论 文

无需手工给出断言的对偶综合

沈胜宇*，秦莹，张建民，李思昆

国防科技大学计算机学院, 长沙410073
*沈胜宇E-mail: syshen@nudt.edu.cn, qy123@nudt.edu.cn, jmzhang@nudt.edu.cn, skli@nudt.edu.cn

国家自然科学基金(批准号: 61070132)资助项目

摘要 对偶综合能够自动生成特定编码器的解码器. 然而其用户需要手工给出配置信号上的断言以防止编码器到达非工作状态.

为了防止该繁琐工作, 我们提出了能够自动推导断言的算法. 对于能够导致解码器不存在的每一个非法配置信号值, 我们使用Craig插值推导出一个包含大量非法配置信号值的公式. 该步骤将重复直至所有非法信号值均被剔除为止. 最终的断言可以通过将所有推导的公式取反再与到一起得到.

然而, 在该推导断言结果下可能存在多个解码器. 为此我们提出了另一个算法以产生所有这些解码器, 通过迭代的测试R, 能够唯一决定输入字符的布尔关系, 是否能够表达为所有已发掘的解码器的组合. 对于所有导致该测试失败的配置信号值, 一个新的解码器的布尔关系可以通过向R中强制该值得到. 该步骤将重复直至所有解码器均被发掘为止. 为了帮助用户从所有发掘的解码器中选择正确的解码器, 第三个算法被提出以计算每个解码器的前提条件.

为了展示该算法的有效性, 我们在多个复杂编码器上运行了本文的算法, 包括PCI-E和以太网. 实验解雇表明本文算法始终能够推导正确的断言并产生解码器. 进一步的, 基于特征化的每个解码器的前提条件, 用户可以非常容易的选择正确的解码器。

关键词
对偶综合
推导断言
Cofactoring
Craig插值

1 Introduction

在设计通讯和多媒体芯片中最为复杂的工作是设计和验证对偶电路对(E, E^{-1}), 其中编码器E将输入数据流编码为易于传输和存储的格式, 而其对偶电路(或解码器)E^{-1}则恢复该信息.

为了提高该工作的效率, 对偶综合算法被提出[1, 2]以自动综合特定编码器的解码器, 通过检查特定编码器的输入能够被输出唯一决定.

然而, 对偶综合的用户需要手工给出配置信号上的断言, 以阻止编码器到达非工作状态. 例如在测试和休眠模式下, 编码器将致力于处理测试命令或者什么也不干. 在这些模式下, 该编码器的输入不能够被其输出唯一决定, 这将导致解码器不存在.

为了避免手工给出断言的繁琐工作, 我们给出如下的自动算法: **首先**, 我们使用由Shen et al. [3]提出的停机算法以找到能够导致解码器不存在的一个配置字符. **其次**, 我们使用cofactoring[4]和Craig插值[5]以推导一个公式, 该公式覆盖了大量非法的配置字符. 这两步被重复直至所有非法配置字符均被覆盖为止. **最后**, 我们通过将所有推导公式取反后与在一起而得到最后断言.

然而, 在这个断言之下有可能同时存在多个解码器. 对于图1中的编码器, 不管c取什么值, in总能够被out唯一决定. 然而该编码器存在两个解码器, 其中一个是$in \stackrel{def}{=} out - 1$, 另一个是$in \stackrel{def}{=} out - 2$.

为此我们提出了另一个算法以得到所有这些解码器, 通过迭代的测试R, 能够唯一决定输入字符的布尔关系, 能否表达为所有已经发掘的解码器的组合. 对于每一个导致该测试失败的配置字符, 通过将该值设置到R中，一个新的解码器的布尔关系可以被得到. 该步骤重复直至所有解码器均被发掘.

为了帮助用户在多个潜在解码器之间选择正确的解码器, 第三个算法被提出以为每一个发掘的解码器计算器前提条件. 通过检验这些前提条件用户可以非常容易的选择正确的解码器.

我们在多个来自于工业界的复杂编码器(如PCI-E[6]和以太网[7])上运行了我们的算法. 试验结果表明我们的算法始终能够为他们推导断言并生成解码器. 进一步的, 通过检验每个解码器的前提条件, 用户可以非常容易的选择正确的解码器. 所有的程序和试验结果可在下列网址下载`http://www.ssypub.org/exp/compsyn_fmcad11.tgz`.

本文的剩余部分如下组织. 节2介绍背景资料. 节3给出了推导断言的算法及其正确性证明. 节4削减节3找到的参数值, 而节5则讨论了如何找到所有的解码器的布尔关系以及他们的前提条件. 节6和7分别给出了试验结果和相关工作. 最后, 节8总结全文.

