algorithm 2 monotonically increases the size of $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$. As the number of decoders that compute Y from X is finite, this loop, and therefore, Algorithm 2 will eventually halt.

C. Characterizing Boolean functions of the discovered decoders

With the set $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ of the discovered decoders' Boolean relations in Algorithm 2, the ALLSAT algorithm proposed by Shen et al. [15] 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. [16].

D. Characterizing $\{IA_1, \ldots, IA_m\}$

Assume $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ is the set of all decoders' Boolean relations discovered by Algorithm 2, and $\{IA_1, \ldots, IA_m\}$ is their corresponding set of configuration letters. To help the user determine which $\mathbb{R}_i$ in $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ is the correct decoder, we need to characterize each IA_i in $\{IA_1, \ldots, IA_m\}$.

According to Fig. 6, the relation between Y and all Y_i can be written as :

$$Y = \bigvee_{i=1}^{m} \{IA_i(c) \wedge Y_i\} \tag{21}$$

Assume Y and all Y_i are vectors of the same length v:

$$\begin{aligned} Y &=< y^0, \ldots, y^{v-1} > \\ Y' &=< y'^0, \ldots, y'^{v-1} > \\ Y_i &=< y_i^0, \ldots, y_i^{v-1} > \\ Y_i' &=< y_i'^0, \ldots, y_i'^{v-1} > \end{aligned} \tag{22}$$

So we can rewrite Equation (21) by splitting it into bits, and the relation between the j-th bits of Y and all Y_i can be written as :

$$y^j = \bigvee_{i=1}^{m} \{IA_i^j(c) \wedge y_i^j\} \tag{23}$$

According to Subsection III-A, it is obvious that Equation (23) represents a functional dependency problem. We can characterize $IA_i^j(c)$ with the functional dependency algorithm proposed by Lee et al. [5] with the following two formulas:

$$\phi_A \stackrel{def}{=} \left\{ \begin{array}{cc} & \mathbb{R}(c, X, Y) \wedge \bigwedge_{i=1}^{m} \mathbb{R}_i(X, Y_i) \\ \wedge & y^j \equiv 1 \end{array} \right\} \tag{24}$$

$$\phi_B \stackrel{def}{=} \left\{ \begin{array}{cc} & \mathbb{R}'(c, X', Y') \wedge \bigwedge_{i=1}^{m} \mathbb{R}_i'(X', Y_i') \\ \wedge & \bigwedge_{i=1}^{m} y_i^j \equiv y_i'^j \\ \wedge & y'^j \equiv 0 \end{array} \right\} \tag{25}$$

It is obvious that $\phi_A \wedge \phi_B$ is very similar to Equation (17), except that only the j-th bits are constrained to be the same, and the y^j and y'^j is constrained to be different constants.

TABLE I
INFORMATION ON 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

So $\phi_A \wedge \phi_B$ is unsatisfiable, and we can generate an interpolant $ITP : C \times \mathbb{B}^m \rightarrow \mathbb{B}$ from its proof, whose support set is $\{c, y_1^j, \ldots, y_m^j\}$. According to Equation (23), ITP is the over-approximation of $\bigvee_{i=1}^{m} \{IA_i^j(c) \wedge y_i^j\}$. Thus, an over-approximation of $IA_i^j(c)$ can be obtained by setting y_i^j to 1, and all other y_k^j to 0:

$$\left\{ \begin{array}{cc} & ITP \\ \wedge & \bigwedge_{k \neq i} y_k^j \equiv 0 \\ \wedge & y_i^j \equiv 1 \end{array} \right\} \tag{26}$$

Because $\phi_A \wedge \phi_B$ is unsatisfiable, this over-approximation of $IA_i^j(c)$ can make $y^j \equiv 1$. So we can just take it as $IA_i^j(c)$. Thus, $IA_i(c)$ can be defined as:

$$IA_i(c) = \bigwedge_{j=0}^{v-1} IA_i^j(c) \tag{27}$$

In Subsection VI-D, we will show that, by inspecting these IA_i, the user can easily select the correct decoder.

VI. EXPERIMENTAL RESULTS

We have implemented this algorithm and solved the generated SAT instances with Minisat [9]. All experiments are run on a PC with a 2.4GHz Intel Core 2 Q6600 processor, 8GB memory and Ubuntu 10.04 linux.

A. Benchmarks

Table I shows information on the following benchmarks.

