计算机科学
COMPUTER SCIENCE

一种基于变量隐藏抽象的IC3硬件验证算法

杨柳, 范洪宇, 李东方, 贺飞

引用本文

杨柳, 范洪宇, 李东方, 贺飞. 一种基于变量隐藏抽象的IC3硬件验证算法[J]. 计算机科学, 2023, 50(11A): 230200112-6.

YANG Liu, FAN Hongyu, LI Dongfang, HE Fei. IC3 Hardware Verification Algorithm Based on Variable Hiding Abstraction [J]. Computer Science, 2023, 50(11A): 230200112-6.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230200112

一种基于变量隐藏抽象的 IC3 硬件验证算法

杨　柳[1,2,3]　范洪宇[1,2,3]　李东方[4]　贺　飞[1,2,3]
1 清华大学软件学院　北京 100084
2 教育部信息系统安全重点实验室　北京 100084
3 北京国家信息科学与技术研究中心　北京 100084
4 北京计算机技术及应用研究所　北京 100854
(529244170@qq.com)

摘　要　随着硬件设计复杂性和规模的大幅度提升,硬件验证工作更加具有挑战性。模型检验技术作为一种自动化验证技术,可以自动构建反例路径,也因此成为硬件验证领域内最重要的研究方向之一。IC3 算法是近些年来最成功的比特级别的硬件验证算法。为了提高验证的规模和效率,硬件验证算法设计逐渐从底层的比特级向更高的抽象级别转变。研究目标是设计一个新型有效的字级 IC3 算法。针对研究目标,提出了一种将变量隐藏抽象和隐式抽象结合的字级 IC3 算法 IC3VA。该方法尝试将变量隐藏抽象和 IC3 算法相结合,并设计了对应的泛化和精化方案。在开源社区和硬件验证大赛收集的测试集上和基于谓词抽象的方法进行对比,实验结果显示了基于变量隐藏抽象的 IC3 算法的有效性。

关键词:硬件验证;IC3 算法;形式化方法;模型检验;变量隐藏抽象
中图法分类号　TP301.2

IC3 Hardware Verification Algorithm Based on Variable Hiding Abstraction

YANG Liu[1,2,3],FAN Hongyu[1,2,3],LI Dongfang[4] and HE Fei[1,2,3]
1 School of Software,Tsinghua University,Beijing 100084,China
2 Key Laboratory for Information System Security,Ministry of Education,Beijing 100084,China
3 Beijing National Research Center for Information Science and Technology,Beijing 100084,China
4 Beijing Institute of Computer Technology and Applications,Beijing 100854,China

Abstract　As the complexity and scale of hardware designs have increased significantly,hardware verification has become more challenging. Model checking techniques,as an automated verification technique,can automatically construct counterexample paths and thus become one of the most important research directions in the field of hardware verification. The IC3 algorithm is the most successful hardware verification algorithm at the bit level in recent years. In order to improve the scale and efficiency of verification,the design of hardware verification algorithms is gradually shifting from the bottom bit level to a higher abstraction level. The research goal is to design a new effective word-level IC3 algorithm. Aimed at this research goal,a word-level IC3 algorithm that speaks of a combination of variable hidden abstraction and implicit abstraction,called IC3VA,is proposed. The approach attempts to combine variable hiding abstraction and IC3 algorithm,and designs a corresponding generalization and refinement scheme. It is compared with the predicate abstraction-based approach on a test set collected by the open-source community and a hardware verification competition. Experimental results show the effectiveness of the IC3 algorithm based on variable hiding abstraction.

Keywords　Hardware verification,IC3 algorithm,Formal method,Model checking,Variable hiding abstraction

1　引言

工业制造能力的提升使得电路的规模和复杂度大幅度增长,因而与之配套的硬件设计技术也发生了翻天覆地的变化。随着电路规模和功能复杂度的增长,设计人员需要在更高的抽象级别看待硬件设计。为了应对这种实际需求,硬件描述语言和工具被设计开发出来。硬件描述语言定义了不同抽象级别的硬件设计表示,并允许用户使用高级别的描述定义硬件设计,且可以自动将其转换为低级别的等效电路表示。因此,设计人员只需要考虑更高层的逻辑和时序设计即可,不用过多考虑底层的电路设计和实现,这无疑大大提升了硬件设计的效率,也成为主流的设计方式。

在安全攸关的领域,如航空航天、铁路运输、高精度仪器等领域保证硬件设计的安全性是一个重要问题。所以自硬件

基金项目:国家重点研发计划课题(2018YFB1308601);国家自然科学基金(62072267,62021002);国防基础科学科研计划(XX2020204B028)
This work was supported by the National Key Research and Development Program of China(2018YFB1308601),National Natural Science Foundation of China(62072267,62021002) and National Defense Basic Scientific Research Program of China(XX2020204B028).
通信作者:贺飞(hefei@tsinghua.edu.cn)