2 背景

2.1 命题逻辑和相关主题

布尔集合记为$\mathbb{B} = \{0, 1\}$. 对于布尔变量集合V上的公式F, 其布尔命题逻辑可满足问题就是寻找满足赋值$A : V \to \mathbb{B}$, 使得F为1. 如A存在则F是可满足的; 否则是不可满足的. 寻找A的一个计算机程序称为SAT求解器. 一般的, 公式F表达为CNF格式, 其中公式是多个短句的与, 而一个短句是多个文字的或, 而一个文字是一个变量或其反. 一个CNF公式也称为SAT实例.

对于一个布尔函数$f : \mathbb{B}^n \to \mathbb{B}$, 我们用$supp(f)$表示其支撑集$\{v_1 \dots v_n\}$. 由Ganai et al. [4], 面向$v$的$f(v_1 \dots v \dots v_n)$的正负cofactors 分别是$f_v = f(v_1 \dots 1 \dots v_n)$和$f_{\overline{v}} = f(v_1 \dots 0 \dots v_n)$. **Cofactoring**是将0或者1赋予v以得到f_v或$f_{\overline{v}}$的动作.

Craig[5]证明了以下定理:

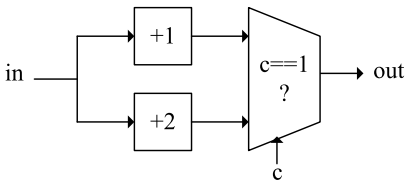

图 1 同时存在的多个解码器

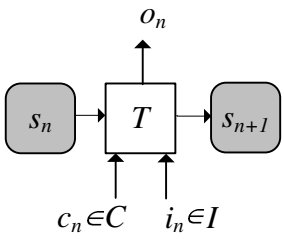

图 2 带配置信息的Mealy有限状态机

Theorem 1 (Craig插值定理). 对于两个公式ϕ_A和ϕ_B, 若$\phi_A \wedge \phi_B$不可满足, 则存在一个仅引用了ϕ_A和ϕ_B共同变量的公式ϕ_I 使得$\phi_A \rightarrow \phi_I$成立且$\phi_I \wedge \phi_B$不可满足. ϕ_I称为ϕ_A面向ϕ_B的插值.

在本文剩余部分, 我们将使用McMillan[10]提出的算法从MiniSAT求解器[11]产生的不可满足证明中产生插值.

对于布尔函数$f : \mathbb{B}^l \rightarrow \mathbb{B}$和布尔函数集合$G = (g_1, ..., g_n)$ with $g_i : \mathbb{B}^l \rightarrow \mathbb{B}$其中$i = 1, \ldots, n$, **函数依赖性**[17]致力于寻找第三个函数$h : \mathbb{B}^n \rightarrow \mathbb{B}$, 以使得$f(X) = h(g_1(X), \ldots, g_n(X))$. Lee et al. [17]构造了一个SAT实例以测试h是否存在, 并用Craig插值[10]特征化他.

2.2 检验解码器的存在

对偶综合[1]包括两步: 检验解码器是否存在和特征化其布尔函数. 我们在这里将仅介绍第一步. 为了建模带配置信号的编码器, 我们扩展了传统的Mealy有限状态机的定义[8]:

Definition 1. 带配置信息的***Mealy***有限状态机是一个6元祖$M = (S, s_0, I, C, O, T)$, 包括状态集合S, 初始状态$s_0 \in S$, 有限输入字符集合I, 有限配置字符集合C, 有限输出字符集合O, 迁移函数$T : S \times I \times C \rightarrow S \times O$从当前状态, 输入字符和配置字符计算下一状态和输出字符.

如图2所示, 状态使用圆角灰色方框表示, 而迁移函数T使用白色方框表示. 我们将第n个周期的状态, 输入字符, 输出字符和配置字符表示为s_n, i_n, o_n和c_n. 我们将第n到m个周期的状态, 输入字符, 输出字符和配置字符序列表示为s_n^m, i_n^m, o_n^m和c_n^m. 一个路径是一个状态序列s_n^m其中$\exists i_j o_j c : (s_{j+1}, o_j) \equiv T(s_j, i_j, c)$对于所有$n \leqslant j < m$. 一个环是一个路径$s_n^m$其中$s_n \equiv s_m$.

一个Mealy有限状态机M的递归半径是其最长无环路径:

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

该定义和状态变量递归半径[9]的区别在于我们的$uirrd$不考虑初始状态. 我们将仅使用该半径来证明我们的定理. 我们的算法并不需要计算他们.

配置信号桑的断言(或者公式)是一个配置字符集合R. 对于配置字符c, $R(c)$意味着$c \in R$. 如果$R(c)$橙路, 我们也说R覆盖c.