1. A XGXS encoder compliant to clause 48 of IEEE-802.3ae 2002 standard [4].
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 [3].
5. The Ethernet module of Sun's OpenSparc T2 processor.

B. Inferred assertions

We will show here the assertions inferred by Algorithm 1.

For XGXS: `!( ( bad_code ) )`

For XFI: `!( ( RESET & TEST_MODE ) | ( !RESET & TEST_MODE ) | ( !RESET & !TEST_MODE & !DATA_VALID ) )`

For scrambler: `True`

TABLE II
EXPERIMENTAL RESULTS

		XG-XS	XFI	scrambler	PCI-E	T2 ether
[16]	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
this paper	Config pin number	3	120	1	16	26
	Runtime	2.33	364.36	14.20	4.49	125.69
	d, p, l	1,3,1	0,3,2	0,2,2	2,3,1	4,4,1
	Number of decoders	1	2	2	1	1

For PCI-E: `!( ( CNTL_RESETN_P0 & TXELECIDLE ) | ( CNTL_RESETN_P0 & !TXELECIDLE & CNTL_TXEnable_P0 & CNTL_Loopback_P0 ) | ( CNTL_RESETN_P0 & !TXELECIDLE & !CNTL_TXEnable_P0 ) | ( !CNTL_RESETN_P0 ) )`

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

C. Comparing the results with previous work

Table II compares the halting algorithm proposed in [16] and this paper's approach. The algorithm in [16] needs an manually specified assertion to be embedded in the benchmarks, while our approach does not.

The 2nd row of Table II shows the runtime of the halting algorithm proposed in [16], which checks whether the decoders exist. And the 3rd row shows the value of d, p and l discovered by that algorithm. These benchmarks all have proper embedded assertions.

In contrast, we remove all these assertions and run this paper's algorithm on them. The 4th row shows the bit numbers of these benchmarks' configuration pins, 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 discovered decoders.

By comparing the 2nd and the 5th rows, it is obvious that our approach is much slower than that of [16], 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 Subsection IV-B.

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 difference between the embedded assertions and the inferred assertion. The assertions inferred by this paper's algorithm contains much more configuration letters than the assertions embedded in benchmarks.

D. Dealing with multiple decoders

According to the last row of Table II, 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 generates only one decoder. For other cases with multiple decoders, the user need to inspect $\{IA_1, \ldots, IA_m\}$ characterized by Subsection V-D to select the correct decoder.

For the two decoders of scrambler, their corresponding IA_1 is $False$, while IA_2 is $!reset$. So the second decoder is the only possible decoder. And the dynamic simulation had confirmed its correctness.

For the two decoders of XFI, their corresponding IA_1 is $RESET$ & $!TEST\_MODE$, while IA_2 is $!RESET$ & $!TEST\_MODE$ & $DATA\_VALID$. The essential difference between them is the value of $RESET$. By inspecting the Verilog source code of XFI, we find that the $RESET$ is used to reset the XFI encoder when it is $True$. So the XFI encoder will work in normal mode when $RESET$ is $False$. So the second decoder is the correct decoder. The dynamic simulation had also confirmed its correctness.

VII. RELATED WORKS

A. Complementary synthesis

The concept of complementary synthesis was first proposed by Shen et al. [1]. Its major shortcomings are that it may not halt, and its runtime overheads while building complementary circuit is large.

The runtime overhead problem was addressed by simplifying the SAT instance with unsatisfiable core extraction [15], while the halting problem was handled by building a set of over-approximations that are similar to onion rings [17]. But this algorithm is too complicated. So another simpler and faster halting algorithm is proposed in [16].

B. Reversible logic synthesis

Reversible logic synthesis [22]–[29] is the problem that uses small reversible gates to build large reversible function, which is a bijection between its input and output. It is somewhat similar to complementary synthesis, but with some major differences.

First, reversible logic synthesis tries to build the encoder whose decoder can easily obtained, while complementary synthesis builds the decoder of an encoder.

Second, the circuit synthesized by reversible logic synthesis is combinational logic, while our approach can deal with sequential logic.

C. Program inversion

According to Gulwani [18], 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.