设计出现以来，硬件验证就成为同样重要的问题。硬件验证就是检查硬件设计对应的电路或系统是否按照指定的功能项或者安全性要求运行。模型检测技术[1-2]将硬件设计视为有限状态的转移系统 M，将待验证的属性视为逻辑公式 P，通过判定 M 是否为 P 的一个模型来进行验证。如果验证发现 M 不满足 P，算法会给出一条反例路径揭示 M 如何违反属性 P。IC3 算法[3-5]是最具代表性的比特级别(bit-level)模型检测算法，它基于归纳推理的思想，通过构造一个归纳不变式来验证属性。

随着硬件技术的不断发展，硬件设计复杂度和规模急剧增长，硬件设计也因此出现了更高的抽象级别，硬件设计语言也逐渐支持开发者在更高的抽象级别上进行硬件设计。同时，相关开发工具可以自动地将其转换为更低层级的硬件描述语言，但这个转换过程本质上是有抽象语义信息丢失的，因为在低级别的硬件描述下，难以定义和获取到更高级别的抽象语义。这种语义信息的丢失，对硬件验证来说很可能就是验证效率和精度的损失。为了适应硬件设计规模急剧增长的现状，字级(word-level)硬件模型检验算法被提出，研究者希望将硬件设计中的字级抽象语义信息提取出来并应用到模型检验过程中，提高验证的效率。IC3 算法作为最成功的比特级别硬件验证算法，如何扩展成字级算法，是研究者们需要关注的焦点。IC3PA[6]算法正是其中一种，它结合了隐式抽象和谓词抽象，将字级抽象的语义信息应用到 IC3 算法中。但是 IC3PA 方法有局限性，不适合处理算术逻辑运算复杂的场景，在特定问题上会出现状态空间爆炸的情况。

论文提出了一种新的基于变量隐藏抽象的字级硬件模型检验算法 IC3VA，并且设计和实现了适配的泛化和精化方案，还进行了实验验证。本文第 2 节介绍了 IC3 算法和 IC3PA 算法的相关背景。第 3 节介绍了 IC3VA 算法的设计过程、关键部分和整体流程说明，包括其正确性证明。第 4 节介绍了 IC3VA 算法泛化和精化的策略。第 5 节设计了相关实验来验证提出方法的有效性。

2 相关工作

2.1 IC3 算法

IC3 算法是一种比特级的符号化模型检验算法，使用归纳推理的思想，尝试构造一个归纳不变式来验证属性。该算法在硬件验证领域得到了广泛的应用。一个硬件设计通常用网表或者硬件描述语言(比如 Verilog)来表示。用 X 表示硬件设计中出现的状态变量，X' 表示状态变量在下个状态时的拷贝。硬件设计的行为可以被建模为一个模型检验的问题 $\langle X, I, T, P\rangle$，其中 $I(X)$ 是代表初始条件的逻辑公式，$T(X, X')$ 是代表变迁关系的逻辑公式，$P(X)$ 是一个代表希望满足的安全属性的逻辑公式。

一个状态 s 是对状态变量 X 的一组赋值，变迁系统的一条执行路径是一个状态的序列 $s_0, s_1, \cdots, s_k$ 使得 $I(s_0)$ 成立并且 $T(s_i, s_{i+1})$ 对于所有的 $0 \leqslant i \leqslant k-1$ 成立。属性公式 P 声明所有可达的状态集合应该满足 P。换句话说，P 应该是该硬件设计的一个不变式。如果经过验证发现 P 不是一个不变式，那么一定有一个有限长度的路径 $s_0, s_1, \cdots, s_k$(叫做反例路径)，使得 $P(s_k)$ 不成立。另外一个归纳不变式 F 是一个公式，满足：

(1) $I \Rightarrow P$

(2) $F \wedge T \Rightarrow F'$

IC3 算法尝试去证明一个硬件设计是否满足给定的安全属性 P。如果该硬件设计不满足 P，那么 IC3 算法返回一个反例路径；否则它返回一个空的路径，并且证明属性成立。

算法 1 是 IC3 算法的伪代码。首先它尝试去寻找 0 步或者 1 步可达的反例路径(第 1 行)。如果没有找到，IC3 算法实例化 F_k 为 P，k 步可达状态的上近似(第 4 行)。接下来，IC3 算法迭代式地检查 F_k 是否能 1 步到达 $\neg P$ 状态(第 6 行)。它会检查每个可满足的赋值 s(记做 *cube*)，判断 s 是否从初始状态 I 可达(第 8 行)。如果从初始状态不可达，IC3 算法移除 s 来使得 F_k 更加准确。这个过程不断重复，直到一个长度为 $k+1$ 的反例路径被发现(第 9 行)，或者所有经过 1 步变迁后会违反 P 的状态都是从初始状态不可达的。如果是后者情况，IC3 算法会检查对于某些 $2 < i \leqslant k+1$，是否存在 $F_i = F_{i-1}$。如果存在，那么 IC3 算法证明了 P 成立(第 13 行)，F_i 就是要找的归纳不变式，否则算法继续增加 k 并且进行下一轮迭代，并通过这种方式来检查反例是否在更长的变迁路径上存在。