如图3所示, 解码器存在当存在三个参数p, d和l, 使得i_n可以被编码器的输出序列o_{n+d-l}^{n+d-1}唯一决定. d是o_{n+d-l}^{n+d-1}和i_n之间的相对延时, 而l是o_{n+d-l}^{n+d-1}的长度, 而p是用于剔除某些不可达状态的前置状态序列. 这可以形式化的定义为:

Definition 2. 参数对偶条件*(PC)*: 对于编码器E, 断言R, 三个参数p, d和l, $E \vDash PC(p,d,l,R)$成立当i_n可以被o_{n+d-l}^{n+d-1}唯一决定, 且R覆盖所有的配置字符c. 这等价于公式*(2)*中$F_{PC}(p,d,l,R)$的不可满足. 我们进一步定义$E \vDash PC(R)$为$\exists p,d,l : E \vDash PC(p,d,l,R)$.

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

公式(2)的行2和3对应于E中的两条路径. 行4强制这两条路径的输出字符序列相等, 行5强制他们的输入字符不等. 最后一行强制c被R覆盖.

基于检查$E \vDash PC(R)$[1, 2]的直接遍历p, d的所有组合l, 从小到大, 直到$F_{PC}(p,d,l,R)$成为不可满足, 这意味着解码器存在.

然而, 如果解码器不存在, 该算法将不会停机. 为了发现一个停机算法, Shen et al.[3]定义了如何检验算法的不存在.

根据定义2和图3, 解码器存在当存在三个参数p, d和l使得$E \vDash PC(p,d,l,R)$成立. 因此, 直观的, 解码器不存在当对任意p, d和l, 我们总能找到另一个p', d'和l'其中$p' > p$,$l' > l$ and $d' > d$, 使得$E \vDash PC(p',d',l',R)$不成立.

这一情形可以被图4中的SAT实例检测, 这与图3非常类似, 除了三个用于检测s_{n-p}^{n+d-l}, $s_{n+d-l+1}^{n}$和s_{n+1}^{n+d}上的环的约束. 如果该SAT实例可满足, 对于任意p, d和l, 我们可以展开这三个环直到找打p' ,d'和l'大于p, d和l. 很明显这个展开的SAT实例仍然是可满足的, 这意味着$E \vDash PC(p',d',l',R)$不成立. 因此解码器不存在. 所以, 解码器不存在可以通过以下定义检验:

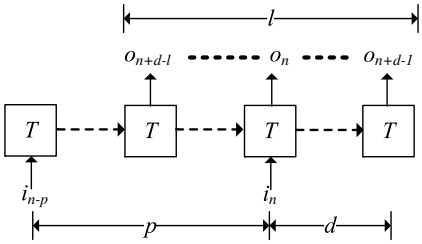

图 3 参数对偶条件

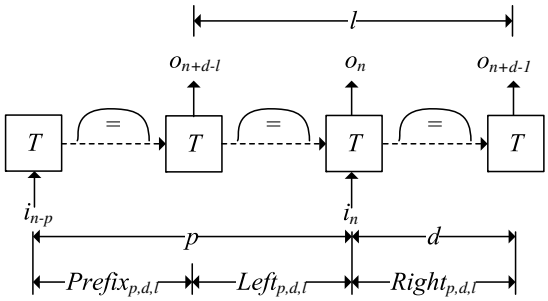

图 4 环形非对偶条件

Definition 3. 环形非对偶条件***(LN):***对于编码器E及其*Mealy*有限状态机$M = (S, s_0, I, C, O, T)$, $E \vDash LN(p, d, l, R)$成立当i_n在s_{n-p}^{n+d-1}上不能被o_{n+d-l}^{n+d-1}唯一决定, 而且在s_{n-p}^{n+d-l}, $s_{n+d-l+1}^{n}$和s_{n+1}^{n+d}上存在环. 这等价于公式*(3)*上的$F_{LN}(p, d, l, R)$的可满足. 我们进一步定义$E \vDash LN(R)$为$\exists p, d, l : E \vDash LN(p, d, l, R)$.

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

公式(2)和(3)的区别在于公式(3)的最后三行, 他们将被用于检测环.

由Shen et al. [3]可知, 我们有以下定理:

Theorem 2. $E \vDash LN \leftrightarrow \neg\{E \vDash PC\}$

因此, 该停机算法[3]简单的遍历p, d and l, 并在每一个循环中检验$E \vDash PC(p, d, l, R)$和$E \vDash LN(p, d, l, R)$. 由定理2可知, 该算法最终总能停机并给出$E \vDash PC(R)$和$E \vDash LN(R)$之间唯一的答案.

