CompSyn : A Tool for Automatically Synthesizing Decoders

ShengYu Shen, Ying Qin, JianMin Zhang, and SiKun Li

School of Computer, National University of Defense Technology, China
{syshen,yingqin,jmzhang,skli}@nudt.edu.cn
http://www.ssypub.org/

Abstract. CompSyn is a tool that automatically synthesizes the decoder circuit of a particular encoder, under some predefined assertion. This tool has two usage modes: the synthesis mode and inferring mode.

The synthesis mode is used, when the correct assertion is known, to determine the existence of the decoder and generate it. On the other hand, when the correct assertion is not known, **the inferring mode** is used to infer this assertion and generate all possible decoders. To help the user select the correct decoder, this mode also infers each decoder's precondition formula, which represents the set of assertions that corresponds to this decoder.

Experimental results show that this tool can always inferring correct assertions and generate decoder circuits for several complex encoders, including PCI-E and Ethernet. And the human effort in specifying assertion and selecting the correct decoder can be significantly reduced.

Keywords: Complementary Synthesis, Inferring Assertion

1 Introduction

One of the most difficult jobs in designing communication 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, while its complementary circuit(or decoder) E^{-1} recovers this information.

In order to facilitate this job, we propose the complementary synthesis[1–4] algorithm, and develop the CompSyn tool to automatically synthesize the decoder circuit of an encoder. This tool has two usage modes:

The synthesis mode: In addition to the encoder's Verilog source code, the user need to specify an assertion on the encoder's configuration pins, to put the encoder into correct working states. With this assertion, CompSyn can automatically determine the existence of the decoder[3] and generate it[2].

The inferring mode: If the correct assertion is NOT known, this mode is used to infer this assertion[4] and generate all possible decoders[5]. To help the user select the correct decoder, this mode also infers each decoder's precondition formula, which represents the set of assertions that correspond to this decoder.

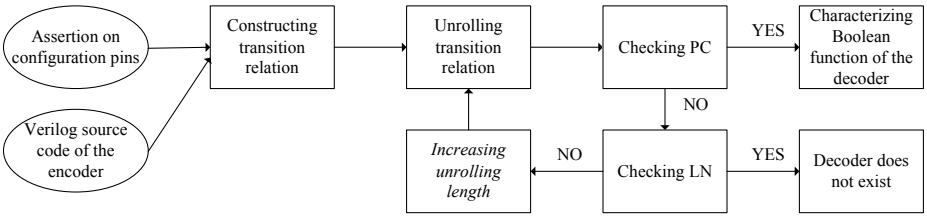

Fig. 1. The flow of the synthesis mode

We have implemented our algorithm with the OCaml language, and solved the generated SAT instances with the Minisat solver[8]. The benchmark set includes several complex encoders from industrial projects (e.g., PCI-E[9] and Ethernet[10]). Experimental results show that this tool can always infer correct assertions and generate decoders for them. And by using the inferring mode, the human effort in specifying assertion and selecting the correct decoder can be significantly reduced. All experimental results and programs can be downloaded from `http://www.ssypub.org`.

2 An Overview of the synthesis mode

The overall flow of the synthesis mode is shown in Figure 1. These steps are connected together by the standard make tool in Linux.

Constructing transition relation

This step takes two input, one is the encoder's Verilog source code, and the other is the assertion on the encoder's configuration pins. Normally, an encoder has several modes, each of which corresponds to a non-overlapped state set:

One of the most important modes is the working mode, in which the encoder encodes its input letters. So the encoder's input letter can be determined by its output sequence, which leads to the existence of its decoder.

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

Thus, the assertion is used here to config the encoder in correct working states.

Unrolling transition relation and checking PC

PC is the abbreviation of Parameterized Complementary Condition defined in [1], which is used to determine the existence of the decoder. Its meaning is intuitively shown in Figure 2a). T is the encoder's transition relation constructed in previous step, while i_n and o_n are respectively the input and output letter. p, d and l are respectively the length of the unfolded transition relation, we also

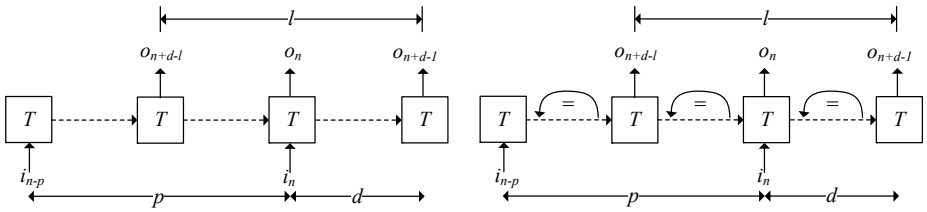

Fig. 2. The PC and LN condition

call them the size of window. This figure, and therefore PC, means that the decoder exists for this encoder if and only if there exists p, d and l, such that the output sequence $< o_{n+d-l}, \ldots, o_{n+d-1} >$ can uniquely determine the input letter i_n.