算法 1 IC3 算法

```
输入：初始状态 I，变迁关系 T，安全属性 P
输出：P 成立，或者反例路径 cex
1. if I∧¬P or I∧T∧¬P holds, then
2.    return counterexample trace cex;
3. 初始化 k=1, F_k=P;
4. while true do
5.    while F_k∧T∧¬P' do
6.       let s be the satisfying assignment;
7.       if Reachable(s, I) then
8.          return counterexample trace cex;
9.    if F_i=F_{i-1} for some 2≤i≤k+1 then
10.      return empty trace; // P holds
11.   k++;
```

2.2 IC3PA 算法

IC3PA 提出一种将隐式谓词抽象[8]、IC3 算法以及反例制导的抽象精化结合的算法，从系统的字级描述表示出发，在经过谓词抽象后的空间上进行 IC3 算法。集合 $\mathbb{P}$ 表示建立在变量集 X 上的一组谓词，即对于每个 $p \in \mathbb{P}$，都是定义在变量集合 X 上的公式；对每个 $p \in \mathbb{P}$，定义一个新的布尔变量 x_p，称作抽象变量。这些抽象变量的集合记作 $X_{\mathbb{P}}$。谓词抽象下的抽象关系 $H_{\mathbb{P}}$ 定义为：

$$H_{\mathbb{P}}(X, X_{\mathbb{P}}) := \bigwedge_{p \in \mathbb{P}} x_p \leftrightarrow p(X) \tag{1}$$

IC3PA 的主要思路是模拟 IC3 算法在谓词集合上 $\mathbb{P}$ 执行，并且用隐式谓词抽象来避免抽象转移关系的计算。如果在抽象空间上发现反例路径，首先需要检查是否为真反例。若为伪反例，需要从伪反例中提取精度精化抽象，再在新的抽象模型上执行 IC3 算法。

集合 $\mathbb{P}$ 包含初始条件 I 和属性 P 中的所有谓词，所以 I 和 P 都是 $\mathbb{P}$ 中谓词的结合。如果一个公式 ϕ 是谓词集合 $\mathbb{P}$ 中的谓词的结合，那么它的抽象版本 $\overset{\wedge}{\phi}$ 逻辑等价于公式：

$$\phi[X_{\mathbb{P}}/\mathbb{P}] \wedge \exists X. H_{\mathbb{P}}(X, X_{\mathbb{P}}) \tag{2}$$

其中，$\phi[X_{\mathbb{P}}/\mathbb{P}]$ 表示将 ϕ 中每个出现在 $\mathbb{P}$ 中的谓词 $p(X)$ 替换成相对应的抽象变量 $X_{\mathbb{P}}$。而 $\exists X. H_{\mathbb{P}}(X, X_{\mathbb{P}})$ 和被抽象的

公式无关,它定义了谓词和谓词变量之间的关系。IC3PA 和 IC3 算法有相同的结构。此外,IC3PA 利用了 CEGAR[7] 框架,所以当找到抽象状态空间上的反例路径时,IC3PA 会检查是否可以具体化,若不可具体化,说明是伪反例,此时 IC3PA 执行精化方法抽取谓词消除该反例。IC3PA 算法忽略了谓词原始在比特级别的含义,所以运行起来会更快,结合 CEGAR 的框架,有效提高了验证的效率。

但是 IC3PA 算法在一些属性与变量取值强相关的情况下,可能会求解出类似于 $a=0$,$a=1$,$a=2\cdots$ 的一系列谓词。极端情况下,若变量 a 的位宽为 n,IC3PA 算法会进行 2^n 次迭代,每次排除 a 的一种可能取值,抽象空间中会有 2^n 个谓词,降低了算法的验证效率。如果直接将变量 a 看作可见变量,此时需要追踪的比特宽度为 n,即 n 个谓词就可以追踪变量的值。综合来说,当发现关于某个变量的谓词个数多于一个阈值时,可以将此变量变为可见变量,这样可以缩小抽象状态空间,减少计算消耗;而对于对属性影响不大或无关的变量,可以直接将其隐藏掉,从而进一步缩减抽象空间。所以论文提出基于变量隐藏抽象的 IC3 算法,将具体状态经过变量隐藏抽象之后,抽象状态集合是可见变量集合。

