基于程序局部性引导的有界模型检测优化方法

王舜，杜晔，韩臻，刘吉强
（北京交通大学智能交通数据安全与隐私保护技术北京市重点实验室，北京 100044）

摘 要：基于多种模型检测方法组合的复合检测方式是当前软件模型检测领域开展研究的热点之一。在当前的研究中，提高检测的规模和检测的对象复杂程度的关键在于如何有效处理抽象的扩张和收缩。证明通过对程序模式或验证信息的利用可以加快状态空间的探索速度。面向有界模型检测（BMC）加速方法展开研究，使用程序中额外的信息和知识对其处理以协助检测器删除冗余和无效的状态。在对程序局部性进行定义的基础上，对其加速性进行讨论，提出一种加速有界检测的方法和一种改进策略，对算法进行了详细描述，并通过实验验证了方法在检测效率和性能上的优越性。

关键词：模型检测；BMC；软件检测；局部性；优化
中图分类号：TP311.1
文献标识码：A
doi: 10.11959/j.issn.1000-436x.2018050

Locality-guided based optimization method for bounded model checker

WANG Shun, DU Ye, HAN Zhen, LIU Jiqiang
Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China

Abstract: For software model checking, approaches that combine with different kind of verification methods are now under research. The key to improve scale and complexity of verifiable software is handling the method for abstraction widening and strengthening wisely and precisely. To archive that, using extra knowledge that extracted from programming pattern or learned through verifying procedure to help eliminate the redundant state has been proved effective. Definition of program locality was given. It took the important role in accelerating software verification, then the strategy was raised and an algorithm was implemented to take advantage of program locality. This method exploits the features of modern BMC (bounded model checker) and scales up the capability of its power in large scale and comprehensive software modules.

Key words: model checking, BMC, software verification, locality, optimization

1 引言

软件模型检测是提高软件可靠性的一种非常重要的方法。在工业环境中，一些对于软件可靠性要求高的安全苛求系统已大量地使用这种方法。软件模型检测是基于程序的形式化理论发展出的一种技术，它的基本思想是对当前需要检测程序的整体状态空间进行遍历，从而寻找其中是否包含问题状态。对于使用基本的遍历思想来检测程序的方法来讲，状态空间爆炸使它在很小的程序上都会耗尽其所有的运算和存储资源，因此，这种检测方法并不具备成规模的实用性。

针对上述问题，现代的软件模型检测方法使

收稿日期：2017-06-14；**修回日期**：2018-01-11
基金项目：北京高校青年英才计划基金资助项目（No.YETP0548）；国家自然科学基金资助项目（No.61672092）
Foundation Items: Beijing Higher Education Young Elite Teacher Project (No.YETP0548), The National Natural Science Foundation of China (No.61672092)

用了多重的优化策略来缓解，这些策略可以按照枚举型和推理型来进行分类[1,2]，2 种类型分别体现了具象和抽象的逻辑思想。而随着研究的进展和深入，很多新的研究使用了同时综合 2 种方法特征的混合策略。特别是随着近年来各种各样的方法被陆续提出，研究者逐渐发现，区分和指导各种方法的根本不同在于如何使用源代码中的信息来引导状态空间的搜索。当然与之相对地，还可以通过表象对软件模型检测的方法进行分类，例如，模型检测算法是否有界。

有界模型检测器（BMC）是一种常见的和使用广泛的软件模型检测工具。在近几年的相关研究中，已经有不少基于其思想的成熟的实现，如文献[3~7]。使用有界模型检测器对程序进行检测的实践可能会因为使用了不合适的参数而导致检测无法完成或检测失败。同时，由于其原理的制约，这种检测方法也无法直接对整个程序的状态空间进行检测。从这个角度来讲，阻碍有界模型检测对大规模代码进行检测的障碍之一在于缺乏一种有效的将代码分解为可以检测的片段并且自动引导检测器来进行检验的方法。

SAT 求解问题是另一个更加基础的研究领域，需要求解的 SAT 问题同样是类似的会引起状态爆炸的 NP 完全问题。在这个研究领域中，前向—后向搜索是常用的加速求解的方法之一，而最为经典的 SAT 前向—后向搜索算法的实现则是 DPLL 算法[8]。这种算法的核心思想是，在遍历搜索时记忆和学习关于逻辑树的结构信息，并使用这个信息来指导新一轮的搜索。在整个 SAT 求解器的研究领域中，大量的优化方法都采用了类似的思路为指导，也就是使用在有限的搜索中发现的信息来缩减后续的搜索树，或选择最优的后续搜索。基于先验信息的启发式算法[9~12]是另一类常用的优化方法。与前一类方法相比，它最大的不同是并不能保证总是有效。但是相对地，启发式算法在状态空间的规模和逻辑复杂度快速增长的情况下是一种更有潜力的方法。

