2022 集成电路 EDA 设计精英挑战赛

作品设计报告

赛题三：高位宽运算电路的逻辑等价性验证

参赛队 ID:EDA220322

目 录

一、问题重述

数字电路中的逻辑等价性验证（Logic Equivalence Checking, LEC），是一种典型的形式化验证方法。其主要作用是验证两个 design 是否等价，可以在两个 RTL 之间或者两个 netlist 之间，也可以在 RTL 和 netlist 之间。LEC 在整个 EDA 流程中，是高频调用的模块。在当前的学术研究中，最主流的方法是将待验证 miter（将待验证的两个 design 通过 XOR 连接构成的组合电路）转成 SAT（布尔可满足性）表达式，而后直接调用 SAT 求解器求解。除此以外，也有通过 BDD（binary decision diagram，二元决策图）等方法来完成求解的学术研究。

在本次比赛中，LEC 工具的 input 是 miter 电路的 AIG（只有 AND gate 和 NOT gate，其文件后缀是 aig），输出是 UNSAT（证明两个 design 等价）或者找到的解（证明两个 design 不等价）。

二、问题分析

一般来讲，LEC 的求解有以下几种思路：

① 将 AIG 文件转换为 CNF 范式从而直接调用 SAT 求解器求解；

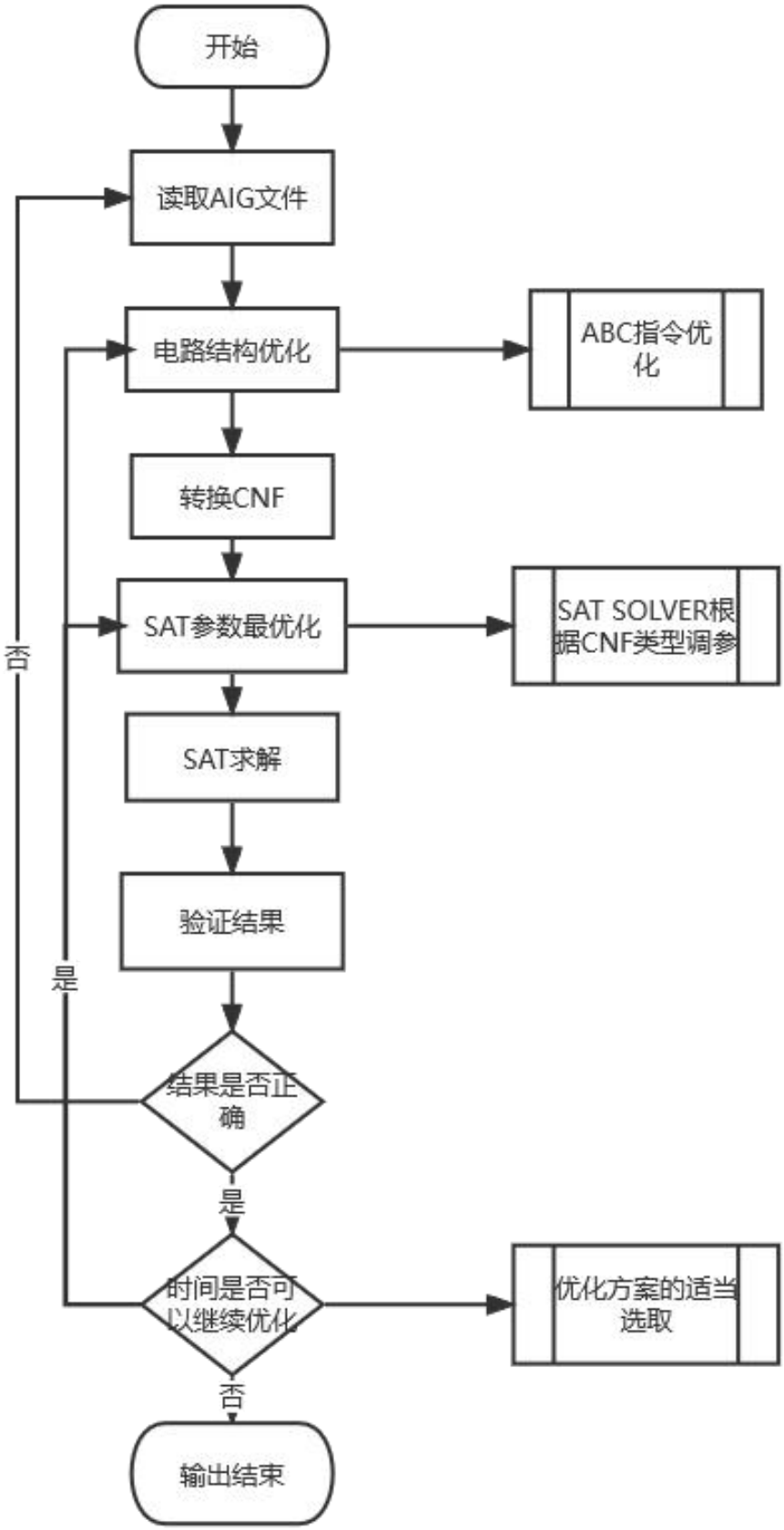
② 在 AIG 上直接求解，或者转成 BDD 等其他格式；

③ 采用定理证明等高阶方法完成证明。

通过理论分析与实际对比，我们采用了将 AIG 文件转换为 CNF 范式从而直接调用 SAT 求解器求解作为我们解决问题的方向。

三、算法思路