3　推导断言

3.1　总体的算法框架

该算法的整体框架如下所示:

Algorithm 1 *InferAssertion*

1: $NA = \{\}$
2: **for** $x = 0 \rightarrow \infty$ **do**
3: $< p, d, l >=< 2x, x, 2x >$
4: **if** $F_{PC}(p, d, l, \bigwedge_{na \in NA} \neg na)$不可满足 **then**
5: **if** $\bigwedge_{na \in NA} \neg na$可满足 **then**
6: 解码器存在且最终断言是$\bigwedge_{na \in NA} \neg na$
7: **else**
8: 解码器不存在
9: **end if**
10: 停机
11: **else**
12: **while** $F_{LN}(p, d, l, \bigwedge_{na \in NA} \neg na)$可满足 **do**
13: 令A为满足赋值, 且$A(c)$为导致解码器不存在的
14: $na \leftarrow InferCoveringFormula(A(c))$
15: $NA \leftarrow NA \cup \{na\}$
16: **end while**
17: **end if**
18: **end for**

在该算法的行1, NA被用于记录所有导致解码器不存在的配置字符公式. 他们是由行14的函数$InferCoveringFormul$导的, 该函数的功能是推导一个不仅能够覆盖$A(c)$, 而且能够覆盖大量其他非法配置字符的公式. 关于该函数的更多细节见下一小节.

行3确保s_{n-p}^{n+d-l}, $s_{n+d-l+1}^{n}$和s_{n+1}^{n+d}的长度全部设置为x, 而其值在行2中被遍历. 如此, 许多p,d和l的冗余组合不必再次被测试.

行4意味着在断言$\bigwedge_{na \in NA} \neg na$下，输入字符能够被输出字符序列唯一圈定. 行5意味着至少存在一个配置字符能够导致解码器存在, 而最终断言是$\bigwedge_{na \in NA} \neg na$.

行7意味着最终断言$\bigwedge_{na \in NA} \neg na$已经剔除了所有配置字符, 即不存在能够导致解码器存在的配置字符. 编码器中必然存在某些错误.

行12意味着行13中的$A(c)$是一个能够导致解码器不存在配置字符. 行14中的函数$InferCoveringFormula$将被用于推导一个公式na不仅覆盖$A(c)$, 而且覆盖大量其他的非法配置字符. 这些配置字符将在行15中被剔除.

我们能够证明算法1是停机的.

Theorem 3. *算法1是停机算法.*

Proof. 由定理2可知, 算法1最终将到达行4或12.

对于前一种情形, 该算法将在行10停机.

对于后一种情形, 一个新的公式na将被推导出来, 他将覆盖$A(c)$. 因为此类$A(c)$是有限的, 他们最终总能够被$\bigwedge_{na \in NA} \neg na$剔除. 因此算法1最终必将到达行4, 并在行10停机. □

3.2　推导覆盖非法配置字符的公式

本小节将介绍算法1行14中的函数$InferCoveringFormula$的实现. 他将被用于推导一个公式na 不仅覆盖$A(c)$, 而且覆盖其他非法配置字符. 该工作将在以下三步中完成:

首先, 将F_{LN}转换为带有目标变量的等价形式, 以便他可以用来定义一个函数f. **其次**, 使用cofactoring [4]将f的支撑集进行削减, 直至只有c剩下. **最后**, 使用Craig插值特征话f. 该f同时也是算法1行14中的na.

这三步将在下面一一描述:

3.2.1　将F_{LN}转换为带有目标变量的等价形式

如上所述, 我们需要将F_{LN}转换为带有目标变量的等价形式.

首先, 我们需要将公式(3)的第4行和最后三行移到一个新的公式中:

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

于是, F_{LN}可以被转换为:

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

很明显F_{LN}和$F'_{LN} \wedge t \equiv 1$是等价的.

同时, 由图4可知, F'_{LN}实际上定义了一个函数$f' : S^2 \times I^{(d+p)*2} \times C \to \mathbb{B}$, 其支撑集为$supp(f')$ is $\{s_{n-p}, s'_{n-p}, i_{n-p}^{n+d-1}, (i')_{n-p}^{n+d-1}, c\}$, 其输出即为公式(5)最后一行的$t$.

3.2.2　使用Cofactoring削减f的支撑集

由算法1的行12, F_{LN}是可满足的, 而且A是其满足赋值. 我们可以将i_{n-p}^{n+d-1}, $(i')_{n-p}^{n+d-1}$, s_{n-p}的$(s')_{n-p}$的值赋予F'_{LN}, 而得到:

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

现在, $F"_{LN}$定义了另一个函数$f"$, 其支撑集被削减为c. 很明显$F"_{LN}(c,t) \wedge t \equiv 1$就已经是一个覆盖非法配置字符的公式, 但是他仍然是一个复杂的CNF公式. 为了削减其尺寸, 我们需要下一小节的基于Craig插值的特征化算法.