最后，程序在设计和运行中具有一个非常本质和重要的特征，即程序的局部性。在调研中，当前模型检测领域并没有相关研究聚焦在程序的局部性和模型检测的优化结合以及程序的局部性对模型检测加速的可能性上。因此，本文主要研究聚焦以上层面。首先，本文给出了一种可能的程序局部性形式化定义，然后，设计了一个策略来对程序进行划分，并提出了一种算法自动化地使用这种划分来指导有界模型检测器的自动运行，最后，通过实验验证了本文方法在加速有界模型检测器对程序的检测上具有较为明显作用。

2 相关工作

近年来，大量的研究关注和聚焦了模型检测和模型检测的相关算法和应用。其中，基于逻辑的枚举和基于逻辑的推理是 2 种重要的类型。在这些研究中，最为活跃和引人注目的部分集中在以下 2 个方向：1) 基于推理的模型检测方法，包括基于符号演算的模型检测方法；2) 不同模型检测方法的可组装性的研究。

其中，第一类研究方向使用归结原理和插值规则通过定理证明的方法[13~15]来进行模型检测。另外，还有一些研究设计并实现了一类框架[16~19]，并在这类框架下实现了基于前向—后向搜索的遍历式模型检测方法。这类方法的基本思想是将目标状态空间简化为一类抽象状态空间，并在其上进行检测。在这类框架之上，文献[20,21]设计了一种懒惰方法，它是针对状态展开的优化，使算法在展开状态时不需要一次性完全展开，而是按照需要展开必要的层数，从而避免状态空间一次性展开过大。最后，文献[22~24]的研究则着眼于设计和实施一套全新的软件模型检测方法。

第二类研究方向中包含了一类构建通用的模型检测方法基础的理论以及其相关的实现，它们都基于一套可以嵌入和组合大量不同方法的基础框架之上。文献[25~30]是其中非常重要的组成部分。它们建立了一个称为 CPA 的模型检测框架。这个框架实际上是一个基于格的状态抽象，在这个抽象之上，使用融合、加强以及停止运算符来将各种算法组合在一起。除了这些工作以外，文献[31]使用了另一种上近似和下近似结合的方法来对不同的模型检测方法进行组合。

本文方法是一类组合方法，但是其手段并非直接组合已有的方法，而是结合第一类的研究和第二类的研究，提出一种基于新的局部性启发式检测加速方法。

3　基本模型

3.1　程序模型

为了简化问题的分析和讨论，本文使用一个简单程序模型来表示需要检测输入的程序抽象。这个模型是一个基于简单命令式程序语言的扩展，使用这个模型可以描述 C 语言的逻辑结构，而其他的细节则直接交给 BMC 工具处理即可。

一个简单命令式程序包含了一系列的操作符，这些操作符包括：赋值操作符、假设操作符、整形变量。简单命令式程序的描述细节可以参考文献[1]。这里对简单命令式程序进行一个扩展，使其可以满足对本文形式化描述的要求，其包含了一些额外的程序结构：函数、分支、循环。它可以通过分支循环展开和函数嵌入的方式很容易地转换回经典的简单命令式程序。实际上，可以通过这个扩展给基础定义添加一些高级结构。

定义 1　简单命令式程序（SIP）。一个简单命令式程序是一个四元组 $P=(X,L,l_0,T)$，其中，包括了一个有类型的变量集合 X，一个程序的位置点集合 L，一个程序入口点 $l_0\in L$ 以及一个程序转换关系集合 T。在转换关系集合中，每个转换关系 $\tau\in T$ 都是一个三元组 (l,ρ,l')。在这个三元组之中，$l,l'\in L$ 是程序的位置点，ρ 是一个在 $X\cup X'$ 上的自由变量的约束关系。其中，X 是在程序位置点 l 处的变量集合，X' 是程序位置点 l' 处的变量集合。在这个定义之下，程序的位置点集合和转换关系集合自然地定义了一个有向带标图。这个图称为该程序的控制流图（CFG）。

一个带条件语句的命令式程序可以转换为一个简单命令式程序，这个转换可以通过合取条件变量 p（以 X 中的元素为变量的布尔表达式）与赋值表达式来获得。

$$p\wedge\bigwedge_{x\in X}x'=x$$

循环语句的处理比条件语句要稍微复杂一些。首先对需要转换的循环语句设定一个展开层数限制，然后展开这个循环为顺序程序。如果循环在到达限制前没有展开完毕，则停止在展开限制处（图 1 为一个简单的例子）。这个展开实际上是一种懒惰（lazy）的循环展开方式。使用这种方法可以在一定程度上阻止可能的状态空间发生爆炸问题。而在循环展开以后，可能会遗留下一个未被展开的循环体，这个尾巴可以被表示为一个特殊的节点，它被标记为尚未被转换和翻译。这也是一个在懒惰策略中常用的技巧。函数的翻译和循环的翻译类似，但是更为简单直接。一个函数既可以被展开成一段平坦的程序片段，同时也可以不翻译。在形式化的语言表达中，可以采用一个函数集合 $\mathcal{F}_{tr}$ 去统一地表示 2 种翻译方法。

