**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 a decoder circuit from an encoder and a predefined assertion. This tool has two usage modes: the synthesis mode and the inferring mode.

When the correct assertion is known, **the synthesis mode** is used to determine the existence of the decoder and generate it. On the other hand, when the 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 all cases that can generate this decoder.

Experimental results show that this tool can infer assertions and generate decoders for several complex encoders, including PCI-E and Ethernet, and the human effort in specifying assertion is significantly reduced.

**Keywords:** Complementary Synthesis, Inferring Assertion, Craig Interpolation, Functional Dependency

**1 Introduction**

One of the most difficult tasks 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.

To facilitate this job, we have proposed the complementary synthesis algorithm [1–4] and developed the CompSyn tool to automatically synthesize the decoder circuit of an encoder. This tool has two usage modes, the synthesis mode and the inferring mode.

When the correct assertion is known, **the synthesis mode** determines the existence of the decoder [3] by iteratively checking whether its input letter can be uniquely determined by its output sequence, and characterizes the decoder’s Boolean function [2] with Craig interpolation [7].

On the other hand, to manually specify an assertion, the user must read extensive documentation and often perform laborious trial-and-error process. That is why we develop the inferring mode to infer the assertion automatically. **The inferring mode** includes three steps: **1)** Inferring the assertion [4] by detecting and removing all cases without decoders. **2)** Using functional dependency [11] to decompose *R*, the Boolean relation uniquely determining the input letter, into all possible decoders [5]. **3)** Inferring each decoder’s precondition formula [5] that represents all cases leading to this decoder’s existence, to help the user select the correct decoder.

For example, our most complex benchmark–the XFI encoder, has 120 configuration pins. Finding out their meaning and correct combination was a very difficult and lengthy process, which I had already experienced when using the synthesis mode. On the contrary, for the two decoders discovered by the inferring mode, their corresponding precondition formulas refer to only three pins, in which only two have different value. Therefore, to select the correct decoder, I only need to find out the meaning of these **TWO** configuration pins instead of all 120 configuration pins. More detail can be found in the experimental results section, which shows that the inferring mode can always significantly save the human effort in specifying assertion and selecting decoder. All the experimental results and programs can be downloaded from http: //www.ssypub.org.

**The remainder of this paper is organized as follows**. Section 2 introduces the CompSyn tool’s software architecture. Section 3 presents the synthesis mode, while Section 4 introduces the inferring mode. Section 5 presents the experimental results, and lastly, Section 6 provides the conclusion.

**2 The software architecture**

The CompSyn tool is implemented in the OCaml language. Its architecture is shown in Figure 1, which comprises the following major components:

**2.1 Syntax analyzer**

This analyzer analyzes the encoder’s Verilog source code and assertions, and generates the transition relation’s circuit description in an And-Inverter graph(AIG)

[12].

Syntax analyzer CNF manager

SAT solver

interface

AIG manager

Craig

interpolant

generator

MiniSAT

solver

BDD interface

and CUDD

package

**Fig. 1.** The architecture of CompSyn

CompSyn: A Tool for Automatically Synthesizing Decoders 3

**2.2 AIG manager**

This AIG manager represents a circuit with an array. Elements of this array are of the following types:

1. TRUE: A logical constant 1 without parameter.

2. FALSE: A logical constant 0 without parameter.

3. VARIABLE: A node with an integer parameter, which is this variable’s encoding number.

4. INVERTER: An inverter with an integer parameter, which refers to the index of the element that drives this inverter.

5. BUFFER: A non-invert buffer with an integer parameter, which refers to the index of the element that drives this buffer.

6. AND: A two-input AND gate with two integer parameters that refer to the indexes of the elements that drive this AND gate. This AIG manager provides some procedures to manipulate the circuit represented in AIG, such as removing redundant elements, propagating constance,

and translating an AIG circuit to CNF formula.

**2.3 CNF manager**

This manager takes care of all CNF formulas generated by the syntax analyzer, unrolled transition relations used to determine the existence of the decoder, the formulas used to generate Craig interpolant, and the formulas used to test functional dependency [11].