3.2.3 使用Craig插值特征化f

我们首先将$F"_{LN}(c,t)$编码为CNF格式, 并记为$CNF(F"_{LN}(c,t))$. 假设$CNF'(F"_{LN}(c,t'))$是$CNF(F"_{LN}(c,t))$的一个拷贝. 他们之间仅仅共享c的编码, 而其他所有变量均独立编码. 因此, 我们可以构造如下所示的ϕ_A和ϕ_B:

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

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

很明显, $\phi_A \wedge \phi_B$是不可满足的. 基于McMillan的插值算法[10], 我们可以生成一个插值函数$ITP : C \to \mathbb{B}$.

由定理1, ITP与公式(8)中的ϕ_B是不兼容的. 因此他特征化了一个能使公式(7)中的ϕ_A可满足的配置字符集合$C' \subseteq C$. 由公式(5)和(6), 很明显:

1. 算法1行13中的$A(c)$被包含于C'. 由定理3, 这确保了算法1是停机的.
2. 所有$c' \in C'$也能导致解码器不存在. 这将极大的提升算法1的速度.

4 剔除冗余

被算法1发掘的p, d和l包含一些冗余, 这将导致在电路面积和特征化时间上存在不必要的开销. 因此, 算法2被用于在将p, d和l传给下一个算法之前削减他们.

该算法迭代的削减p, d和l, 并测试他们是否仍然能够使得$E \vDash PC(R)$成立. 我们在这里将不再进一步给出细节.

5 发掘多个解码器的布尔关系

小节5.1介绍了如何发掘多个解码器的布尔关系, 而小节5.2介绍了算法实现. 小节5.4如何得到每个解码器的前提条件, 以便用户选择正确的解码器.

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

for $p' = p \to 0$ **do**
  if $F_{PC}(p'-1, d, l, R)$可满足 **then**
    break
  end if
end for
for $d' = d \to 0$ **do**
  if $F_{PC}(p', d'-1, l, R)$可满足 **then**
    break
  end if
end for
for $l' = 1 \to l-(d-d')$ **do**
  if $F_{PC}(p', d', l', R)$不可满足 **then**
    break
  end if
end for
print "最终结果是$< p', d', l' >$"

5.1 构造SAT实例以发掘解码器

假设被算法1发掘的断言为:

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

同时, 我们也使用IA表示被起覆盖的配置字符集合. 因此IA的具体含义依赖于上下文语境. 我们进一步假设被算法2发掘的参数组为$< p, d, l >$, 而从o_{n+d-l}^{n+d-1}和c唯一决定i_n的布尔关系为$F_{PC}(p, d, l, IA)$. 为了简化表达我们重新记i_n和o_{n+d-l}^{n+d-1}为:

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

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

因此, 从o_{n+d-l}^{n+d-1}和c唯一决定i_n的布尔关系可以重写为:

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

假设f是被R定义的函数, 而该函数从X和c计算Y:

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

同时, 对于每个配置字符$c_i \in IA$, 由一个对应的R_{c_i}:

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

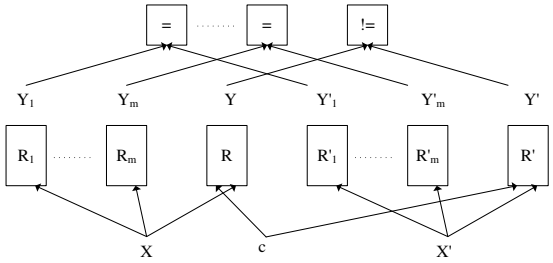

图 5 The SAT instance that discovers decoders

而两个不同的c_i和c_j可能分享同一个R_{c_i}, 即, $R_{c_i} \equiv R_{c_j}$. 因此IA可以被划分为$\{IA_1, \ldots, IA_n\}$, 使得:

1. 同一个IA_i中的所有c分享同一个R_i.
2. 在两个不同的IA_i和$IA_{i'}$中的c和c'不共享同一个R_i.