我们采用了 AIG 转 CNF 采用 SAT SOLVER 的方式进行求解。因此需要：①AIG 转 CNF，②求解 CNF，流程如下图所示：


开始
读取AIG文件
电路结构优化
ABC指令优化
转换CNF
SAT参数最优化
SAT SOLVER根据CNF类型调参
SAT求解
验证结果
结果是否正确
否
是
时间是否可以继续优化
是
否
优化方案的适当选取
输出结束

四、算法实现

1.AIG 转 CNF 的实现

ABC 是一个不断发展的软件系统，用于合成和验证出现在同步硬件设计中的二进制时序逻辑电路。ABC 结合了基于 And-Inverter Graphs（AIGs）的可扩展逻辑优化、基于最优延迟 DAG 的查找表和标准单元的技术映射，以及用于顺序综合和验证的创新算法。ABC 提供了这些算法的实验实现，并为构建类似的应用程序提供了一个编程环境。我们使用这样的编程环境来对 aig 文件进行了转换，期望转换为一个能够高速求解的 cnf 文件。

① resyn 转换

resyn 转换能够将当前的 aig 网络转换为一个平衡性良好的 aig 网络文件。resyn 共有以下几个参数，如下表所示

序号	参数	作用
1	l	切换最小化级别数[默认值=是]
2	d	切换逻辑重复[默认=否]
3	s	在关键路径上切换重复[默认值=否]
4	x	切换平衡多输入 EXOR[默认值=否]

经讨论与实践检验，我们采用了-l 参数作为我们的转换方案，即

resyn -l 文件名.aig

② choice 转换

通过破坏 AIG 数据库来创建当前网络。

序号	参数	作用
1	R num	随机模式数（127<num<32769）[默认值=取决于设计]
2	D num	系统模式数（127<num<32769）[默认值=设计相关]
3	C num	一个 SAT 问题的回溯次数[默认值=1000]

