基于模块化 Abstract-Refine 算法框架的软件模型检测方法

王 舜,杜 晔,韩 臻
(北京交通大学计算机与信息技术学院,北京 100044)

摘 要: Abstract-Refine(抽象—精炼)方法是软件模型检测领域中较为有效的设计思想,具有较高的通用性和效率优势,但目前并没有一个框架可以对其精确进行描述及实现有效的模块化使用和替换. 本文提出了一种模块化的 Abstract-Refine 算法框架,分析和解释了 Abstract-Refine 算法所接受的输入程序的精细结构和特性,并对 Abstract-Refine 算法和相关子算法运用平衡操作符做以模块化解耦,使得子算法的修改和更换不需要依赖对上层的变更. 经过实验验证,本方法可有效实现传统算法模块化解耦,同时不对原算法的性能造成冲击.

关键词: 软件模型检测; 模块化方法; 抽象—精炼 (Abstract-Refine); 通用算法; 抽象程序
中图分类号: TP311.5 **文献标识码:** A **文章编号:** 0372-2112 (2020)05-0997-06
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2020.05.022

Software Model Checking Method Based on Modular Abstract-Refine Algorithm Framework

WANG Shun, DU Ye, HAN Zhen
(*School of Computer and Information Technology, Beijing Jiaotong University, Beijing* 100044, *China*)

Abstract: Abstract-Refine is a relatively effective method in the field of software model checking, which has the advantages of high generality and efficiency. However, there is no framework for precise description and effective modular use or replacement of this method so far. This paper introduces a modular Abstract-Refine algorithm framework which analyzes and explains the structure of input program in fine-grained level. Also, this method modularly decouples Abstract-Refine algorithm from its sub-algorithms with the balancing operator, so that any modifications on sub-algorithms will not affect the upper level. Experiments verify that our approach can effectively implement modular decoupling of traditional algorithms, and will not impact the performance of original algorithms.

Key words: software model checking; modularity; Abstract-Refine; general algorithm; abstract program

1 引言

软件模型检测是一种用于验证程序代码是否符合相关规约的重要方法. 目前,软件模型检测有三种主流的遍历方式:(1)下逼近式,(2)上逼近式和(3)混合式[1]. 有界模型检测(BMC)是下逼近式方法的重要成员,同时也是最主要的下逼近方法,其核心思路是模拟软件执行[2]. BMC 方法可以保证所有检测到的规约违反都是真实可靠的,但是却不能保证穷尽整个程序的状态空间. 也就是说,BMC 得到的没有违反规约的结果是不可靠的.

为了解决这个问题,研究者提出了上逼近式的方法. 最为广泛被使用的是谓词抽象(Predicate Abstract)方法[3]. 谓词抽象更加关心执行的覆盖性,因此这类方法可以保证所有检测到的规约保持是真实可靠的,但是却无法保证发现的规约违反的真实性.

混合式方法兼有二者优点,其代表是 Abstract-Refine 方法. 在 Abstract-Refine 方法中,Abstract 基本上可以认为是指代上逼近方法集,而 Refine 则是指代下逼近方法集. 反例引导的 Abstract-Refine 方法就是其中的典型[4].

本文提出了一种对输入程序的更为精确的刻画,

收稿日期:2019-03-29;修回日期:2019-09-02;责任编辑:李勇锋
基金项目:国家自然科学基金(No.61672092)

使得它们可以描述程序的一些重要的特性. 同时,本文针对这种描述提出了一种操作关系的算符,称为平衡算符用来统一描述 Abstract 和 Refine 过程,并演示了将不同子算法嵌入的方案. 最后,通过实际测试,证明了本文的方案并不会对原有的基于 Abstract-Refine 的算法引入额外开销.

2 相关工作

抽象方法 基于逼近的目标不同,许多上逼近方法都可以被看作是抽象方法[5]. 其中的代表就是谓词抽象,文献[6,7]提供了典型的谓词抽象的方案. 另外,如果将程序位置作为一种抽象的话,那么文献[8,9]两项关于区块抽象分析的研究可以认为是采取了此种思路.