3　基于变量隐藏抽象的 IC3 算法 IC3VA

3.1　变量隐藏抽象的形式化定义

文献[9-10]中设计了变量隐藏抽象方法。变量隐藏抽象将状态变量集合划分为可见变量和不可见变量两个子集,其中可见变量子集被认为是对待验证属性来说很重要的集合。抽象模型中只保留可见变量的转移函数,不可见变量被抽象为无限制的基础输入。具体模型中,在可见变量上取值相同的具体状态被抽象到同一个抽象状态。由于消除了不可见变量上的逻辑限制,这种抽象会比原模型产生更多的执行路径。所以该抽象会导致伪反例出现,可以通过恢复不可见变量为可见的方式来消除伪反例。

若 $X_v=\{x_1,x_2,\cdots,x_n\}\subseteq X$ 为可见变量集合,初始时,X_v 包含属性 P 中出现的所有状态变量。假设每个 x_i 的宽度是 n_i,那么一个变量 x_i 可以拆解为元组 $\langle x_{i1},x_{i2},\cdots,x_{in_i}\rangle$。所以拆解后的比特集合为:

$$B_{x_v}=\bigcup_{i=1}^{n}\{b_{i1},b_{i2},\cdots,b_{in_i}\} \tag{3}$$

由于是 1 个比特,所以可以看作是布尔变量。抽象状态变量集合则为 B_{x_v},此时可见变量到抽象变量的映射关系是:

$$H_{x_v}(X_v,B_{x_v})=\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n_i}slice(x_i,j,j)\leftrightarrow b_{ij} \tag{4}$$

函数 *slice* 表示对变量进行切片,选中的是变量第 j 位的取值,即集合 B_{x_v} 中的每一个布尔变量对应到原变量的位置上的取值,若取值为 0,则布尔变量为 false,反之为 true。所以,对于一个具体状态 s,其抽象状态 $\hat{s}$ 是 s 在 B_{x_v} 和 B_{x_v}' 上的投影 $proj(s,B_{x_v}\cup B_{x_v}')$。即对于 IC3VA 算法,抽象状态变量是可见变量集合的每一位比特。变量隐藏抽象的抽象条件是,若两个具体变量在可见变量集合上的取值相同,则应该被抽象到同一个抽象状态,所以对于变量隐藏抽象有:

$$EQ_{x_v}(X,\bar{X})=\bigwedge_{x\in x_v}x(X)\leftrightarrow x(\bar{X}) \tag{5}$$

如果两个具体状态之间有转移,那么它们所属的抽象状态之间存在抽象转移。

IC3 算法中的泛化算法对算法的效率有着非常大的影响;另外,由于使用了变量隐藏抽象,所以会有伪反例的问题,需要精化以消除伪反例。下面介绍 IC3VA 泛化和精化算法。

3.2　IC3VA 的泛化算法

在 IC3 算法流程中,帧 F_i 一般用若干子句的集合来表示系统迁移 i 步可以到达的状态集合的上近似,当某个 *cube*(c,i) 被阻塞成功,说明前 i 步内该立方体表示的状态集合不可达,所以将该立方体取反变成子句加入到帧 F_i 中,达到去掉该 *cube* 的目的。若某个 *cube* 已经被成功阻塞,需要将其泛化成更大的集合,取反后则可以去除更多不可达状态。IC3VA 的抽象状态变量是可见变量的每一位比特,因此在抽象泛化的部分,IC3VA 可以使用改进的比特级 IC3 的泛化算法。

由于 *cube* 可以看作是若干文字的集合,所以标准的 IC3 算法的泛化会遍历集合中的文字,尝试丢掉每个文字来加强子句,主要是对丢弃后得到的 *cube* 进行 drop 检查。如果 drop 检查返回 true,则可以丢弃;如果返回 false,则说明不能丢弃,泛化算法将其加入到原子句中。drop 检查的作用是寻找 $\neg$*cube* 的最大归纳子句,如果能找到,返回 true,否则返回 false。具体来说,第 k 步迭代时,首先检查有没有包含初始状态,如果有则返回 false;如果没有,继续检查 *cube* 是不是归纳相关于 F_i,如果是,则返回 true,否则提取 *cube* 的前驱 s。由于每次迭代中 *cube* 中文字的个数都在逐渐减少,因此 *cube* 中的某个状态的转移得到有效扩展。

但是这种泛化方式也有局限性。图 1 给出了一个泛化不够好的例子。假设 000 是初始状态,001 是违反属性的状态,它有两个前驱:110 和 100。假设 IC3 算法首先找到了 100,由于 100 没有前驱,所以它的反是归纳的,IC3 判定不可达。IC3 使用本节提到的方式进行泛化,假设丢弃第三个文字,变成 10x,其中 x 表示不关心该文字的取值。由于 101 有前驱 011,所以 10x 不是归纳的。为了能找到一个可以把 100 和 101 都包含的且不可达的 *cube*,该 *cube* 必须包含 011,否则它的反就不归纳。同时包含 100,101 和 011 的 *cube* 是 xxx,就把初始状体包含在内了,所以 drop 会返回 false。类似的原因,第一个和第二个文字不能被丢弃。所以 IC3 算法只能阻塞 100,效率太低。