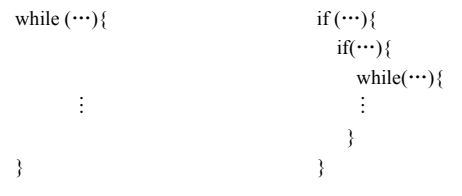


图 1　程序的循环展开示例

上面所述的描述方式可以表示一个具体的程序。这个具体的程序模型描述了一个程序精确的运行特性，但是它的状态集通常都十分巨大甚至难以使用有限的描述来表示。如果使用这个较为接近底层的模型去指导一个状态空间的搜索，难免会遇到困难和失败。因此，需要一个更加抽象的模型去描述一个更小以及更有限的状态空间。

3.2　抽象可达图

抽象程序模型可以用一张描述程序位置点的可达图来表示。依照上文定义，将程序记作四元组 $P=(X,L,l_0,T)$，使用 err 表示这个程序的错误点。

定义 2　可达图（RG）。程序 P 的可达图是一个五元组 (V,E,v_0,ν,τ)。其中，三元组 (V,E,v_0) 表示一个有向无环图（DAG）。入口点是 $v_0\in V$，而映射 $\nu\to L$ 将节点映射到程序 P 的位置点，这个映射可以表示为 $\nu(v_0)=l_0$。另外一个映射 $\tau:E\to X$ 将边映射到了程序 P 的动作集。

定义 3　抽象可达图（ARG）。一个抽象可达图是一个四元组 $(U,\psi,\subseteq,\subseteq_t)$，其中，$U=(V,E,v_0,\nu,\tau)$ 是它关联的程序 P 的可达图。ψ 是从节点集 V 到公式集 X 的映射，$\subseteq$ 是节点集 U 上的父节点关系。如果 $\nu(v)=err$，那么节点 v 称为错误节点。

3.3　有界模型检测

有界模型检测（BMC）只检测 k 步以内的可达性，其形式化的表达如下。

$$BMC_0^k=I\wedge\bigwedge_{i=0}^{k-1}T\wedge\neg P$$

其中，I 表示初始状态集合，T 表示程序的转换关系集合，前方下标是转换关系索引，P 表示程序不变的属性集合，这些属性需要在程序的整个执行期中保证不被违反。一个有界模型检测方法的工作过程就是对这个命题进行验证，证明它恒为真或找到一个使它为假的证明。如果一个有界模型检测方法找到了一个使命题为假的证明，这个证明肯定可以应用在原始的程序中，使程序也有一条相应的在这个赋值下的路径，这条路径通常被称为反例。如果反例存在，那它一定是充分的。但是当这个有界模型检测方法给出了一个恒为真的证明时，这个证明却不足以说明整个原始程序的安全是充分的。所以，有界模型检测方法是一种下近似的验证方法。这样的有界模型检测可以保证整个验证过程一定能在一定步骤后停止，无论它检测的程序是什么结构。因此，这种方法配合其他一些优化方法对于检测实用的程序非常有效。

有界模型检测的基本算法框架如算法 1 所示。这种方法需要配合使用一个 SAT 或 SMT 求解器来作为内部的核心组件。一个基本的有界模型检测算法也可以看作是符号模型检测算法添加一个界限的产物。

算法 1 有界模型检测的基本算法

输入 简单程序 $P=(X,L,l_0,T)$，错误点 $\mathcal{E}\in L$，检测边界 k

输出 如果程序安全则输出 SAFE，如果程序检测逾越边界则输出 UNKNOWN，否则输出 UNSAFE

函数 BMC($P,\mathcal{E},k$)：
可达集设置为 $(l_0,\varnothing)$
工作队列设置为 $(l_0,\varnothing)$
步数设置为 0
while 工作列表非空 do
　从工作列表中弹出元素 (l,r)
　if r 不在 l 的可达集合中 then
　　将 r 加入 l 的可达集中
　　步数加 1
　　if 步数大于 k，则返回 UNKNOWN
　　for $(l,\rho,l')\in T$ do
　　　为工作集添加元素的像
　if 可达集在错误点非空，则返回 UNSAFE
　否则，返回 SAFE

如果把有界模型检测算法看作是对程序模型的树型遍历，那么很显然，这种方法使用的是深度优先策略。

4 程序局部性定义与局部性模型描述

局部性又被称为参考局部性或者局部性原理，是计算机工程领域的一个重要的原理。这个原理描述了计算机系统在数据访问和程序执行的过程中会大概率地优先访问或者执行上一个访问位置附近的相关数据，对于指导计算机工程上的很多问题的优化方案有重要的作用。在局部性中，2 种重要的类型分别是时间局部性和空间局部性。时间局部性描述了在一段连续的时间片段中，特定的数据或位置会被频繁地访问。相似地，空间局部性描述了数据的访问会经常地发生在相距很近的地方。在很多启发式算法中，局部性是指导优化内在原理。