精炼方法 在值和变量层上最常用的精炼方法是 Craig 插值,相关研究包含在文献[10~12]中. 如前文所述,BMC 类方法也可以很好地扮演精炼的角色,例如文献[13],这种精炼方法被称为反例引导的抽象-精炼(CEGAR),文献[14]使用该方法检测了 C 语言中的空指针. 除此之外还有文献[15]同样是直接基于抽象的精炼方法,它们并不依赖具体的反例来对精炼过程进行引导.

框架研究 框架研究主要关注方法的结合,比如文献[16,17]组合了抽象方法和具体方法. 可配置模型检测(CPA)[18]是其中重要的研究,它在一定程度上解决了一些方法进行组合的问题,但是并未涉及到抽象程序本身的特征和 Abstract-Refine 检测的通用结构. 因此,这种解耦合,其仍然难以得到一个稳定的算法框架,需要不断地修改以适应相关算法的接插. 这使得基于框架的算法实现出现了重复的部分,从而降低了其通用性. 另外,CEGAR 只是 Abstract-Refine 方法中的一类,其他子类型的方法也应该包含在 Abstract-Refine 方法的模块化描述中.

3 输入程序的精细结构定义与属性分析

定义 1 (抽象程序)抽象程序 $\hat{P}$ 是一个元组($\hat{F}$,$\hat{S}$),其中 $\hat{S}=\{(l,x,v)\mid l\in\hat{L},x\in\hat{X},v\in\hat{V}\}$. $\hat{L}$ 表示程序位置抽象的集合,$\hat{X}$ 表示程序变量抽象的集合,$\hat{V}$ 表示程序值的集合. $\hat{F}=End(\hat{S})$.

定义 1 描述了一个通用程序模型,其不仅可以描述静态程序,还可以描述程序在动态执行过程中的任意状态,并且这些状态都可以通过对原模型进行局部细分得到.

基本模型的描述是受限的,其无法表述带有循环指令结构或者循环数据结构的程序. 为了突破这个限制,可以从基本版的模型导出一个更为复杂的版本. 递归地定义 $\hat{L}\supset\{l\mid l\to\hat{L}\}$,同时 $\hat{L}\supseteq L$. 相似地,可以定义 $\hat{X}\supset\{x\mid x\to\hat{X}\}$ 且 $\hat{X}\supseteq X$. 最后,由于值没有使用映射定义,所以这个版本的值也可以简单地定义成 $\hat{V}=V*$.

本文的研究主要聚焦在程序在运行时抽象模型体现出的性质,这也正是 Abstract-Refine 方法所在的作用域. 在程序运行过程中,新状态的产生可以被看作是由于条件变化导致状态空间产生矛盾,从而不得不引入来解决矛盾的规则. 这套推理规则可以被描述如下:

性质 1 (递归解析性)$\forall m,p,q\in L\cup X\cup V$ 有

$$m\to p,m\to q,p\neq q\Rightarrow m\to\{p,q\} \tag{1}$$

$$m\to p,n\to q,m\to n,p\neq q\Rightarrow m\to q \tag{2}$$

对任意两个状态应用递归合并,实际上是按顺序对其位置、变量、值抽象分别应用以上性质. 这个性质使得不同层级上的抽象状态进行比较和一定程度上的合并成为可能. 而如果需要处理的是一个包含复杂子状态的抽象状态,那么就需要将其分步展开成多个简单状态,然后再做进一步的组合和展开. 为了将复杂状态展开成为简单状态,我们又引入了下面的另一个性质.

性质 2 (可分配性)$\forall l_1,l_2,l\in L,x_1,x_2,x\in X,v_1,v_2,v\in V$ 有

$$(l_1\cup l_2,x,v)=(l_1,x,v)\cup(l_2,x,v) \tag{3}$$

$$(l,x_1\cup x_2,v)=(l,x_1,v)\cup(l,x_2,v) \tag{4}$$

$$(l,x,v_1\cup v_2)=(l,x,v_1)\cup(l,x,v_2) \tag{5}$$

可分配性的存在保证了包含多个复杂子状态的抽象状态可以被依次分割成多个简单状态.