The set of clauses of a CNF formula is stored in a OCaml list, while each element of this list is another list that stores this clause’s literal set. The reason for using list is that the clauses in a CNF formula do not need to be randomly accessed, and the list type provides the flexibility to increase the size of the formula dynamically.

**2.4 The SAT solver interface and the SAT solver**

The SAT solver used here is the minisat solver v1.14 [8]. This solver provides the ability to generate a proof for an unsatisfiable formula, which can be analyzed to generate a Craig interpolant.

The proof generated by the minisat solver often includes most of the clauses appearing in the original formula, which causes huge runtime overhead in generating Craig interpolant. To reduce this overhead, the minisat solver is modified to minimize the proof, by removing those redundant clauses.

The OCaml to C interface that links CompSyn and minisat together is MiniSat-ocaml [13] developed by Flavio Lerda. However, one of its major shortcomings is that it does not provide the ability to read back the proof from the minisat solver. Hence, such a procedure is added.

The other major shortcoming of this interface is that it only provides a procedure to allocate new variables one by one. As the overhead of calling C procedure from OCaml is very high, this will lead to very large runtime overhead for large formulas. To reduce this overhead, a new procedure that allocates all new variables in one calling is added.

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

Verilog source

code of the

encoder

Assertion on

configuration pins

Constructing

transition

relation

Determining

the decoder's

existence by

checking *PC*

YES

Characterizing

Boolean

function of the

decoder

Determining

the decoder's

nonexistence

by checking *LN*

NO

Unrolling

transition

relation for a

window size

Increasing the

window size

YES Decoder does

not exist

NO

**Fig. 2.** The flow of the synthesis mode

**2.5 The Craig interpolant generator and the BDD interface**

The Craig interpolant generator works on the proof return from the minisat solver, and generates the interpolant in AIG form. One shortcoming of our Craig interpolant generator, which is also shared by other implementations of the same algorithm [14], is that the generated interpolant contains lots of redundant gates.

To remove these redundant gates, the CUDD package is invoked to generate a canonical representation of the interpolant. The OCaml to C interface that links CUDD to CompSyn is taken from Blast model checker [15]. After the simplification, the BDD is converted back to a much more compact AIG by enumerating all cubes of this BDD.

**3 An overview of the synthesis mode**

The overall flow of the synthesis mode is shown in Figure 2. The loop iteratively increases the window size and unrolls the transition relation on that window. Each iteration determines the decoder’s existence by checking *PC* condition, and determine the decoder’s nonexistence by checking *LN* condition. These two conditions will be introduced intuitively in the following subsections.

**3.1 Constructing transition relation**

This step takes two inputs, one is the encoder’s Verilog source code, 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. Hence, the encoder’s input can be determined by its output, 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 processes test commands or does nothing, respectively.

Therefore, in these modes, the encoder’s input cannot be determined by its output, which leads to the nonexistence of its decoder. Therefore, the user needs to specify an assertion here to constrain the correct value of the encoder’s configuration pins, such that the encoder can be put in the correct working modes.

CompSyn: A Tool for Automatically Synthesizing Decoders 5

*on+d-l on+d-1*

*in-p*

*T*

*in*

*T*

*l*

*p d*

*on*

*T T*

*on+d-l' on+d-1'*

*in-p'*

*T*

*in'*

*T*

*on'*

*T T*

*l*

=

*!=*

**Fig. 3.** The *PC* condition

**3.2 Unrolling the 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 3, where *T* is the encoder’s transition relation constructed in the previous step, while *in* and *on* are the input and output letter, respectively, and *p*, *d*, and *l* are the lengths of the unfolded transition relation, which are also called **the window size**.

This figure, and therefore *PC*, indicates that the decoder exists if and only if there exists *p*, *d*, and *l*, such that the output sequence *< on*+*d−l, . . . , on*+*d−*1 *>* can uniquely determine the input letter *in*. In other words, there does not exist two different input letters *in* and *i\_n* that can be recovered from the same output sequence *< on*+*d−l, . . . , on*+*d−*1 *>*.