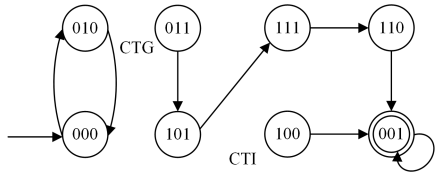

图 1　泛化失败的样例

Fig. 1　Sample of generalization failure

为了解决上述问题,使用了一种基于泛化反例(Counter-example to Generalization,CTG)[11] 的泛化方法来替代传统的 IC3 泛化算法。算法流程如算法 2 所示,其中 CTG 指类似于图 1 中 011 那种导致泛化失败的前驱。

此算法也是遍历 *cube* 中的每个文字,并检查剔除之后的 *cube* 是不是归纳的(第 5－8 行),如果不是归纳的,不会立即把 q 和 s 连接起来,而是试图在 F_{i-1} 层去阻塞 s(第 12 行)。如果尝试成功,就将 CTG 的个数增加 1,并尝试在更高层去阻塞(第 14－15 行)。找到最高层后,就在最高层递归调用算法

去泛化 s，并增加递归深度 d（第 16 行）。如果一个 CTG 的深度很大，就不进行后续的步骤（第 10 行）。在解决了一个导致 q 不归纳的原因后，算法继续迭代。如果 CTG 的个数太多，超过了限制，就把找到的 CTG 和 q 连接起来，并把 CTG 数目置 0（第 19－20 行）。这种泛化方式比经典的泛化方式的效果更好。

算法 2 基于泛化反例的 IC3 算法泛化 MIC

输入：一步变迁违反属性的 cube q，帧的下标 i，当前检查深度 d

输出：无返回值

```
1. for literal l in q:
2.     q̂ := q\l;
3.     ctgs := 0
4.     while true:
5.         if I ⇏ ¬q: break;
6.         if F_i ∧ ¬q ∧ T ⇏ ¬q': 
7.             q := q̂; break;
8.         with(F_i ∧ ¬q)-state s:
9.             if d > maxDepth: break;
10.            if ctgs < maxCTGs and i > 0:
11.                if F_{i-1} ∧ ¬s ∧ T ⇏ ¬s': break;
12.                ctgs := ctgs + 1;
13.                for j := ito k:
14.                    if F_j ∧ ¬s ∧ T ⇏ ¬s': break;
15.                    MIC(s, j-1, d+1);
16.            add ¬s to clause(F_j);
17.            else:
18.                ctgs := 0;
19.                q := q ∪ s.
```

3.3 IC3VA 的精化算法

根据反例制导的抽象精华框架（CEGAR），如果在抽象状态空间中发现了反例，首先需要在具体系统中模拟该抽象反例路径的执行，如果该检查确认反例在抽象状态空间可达，则验证为真反例，证明属性违反；否则，证明为伪反例，需要据此伪反例提取精度，继续精化。而对于 IC3VA 算法而言，精化抽象空间的方式是添加新的可见变量。下面介绍 IC3VA 的精化算法。

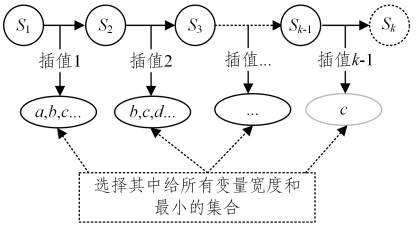

图 2 IC3VA 精化示例图

Fig. 2 Example of IC3VA refinement

假设在抽象空间上得到的反例路径是 $\pi=(\hat{s}_0,\hat{s}_1,\cdots,\hat{s}_k)$，$\hat{s}$ 表示一个抽象状态。因为使用了抽象技术，所以这条反例路径实际包含了具体空间上的多条路径。利用限界模型检测将该路径进行展开，具体流程如下。

(1)将每个抽象状态 $\hat{s}$ 具体化为具体状态 s，即将抽象状态替换成具体状态的语义：

$$s=\hat{s}(B_{x_v})\wedge H_{x_v}(X_v,B_{x_v})=\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n_i} slice(x_i,j,j)\leftrightarrow\hat{s}(b_{ij})$$