根据上文性质,一个程序同样也可以被分解成多片子程序,这使得基于这些属性的模型检测具有潜在的并行性. 进一步地,为了降低实际检测程序的复杂性,我么也需要提出一种方法来将程序状态分割成片.

如果有两个抽象状态(l_1,x_1,v_1)和(l_2,x_2,v_2)满足 $l_1\subseteq l_2,x_1\subseteq x_2,v_1\subseteq v_2$,此时可记$(l_2,x_2,v_2)\geq(l_1,x_1,v_1)$. 特别地,如果$(l_2,x_2,v_2)\geq(l_1,x_1,v_1)$且$(l_1,x_1,v_1)\geq(l_2,x_2,v_2)$,记$(l_1,x_1,v_1)=(l_2,x_2,v_2)$.

定义 2 (抽象层)对于一个抽象模型 $\hat{P}_f$,它的一个抽象层是模型的一个抽象状态集 $\hat{S}_l$,对于 $\forall s_l\in\hat{S}_l$ 满足 $\exists s_f\in\hat{S}_f$ 有 $s_l\geq s_l$ 或 $s_f\geq s_l$ 或 $s_f=s_l$.

本文的讨论只聚焦于两类特殊的抽象层,即上逼近抽象层和下逼近抽象层. 如果将模型检测需要找到的规约保持或者规约违反的路径看作是一个抽象层,称为目标抽象层. 上逼近抽象层和下逼近抽象层对于这个目标抽象层的所有状态都是可分配的. 这意味着上逼近抽象层和下逼近抽象层被目标抽象层分离,从而使得这两层中的推理可以相对独立地进行.

最后在抽象模型 $\hat{P}=(\hat{F},\hat{S})$ 中转换映射 $\hat{F}$ 和抽象状态集 $\hat{S}$ 间尚存在形式化描述上的差别. 在传统的程序形式化描述中,$\hat{F}$ 通常被视作一个操作符集合,其中的操作符有限,并且不可分. 在这种定义下,状态间的转换难以做到合并,分离和抽象,使得其难以被拓展到抽象计算的层面中.

定理 1　在抽象模型 $\hat{P}=(\hat{F},\hat{S})$ 中,满足 $\forall f=s_1\rightarrow s_2, f\in\hat{F}$,总是存在与 f 相关的状态 s_{op},使得 $f(s_1)=s_1\cup s_{op}$.

我们可以使用构造性证明方法来证明这个定理. 如果 s_1 的变量与 f 所指代的操作符不共享相同的变量,这时操作符可以独立地生成一个与 s_1 无关的状态空间,很明显地,将操作符应用在状态 s_1 上就相当于对两个状态空间进行合并,从而 $f(s_1)=s_1\cup s_{op}$ 成立. 如果 s_1 的变量所指代的操作符与 f 所指代的操作符共享相同变量,可以分别对 s_1 和 f 的生成状态空间应用性质 2,从而拆分出二者独立的部分,并进行合并. 而二者不独立的部分可以进一步应用性质 1,得到新的合并状态. 最后将两个结果合并,同样可以得到 $f(s_1)=s_1\cup s_{op}$. 基于这个定理,抽象模型 $\hat{P}$ 也可以记作 $(\hat{S},\hat{S}_{op})$.

定理 1 使得抽象状态可以彻底地从抽象模型的转换映射中剥离出来,我们可以用 $\hat{S}_{op}$ 来指代从 $\hat{F}$ 中剥离出的抽象状态. 释放出操作符中的抽象状态后,整个抽象模型在状态转换上就具有了更大的灵活性. 自此,整个抽象模型中的转换也就被统一了. 在抽象模型的状态转换中不需要再区分不同的转换方式,只需要关注相关的抽象状态的拆分和组合就足够了. 我们使用一个新的操作符来定义这个统一的转换方式,称为平衡操作符.