当前的程序设计语言给予了程序设计人员强大的能力，这个能力体现在程序中就是程序的多维度上的自由度。具体来讲，例如，在面向对象的程序设计（OOP）范式中，2 个重要的自由度是继承和包含关系。另一个重要的例子是，在堆栈式语言编码的程序当中，2 个重要的维度是深度和广度。程序的局部性在这 2 个例子中就可以体现在它们的自由维度上。在每一个维度上，程序本身都具有依赖这个维度的局部性。形式化地，这些局部性可以体现在相关程序的语法和语义的邻接关系上。在这里将要给出一个基于扩展版本的简单命令程序的局部性的定义。首先，需要先介绍几个前置概念。

定义 4 分层 ARG。程序 P 的一个分层 ARG 是一个二元组 $(\mathcal{U},\mathcal{N})$，其中，$\mathcal{U}$ 是程序 P 的抽象可达图，$\mathcal{N}$ 是在 $\mathcal{U}$ 上定义的节点集合，当且仅当 $n\in\mathcal{N}$ 时，有 $\exists v_f\in n,\ v_f\subseteq v_{\text{anc}},v_{\text{anc}}\notin n$ 且 $\forall v\in\{v_f\}\setminus n,v\subseteq v_{\text{anc}},v\in n$。

在分层抽象可达图中，节点的有向连接体现了程序执行序列的方向。实际上，分层抽象可达图是一个二维视角下程序流语义的展现。

在分层抽象可达图中，量化的距离定义不妨借鉴经典的距离表达方法，即曼哈顿距离来表示。在这个前提下，可以进一步定义一个曼哈顿抽象可达

图为一种分层抽象可达图，其中，每一层都是一个由分支、循环或函数节点展开得到的子抽象可达图。在曼哈顿抽象可达图中，每一层的子分支，循环和函数都默认不被展开。在这个抽象可达图上，可以比较方便地定义距离，同时，程序语法和语义上的信息也被有效地保留了下来。

定义 5　曼哈顿 ARG。可达图 $U_{\mathrm{AL}}=(\mathcal{U},\mathcal{N})$。其中，如果 $\forall n\in\mathcal{N},v\in n,v\sqsubseteq v_{\mathrm{out}},v_{\mathrm{out}}\notin n$，那么 v 和 v_{out} 中间的边 e 满足 $\tau(e)\in\mathcal{F}_{\mathrm{tr}}$。

简单命令式程序的局部性是程序曼哈顿抽象可达图上使用边的权重定义的曼哈顿距离。可以为这个抽象可达图上的每一条边设定一个权重，即 $\omega(e)$，当 $e\in E$ 时。而一条在节点 v_1、v_2 中间的路径 $path(v_1,v_2)$ 的曼哈顿距离就可以定义为连接这 2 个节点的边的权重之和。

定义 6　程序局部性度量。程序的局部性度量是这个程序的曼哈顿抽象可达图上 2 个节点 v_1、v_2 的曼哈顿距离。

$$\sum_{e\in path(v_1,v_2)}\omega(e)$$

假设 1　一个程序的执行路径会有较大概率选择局部性强的路径。

本文给出的假设 1 是合理的。首先，其符合程序员的编程直觉。使用局部性较强的路径更加容易设计逻辑连贯的程序，因为在处理逻辑的连接时，需要进行的额外记忆将会降低。其次，局部性符合计算机体系结构的基本原理，程序在执行时选择局部性较强的路径会节约计算机运算的时间资源和空间资源。

显然，函数 ω 是影响程序局部性度量的一个关键因素。另一个关键因素是如何处理 $\mathcal{F}_{\mathrm{tr}}$ 使 BMC 可以在其上运行。

对于一个具体程序的抽象可达图，符合假设 1 的 ω 函数是简单的。只需要给每一条边一个相同的单位权重就行了。这类赋值是符合一般的直觉判断的，即看起来的距离越近，相关性越强。但继续深入地分析就会发现，程序中任意 2 个位置间的程序局部性连接并不是精确地反映在 2 个位置中间的边的数量。这个相关性实际上是依赖于 2 个位置间各个变量的因果联系。如果某个位置上并没有任何变量，那么这个关系会连接在通向这个位置的控制变量上。对于一个更加复杂的情形，例如，程序的抽象可达图包含了有很多具体转换连接的抽象节点。在这种情形下，边的权值显然和最简单的版本的计量方法不同。即便如此，依然可以将多个边的连接看作是一种抽象的边，这样的话，这条边的权值可以用其包含的所有具体边权值的和来表示。

为了将程序的局部性性质应用到传统的有界模型检测算法中去，原始的算法需要进行一定的扩展，从而使它可以接受和处理现代程序设计语言中蕴含的特定的因果关系。具体地，就是让原始的有界模型检测算法可以接受和处理来自广度方向上的局部性信息。有界模型检测算法是一个典型的深度优先搜索算法，这个算法只接受具体的符号化程序作为输入，然后按照这个程序的运行路径进行搜索和计算。综上所述，本文的程序局部性的定义是包含抽象状态的，因此，有界模型检测算法并不能直接应用在其上。特别地，如果搜索方向是广度优先，那么在执行过程中将会有很大概率碰到抽象状态。所以需要让有界模型检测算法可以处理一些抽象状态。