经讨论与实践检验，我们采用了默认的参数作为我们的转换方案，即

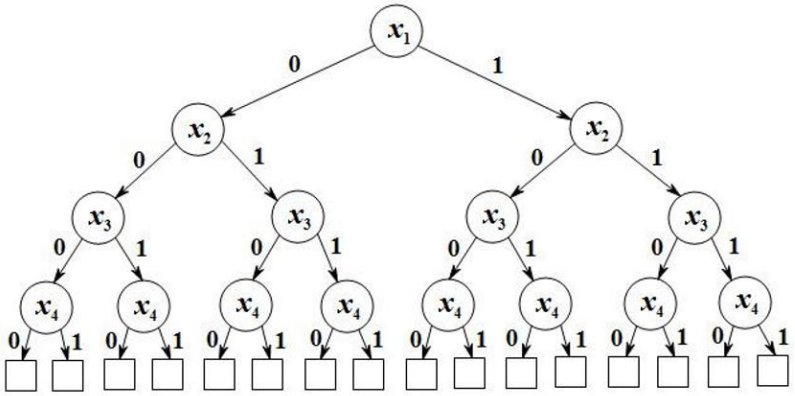
choice 文件名.aig

综上所述，我们采用二者结合的方式来改进，执行转换脚本的具体内容如下

/home/eda220322/abc-try/abc-master/abc -q "read_aiger $1; write_cnf t.cnf; quit"

2.SAT SOLVER 的实现

在 DPLL 算法中，每一个文字 literal 的赋值都可以看作在当前树节点，选择遍历它的左子树（真值为 0，向左），或者右子树（真值为 1，向右）；故而 DPLL 的求解过程，就可以看作是从一棵树的根出发，然后通过选择不同变元赋值情况到达最底部：二叉树中，从树根到叶子节点的一条路径就表示子句集中的一组变量赋值序列，示意图如下：



为了找到这个路径，采取如下方法：根据某个条件先确定树根，接着从树根出发：往前进行试探搜索，该试探搜索遵循先访问左子结点，再访问右子结点的顺序，如果在搜索过程中发生冲突（即遇到不可满足的结点），则利用回溯策略选择其他的路径继续进行搜索，直至找出一条满足所有条件的路径，即找到使 SAT 问题可满足的解，或搜索完均不满足所有条件的全部路径，即原 SAT 问题不可满足。上述过程的伪码如下图所示：

```
输入：子句集 S
输出：S 的可满足性
1:   procedure DPLL(S)
2:       decisionlevel ← 0
3:       repeat
4:           status ← BooleanConstraintPropagation( )
5:           if status == CONFLICT then
6:               if decisionlevel == 0 then return UNSAT
7:               else blevel ← AnalyzeConflict( )
8:                    decisionlevel ← blevel
9:                    BacktrackTo(blevel)
10:              end if
11:          else
12:              if all variables are assigned then return SAT
```

```
13:             else PickBranchingVariable( )
14:                 decisionlevel ← decisionlevel+1
15:             end if
16:         end if
17:  end procedure
```

其中涉及到 3 个算法关键点：**确定树根，回溯策略，变元寻找策略**（寻找其他路径）；在简单 DPLL 算法中，这三者要点都采用最简单的形式实现：

（1） 确定树根

以 CNF 变元表中的第一个 literal 为树根开始尝试赋值；

（2） 回溯策略

直接回溯，即在二叉树中返回上一层，并且按舒徐选取另一条路径（如果之前是向左走，那么现在向右走；如果之前向右走，说明当前层不成立，继续回溯至上一层）；

（3） 变元寻找策略

在 CNF 变元表中按照序号从小到大依次选取。

相比于 DPLL，CDCL 主要从**确定树根，回溯策略，变元寻找策略**等三个关键方面进行了优化：

（1） 确定树根

采用统计出现频率的方法确定树根（出发点）；

（2） 回溯策略

相较于 DPLL 的时许回溯，CDCL 可以通过“**学习**”，来完成非时序性回溯，即遇到问题是可能可以回溯不止一层；

学习的简要原理（此处根据程序写出必要的算法思想，涉及到的理论知识请见报告的 3.4.3 部分 CDCL 涉及到理论的部分简介）。小节最后的伪码即是把部分遍历的解决计算结果生成了新的 clause 又插入 CNF 中作为了新的子句；