This condition can be encoded into a SAT instance and solved with the minisat solver [8]. If the result is unsatisfiable, then the decoder exists; otherwise, the decoder does not exist for this particular value of *p*, *d*, and *l*. CompSyn needs to check *LN* or increase the window size.

**3.3 Checking *LN***

In addition to the *PC* mentioned in the last subsection, another condition *LN* had been defined in [3] to determine the nonexistence of the decoder. Its meaning is intuitively shown in Figure 4, indicating that the decoder does not exist if and only if there exists *p*, *d*, and *l*, such that the output sequence *< on*+*d−l, . . . , on*+*d−*1 *>* can **NOT** uniquely determine the input letter *in*, and there are three loops on state sequences *< sn−p, . . . , sn*+*d−l >*,*< sn*+*d−l*+1*, . . . , sn >*, and *< sn*+1*, . . . , sn*+*d >*.

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

*on+d-l on+d-1*

*in-p*

*T*

*in*

*T*

*l*

*p d*

*on*

*T T*

*on+d-l' on+d-1'*

*in-p'*

*T*

*in'*

*T*

*on'*

*T T*

*l*

=

*!=*

= = = = = =

**Fig. 4.** The *LN* condition

This condition can be encoded into a SAT instance, and solved with the minisat solver [8]. If this SAT instance is satisfiable, then the nonexistence of the decoder for all those longer paths can be proved by unfolding these three loops. Otherwise, the window size will be increased and a new iteration will begin. We have proven in [3] that the loop between *LN* and *PC* will eventually terminate.

**3.4 Characterizing the Boolean function of the decoder**

If the *PC* checking succeeds, then there exists a function that maps the output sequence *< on*+*d−l, . . . , on*+*d−*1 *>* back to the input letter *in*. This function can be characterized from the Boolean relation shown in Figure 3, with the ALLSAT algorithm proposed in [2].

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

This algorithm had also been implemented in CompSyn. To simplify the presentation, we denote *in* as *Y* , the *j*-th bit of *Y* as *Y j*, and*< on*+*d−l, . . . , on*+*d−*1 *>* as *X*. *R*(*X, Y* ) is the Boolean relation shown in Figure 3, which succeeds in checking *PC*. As *R* can uniquely determine the value of *Y* from *X*, a Craig interpolant of *R*(*X, Y* ) *∧ Y j ≡* 1 with respect to *R*(*X, Y \_*) *∧ Y \_j ≡* 0 can be generated. This Craig interpolant is exactly the Boolean function that computes the *j*-th bit of *Y* from *X*.

**4 An overview of the inferring mode**

To manually specify an assertion, the user must read extensive documentation and often perform laborious trial-and-error process. This is why we develop the inferring mode that infers the assertion automatically. The flow of the inferring mode is shown in Figure 5. It is similar to Figure 2, with some new steps in gray color. The assertion can be automatically inferred in this mode, so the user does not need to specify it here.

CompSyn: A Tool for Automatically Synthesizing Decoders 7

Verilog source

code of the

encoder

Constructing

transition

relation

YES

Discovering all

decoder's

Boolean

relations

NO

Unrolling

transition

relation

NO Inferring

precondition

formulas

Ruling out

invalid

configuration

values

YES

Characterizing

Boolean

function of the

decoder

Outer loop

Inner loop

*Increasing*

*window size*

Checking *PC*

Checking *LN*

**Fig. 5.** The flow of the inferring mode

One major difference while compared to the synthesis mode is the inner loop on the Checking *LN* step. Because the Checking *LN* step can find out a configuration pin value that leads to the decoder’s nonexistence, a new step is inserted after it to remove all such values, to ensure all remained configuration pin values have corresponding decoders.

The other major difference is the addition of two steps after the outer loop. Because the inferred assertion actually includes multiple configuration pin values, there may simultaneously exist multiple decoders. Therefore, a new step is inserted to discover all of them, and another step is inserted to infer a precondition formula for each decoder, which represents the set of configuration values that can lead to this decoder’s existence.