(2)给 s 对应的公式加上“时间”属性，即需要显式标注是迁移中的第几步。

(3)给具体状态之间加上转移函数，使其连接起来形成具体路径 $Path^{\pi}_{B_{X_v}}$。若 $Path^{\pi}_{B_{X_v}}$ 可满足，则说明该抽象反例存在对应的具体反例；若不可满足，则说明该抽象反例为伪反例，系统抽象层次太高，需要增加可见变量来消除反例。

论文设计的求解精度的方式是对 $Path^{\pi}_{B_{X_v}}$ 求插值，由于 $Path^{\pi}_{B_{X_v}}$ 不可满足，所以可以调用求解器求该路径的插值，长度为 k 的路径产生 $k-1$ 个插值，若某插值公式中只有一个隐藏变量，则将该隐藏变量变为可见，消除反例。由于可见变量集合是单调增加的，抽象转移函数也是单调增强，即 $X_v\subseteq X_v'$，所以 $\hat{T}_{X_v'}\rightarrow\hat{T}_{X_v}$。所以精化之前帧中的子句仍然可以保留，算法是增量的。

3.4 IC3VA 算法流程

整个 IC3VA 算法的流程如算法 3 所示。IC3VA 和 IC3 一样，维护了帧的序列 F。初始的精度集合 X_v 为属性 P 中出现的变量。IC3VA 中的状态变量是 B_{x_v} 中变量与或非的合取，子句则是析取。IC3VA 的帧是这些子句的合取。

算法 3 IC3VA 算法

输入：初始条件 I，变迁关系 T，安全属性 P，可见变量集合 X_v

输出：属性证明，或者返回反例路径

```
1. 初始化X_v为属性 I 和 P 中出现的变量集合
2. X_v = X_v ∪ {x | x∈I ∨ x∈P}
3. if Ī |≠ P̄ return false// 初始状态不满足
4. F_0 := Ī, k := 1, F_k := P̄// 添加新帧
5. while true:
6.     while F_k ∧ ¬P̄ is sat: //阻塞阶段
7.         从模型中抽取一个 cube c
8.         if c 在F_k无法被阻塞:
9.             构造抽象反例 π=(ŝ_0, ŝ_1, ..., ŝ_k)
10.            if π 还原到具体状态空间 unsat:
11.                精化并且添加新的可见变量
12.            else return false
13.    k := k+1, F_k := P̄// 传播阶段
14.    for i := 1 to k-1:
15.        for each clause∈F_i:
16.            if c 在变量隐藏抽象下经过一步变迁下无法到达¬P̄
                   状态:
17.                add c to F_{i+1}
18.        if F_i = F_{i+1}:
19.            return true //
```

算法是一个大的循环体，每次迭代首先是阻塞阶段，然后是传播阶段。阻塞环节首先检查上一个帧中违背属性的抽象的 $cubec(B_{x_v})$ 是否可以被阻塞（第 8 行），如果可以被阻塞，则对 $\neg c$ 进行泛化，并且将泛化之后的子句加到 F_1 到 F_k 中，来强化它们。如果 c 无法被阻塞，则需要继续抽取 c 的抽象前驱，也就是 $F_{i-1}\wedge\neg c$ 中可以一步到达 c 的抽象 $cube\ s$，然后递归尝试去阻塞 s。阻塞过程是递归的，它要么不断加强队列中的帧，要么不断泛化抽象反例。如果发现需要被阻塞的处于第 0 级的状态，该递归函数退出，说明发现了一个抽象反例。然后尝试对抽象路径进行具体化，如果不能具体化，则说明是伪反例，从该抽象路径中提出精度加入原始精度集合 X_v

(第 11 行)。具体化成功,则属性为假,返回 false。

如果当前的帧找不到违反属性的抽象 *cube*,则算法进入到传播阶段。新增一个不包含任何子句的帧(第 17 行),并且从F_1开始检查,将F_i中归纳相关于它的子句加入到F_{i+1}。因为$\neg c$可以在F_i被阻塞,所以 c 在F_{i+1}是不可达的。如果在传播阶段发现$F_i=F_{i+1}$,则属性成立(第 18 行)。

4　实验与分析

论文对 IC3VA 方法的有效性进行了验证;按照算法 3 的设计,在 IC3PA 的开源工具基础上实现了 IC3VA 算法;选取了相同的硬件验证数据集进行了对比实验,分析了 IC3VA 算法在时间效率上的效果。

4.1　实验环境与数据集说明

实验在同一台服务器上完成,服务器操作系统是 64 位的 Ubuntu 20.04 LTS,处理器是 AMD EPYC 7282 16-core Processor @2.80 GHz,内存是 128 GB,硬盘是 7200 rpm 的 4T 机械硬盘。