定义 3　(平衡操作符)平衡操作符 $\hookleftarrow\!\!\rho$ 是一个二元操作符,它接受两个抽象状态空间,并对其应用性质 1. 形式化地,对于一个抽象状态空间 $\hat{S}$,和一个策略抽象状态空间 $\hat{S}_{op}$,$\hat{S}\hookleftarrow\!\!\rho\hat{S}_{op}$ 得到另一个抽象状态空间 $\hat{S}'$ 是执行结果,即 $\hat{S}'$ 不包含任何矛盾的状态. 为了简化表达,我们定义 $\hat{S}'\hookleftarrow\!\!\rho\hat{S}_{op}=\text{true}$ 表示 $\hat{S}'$ 关于 $\hat{S}_{op}$ 已经平衡,即 $\hat{S}_{op}=\hat{S}'\hookleftarrow\!\!\rho\hat{S}_{op}$.

定理 2　对于算符 $\hookleftarrow\!\!\rho$,若 $\hat{S}'=\hat{S}\hookleftarrow\!\!\rho\hat{S}_S, S_r\subseteq\hat{S}'$,则有 $\hat{S}\hookleftarrow\!\!\rho\hat{S}_S\backslash S_r=\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S$.

证明　根据性质 2,$\hat{S}'=\hat{S}\hookleftarrow\!\!\rho\hat{S}_S$ 等价于 $\hat{S}'=(\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S)\cup(S_r\hookleftarrow\!\!\rho\hat{S}_S)$. 对等式两边进行移项可得 $\hat{S}'\backslash(S_r\hookleftarrow\!\!\rho\hat{S}_S)=\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S$. 对等式的两边同时应用平衡操作符可得 $(\hat{S}'\backslash(S_r\hookleftarrow\!\!\rho\hat{S}_S))\hookleftarrow\!\!\rho\hat{S}_S=(\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S)\hookleftarrow\!\!\rho\hat{S}_S$. 同时,又 $\hat{S}'=\hat{S}\hookleftarrow\!\!\rho\hat{S}_S=(\hat{S}\hookleftarrow\!\!\rho\hat{S}_S)\hookleftarrow\!\!\rho\hat{S}_S$,所以 $(\hat{S}'\backslash(S_r\hookleftarrow\!\!\rho\hat{S}_S))\hookleftarrow\!\!\rho\hat{S}_S=\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S$. 继续对等式左边应用性质 2 可得 $(\hat{S}'\hookleftarrow\!\!\rho\hat{S})\backslash((S_r\hookleftarrow\!\!\rho\hat{S}_S)\hookleftarrow\!\!\rho\hat{S}_S)=\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S$,对等式两边进行约简后可以得到 $(\hat{S}'\hookleftarrow\!\!\rho\hat{S})\backslash(S_r\hookleftarrow\!\!\rho\hat{S}_S)=\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S$. 因为 $S_r\hookleftarrow\!\!\rho\hat{S}_S$,可得 $\hat{S}\hookleftarrow\!\!\rho\hat{S}_S\backslash S_r=\hat{S}\backslash S_r\hookleftarrow\!\!\rho\hat{S}_S$,证毕.

可以看出,对于任意抽象模型 $\hat{P}$,都可以通过平衡操作符进行统一的抽象模拟执行,而这种抽象执行已经脱离了具体程序实际执行过程中对于操作符和程序指针的依赖. 到此为止,我们构建了一个具有统一的抽象状态和执行方式的抽象程序模型.

4　模块化 Abstract-Refine 算法

首先我们使用上文所描述的抽象模型来表达的基本模型检测算法,该算法可以看作是 BMC^{∞},即 BMC 的无界版本. 如果跟踪这个算法的执行过程,我们会发现,它有可能多次触发性质 1(2). 再考虑性质 1(1),其在设定了初始化状态到程序初状态集中时,对于每一个初状态的计算都不需要应用该性质,只有在不同初状态的混合计算时才需要使用.

然而性质 1(1)在计算中会破坏基本版本模型中与 $\mathbb{N}$ 的对应的连续性,导致随着计算的深入,计算量会随之按指数级增大. 解决这个问题需要直接限制性质 1 的过度应用. 上文提到的平衡操作符在这里将扮演控制抽象的核心角色.