**4.1 Ruling out invalid configuration values**

If the *LN* checking succeeds, an invalid configuration value that leads to the nonexistence of the decoder can be obtained from the minisat[8] solver’s satisfying assignment. We can simply rule out this invalid configuration value and return to the previous step to check *LN* again. However, to reduce the runtime overhead, an algorithm has been proposed in [4] to enlarge this value to a larger set of invalid configuration values with Craig interpolant [7], such that they can be rule out altogether.

The formula used in enlarging is shown in Figure 6. It is very similar to that of Figure 4, except that the existence of the six loops and the equality of the two output sequences is anded together to generate a new Boolean variable *t*. By assigning all *in*’s and *i\_n*’s satisfying assignment of the SAT instance shown in Figure 4, this formula can be transformed into a new formula *F*(*c, t*), which defines a circuit, whose input is the configuration value *c*, and output is *t*.

The Boolean function of this circuit can be characterized by generating the Craig interpolant of *F*(*c,* 1) with respect to *F*(*c,* 0). Thus, this Boolean function is exactly the enlarged set of invalid configuration values.

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

*on+d-l on+d-1*

*in-p*

*T*

*in*

*T*

*lo*

*n*

*T T*

*on+d-l' on+d-1'*

*in-p'*

*T*

*in'*

*T*

*on'*

*T T*

*l*

=

= = = = = =

and

t

**Fig. 6.** Enlarging the invalid configuration value with Craig interpolant

The inner loop between this step and the Checking *LN* step will eventually terminate after such invalid configuration values are all ruled out. Then, the window size is increased to check *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 *∧na∈NA¬na*.

**4.2 Discovering all decoders’ Boolean relations**

The final inferred assertion is a formula that actually contains many different configuration values. This means that multiple decoders may exist simultaneously for this inferred assertion. Thus, we need to find out the set of all decoders’ step by step.

1 m

1 m

1 m

1 m

**Fig. 7.** The SAT instance that discovers all decoders

CompSyn: A Tool for Automatically Synthesizing Decoders 9

We denote *in* as *Y* , *< on*+*d−l, . . . , on*+*d−*1 *>* as *X*, and the configuration value as *c*. Then *R*(*c,X, Y* ) is the Boolean relation shown in Figure 3 that succeeds in checking *PC*.

Assume *{R*1*, . . . , Rm}* is a set of decoders’ Boolean relations that has already been discovered. To test whether all decoders have already been discovered, the SAT instance shown in Figure 7 is constructed based on functional dependency [11].

According to [5], if this SAT instance is unsatisfiable, then *Y* can be uniquely determined by *{Y*1*, . . . , Ym}*, which means all decoders have been discovered. Otherwise, a new decoder’s Boolean relation can be discovered by assigning the satisfying assignment of *c* to *R*.

A new round of functional dependency test will be performed again, until no more decoder can be discovered. Subsequently, these Boolean relations will be used to characterize their corresponding decoders’ Boolean functions.

**4.3 Inferring precondition formulas**

Assume that *{R*1*, . . . , Rm}* is the set of all decoders’ Boolean relations discovered in the last step, and *{IA*1*, . . . , IAm}* is their corresponding set of configuration letters. To help the user determine which *Ri* in *{R*1*, . . .,Rm}* is the correct decoder, each *IAi* in *{IA*1*, . . . , IAm}* must be characterized.

Assume *Y* and all *Yi* in Figure 7 are vectors of the same length *v*, and their *j*-th bit are *Y j* and *Y j*

*i* ,respectively. If the precondition formulas *IAj i* for the *j*-th bit of *IAi* can be characterized, then *IAi* can be defined as \_*v−*1 *j*=0 *IAj i* .

To achieve this goal, the unsatisfiable SAT instance shown in Figure 8 is constructed. This SAT instance is very similar to that of Figure 7, except that only the *j*-th bit is constrained. Therefore it is unsatisfiable. As the set of common variables between formula *A* and *B* is *{c, Y j*

1 *, . . . , Y jm }*, the generated interpolant of *A* with respect to *B* is a function *F*(*c, Y j* 1 *, . . . , Y jm*

) that computes the value of *Y j* . By setting the value of *Y j i* to 1 and all other *{Y j k |k \_*= *i}* to 0 in

*F*(*c, Y j* 1 *, . . . , Y jm* ), we can obtain the formula *IAji* .