论文从多学术论文[12]和开源社区中收集了 487 个硬件安全属性验证任务,包括 vcegar[13],v2c[14],verilog2smv[15]等开源的标准硬件,验证测试集中的 163 个验证任务,记作测试集 A。实验还选取了硬件模型检验大赛(HWMCC'20)[16]的 bit-vector 分支上的验证任务,共 324 个,记作测试集 B。为了提供公平的对比,所有的测试集文件通过 yosys[17-18]和 vmt-tools[19]工具综合成面向验证的统一输入 vmt 格式。实验将每个验证任务的验证时间上限设置为 15 min,即 900 s,超过上限会被强行终止。

4.2　实验与分析

论文将 IC3PA 和 IC3VA 算法在任务集 A 和 B 总计 487 个任务上进行了验证实验,实验总体结果如表 1 所列。其中,safe 表示安全属性验证通过的任务数量,unsafe 表示验证为不安全的任务数量,TO 表示验证时间超过 900 s 从而被强行终止的任务数量,unknown 表示不能返回验证结果的任务的数量,unique 表示当前验证算法能验证但是其他验证算法无法验证的任务数量。

表 1　总体实验结果

Table 1　Overall experimental results

任务集	A(共 163 个)		B(共 324 个)	
算法	IC3PA	IC3VA	IC3PA	IC3VA
safe	116	106	74	66
unsafe	40	41	15	12
TO	7	12	230	171
unknown	0	4	5	75
unique	12	3	24	12

根据实验结果,IC3VA 和 IC3PA 共同可解的任务,在任务集 A 上有 144 个,在任务集 B 上有 66 个,总计 210 个。在 900 s 内可以解决的任务的个数上,IC3PA 要多于 IC3VA,在数据集 A 中,IC3PA 可以求解 116 个例子,IC3VA 求解了 106 个例子;在数据集 B 中,IC3PA 求解了 74 个例子,IC3VA 求解了 66 个例子。从 unique 这个指标可以看出两个算法的区别,IC3VA 可以解决部分 IC3PA 算法无法解决的任务,其中任务集 A 中有 3 个,任务集 B 中有 12 个。数据集 B 的任务平均验证复杂度要远高于任务集 A,可以看出 IC3VA 的改进策略是有效的,在验证任务本身较为复杂的情况下,实验效果体现得更加明显。下面从验证时间、精化次数以及求解次数等角度来分析 IC3VA 算法的验证效率问题。

从表 2 的数据中可以看出,在任务集 A 上,对于 IC3VA 和 IC3PA 都可以解决的 144 个任务,IC3VA 求解时间是 2584 s,IC3PA 是 1059 s,前者耗时是后者的 2.4 倍,同时求解次数是后者的 2.82 倍;在任务集 B 上,对于两者都可以解决的 66 个任务,IC3VA 的总体耗时少于 IC3PA,求解次数也要少于 IC3PA。在不同的任务集上,两者的验证效果对比有所不同。IC3PA 在两个任务集上的平均精化次数均为 2.8 次左右。

表 2　两种算法验证效率指标的对比

Table 2　Comparison of two algorithm validation metrics

任务集	A(同解 144 个)		B(同解 66 个)	
算法	IC3PA	IC3VA	IC3PA	IC3VA
时间/s	1059	2584	5671	5211
精化数	398	31	189	6
求解数	7	12	230	171

虽然任务集 A 中测例较为简单,但是 IC3PA 验证过程中出现的变量的宽度可能并不小,因而对于其中大部分任务,更加灵活的 IC3PA 可以使用较少的谓词求解。所以,在数据集 A 上,144 个验证任务净化了 398 次,平均为 2.76 次。IC3VA 在这种情况下整体耗时较高。但在更为复杂的数据集 B 上,对于其中大部分验证任务,其验证过程更为复杂,且复杂度主要体现在转移函数以及变量数量上,而不是变量位宽。因此在数据集 B 两者都可以解决的任务中,相比于 IC3PA 使用的谓词抽象的方式,IC3VA 使用变量隐藏抽象提高了精化部分的效率,总的精化次数只有 6 次,所以在验证耗时和求解次数上更具优势。

IC3VA 验证的实验结果基本符合设计的初衷,在较为复杂的验证任务上,它缓解了 IC3PA 验证过程中出现的某个变量谓词数量过大的问题,能够提高验证的效率。综上所述:

(1)IC3VA 算法能够解决部分 IC3PA 无法解决的验证任务,IC3VA 算法是有效的。

(2)对于数据集 B 上的共同可解的验证任务,IC3VA 算法耗时减少了 460 s,验证效率有提升。此外,在 IC3PA 超时而 IC3VA 可解的任务上,IC3VA 平均耗时不超过 200 s。这说明 IC3VA 基本符合算法设计的预期,在一些 IC3PA 算法关于某个变量谓词过大的情况下,IC3VA 可以加速验证。