算法 1　抽象状态空间上的模型检测算法

输入:抽象模型 $\hat{P}=(\hat{S},\hat{S}_{op})$,错误状态 s_e,平衡策略 $\hat{S}_S$
输出:若错误状态不可达,则 *SAFE*,否则 *UNCERTAIN*

```
Function AbstractMC(P̂, s_e, Ŝ_S)
reach := ∅, loctop := Ŝ
While loctop = Ŝ do
  s ∈ loctop, loctop := loctop\s
  If s ∉ reach then
    If s ↩ρ Ŝ_S then
      reach := reach ∪ {s}
      For s_op ∈ Ŝ_op do
        S_n := SGA(s, s_op)
        For s_n ∈ S_n do
          If s_e ↩ρ s_n then Return UNCERTAIN
          End If
        End For
        loctop := loctop ∪ S_n
      End For
    Else
```

```
        reach := reach ∪ {s}
      End If
    End If
  End While
  Return SAFE
End Function
```

算法 1 是运行在抽象空间上的模型检测算法,该算法在后文中简记为 AbstractMC. 当状态空间满足策略空间的抽象要求时,状态空间关于操作符抽取的状态空间的进一步平衡操作就被跳过了. 策略空间 $\hat{S}_S$ 和操作符抽取的状态空间 $\hat{S}_{op}$ 在形式上是一样的,这意味着它们具有在某种情况下的可替换性.

对于精炼用 BMC^{∞} 来讲,其输入抽象程序模型是一个包含潜在错误状态的抽象层. 这个抽象层所生成的模型状态集是 $\hat{S}_{op}$ 而操作符抽取的状态集是 $\hat{S}_r$,正好与正常的检测输入相反. 按照这种输入,如果 *AbstractMC* 执行的是一种浅而广的搜索的话,BMC^{∞} 就会相对地执行一个深而窄的状态搜索,反之亦然. 通过对 AbstractMC 和 BMC^{∞} 的返回值添加可达的状态子空间,这两个算法可以被简记如下:

$$(r,S_r) := ConcreteMC(\hat{P},s_e) \tag{6}$$

$$(r,S_r) := AbstractMC(\hat{P},s_e,\hat{S}_S) \tag{7}$$

这里的 *ConcreteMC* 指 BMC^{∞}. 于是可得 $(r,S_r) := AbstractMC(\hat{P},\overline{s_e},\varnothing)$ 与 $(r,S_r) := ConcreteMC(\hat{P},s_e)$ 等价,后续就可以统一使用 AbstractMC 来描述.

算法 2 模块化 Abstract-Refine 算法

```
输入:抽象模型 P̂ = (Ŝ, Ŝ_op),错误状态 s_e,平衡策略 Ŝ_S = (Ŝ_S^A, Ŝ_S^R)
输出:若错误状态不可达,则 SAFE,否则 UNSAFE
Function UnifiedARMC(P̂, s_e, Ŝ_S)
r := UNCERTAIN
While r = UNCERTAIN do
  (r,S_r) := AbstractMC(P̂, s_e, Ŝ_S^A)
  If r = SAFE then
    Return SAFE
  End If
  (r,S_r) := AbstractMC((S_op, S_r), s_e, Ŝ_S^R ∩ Ŝ)
  If r = SAFE then
    Return UNSAFE
  End If
  P̂ := (Ŝ \ S_r, Ŝ_op)
End While
EndFunction
```

模块化 Abstract-Refine 算法框架如算法 2 所示,该算法在后文中简记为 UnifiedARMC. 其中原本 ConcreteMC 所占据的精炼部分已经被一个更广义的 AbstractMC 算法所替代. 在这个算法中,精炼过程也引入了策略空间,从而使得精炼过程也可以摆脱具体状态空间的束缚,进入到抽象状态空间中去. 这个扩展使得 Abstract-Refine 方法成为了一个真正通用的方法,它的整个工作机制就是围绕程序状态空间进行上下逼近.

在 UnifiedARMC 中,原本运行在抽象域上的和运行在具体域上的算法,都被统一抽象成了策略参数 $\hat{S}_S$,是一组抽象状态集,实现了算法的模块化解耦. 这种解耦显而易见的好处是,算法之间的组合可以通过集合的相关运算来完成. 本文将在下一节通过实例来展示这种能力.