Checking LN

In addition to the PC mentioned in last subsection, another condition, LN is defined in [3] to determine the non-existence of the decoder. Its meaning is intuitively shown in Figure 2b). It means that the decoder does not exist if and only if there exists p, d and l, such that the output sequence $< o_{n+d-l}, \ldots, o_{n+d-1} >$ can **NOT** uniquely determine the input letter i_n, and there are three loops on state sequences $< s_{n-p}, \ldots, s_{n+d-l} >$,$< s_{n+d-l+1}, \ldots, s_n >$ and $< s_{n+1}, \ldots, s_{n+d} >$.

If this test succeeds, then there does not exist decoder for this encoder and its assertion. Otherwise, the window size will be increased and a new iteration will begin. We have proven in [3] that this loop between LN and PC will eventually terminate.

Characterizing the Boolean function of the decoder

If the PC test succeeds, then there exist a function that map the output sequence $< o_{n+d-l}, \ldots, o_{n+d-1} >$ back to the input letter i_n. This function can be characterized from the Boolean relation shown in Figure 2a).

Currently, we are using the ALLSAT algorithm proposed in [2] to characterize this function. Recently, Liu et al.[6] proposes a much faster algorithm based on Craig interpolant[7] to characterize this function, and the timing and area of the decoder generated by it is comparable to our ALLSAT algorithm.

3 An Overview of the inferring mode

The overall flow of the inferring mode is shown in Figure 3. It is similar to Figure 1, with some new steps in gray color. The user does not need to specify the assertion any more.

Ruling out invalid configuration values

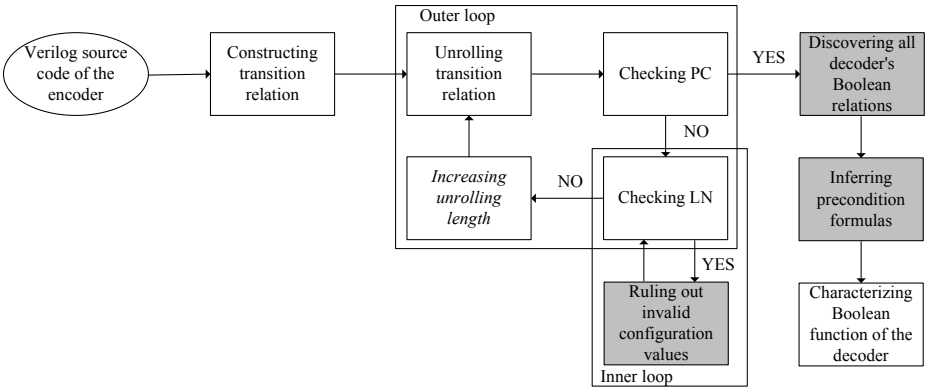

Fig. 3. The flow of the inferring mode

If the LN test succeeds, an invalid configuration value that leads to the non-existence of the decoder can be obtained from the satisfying assignment returned from the SAT solver. We can simply rule out this invalid configuration value and return to the previous step to test LN again. But to reduce the runtime overhead, we propose an algorithm in [4] to enlarge this value to a larger set of invalid configuration value with Craig interpolant[7].

The inner loop between this step and Checking LN step will eventually terminate after such invalid configuration values are all ruled out. Then the window size is increased to test PC again.

We have proven in [4] that the outer loop between LN and PC will eventually terminate. Assume that the set of all configuration values ruled out is NA, then the final inferred assertion is $\wedge_{na \in NA} \neg na$.

Discovering all decoders' Boolean relations

The final inferred assertion is a formula that actually contains many different configuration values. This means that there may exist multiple decoders for this inferred assertion.

Thus, our job here is to find out the set of all decoders' Boolean relations $\{\mathbb{R}_1, \ldots, \mathbb{R}_n\}$ 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 shown in Fig. 4 based on functional dependency[11].

To simplify presentation, we denote i_n as Y, and $< o_{n+d-l}, \ldots, o_{n+d-1} >$ as X. $R(X, Y)$ is the Boolean relation shown in Figure 2a) that succeeds in checking PC.

If this SAT instance is unsatisfiable, then all decoders' have been discovered. Otherwise, we can discover a new decoder's Boolean relation by assigning the satisfiy assignment of c to R.