(3)IC3VA 算法仍然有优化的空间。根据实验的日志数据,在 75 个 unknown 验证任务上,IC3VA 都有 1 次精化,但这其中只有 6 次是精化成功的,其余的 69 次都失败了。此外,IC3VA 总体验证的任务数量还少于 IC3PA,这种表现可能的原因是,IC3VA 把初始条件和属性中的所有变量都视为可见变量,这种方式导致初始抽象的程度太低,所以对于某些验证任务来说,如果将所有相关变量都视为可见变量,那么 IC3VA 算法就退化到了原始的比特级别 IC3 算法,而有些任务本来只需要一两个谓词就可以解决,相比于 IC3PA 算法,这时 IC3VA 会迅速增大状态空间而超时。

结束语　论文提出了一种基于变量隐藏抽象的字级 IC3 算法 IC3VA,并且介绍了 IC3VA 中新的泛化和精化方法,同时给出了 IC3VA 算法完整的描述。为了验证 IC3VA 算法的有效性,对 IC3VA 算法进行了实现,并且收集了两个开源数据集作为标准硬件验证测试集,在其上进行了实验和结果分析。

实验结果表明，IC3VA 算法确实可以解决部分 IC3PA 算法超时的任务，并且在较为复杂的数据集共同可解的例子上耗时更少。当然，论文提出的方法也有局限性，即从初始状态和属性中提起可见变量的方式比较简单，容易造成状态空间迅速增大，导致超时的问题。更加合理和灵活的可见变量选择以及与之配套的泛化和精化方案，是未来一个研究方向。

参 考 文 献

[1] CLARK E M. Model checking[C]// 17th Foundations of Software Technology and Theoretical Computer Science. Springer Berlin Heidelberg, 1997: 54-56.

[2] CLARK EM, JOOST P K. Principles of model checking [M]. MIT Press, 2008.

[3] BRADLEY A R. SAT-based model checking without unrolling [C]// 12th Verification, Model Checking, and Interpretation (VMCAI), 2011: 70-87.

[4] BRADLEY A R. Understanding ic3[C]// 15th Theory and Application of Satisfiability Testing-SAT. 2012: 1-14.

[5] EEN N, MISHCHENKO A, BRAYTON R. Efficient implementation of property directed reachability[C]// 2011 Formal Methods in Computer-Aided Design (FMCAD). IEEE, 2011: 125-134.

[6] CIMATTI A, GRIGGIO A, MOVER S, et al. IC3 Modulo Theories via Implicit Predicate ion[C]// International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2014: 46-61.

[7] CLARKE E, GRUMBERG O, JHA S, et al. Counterexample-guided abstraction refinement [C]// Proceedings of Computer Aided Verification: 12th International Conference (CAV 2000). 2000: 154-169.

[8] GRAF S, SAIDI H. Construction of abstract state graphs with PVS[C]// CAV. 1997: 72-83.

[9] LONG D E. Model checking, abstraction, and compositional verification [M]. Carnegie Mellon University, 1993.

[10] KURSHAN R P. Computer-aided verification of coordinating processes [M]// Computer-Aided Verification of Coordinating Processes. Princeton University Press, 2014.

[11] HASSAN Z, BRADLEY A R, SOMENZI F. Better generalization in IC3[C]// 2013 Formal Methods in Computer-Aided Design. IEEE, 2013: 157-164.

[12] GOEL A, SAKALLAH K. Empirical evaluation of ic3-based model checking techniques on verilog rtl designs[C]// Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2019: 618-621.

[13] JAIN H, KROENING D, SHARYGINA N, et al. VCEGAR: Verilog counterexample guided abstraction refinement [C]// Tools and Algorithms for the Construction and Analysis of Systems[C]// 13th International Conference (TACAS 2007). 2007: 583-586.

[14] MUKHERJEE R, TAUTSCHNIG M, KRONENING D. v2c-A verilog to C translator [C]// 22nd International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016). 2016: 580-586.

[15] IRFAN A, CIMATTI A, GRIGGIO A, et al. Verilog2SMV: A tool for word-level verification[C]// 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2016: 1156-1159.

[16] PREINER M, BIERE A, FROLEYKS N. Hardware model checking competition[EB/OL]. https://fmv.jku.at.hwmcc20.

[17] YOSYS. Yosys open synthesis suite[EB/OL]. https://www.yosyshq.com/about.

[18] WOLF C, GLASER J, KEPLER J. Yosys-a free Verilog synthesis suite[C]// Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip). 2013: 97.

[19] CIMATTI A, GRIGGIO A, TONETTA S. The VMT-LIB language and tools [J]. arXiv: 2109.12821, 2021.

YANG Liu, born in 1997, master. Her main research interests include predicate abstraction based on reinforcement learning and hardware model checking.

HE Fei, born in 1980, Ph. D, associate professor, Ph. D supervisor. His main research interests include program verification, model checking, and automated reasoning.