5 模块化嵌入方案设计和性能评价

本文所述的 Abstract-Refine 模型检测算法除了输入以外,只有两个附加参数,即抽象策略状态空间和精炼策略状态空间,分别记作 $\hat{S}_S^A$ 和 $\hat{S}_S^R$,也可以记 $\hat{S}_S = (\hat{S}_S^A,\hat{S}_S^R)$. 每个状态空间 S 都具有位置,变量,值的抽象层结构. 方便起见,可称其为位置层,变量层,值层,分别记作 $S.l$,$S.x$,$S.v$.

可嵌入谓词抽象方案. 谓词抽象的策略空间需要定义变量层和值层的抽象,而位置层则可以是全空间. 根据这个分析,程序 $P=(S,S_{op})$ 谓词抽象的策略空间 $\hat{S}_S$ 具有以下形式:

$$\hat{S}_S = \{(l_i,x_i,v_i)\} \cup \bigcup \{(S.l,x,S_{op}.v_x)\}$$

其中 (l_i,x_i,v_i) 可以不必特意设置,直接留空,也可以是程序的初始位置,或是其他的具体位置,这取决于程序检测的入口点如何设置. 当为 (l_i,x_i,v_i) 全空间时,检测的入口点是随机设置的. $x \in S.x$ 而 $S_{op}.v_x$ 表示 x 取值处的 v 的值抽象. 在实际的抽象模型检测算法中,$\hat{S}_S$ 是一个动态策略,也就是说随着模型检测算法的执行,策略会被动态填充. 可以看出,上面这个策略并非是动态策略,但是其效果却与动态策略等价. 如果算法 2 中谓词抽象扮演抽象算法的角色,那么 $\hat{P}=(\hat{S}\setminus S_r,\hat{S}_{op})$ 被替换成 $\hat{S}_S^A := \hat{S}_S^A \cup S_r$,则它就成为了一个动态策略的谓词抽象的版本. 因为两个版本的抽象状态在 AbstractMC 算法中会进行平衡性比较,而根据定理 2,这个比较对于差集运算是可以移项的.

可嵌入插值方案 以 Craig 插值法为例,对于两个状态子空间 (φ^+,φ^-),如果 $\varphi^+ \nhookleftarrow \varphi^-$ 无法被满足,那么它们的一个 Craig 插值 ψ 就是满足以下条件的状态子空间:(1)$\psi \nhookleftarrow \varphi^-$;(2)$\varphi^+ \nhookleftarrow \psi$ 不满足;(3)$\psi.x \subseteq \varphi^+.x \cap \varphi^-.x$.

Craig 插值实际上包含了两个平衡过程,它与抽象

的 Abstract-Refine 算法可能具有一定的相似性. 传统的 Craig 插值在实现中也包含两个过程,第一个过程是清理无用变量,也就是使其满足(3)的要求. 第二个过程是状态空间的推理,使其正好夹在(φ^+,φ^-)两个子空间之间. 考虑变量消除可以看作是在状态空间变量层上的平衡过程,Craig 插值的三个条件就变成了状态空间上的三个平衡过程. 由于变量消除过程与其它推理过程具有独立性,所以根据性质 2,这个过程可以与另外两个平衡过程合并进行. 最后,所求子空间 ψ 是另外两个子空间上下逼近的结果,所以 Craig 插值算法实际上就是一个 Abstract-Reine 算法.

对 Craig 插值的分析表明,在 Abstract-Refine 算法中,不仅抽象和精炼过程都可以统一表示为抽象过程,而且抽象过程本身也是 Abstract-Refine 过程的一个特例,即 Abstract-Refine 算法本质上是一个具备递归调用结构的算法. 这同时也表明,Craig 插值可以是一组方法类,它可以接受和 Abstract-Refine 方法一样的参数来调整算法的抽象和精炼策略. 这个思路未来可以引导对 Craig 插值具体实现的改进.

形式地,Craig 插值可以表示为如下结构:

$$(r,S_r) := AbstractMC(\hat{P},s_e,\hat{S}_S^A,\hat{S}_S^R)$$

其中典型的 $\hat{S}_S^A$ 是一个 SMT 求解器策略,而典型的 $\hat{S}_S^R$ 则是一个谓词抽象策略.