不翻译函数（uninterpreted function）是 SAT 求解领域里的一个重要的理论模型。这个模型扩展了 SAT 求解器的能力，使它可以处理的逻辑范围不仅限于命题逻辑[32]。不翻译函数的基本理念是使用一个不展开的符号来表示一组命题，而这组命题中的符号被当作这个符号的参数来处理。这样一来，这个不展开的符号就符合了一般函数的定义，而它的值就是这组命题计算式的合取。这种表示方式可以跳过对这组计算式的原地求值（on-site evaluation），从而形成一个懒惰求值的节点。因此，可以仿照不翻译函数的形式逻辑来表示本文的抽象可达图中的抽象部分。

定义 7　不翻译函数的扩展逻辑。一个定义了不翻译函数的扩展逻辑包含以下语法。

$$form:form\wedge form\,|\,\neg form\,|\,(form)\,|\,atom$$

$$atom:term\ rel\ term\,|\,predicate\_symbol(list\_of\_terms)$$

$$term:ident\,|\,function\_symbol(list\_of\_terms)\,|\,list\_of\_terms$$

$$rel:=|>|<|\geqslant|\leqslant$$

这种扩展逻辑利用了不翻译函数的能力并且可以处理多重返回值以及各种各样的二元关系运算符。它可以将程序片段转换成逻辑公式使特定的 SAT 或 SMT 求解器得以求解。对应地，它也可以将程序变为可被 BMC 接受的抽象程序。

5 局部性引导的 BMC 算法

本节将会讨论如何使用局部性来指导有界模型检测算法。先从设计和实现一个简单的、使用局部性策略的算法开始，接着讨论一个添加了更多优化的版本。

5.1 基础算法

直接使用程序的局部性来指导有界模型检测算法可以通过计算目标程序的 2 个节点间的局部性度量来实现。这个想法相对比较易于实现，同时又易于理解。通过搜索当前节点到错误点的局部度量可以获得一个整体局部性。反复进行搜索便可以得到一个迭代更新的局部性参考。

与原始的有界模型检测算法类似，每一轮有界模型检测算法的运行并不是完全覆盖整个程序的状态集。但对比传统的算法，本文算法可以降低在深度或广度方向上的运行消耗。另外，将一个大型的有界模型检测搜索拆分成数个较小检测可以增加检测的灵活性，使检测可以在有限的空间下对更大型的状态空间执行搜索。

还有一个需要讨论的非常重要的因素是何时来切换深度搜索和广度搜索。这里使用一个阈值 t_s 来控制这 2 种搜索的切换。这个阈值可以通过计算深度方向的局部性度量和广度方向的局部性度量的商来得到。具体如算法 2 所示。

函数 *CHKSD* 是递归逐层检测程序的抽象可达图的方法。其他一些参数的定义如下。

larg 是分层抽象可达图的有序列表。

root 用来获取抽象可达图的根元素的函数。

nextstep 是一个获取目标程序后置操作的迭代器。

parent 是一个获取抽象可达图中父节点的函数。

child 是一个获取抽象可达图中子节点的函数。

abstract 是一个使用扩展逻辑将节点变为对应的抽象节点的函数。

next 是一个获取分层抽象可达图当前层的节点的迭代器。

concrete 是一个从目标抽象可达图的节点上获取其所包含的具体状态的函数。这些状态包括所选择节点在深度或广度方向上的状态。如果深度或广度方向没有在参数上被指定，那么就直接获取节点在所有方向上的具体状态，这时，这个函数与 *abstract* 函数互逆。