对于集合$\{IA_1, \ldots, IA_n\}$, 假设被所有$c \in IA_i$共享的布尔关系为R_i, 而R_i定义的函数是f_i, 则f可以被重写为:

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

因此, 在这里我们的工作是一步步的发掘$\{R_1, \ldots, R_n\}$. 假设我们已经发掘了一部分解码器的布尔关系$\{R_1, \ldots, R_m\}$. 为了测试他是否包含$\{R_1, \ldots, R_n\}$, 即, 是否所有的解码器均已经被发掘, 我们构造如下SAT实例, 如图5所示:

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

公式(16)行1表示$\{R_1, \ldots, R_m\}$和R中的布尔关系. 行2是行1的拷贝. 他们之间共享的唯一变量是c. 行3强制所有的Y_i和Y'_i相等, 而最后一行强制Y和Y'不相等.

下面的定理表明, 如果公式(16)不可满足, 则所有解码器已经被发掘.

Theorem 4. *如果公式(16)是不可满足的, 则$\{R_1, \ldots, R_m\}$包含$\{R_1, \ldots, R_n\}$.*

Proof. 使用反证法. 假设$R_n \notin \{R_1, \ldots, R_m\}$, 而$IA_n$是其对应的配置字符集合, 而且$c_n \in IA_n$.

我们可以构造一个满足赋值A使得$A(c) \equiv c_n$. 因此我们有$\{R(c, X, Y) \wedge A(c) \equiv c_n\} \equiv R_n(X, Y)$, 即我们用$A$将图5中的$R$和$R'$转变为$R_n$.

因为$R_n \notin \{R_1, \ldots, R_m\}$, 必然存在$A'$, 使得当我们将$A'(X)$赋予$X$而$A'(X')$赋予$X'$, 我们能够使$\bigwedge_{i=1}^{m} Y_i \equiv Y'_i$和$Y \neq Y'$都成立.

通过组合A和A', 公式(16)变成可满足的. 该矛盾导致定理得证. □

另一方面, 如果公式(16)可满足, 我们需要证明:

Theorem 5. 如果*(16)*可满足, 则必然存在一个尚未被发掘的解码器.

Proof. 使用反证法. 假设所有解码器均已经被发掘, 即, $\{R_1,\ldots,R_m\}$包含$\{R_1,\ldots,R_n\}$.

这意味着f可以被重写为:

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

因此, 对于任意能使的公式(16)前三行满足的A, 函数f可以被重写为:

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

因此, 公式(16)的最后一行将不能满足. 因此公式(16)是不可满足的. 这个矛盾导致定理得证. □

由A, 下面的R_{m+1}是一个新发掘的解码器.

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

为了证明我们的算法没有重复冗余的工作, 我们需要证明R_{m+1}以前没有被发掘过:

Theorem 6. $R_{m+1}\notin\{R_1,\ldots,R_m\}$

Proof. 使用反证法. 假设存在$0\leqslant i\leqslant m$使得$R_i\equiv R_{m+1}$, 且存在$c'\in IA_i$.

因为R_i可以从X唯一决定Y, 而R可以从X和c唯一决定Y, 而$R_{m+1}\equiv R_i$, 很明显成立通过强制c等于$c'\in IA_i$我们可以使得$Y\equiv Y_i$.

同样的, 我们有$Y_i'\equiv Y'$.

从公式(16)的行5, 我们有$Y\equiv Y_i\equiv Y_i'\equiv Y'$. 这与公式(16)最后一行的$Y\neq Y'$矛盾. 该矛盾导致定理得证. □

5.2 算法实现

基于上述讨论, 算法3描述了如何发掘多个解码器的布尔关系.

Algorithm 3 *DiscoveringDecoders*

1: **while** 公式(16)可满足 **do**
2: 假设A是满足赋值
3: 将公式(19)的R_{m+1}插入$\{R_1,\ldots,R_m\}$
4: **end while**
5: 发掘的布尔关系集合为$\{R_1,\ldots,R_m\}$

行1意味着$\{R_1,\ldots,R_m\}$并不包含所有的解码器, 还有某些解码器没有被发掘.

从表达式(16)中返回的A, 行3的R_{m+1}是公式(19)中发掘的新解码器. 他将被插入$\{R_1,\ldots,R_m\}$以在行1中再次进行测试.

算法3的循环单调的增加$\{R_1,\ldots,R_m\}$的尺寸. 因为这样的解码器的个数是有限的, 这个循环, 和算法3最终总能够停机.

5.3 特征化发掘的解码器的布尔函数

从算法3发掘的$\{R_1,\ldots,R_m\}$, Shen et al. [2]提出的ALLSAT算法将被用来特征化他们的布尔函数. 其细节将不在这里给出.

对于那些对解码器面积和时许感兴趣的读者, 请阅读Shen et al. [3]的小节V.B和V.C.