（3） 变元寻找策略

在 DPLL 优化后的策略（定时更新，根据频率选则出现次数最高的变元进行赋值）基础

上，在 CDCL 这种优化策略已经不再起作用；

原因在于，“学习”到的结果是作为新的子句加入到 CNF 表中的，所以有两方面的影响：①有些关键变元排序，会由于学习结果“新子句”的加入而有改变的可能，如果每次学习之后都重新进行排序则开销过大；②“学习”结果在作为“新子句”的时候，所占的各种权重与初始的各个 clause 与 literal 是相同的，但是实际上，由于学习结果是由于程序根绝部分基础数据计算出来的“析取范式”，因此学习到的“clause”更具有效力，在促使程序向着正确方向进行求解的“作用力”更大；

基于①②，想要让不同的 literal 的频率的占比随着求解过程的进行，逐渐的衰减，这样最新加入的“学习结果”中的子句就自然占据了更高的比重；根据相关论文的查找结果，初步设置了衰减系数；

```
输入：子句集 S
输出：S 的可满足性
1:   procedure CDCL(S)
2:       decisionlevel ← 0
3:       repeat
4:           status ← BooleanConstraintPropagation( )
5:           if status == CONFLICT then
6:               if decisionlevel == 0 then return UNSAT
7:               else (C, blevel) ← AnalyzeConflict( )
8:                    S ← S ∪ C
9:                    decisionlevel ← blevel
10:                   BacktrackTo(blevel)

11:              end if
17:          end if
18:  end procedure
```

五、算法性能分析与结果检验

1. 结果检验

aig 优化前后的检验

使用 abc 自带的检验命令检验即可，根据 abc 的相关文档，我们采用 abc 的检验方式

为 cec file1.aig file2.aig。

cnf 求解的检验

通过程序读入变元的值，再代入每一个子句中检查，若每个子句均无误，则正确，反之则求解错误。部分核心代码如下图所示，

```
int main()
{
    int num_vary,num_sentence,*num1=&num_vary,*num2=&num_sentence;
    int flag[500000];
    char s[20];//验证的文件名
    printf("请输入读取的文件名，输入正确后会输出各个子句：");
    scanf("%s",s);
    FORM ziju;
    ziju=cnf_read(s,num1,num2);
    cnftraverse(ziju);
    printf("\n输入各变量的真值：");
    for(int i=1;i<=num_vary;i++)
    {
        int temp;
        scanf("%d",&temp);
        if(temp<0) flag[i]=-1;
        else flag[i]=1;
    }
    FORM head=ziju->next;
    for(int i=1;i<=num_sentence;i++)
    {
        LETTER *prop;
        int sympol=0,cur;
        prop=head->headp->next;
        while(prop){
            int k=prop->prop;
            if(k<0){
                k*=-1;
                if(flag[k]==-1){sympol=1;}
            }
            else{
                if(flag[k]==1){sympol=1;}
            }
            if(sympol)
            {
                cur=k;
                break;
            }
            prop=prop->next;
        }
        if(!sympol){
            printf("第%d子句有问题\n",i);//输出哪个子句有问题
            break;
        }
        head=head->next;
    }
    printf("Test End!\n");
    return 0;
}
```

2. 性能对比

在完成结果正确性检验后，我们对求解 aig 在时间上的优化进行了比较。选取了市面上较优的开源软件对 cnf 文件进行求解，并计算得到了他们的优化比例。如下表所示，我们的求解器（实验组）相较于市面上广泛使用的求解器（对照组）在求解效率上有较大的提升，在求解时间上最高可以优化 98.2%（test2）。由此可见，我们在电路逻辑等价性验证这一问题上有较大的优化效果。