count 是一个统计集合内元素个数的函数。

related 是一个获得节点和路径间具有因果关系的所有节点的集合。

last 是获取当前节点沿特定方向到错误点的路径。

算法 2 基础算法

输入 扩展的简单程序 $P=(X,L,l_0,T)$，错误点 $\varepsilon \in L$

输出 如果程序安全则输出 SAFE，否则输出 UNSAFE

函数局部引导模型检测(P,ε)：
初始化结果为 UNKNOWN，抽象状态 *as* 为空
调用函数创建分层 *ARG*，赋值给 *larg*
while 结果是 UNKNOWN do
    调用函数 *CHKSD*(*larg*,*root*(*arg*), ε)并把结果传递给检测结果和 *as*
    for 对于集合 *as* 的元素 do
        对元素调用函数 *concrete*
返回结果
函数创建分层 ARG(*P*):
初始化 *larg* 为 *root*
for 对于程序 *P* 执行的各步骤 do
    if 该步骤符合曼哈顿条件 then
        调用函数 *child*(*larg*, *parent*)并给其结果赋值为该步骤调用 *abstract* 函数的返回值。
    否则，调用函数 *next*(*larg*)并为其结果赋值为该步骤调用 *abstract* 函数的返回值
函数 *CHKSD*(*larg*, *v*, ε)：
调用函数 *CALCL* 分别计算 *v* 广度方向上的距离和深度上的距离，并求二者之差 t_s
设置探索方向为深度
if t_s 大于 1 then
    设置探索方向为广度
调用函数 *BMC*(*concrete*(*v*,当前探索方向), ε, 1)并赋值给结果
if 结果是 UNSAFE then 返回(UNSAFE, ϕ)
else if 结果是 UNKNOW then
    初始化 *sresult* 和 *sresults* 为空
    for 集合 *child*(*larg*, *v*)的元素 do
        调用函数 *CHKSD*(*larg*, 当前元素, ε)并将结果赋值给 *tresult* 和 *sresults*
        if *tresult* 取值为 UNKNOWN then 将 *sresult* 合并入 *sresults*
返回(UNKNOWN, *sresults*)

否则，返回 SAFE

函数 *CALCL*(*v*, *dir*):

if 方向 *dir* 是深度方向 then

返回 *count*(*related*(*v*, $last_b(v)$))

否则 返回 *count*(*related*(*v*, $last_d(v)$))

5.2 改进算法

基于上文所述的基础算法（算法 2），可以进一步改进来获得一个更加优化的新算法。这个算法利用了有界模型检测过程所生成的反例来加强抽象条件的准确性。函数 *CHKSD* 生成的内层函数的反例可以就地加强外层函数的抽象。这样做可以降低重复调用有界模型检测所造成的重复检测的额外开销，同时使抽象信息的发现和使用更加高效。

算法 3 是改进算法的详细描述。在改进算法的描述中，略去与基础算法中一致的部分。需要说明的是，*getcounterexample* 是一个获得有界模型检测器证明 UNSAFE 时的反例详细信息的函数。

算法 3　改进算法

输入　扩展的简单程序 $P=(X,L,l_0,T)$，错误点 $\varepsilon \in L$

输出　如果程序安全则输出 SAFE，否则，输出 UNSAFE

函数局部引导模型检测(P,ε)：

调用函数创建分层 ARG(*P*)并将结果赋予 *larg*

返回 *CHKSD*(*larg*, *root*(*larg*), ε)

函数 *CHKSD*(*larg*, *v*, ε)：

调用函数 *CALCL* 分别计算 *v* 广度方向上的距离和深度上的距离，并求二者之差 t_s

设置探索方向为深度

调用函数 BMC(*concrete*(*v*,当前探索方向), ε_1 ,1)并赋值给结果

if 结果为 UNSAFE then

调用函数 *getcounterexample*(*concrete*(*v*,探索方向), ε)并用返回值初始化 *ce*

返回 (UNSAFE, *ce*)

else if 结果为 UNKNOWN then

初始化 *sresult*，*sresults* 为空

初始化 *tresult* 为真

for 集合 *child*(*larg*, *v*)的元素 do

调用函数 CHKSD(*larg*, 当前元素, ε)并将结果赋值给 *tresult* 和 *sresults*

if *tresult* 取值为 UNSAFE then

将 *sresult* 合并入 *sresults*

返回 BMC(*concrete*(*v*, *sresults*), ε)

否则 返回 SAFE

在整个改进的算法中，参考了反例引导的抽象—精炼模型检测方法（counterexample-guided abstraction-refinement）中的反例引导思想。通过提取每次有界模型检测器获取的一个证明来加强算法中的局部性寻找，从而使局部性引导过程的搜索空间可以被有效限制，并且避免了一些重复的状态遍历。这种做法可以加强在局部性寻找失败时的算法效率。

6 实验设计与性能分析

本文方法使用有界模型检测工具作为内部的实现核心，这个工具本身具有独立地检测软件可靠性的能力。综上所述，本文的目标是针对传统的有界模型检测工具的一个改进和优化。将优化方法实施在一个现有的有界模型检测器上而不是使用一个定制的检测器具有多重优势。其中之一是，现有的模型检测器除了实施基本的有界模型检测算法以外，同时还考虑了多种与模型检测和程序分析相关的优化措施。使用现有的模型检测工具可以充分地利用领域里已经实施的相关优化方法，从而可以获得更佳的性能表现。另外一点是，使用现有的工具可以充分利用其对于程序的表达和处理的能力，从而可以使方法能够应对更多样和复杂的样本。

通过对现有各种有界模型检测工具进行考察发现，其中很多工具并不能很好地分析和处理实际的代码片段，例如代码具有复杂的循环结构、跳转、指针或内存操作。在部分有界模型检测工具中，对这些结构的检测体现为直接跳过或出现误报或者提示错误并拒绝，因此本文实验还需要对测试样本进行筛选，以去除不良用例。