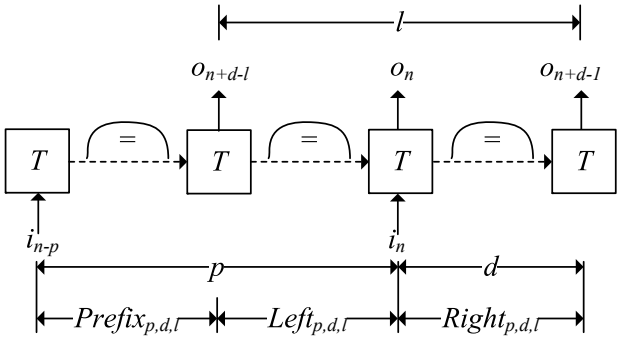

Fig. 7. The loop-like non-complementary condition

The initial work on deriving program inversion used proof-based approaches [19], but it could only handle very small programs and very simple syntax structures.

Glück et al. [20] 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. [21] 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 .

D. 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. [30] 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. [31] 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 [3] and Ethernet [4]. Moreover, with the characterized formulas on the configuration pins, the user can easily select the correct decoder when there exists multiple decoders.

Appendix
Proof of Theorem 2

In this section, we will prove $E \models LN(R) \leftrightarrow \neg\{E \models PC(R)\}$, that is, LN and PC are exclusive to each other. To facilitate our presentation, we place Figure 5 here again in Figure 7.

We first need to define how to unfold the three loops in Figure 7. 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 8. It is obvious that, the unfolded SAT instance corresponds to $F_{LN}(p", d", l", R)$, where:

$$\begin{array}{rcl} p" & = & l_1 * q + (d - l + p - l_1) + l_2 * q + (l - d - l_2) \\ d" & = & l_3 * q + (d - l_3) \\ l" & = & l_2 * q + (l - d - l_2) + l_3 * q + (d - l_3) \end{array} \quad (28)$$

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.

We then need to prove some lemmas about this unfolded SAT instance.

***Lemma** 1 ():* For $F_{LN}(p", d", l", R)$ in Figure 8, 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:

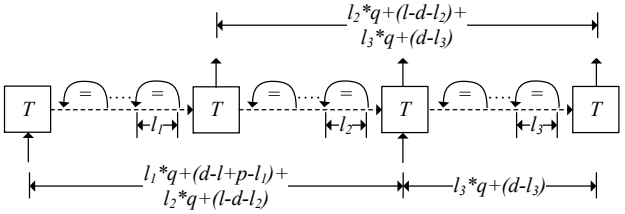

Fig. 8. The loop-like non-complementary condition unfolded for q times

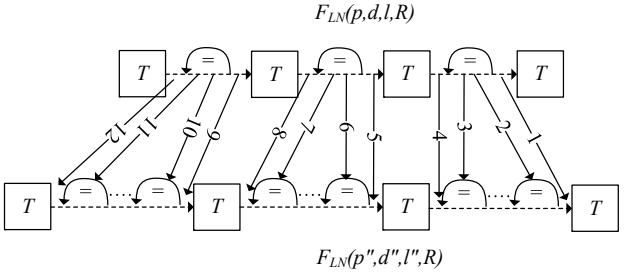

Fig. 9. Correspondance between $F_{LN}(p, d, l, R)$ and $F_{LN}(p", d", l", R)$

$$F_{LN}(p", d", l", R) \stackrel{def}{=}$$
$$\left\{\begin{array}{cc} & \bigwedge_{m=n-p"}^{n+d"-1}\{(s_{m+1}, o_m) \equiv T(s_m, i_m, c_m)\} \\ \wedge & \bigwedge_{m=n-p"}^{n+d"-1}\{(s'_{m+1}, o'_m) \equiv T(s'_m, i'_m, c'_m)\} \\ \wedge & \bigwedge_{m=n+d"-l"}^{n+d"-1} o_m \equiv o'_m \\ \wedge & i_n \neq i'_n \\ \wedge & \bigwedge_{x=n-p"}^{n+d"-1} c_x \equiv c \\ \wedge & \bigwedge_{x=n-p"}^{n+d"-1} c'_x \equiv c \\ \wedge & R(c) \\ \wedge & \bigvee_{x=n-p"}^{n+d"-l"-1} \bigvee_{y=x+1}^{n+d"-l"}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+d"-l"+1}^{n-1} \bigvee_{y=x+1}^{n}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \\ \wedge & \bigvee_{x=n+1}^{n+d"-1} \bigvee_{y=x+1}^{n+d"}\{s_x \equiv s_y \wedge s'_x \equiv s'_y\} \end{array}\right. \quad (29)$$

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 9, show the correspondence between $F_{LN}(p, d, l, R)$ and the $F_{LN}(p", d", l", R)$.