5.4 特征化$\{IA_1,\ldots,IA_m\}$

假设$\{R_1,\ldots,R_m\}$是被算法3发掘的解码器全集, 而$\{IA_1,\ldots,IA_m\}$是他们对应的配置字符集合. 为了帮助用户从$\{R_1,\ldots,R_m\}$选择正确的解码器, 我们需要特征化$\{IA_1,\ldots,IA_m\}$里的每一个IA_i.

由图5, Y和所有的Y_i之间的关系可以表述为:

$$Y = \bigvee_{i=1}^{m}\{IA_i(c) \wedge Y_i\} \tag{20}$$

假设Y和所有Y_i是等长的，长度均为v:

$$\begin{aligned} Y &=< y^0,\ldots,y^{v-1} > \\ Y' &=< y'^0,\ldots,y'^{v-1} > \\ Y_i &=< y_i^0,\ldots,y_i^{v-1} > \\ Y_i' &=< y_i'^0,\ldots,y_i'^{v-1} > \end{aligned} \tag{21}$$

我们可以将公式(20)按照bit重写, 而Y和Y_i的第$j$bit可以被重写为:

$$y^j = \bigvee_{i=1}^{m}\{IA_i^j(c) \wedge y_i^j\} \tag{22}$$

由小节2.1, 很明显公式(22)是一个经典的函数依赖问题. 我们可以使用Lee et al. [17]提出的函数依赖求解算法特征化$IA_i^j(c)$，首先构造如下两个公式:

$$\phi_A \stackrel{def}{=} \left\{ \begin{array}{cc} & R(c,X,Y) \wedge \bigwedge_{i=1}^{m} R_i(X,Y_i) \\ \wedge & y^j \equiv 1 \end{array} \right\} \tag{23}$$

$$\phi_B \stackrel{def}{=} \left\{ \begin{array}{cc} & R'(c,X',Y') \wedge \bigwedge_{i=1}^{m} R_i'(X',Y_i') \\ \wedge & \bigwedge_{i=1}^{m} y_i^j \equiv y_i'^j \\ \wedge & y'^j \equiv 0 \end{array} \right\} \tag{24}$$

很明显$\phi_A \wedge \phi_B$非常类似于公式(16), 除了第j位被约束为相同, 而y^j和y'^j被约束为不同的常数.

表 1 Benchmarks的相关信息

	XGXS	XFI	scrambler	PCIE	T2 ethernet
Verilog 代码行数	214	466	24	1139	1073
寄存器个数	15	135	58	22	48
数据路径宽度	8	64	66	10	10

因此$\phi_A \wedge \phi_B$是不可满足的, 我们可以从其不可满足证明产生一个Craig插值$ITP : C \times \mathbb{B}^m \to \mathbb{B}$, 其支撑集为$\{c, y_1^j, \ldots, y_m^j\}$. 由公式(22), ITP是$\bigvee_{i=1}^m \{IA_i^j(c) \wedge y_i^j\}$的上估计. 因此, 通过设置$y_i^j$为1而其他$y_k^j$为0，即可得到$IA_i^j(c)$的上估计:

$$\left\{ \begin{array}{ll} & ITP \\ \wedge & \bigwedge_{k \neq i} y_k^j \equiv 0 \\ \wedge & y_i^j \equiv 1 \end{array} \right\} \tag{25}$$

因为$\phi_A \wedge \phi_B$不可满足, 这个$IA_i^j(c)$的上估计也可以使得$y^j \equiv 1$. 我们可以直接将其视为$IA_i^j(c)$. 因此, $IA_i(c)$可以定义为:

$$IA_i(c) = \bigwedge_{j=0}^{v-1} IA_i^j(c) \tag{26}$$

在小节中6.4, 我们可以看到, 通过检查这些IA_i, 用户可以非常容易的选择正确的解码器.

6 实验结果

我们实现了该算法并使用Minisat求解其[11]求解. 所有实验在一台具有一个2.4GHz Intel Core 2 Q6600处理器, 8GB内存和Ubuntu 10.04 linux操作系统的PC上进行. 所有的程序和试验结果均能从`http://www.ssypub.org/exp/compsyn_fmcad11.tgz`下载.

6.1 Benchmarks

图1给出了以下Benchmark的信息.

1. 一个满足IEEE-802.3ae 2002标准[7]短句48的XGXS编码器.

2. 一个满足同一个标准的短句49的XFI编码器.

3. 一个66位的加扰器，用于使输入数据具有足够多的0-1翻转，以便能够可靠的在高噪声通讯系统中传输.

表 2 实验结果

		XG-XS	XFI	scra-mbler	PCI-E	T2 e-ther
[3]	检查*PC*的时间(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
本文	配置信号个数	3	120	1	16	26
	运行时间	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
	解码器个数	1	2	2	1	1

4. 一个PCI-E[6]的编码模块.