样例	对照组			实验组	优化比例		
	kissat	cadical	nflsat	EDA220322	相对kissat	相对cadical	相对nflsat
easy1	0.04	0.04	0.05	0.04	0.0%	0.0%	18.5%
easy2	0.48	0.28	0.37	0.28	40.5%	0.0%	22.8%
middle1	0.10	0.09	0.23	0.09	5.2%	-4.6%	59.6%
middle2	3.20	2.27	6.96	2.23	30.2%	1.5%	67.9%
middle3	17.58	18.72	21.92	18.01	-2.4%	3.8%	17.8%
middle_new_1	57.00	45.42	155.75	39.58	30.6%	12.8%	74.6%
middle_new_2	92.87	83.14	220.67	64.82	30.2%	22.0%	70.6%
middle_new_3	94.25	85.78	221.53	64.41	31.7%	24.9%	70.9%
test1	11.87	6.64	19.89	6.79	42.8%	-2.4%	65.8%
test2	1.58	1.66	97.59	1.80	-14.4%	-8.9%	98.2%
总时间	278.96	244.03	744.96	198.06	29.0%	18.8%	73.4%

注：优化比例>0 为时间减少的比例，即正优化；优化比例<0 为负优化

六、主要创新点

1. 对 aig 转 cnf 进行了创新，不是简单地利用 abc 转换为 cnf，而是先转化为了一个平衡性更加良好的 aig，然后再进行后续的转换，详见 4.1.1resyn 转换一节所示。

2. 在 aig 转 cnf 过程中进行的另一个创新是通过随机或 sat 流程破坏 AIG 数据库来创建新的网络，由此减少子句数与变元数，加快 sat 的求解，详见 4.1.2choice 转换一节所示。

3. 由表格可知，cadical 通过子句学习与预测机制，在各个规模的 aig 文件转为的 cnf 文件的 SAT 求解过程都有比较优秀的求解性能。值得注意的是，cadical 求解器与 abc 工具的预处理工具结合起来之后还可以在现有基础上得到进一步的优化性能。

七、困难与解决方式

1. AIG 文件转换后信息冗余

问题：对 aig 文件不够了解，在 aig 转 cnf 的过程中直接采用 ABC 的简单 read 和 write 指令进行 cnf 的生成，使得生成的 cnf 信息有很大的冗余。在用 cadical 求解的过程中，可以从时间以及输出参数得到某些算例的结构特征还没有得到充分利用的挖掘，然后就在转为 cnf 之后求解的构成中被淹没掉了。从而不能得到冲抵的利用。

解决方式：采用特殊文件阅读工具对 AIG 文件进行详细的解析，并发现 ABC 的一些指令可以对 AIG 文件进行优化，因此可以利用 ABC 这一工具对 AIG 进行结构的重构以及特征的挖掘，从而重复利用电路结构特征使得最终的求解速度得到提升。

2. 部分算例求解时间过长

问题：相同的求解器对不同的算例具有不同的适配性，对于几个不同的 SAT 求解器均有此特点并且适配的算例不完全重合。

解决方式：分析算例的适配特点并计算适配度，对算例的求解时间进行控制，在超出控制时间后更换求解器以试图匹配更高适配度的求解器来缩短时间。

3. ABC 的负优化

问题：在一些算例上，对 AIG 文件进行一些 ABC 的指令后，转 CNF 后求解时间反而变长。

解决方式：这是因为一些指令执行的过程中是按照某些特定的规则对 AIG 进行“优化”，因此并没有考虑后续的求解过程所需要的结构特点，因此我们对相同的指令进行了算例试验扩充，根据负优化的存在情对其进行筛选。此外，经过多次实验发现，一些负优化指令在搭配其他的一些指令后会产生意想不到的正优化效果，因此搭配其他指令也可以作为解决该问题的一种方式。

八、参考文献

[1] 第四届（2022）集成电路 EDA 设计精英挑战赛赛题指南

[2] https://github.com/arminbiere/cadical

[3] 范全润,段振华,徐国培.用 AIG 推理检验组合电路的等价性[J].西安电子科技大学学报,2009,36(05):877-884.