与传统硬编程的 Abstract-Refine 算法相比,本文中的方法具有两个明显的优点:首先,在传统 CEGAR 算法中抽象、反例寻找、精炼三种截然不同的算法被合并表述成了同一种算法. 将基本算法并行化,可以实现对多种特定程序结构的更广泛适应性,同时大大减轻了设计组合算法的难度. 其次,策略可以由算法自动生成. 通过策略的自动化生成,可以提高新算法设计中的策略自动化程度.

性能评价　本文实现的比较分为两部分,第一部分是传统的抽象方法和嵌入本文框架中的抽象方法的比较,第二部分是传统的 Abstract-Refine 方法和嵌入本文框架中的 Abstract-Refine 方法的比较.

本文的测试集选用了简单的 C 语言程序测试集,这些测试程序片段被分为几个类型,其中包括顺序程序 simple,分支程序 cond,数组相关程序 array1,array2,array3,以及循环程序 loop. 顺序程序反映了相应检测算法的一般性能. 分支程序反映了相应检测算法对于分支结构的检查能力. 本文的测试集总共包含 482 个程序片段. 测试结果是该组程序运行时间之和,其中 FP 是测试中出现假阳性测试结果. 本文测试实例运行的硬件配置为 8 核 3.4GHz 处理器、32G 内存、1T 硬盘空间,操作系统为 Ubuntu 18.04 x64,运行时环境为 Java 8、Clojure 1.8.

表 1　分立算法和模块化算法的性能比较

样本类型	传统抽象	传统抽象精炼	模块化抽象	模块化抽象精炼
simple	530s	386s	502s	391s
cond	146s	107s	151s	105s
array1	413s	340s	424s	348s
array2	495s	403s	500s	420s
array3	4926s	622s	5038s	639s
loop	FP	943s	FP	918s

表 2　分立算法和模块化算法的内存消耗比较

样本类型	传统抽象	传统抽象精炼	模块化抽象	模块化抽象精炼
simple	2.93G	2.13G	3.37G	2.46G
cond	580M	577M	574M	585M
array1	3.42G	3.25G	3.78G	3.69G
array2	5.01G	4.82G	5.24G	5.11G
array3	13.0G	9.75G	14.2G	10.5G
loop	FP	16.8G	FP	16.3G

本文中的测试结果为各测试集中代码片段执行资源消耗的算数平均值,其中空间消耗取三个有效数字. 在这个测试中,从表 1 可以发现本文提出的模块化的 Abstract-Refine 框架并不会对传统方法的时间性能造成冲击性的影响. 实际上在某些案例中,它的性能还微幅好过传统方法. 而通过表 2 可以发现,本文中的方法在空间效率上略微落后,经过分析,这主要是由于模块化算法在实现中使用了较多的空间分配操作,而 JVM 在自动垃圾回收过程中会有延时而造成的. 这个测试对软件模型检测实践中基本的和具有代表性的两种方法进行了新旧比较,可以预期基于这两种方法的其他方法都会有相似的性能表现.

6　总结

本文提出了一种新的抽象程序的描述方法,更细粒度地揭示了抽象程序的内部结构和规律,并在这个基础上提出了 Abstract-Refine 方法的模块化算法结构框架. 该框架实现了对此类方法的统一表示,和参数化的模块替换. 不同的算法可以通过框架实现部分或者整体的交换和嵌入,从而使得新方法的提出变得更加简单. 本文设计了谓词抽象和 Craig 插值的具体模块化方法,并通过实验测试了其在模块化方法上的性能表现,证明了本文方法并不会对已有方法性能造成冲击. 本文方法揭示的程序检测的潜在的递归结构表明了软件模型检测可能也具有类似结构,可在未来做进一步的研究.

参考文献

[1] ALBARGHOUTHI A,GURFINKEL A,CHECHIK M. From under-approximations to over-approximations and back[A]. Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)[C]. Berlin:Springer,2012. 157 – 172.

[2] BIERE A, CIMATTI A, CLARKE E M, et al. Bounded model checking [J]. Advances in Computers, 2003, 58 (11): 117 – 148.