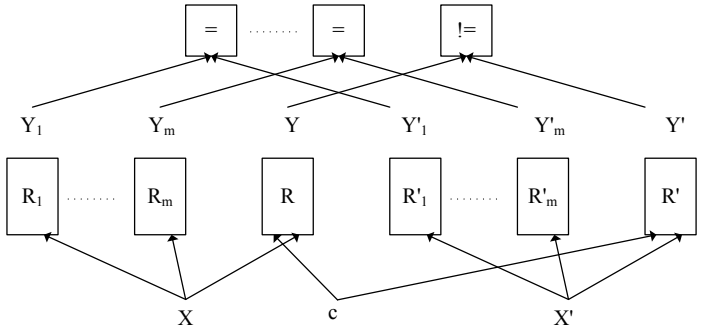

Fig. 4. The SAT instance that discovers decoders

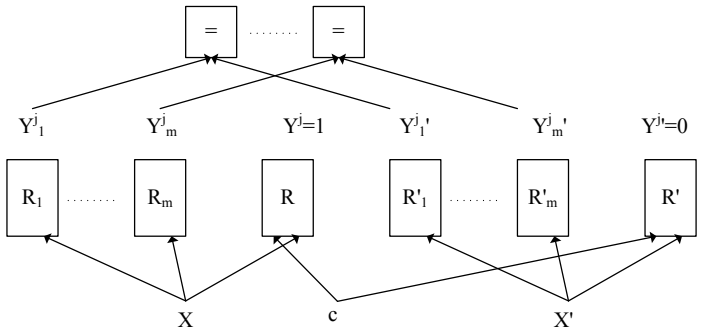

Fig. 5. The SAT instance that infers precondtion formulas

A new round of functional dependency test will be perform again until no more decoder can be discovered. After that these Boolean relations will be used to characterize their corresponding decoders' Boolean functions.

Inferring precondition formulas

Assume $\{\mathbb{R}_1, \ldots, \mathbb{R}_m\}$ is the set of all decoders' Boolean relations discovered in last step, 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 $\{IA_1, \ldots, IA_m\}$.

Assume Y and all Y_i in Figure. 4 are vectors of the same length v, and the j-th bit if them are respectively Y^j and all Y_i^j. Then the SAT instance shown in Fig. 5 will generate the set of formula $\{IA_1^j, \ldots, IA_m^j\}$, which represents the precondtion formulas for the j-th bit of IA_i in $\{IA_1, \ldots, IA_m\}$.

Thus, IA_i can be defined as $\bigwedge_{j=0}^{v-1} IA_i^j$.

4 Experimental results

We have implemented our algorithm with the OCaml language, and solved the generated SAT instances with the Minisat solver[8]. The benchmark set includes several complex encoders from industrial projects (e.g., PCI-E[9] and Ethernet[10]).

According to [3], when given the assertion, no matter correct or not, the synthesis mode can determine the existence of the decoder within 40 seconds, and build the decoders within 10 seconds.

According to [4, 5], the runtime overhead for inferring assertion, generating decoders and inferring their precondtion formulas are no more than 400 seconds. Moreover, 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 $\{IA_1, \ldots, IA_m\}$ to select the correct decoder.

For example, for the two decoders of the most complex XFI encoder that compliant to 10G Ethernet standard[10], 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.

In this process, the user only need to inspect the meaning of one configuration pin, instead of all 120 configuration pins of the XFI encoder. In this way, the human effort in specifying assertion and selecting the correct decoder is significantly reduced.

5 Conclusions

Experimental results show that this tool can always inferring correct assertions and generate decoder circuits for several complex encoders, including PCI-E and Ethernet. And the human effort in specifying assertion and selecting the correct decoder can be significantly reduced.

References

1. Shen, S., Zhang, J., Qin, Y., Li, S. : Synthesizing Complementary Circuits Automatically. In: 2009 International Conference on Computer-Aided Design, pp. 381–388. IEEE Press, New York (2009)
2. Shen, S., Qin, Y., Wang, K., Xiao, L., Zhang, J., Li, S. : Synthesizing Complementary Circuits Automatically. IEEE transaction on CAD of Integrated Circuits and Systems 29(8), 1191–1202 (2010)
3. Shen, S., Qin, Y., Xiao, L., Wang, K., Zhang, J., Li, S. : A halting algorithm to determine the existence of the decoder. IEEE transaction on CAD of Integrated Circuits and Systems 30(10), 1191–1202 (2011).

4. Shen, S., Qin, Y., Zhang, J., Li, S. : Inferring Assertion for Complementary Synthesis. *accepted by ICCAD11.*
5. Shen, S., Qin, Y., Xiao, L., Wang, K., Zhang, J., Li, S. : Inferring Assertion for Complementary Synthesis. submitted to IEEE transaction on CAD of Integrated Circuits and Systems.
6. Liu H., Chou Y.C., Lin C.H., Jiang J.-H. R. : Completely Automatic Decoder Synthesis. *accepted by ICCAD11.*
7. Craig, W. : Linear reasoning: A new form of the Herbrand-Gentzen theorem. J. Symbolic Logic. 22(3), 250–268 (1957)
8. Eén, N., Sörensson, N. : Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2003)
9. PCI Express Base Specification Revision 1.0. Download from `http://www.pcisig.com`
10. *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.
11. Lee C.-C. , Jiang J.-H. R. , Huang C.-Y. , Mishchenko A. : Scalable exploration of functional dependency by interpolation and incremental SAT solving. In: 2007 International Conference on Computer-Aided Design, pp. 227–233. IEEE Press, New York (2007)