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 (29) is satisfied with the assignment A. Similarly, the 3rd to 8th lines of Equation (29) 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 (29) 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** 7:* $E \vDash LN(R) \rightarrow \neg\{E \vDash PC(R)\}$

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** 8:* $E \vDash LN(R) \leftarrow \neg\{E \vDash PC(R)\}$

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:

$$\begin{array}{c} p = uirrd(M^2) * 2 + 2 \\ d = uirrd(M^2) + 1 \\ l = uirrd(M^2) * 2 + 2 \end{array} \quad (30)$$

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. ■

With Theorem 7 and 8, Theorem 2 is proved.

ACKNOWLEDGMENT

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, J. Zhang, Y. Qin, and S. Li, "Inferring Assertion for Complementary Synthesis,"*accepted by ICCAD11. http://www.ssypub.org/pub/iccad11_ssy.pdf*.
[3] "PCI Express Base Specification Revision 1.0". [Online]. Available: http://www.pcisig.com
[4] "IEEE Standard for Information technology Telecommunications and information exchange between systems Local and metropolitan area networks Specific requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications Amendment: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation", IEEE Std. 802.3, 2002.
[5] 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.
[6] 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.
[7] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD96*, pages 220–227. IEEE, November 1996.
[8] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *DATE02*, pages 142–149. IEEE, March 2002.
[9] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT03*, pages 502–518. Springer, May 2003.
[10] 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.

[11] W. Craig, “Linear reasoning: A new form of the Herbrand-Gentzen theorem,” *J. Symbolic Logic*, vol. 22, no. 3, pp. 250–268, 1957.
[12] K. L. McMillan, “Interpolation and SAT-based model checking,” in *CAV03*. Springer, July 2003, pp. 1–13.
[13] G. H. Mealy, “A method for synthesizing sequential circuits,” *Bell Systems Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.
[14] D. Kroening and O. Strichman, “Efficient computation of recurrence diameters,” in *VMCAI03*. Springer, January 2003, pp. 298–309.
[15] 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.
[16] 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/tcad11.pdf*.
[17] 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.
[18] S. Gulwani, “Dimensions in program synthesis,” in *PPDP10*. ACM, July 2010, pp. 13–24.
[19] E. W. Dijkstra, “Program inversion,” in *Program Construction 1978*, 1978, pp. 54–57.
[20] 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.
[21] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. Foster, “Program inversion revisited,” *MSR-TR-2010-34, Microsoft Research*, 2010.
[22] Y. Zheng, and C. Huang, “A novel Toffoli network synthesis algorithm for reversible logic,” in *ASPDAC09*. IEEE, Jan. 2009, pp. 739–744.
[23] R. Wille, M. Soeken, and R. Drechsler, “Reducing the number of lines in reversible circuits,” in *DAC10*. IEEE, Jun. 2010, pp. 647–652.
[24] V. Shende, A. Prasad, I. Markov, and J. Hayes, “Synthesis of reversible logic circuits,” *IEEE transaction on CAD of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 710–722, Jun. 2003.
[25] W. Hung, X. Song, G. Yang, J. Yang, and M. Perkowski, “Quantum logic synthesis by symbolic reachability analysis, “ in *DAC04*. IEEE, Jun. 2004, pp. 838–841.
[26] P. Kerntopf, “A new heuristic algorithm for reversible logic synthesis, “ in *DAC04*. IEEE, Jun. 2004, pp. 834–837.
[27] D. Maslov, G. Dueck, and D. Miller, “Toffoli network synthesis with templates, “ *IEEE transaction on CAD of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 807–817, Jun. 2005.
[28] P. Gupta, A. Agrawal, and N. Jha, “An algorithm for synthesis of reversible logic circuits, “ *IEEE transaction on CAD of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2317–2330, Nov. 2006.
[29] D. Maslov, and M. Saeedi, “Reversible Circuit Optimization Via Leaving the Boolean Domain, “ *IEEE transaction on CAD of Integrated Circuits and Systems*, vol. 30, no. 6, pp. 806–816, Jun. 2011.
[30] 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.
[31] K. Avnit and A. Sowmya, “A formal approach to design space exploration of protocol converters,” in *DATE09*. IEEE, Mar. 2009, pp. 129–134.