实验使用了来自 SV-COMP 和其他项目中的工业代码作为测试用的样本。这些样本中包含了规模不一的代码片段。其中，最长的代码片段长度为 1 762 行，最短的片段长度为 341 行。使用这个长度范围内的片段是因为短于这个长度的片段有界模型检测工具可以很快地完成代码的检查，无法区分本文所述方案的优越性；而长于最长片段的代码将会导致内部的有界模型检测工具出现错误中止，从而使比较产生系统性误差。在测试样本中，除了代码长度不同，代码的类型也有所区别。样本

中的代码包含顺序代码、分支代码、循环代码、跳转代码、指针及内存操作代码。所检查的程序安全性属性包含可达、不可达、控制逻辑以及一些访问相关的属性。

大多有界模型检测器最大的问题都是对指针和内存的支持不完整，导致实际生产中的代码难以进行有效的检测。通过大量的研究和测试发现，LLBMC[33]对于工业代码的支持最为完整，同时因为其使用了显式内存分析，它对代码规模的支持也很强大。同时根据 SV-COMP[34]的测试结果显示，其在整个模型检测业界也具有很好的性能表现。基于这些研究，在测试中使用 LLBMC 作为算法中有界模型检测器的实现。

本文实验平台配置如下。Intel Core 2，4 核 8 线程处理器，64 GB 物理内存以及 64 位 Ubuntu 12.04 操作系统。保留测试代码中的各种瑕疵以及不好的编程范式，从而使其更加贴近真实的生产中会产生的代码。同时，也使程序员在编写代码时遗留在程序中的局部性得以保留。实验分为 2 个部分，第一个部分将实验样本按照代码规模从低到高分为 5 个阶梯。因为代码的行数与其复杂程度具有一定的关系但又不是严格的线性关系，使用这种阶梯式的分类方法可以更好地体现出代码复杂程度的演进。第二个部分将代码按照其逻辑类型分为 4 个大类。使用这个分类的目的是观察程序的逻辑类型对算法的影响。

本文实验并未涉及比较算法在空间上的复杂度差异，其原因是本文方法涉及多次使用检测工具，其内存消耗包括了多次的预分配和其他支持库的共享部分与单一使用不具备可比性。同时，本文用以比较的其他算法来自不同框架，其本身内存分配和回收机理不同，也会造成结果的说明性不佳。

实验使用的测试样本代码总共有 115 段，来自于其他测试样例和一些实际项目中的典型代码片段。在实验的第一个部分中，按照大约 200 行代码为一块对这些代码进行分片。

单独使用 LLBMC 以及 2 种算法在这些分片上运行的时间消耗比较如表 1 所示。其中所列的时间是代码检测的平均时间。通过比较可以发现，本文所述的算法在规模较小的程序检测中体现出更高的时间消耗。这是因为本文方案对代码进行了多次扫描，这种扫描造成了潜在的时间消耗。同时，本文使用的方法具有一个启动过载，这个因素同样影响了时间消耗。在规模较大的代码检测中，本文方法展现出了优势。

表 1　　算法时间消耗比较

样本平均数	算法时间消耗/s		
	LLBMC	基础算法	改进算法
533	611	735	708
729	971	832	820
1 002	3 613	3 570	3 439
1 291	N/A	3 902	3 219
1 347	N/A	6 037	5 791

第二部分实验结果如表 2 所示。其中，数据代表当前类别在所列算法的检测中成功返回并返回正确结果的数量。在前面实验中已经发现当代码规模增大到一定程度时，传统的有界模型检测器已无法完成针对代码的检测。因此也将代码规模限制在这个边界的附近，从而可以获得具有比较意义的结果。在第二部分的实验中也可以发现，LLBMC 在各个类型的代码中均有检测失败的案例。由于给 LLBMC 设置了无限长的代码展开，所以这种检测失败基本上都是由于状态空间过大所引起。本文的 2 种方案在其上表现很好，这是由于本文方法将模型检测空间进行了有效限制。

表 2　　算法成功检测数量比较

样本类型	成功检测数量		
	LLBMC	基础算法	改进算法
简单程序	39	44	45
条件程序	21	27	27
循环程序	17	23	23
内存操作程序	19	20	20

从实验结果可以看到，本文方法在 LLBMC 上可以有效地运行并成功地对相对大规模的代码进行检测，比单纯的有界模型检测算法在规模上更加强壮，同时在相对较大的规模代码上具有一定的时间优势。当然，在实验中也发现了在不同的代码片段上有一定的不稳定性。这个现象的原因可能是代码的局部性在不同的代码片段上的强弱程度表现不一，这也同时说明了实现属于一种启发式的算法实例。

7　结束语

本文提出了局部性在模型检测方法中应用的可能性，在描述了形式化定义的基础之上提出了对应的算法，并通过实验验证了其有效性。该方法实际上属于一种上近似的算法，通过获取一种局部性的上近似来对目标程序模型的状态空间进行分片，然后使用有界模型检测工具来完成状态子空间的检测。算法理论上可以结合各种有界模型检测算法，具有可扩展性，同时也具有结合其他模型检测算法的潜力。在未来的研究中，可以进一步地探讨将本文算法嵌入其他模型检测算法中或将其他模型检测算法嵌入本文算法中的可能性。

参考文献：

[1] JHALA R, MAJUMDAR R. Software model checking[J]. ACM Computing Surveys. 2009, 41(4): 1-54.