5. Sun公司的OpenSparc T2处理器的以太网模块.

6.2 推导得到的断言

我们在这里将给出算法1推导的断言.

对于**XGXS**: `!( ( bad_code ) )`

对于**XFI**: `!( ( RESET & TEST_MODE ) | ( !RESET & TEST_MODE ) | ( !RESET & !TEST_MODE & !DATA_VALID ) )`

对于**scrambler**: `True`

对于**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 ) )`

对于**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] ) )`

6.3 与现有工作的比较

表2的行2给出了Shen et al. [3]的算法的运行时间. 而行3给出了他发现的d, p和l. 所有这些benchmark均具有正确的断言.

相反, 我们移除了所有断言并运行我们的算法. 行4给出了benchmark中配置信号的个数, 行5给出了本文算法的运行时间, 而行6给出了发掘的d, p和l, 而最后一行给出了解码器的个数

通过比较行2和5, 很明显本文算法大大慢于[3], 这是由更为复杂的$InferCoveringFormula$和$DiscoveringDecoders$造成的.

然而, 通过比较行4和5, 虽然XFI和T2 Ethernet分别有120和26个配置信号, 他们的运行时间并不是特别长. 这是由节3中的高效特征化算法导致的.

通过比较行3和6, 很明显在发掘的参数值上有一些细微的差别. 这是由内嵌的断言和推导的断言之间的差异造成的. 本文算法推导的断言比内嵌断言包含多得多的情形.

6.4 处理多个解码器

由表2的最后一行, 在五个benchmark中只有两个具有两个解码器, 而其他三个均只有一个解码器. 这意味着, 在大多数情况下, 我们的算法只产生一个解码器. 对于其他具有多个解码器的情形, 用户需要自己查看节5.4推导的$\{IA_1, \ldots, IA_m\}$.

对于加扰器的两个解码器, 他们对应的IA_1是$False$, 而IA_2是$!reset$. 因此第二个解码器是唯一可能的解码器. 动态模拟也确认了我们的选择.

对于XFI的两个解码器, 他们对应的IA_1是$RESET$ & $!TEST\_MODE$, 而IA_2是$!RESET$ & $!TEST\_MODE$ & $DATA\_VAL$ 他们之间的根本差别在于$RESET$. 通过检查XFI的Verilog源代码, 我们注意到在$RESET$为$True$是将复位整个编码器. 因此XFI编码器只有在$RESET$为$False$时才正常工作. 因此第二个解码器是正确的解码器. 动态模拟确认了我们的选择.

7 相关工作

7.1 程序取反

由Gulwani[13], 程序取反是针对特定程序P自动生成具有相反功能的程序P^{-1}. 因此程序取反和对偶综合非常相似.

最早的程序取反算法使用基于证明的方法[14], 但是该方法只能处理非常简单的语法和非常小的程序.

Glück et al. [15]使用基于LR的方法去除一阶程序中的不确定性. 然而使用函数语言导致该方法与我们的应用背景不兼容.

Srivastava et al. [16] 迭代的剔除程序中不能满足取反要求的路径, 直到只剩下一个可能的程序为止. 因此该方法只能保证存在一个解, 但是并不保证其正确性.

7.2 协议转换综合

协议转换综合问题是针对两个不兼容的通讯协议，自动生成转换逻辑. 这与本文工作相关因为他们都专注于通讯电路的综合.

Avnit et al. [18]首先提出描述通讯协议的通用模型. 而后他们提出一个算法以检验是否存在某些功能在一个协议中能够实现而在另一个协议中不能实现. 最后, 通过为缓冲区的控制函数计算一个最大不动点从而生成转换电路. Avnit et al.[19]通过提出更为有效的空间探索算法以提高整体算法效率.

8 Conclusions

本文提出一个完全自动化的算法以为特定编码器推导断言并生成所有合法的解码器. 试验结果表明本文算法能够处理许多来自于工业界的复杂编码器, 包括PCI-E[6]和以太网[7]. 进一步, 基于每个解码器的前提条件, 用户可以非常容易的选择正确的解码器.

Acknowledgment

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

参考文献

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

2 S. Shen, Y. Qin, K. Wang, L. Xiao, J. Zhang, and S. Li, "Synthesizing complementary circuits automatically," *IEEE transaction on CAD of Integrated Circuits and Systems*, vol. 29, no. 8, pp. 1191–1202, Aug. 2010.

3 S. Shen, Y. Qin, L. Xiao, K. Wang, J. Zhang, and S. Li, "A halting algorithm to determine the existence of the decoder," *accepted by IEEE transaction on CAD of Integrated Circuits and Systems. http://www.ssypub.org/tcad11.pdf*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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