1 m

j

1

j

m

j

1 m

1

j

m

j j

**Fig. 8.** The SAT instance that infers precondition formulas

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

**Table 1.** Information on Benchmarks

XGXS XFI scrambler PCIE T2 Ethernet

Line number

of verilog 214 466 24 1139 1073

source code

#state variables 15 135 58 22 48

Data path width 8 64 66 10 10

**5 Experimental results**

**5.1 The usage model**

All these steps mentioned above are connected together by the standard make tool in Linux. To use the synthesis mode, the user needs to run the command ”make halting” under the bash shell in the benchmark directory. To use the inferring mode, the user needs to run the command ”make infer multidec not charfirst nowall”.

**5.2 Benchmarks**

The benchmarks used in the experiments include several complex encoders from industrial projects,

1. A XGXS encoder compliant to clause 48 of IEEE-802.3ae 2002 standard [10].

2. A XFI encoder compliant to clause 49 of the same IEEE standard.

3. A 66-bit scrambler used to ensure that a data sequence has sufficient 0-1 transitions, so that it can run through a high-speed noisy serial transmission channel.

4. A PCI-E physical coding module [9].

5. The Ethernet module of Sun’s OpenSparc T2 processor.

The profiles of these benchmarks are shown in Table 1. Some of these large benchmarks have more than 1000 lines of source code, while the XFI encoder has more than 100 state variables.

**5.3 The experimental result of the synthesis mode**

According to Table 2 and [3], when given the assertion, the synthesis mode can determine the existence of the decoders within 40 seconds, and builds the decoders within 10 seconds with an algorithm similar to that of Liu et al. [6]. Moreover, we inserted some bugs into these encoders, which generated the same output letter for two different input letters. CompSyn successfully detected all these bugs within 10 seconds.

CompSyn: A Tool for Automatically Synthesizing Decoders 11

**Table 2.** Experimental results on the correct and incorrect encoders

XGXS XFI scrambler PCI-E T2 Ethernet

For Time to chcorrect

eck *PC*(sec) 0.29 17.86 2.67 0.47 29.64

encoder *d, p, l* 1,2,1 0,3,2 0,2,2 2,2,1 4,4,1

For Time to chincorrect

eck *PC*(sec) 0.16 7.59 1.17 0.33 2.19

encoder

**5.4 The experimental result of the inferring mode**

According to [4, 5], when the assertions are not known, the inferring mode can infer assertions, generate decoders and infer these decoders’ precondition formulas within 4000 seconds.

The set of inferred assertions are shown below:

**For XGXS**: ( ( !bad code ) )

**For XFI**: ( ( TEST MODE ) | ( !TEST MODE & RESET ) | ( !TEST MODE

& !RESET & DATA VALID ) )

**For scrambler**: True

**For PCI-E**: ( ( !TXELECIDLE & CNTL TXEnable P0 & CNTL RESETN P0 &

!CNTL Loopback P0 ) )

**For T2 ethernet**: ( ( link up loc & !reset tx & !txd sel[1] & !jitter study pci[1]

& !txd sel[0] & !jitter study pci[0] ) | ( !link up loc & !reset tx &

!txd sel[1] & tx enc conf sel[3] & tx enc conf sel[2] & !jitter study pci[1]

& !txd sel[0] & !jitter study pci[0] ) | ( !link up loc & !reset tx &

!txd sel[1] & tx enc conf sel[3] & !tx enc conf sel[2] & tx enc conf sel[1]

& !jitter study pci[1] & !txd sel[0] & !jitter study pci[0] ) | ( !link up loc

& !reset tx & !txd sel[1] & !tx enc conf sel[3] & !tx enc conf sel[2]

& tx enc conf sel[1] & !jitter study pci[1] & !txd sel[0] & !jitter study pci[0]

& tx enc conf sel[0] ) )

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 needs to inspect the inferred precondition formulas to select the correct decoder.