[3] FLANAGAN C, QADEER S. Predicate abstraction for software verification [A]. ACM SIGPLAN Notices [C]. New-York: ACM. 2002, 37(1): 191 – 202.

[4] RAJAMANI S K, GULAVANI B. Counterexample driven refinement for abstract interpretation [P]. US Patent 7509534. 2009 – 3 – 24.

[5] 魏欧,石玉峰,徐丙凤,等. 软件模型检测中的抽象模型研究综述 [J]. 计算机研究与发展, 2015, 52 (7): 1580 – 1603.
WEI O, SHI Y, XU B, et al. Abstract modeling formalisms in software model checking [J]. Journal of Computer Research and Development, 2015, 52 (7): 1580 – 1603. (in Chinese)

[6] SHVED P, MANDRYKIN M, MUTILIN V. Predicate analysis with BLAST 2. 7 [A]. Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012) [C]. Berlin: Springer, 2012. 525 – 527.

[7] BALL T J, BOUNIMOVA E O, LEVIN V A, et al. Program analysis through predicate abstraction and refinement [P]. US Patent 8402444. 2013-3-19.

[8] FRIEDBERGER K. CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis [A]. Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016) [C]. Berlin: Springer, 2016. 912 – 915.

[9] WONISCH D. Block abstraction memoization for CPAchecker [A]. Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012) [C]. Berlin: Springer, 2012. 531 – 533.

[10] MCMILLAN K L. Interpolation: Proofs in the service of model checking [A]. Handbook of Model-Checking [M]. Berlin: Springer, 2014.

[11] FEDYUKOVICH G, HYVARINEN A E J, SHARYGINA N. Interpolation-based model checking for efficient incremental analysis of software [A]. Proceedings of IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS 2013) [C]. IEEE, 2013. 8 – 9.

[12] VIZEL Y, GURFINKEL A, MALIK S. Fast interpolating BMC [A]. Proceedings of International Conference on Computer Aided Verification (CAV 2015) [C]. Cham: Springer, 2015. 641 – 657.

[13] BEYER D, LOWE S. Interpolation for value analysis [A]. Proceedings of Software Engineering & Management [C]. Bonn: Gesellschaft für Informatik, 2015. 73 – 74.

[14] 段钊,田聪,段振华. 基于 CEGAR 的 C 程序空指针解引用检测 [J]. 计算机研究与发展, 2016, 53(1): 155 – 164.
DUAN Z, TIAN C, DUAN Z. CEGARbased null-pointer dereference checking in C programs [J]. Journal of Computer Research and Development, 2016, 53(1): 155 – 164. (in Chinese)

[15] CIMATTI A, GRIGGIO A. Software model checking via IC3 [A]. Proceedings of International Conference on Computer Aided Verification (CAV 2012) [C]. Berlin: Springer, 2012. 277 – 293.

[16] BEYER D, LOWE S. Explicit-state software model checking based on CEGAR and interpolation [A]. Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE 2013) [C]. Berlin: Springer, 2013. 146 – 162.

[17] Beyer D, Löwe S. Explicit-state software model checking based on CEGAR and interpolation [A]. Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE 2013) [C]. Berlin: Springer, 2013. 146 – 162.

[18] BEYER D, KEREMOGLU M E. CPAchecker: A tool for configurable software verification [A]. Proceedings of International Conference on Computer Aided Verification (CAV 2011) [C]. Berlin: Springer, 2011. 184 – 190.

作者简介

王 舜 男,1988 年生,河南洛阳人,北京交通大学博士生,主要研究方向为形式化方法、程序分析技术、信息安全等.
E-mail: shunwang@ bjtu. edu. cn

杜 晔 男,1978 年生,黑龙江哈尔滨人,北京交通大学教授、博士生导师,主要研究方向为网络安全、态势感知、软件可靠性分析与评估等.
E-mail: ydu@ bjtu. edu. cn

韩 臻 男,1962 年生,浙江宁波人,北京交通大学教授、博士生导师,主要研究方向为可信计算、系统安全、保密技术等.
E-mail: zhan@ bjtu. edu. cn