**Table 3.** Experimental Results of the inferring mode

XGXS XFI scrambler PCI-E T2 Ethernet

Config

pin number 3 120 1 16 26

Runtime 3.83 3841.34 18.73 8.51 1791.22

*d, p, l* 1,5,1 0,5,2 0,4,2 2,5,1 4,5,1

Number of

decoders 1 2 2 1 1

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

For the two decoders of scrambler, their corresponding precondition formulas are *reset* and !*reset*. By inspecting the Verilog source code of scrambler, we found that the *reset* is used to reset the scrambler when it is *True*. Thus, the XFI encoder will work in normal mode when *reset* is *False*. Therefore, the second decoder is the correct decoder. And the dynamic simulation had confirmed its correctness.

For the two decoders of the most complex XFI encoder [10], their corresponding precondition formulas are *RESET* &!*TEST MODE* & !*DATA V ALID* and !*RESET* & !*TEST MODE* & *DATA V ALID*. The only differences between them are the value of *RESET* and *DATA V ALID*. By inspecting the Verilog source code of XFI, we found that the *RESET* is used to reset the XFI encoder when it is *True*, and *DATA V ALID* means that the input data is valid when it is *True*. So, the XFI encoder will work in normal mode when *RESET* is *False* and *DATA V ALID* is *True*. Therefore, the second decoder is the correct decoder. The dynamic simulation had also confirmed its correctness. Haha

In this process, the user only needs to inspect the meaning of two configuration pins, 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.

**6 Conclusions**

The CompSyn tool can infer correct assertions and generate decoder circuits for several complex encoders. Furthermore, it can significantly reduce the human effort in specifying assertion and selecting the correct decoder.

**References**

1. ShengYu Shen, Jianmin Zhang, Ying Qin, Sikun Li: Synthesizing complementary

circuits automatically. ICCAD 2009: 381-388

2. ShengYu Shen, Ying Qin, KeFei Wang, LiQuan Xiao, Jianmin Zhang, Sikun Li:

Synthesizing Complementary Circuits Automatically. IEEE Trans. on CAD of Integrated

Circuits and Systems 29(8): 1191-1202 (2010)

3. ShengYu Shen, Ying Qin, LiQuan Xiao, KeFei Wang, Jianmin Zhang, Sikun Li:

A Halting Algorithm to Determine the Existence of the Decoder. IEEE Trans. on

CAD of Integrated Circuits and Systems 30(10): 1556-1563 (2011)

4. ShengYu Shen, Ying Qin, Jianmin Zhang, Sikun Li: Synthesizing complementary

circuits automatically. *accepted by ICCAD11*.

5. ShengYu Shen, Ying Qin, LiQuan Xiao, KeFei Wang, Jianmin Zhang, Sikun Li:

Inferring Assertion for Complementary Synthesis. submitted to IEEE transaction

on CAD of Integrated Circuits and Systems.

6. Siou-Yuan Liu, Yen-Cheng Chou, Chen-Hsuan Lin, Jie-Hong Roland Jiang : Completely

Automatic Decoder Synthesis. *accepted by ICCAD11*.

CompSyn: A Tool for Automatically Synthesizing Decoders 13

7. William Craig, : Linear reasoning: A new form of the Herbrand-Gentzen theorem.

J. Symbolic Logic. 22(3), 250–268 (1957)

8. Niklas E′en, Niklas S¨orensson: An Extensible SAT-solver. SAT 2003: 502-518

9. PCI Express Base Specification Revision 1.0. 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. Chih-Chun Lee, Jie-Hong Roland Jiang, Chung-Yang Huang, Alan Mishchenko:

Scalable exploration of functional dependency by interpolation and incremental SAT

solving. ICCAD 2007: 227-233

12. Alan Mishchenko, Satrajit Chatterjee, Robert K. Brayton: DAG-aware AIG rewriting

a fresh look at combinational logic synthesis. DAC 2006: 532-535

13. https://github.com/flerda/MiniSat-ocaml/wiki

14. Kenneth L. McMillan: Interpolation and SAT-Based Model Checking. CAV 2003:

1-13

15. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar: The software

model checker Blast. STTT 9(5-6): 505-525 (2007)