[2] BJORNER N, MCMILLAN K, RYBALCHENKO A. On solving universally quantified horn clauses[C]//The International Symposium on Static Analysis. 2013: 105-125.

[3] CLARKE E, KROENING D, LERDA F. A tool for checking ansi-c programs[C]//The 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2004: 168-176.

[4] CORDEIRO L, LISCHER B, MARQUES S J. SMT-based bounded model checking for embedded ANSIC software[J]. IEEE Transactions on Software Engineering, 2012, 38(4): 137-148.

[5] YANG Z, GANAI M K, GUPTA A, et al. Efficient SAT-based bounded model checking for software verification[J]. Theoretical Computer Science, 2008, 404 (3) : 256-274.

[6] MERZ F, FALKE S, SINZ C. LLBMC: bounded model checking of C and C++ programs using a compiler IR[C]//The International Conference on Verified Software: Theories, Tools, Experiments. 2012: 146-161.

[7] MORSE J, CORDEIRO D, NICOLE D, et al. Handling unbounded loops with ESBMC 1.20[C]//The International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2013: 619-622.

[8] DAVIS M, LOGEMANN G, LOVELAND D. A machine program for theorem-proving[J]. Communications of the ACM, 1967, 5(5): 394-397.

[9] HOOKER J N, VINAY V. Branching rules for satisfiability[J]. Journal of Automated Reasoning, 1995 , 15 (3) :359-383.

[10] LI C M, ANBULAGAN A. Heuristics based on unit propagation for satisfiability problems[C]//The 15th International Joint Conference on Artificial Intelligence. 1997 :366-371.

[11] MOURA D, BJORNER N. Z3: an efficient SMT solver[C]//The International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2008: 337-340.

[12] LIU L, KONG W, ANDO T. A survey of acceleration techniques for SMT-based bounded model checking[C]//The International Conference on Computer Sciences and Applications. 2013: 554-559.

[13] HENZINGER T A, JHALA R, MAJUMDAR R, et al. Abstractions from proofs[J]. ACM SIGPLAN Notices, 2004 , 39 (1) :232-244.

[14] JHALA R, MCMILLAN K L. Array abstractions from proofs[C]//The International Conference on Computer Aided Verification. 2004: 232-244.

[15] MCMILLAN K L. Lazy abstraction with interpolants[C]//The International Conference on Computer Aided Verification. 123-136.

[16] GULAVANI B S, RAJAMANI S K. Counterexample driven refinement for abstract interpretation[C]//The International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2006: 474-488.

[17] FLANAGAN C, QADEER S. Predicate abstraction for software verification[J]. ACM SIGPLAN Notices, 2002, 37 (1) :191-202.

[18] KOMURAVELLI A, GURFINKEL A, CHAKI S. Automatic abstraction in SMT-based unbounded software model checking[C]//The International Conference on Computer Aided Verification. 2013: 846-862.

[19] APEL S, BEYER D, FRIEDBERGER K. Domain types: abstract- domain selection based on variable usage[C]//The International Conference on Hardware and Software: Verification and Testing. 2013: 262-278.

[20] BEYER D, HENZINGER T A, THEODULOZ G. Lazy shape analysis[C]//International Conference on Computer Aided Verification. 2006: 532-546.

[21] HENZINGER T A, JHALA R, MAJUMDAR R, et al. Lazy abstraction[J]. ACM SIGPLAN Notices, 2002, 37(1): 58-70.

[22] BRADLEY A R. SAT-based model checking without unrolling[C]//The International Conference on Verification, Model Checking, and Abstract Interpretation. 2011: 70-87.

[23] RRADLEY A R, MANNA Z. Checking safety by inductive generalization of counterexamples to induction[C]//The International Conference on Formal Methods in Computer Aided Design. 2007: 173-180.

[24] CHAKI S, CLARKE E M, GROCE A, et al. Modular verification of software components in C[J]. IEEE Transactions on Software Engineering, 2004 , 30 (6) :388-402.

[25] BEYER D, KEREMOGLU M E. CPAchecker: a tool for configurable software verification[C]//The International Conference on Computer Aided Verification. 2011:184-190.

[26] BEYER D, LEMBERGER T. Symbolic execution with CEGAR[M]// Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. Springer International Publishing, 2016.

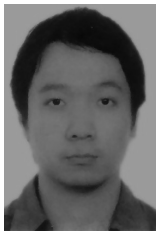
[27] BEYER D, DANGL M, WENDLER P. Boosting K-induction with Continuously-refined Invariants[M]//Computer Aided Verification. Springer International Publishing, 2015: 622-640.

[28] BEYER D, LOWE S. Interpolation for value analysis[J]. Software-Engineering and Management, 2015: 73-74.

[29] BEYER D, WENDLER P. Reuse of verification results[C]//The International Symposium on Model Checking Software. 2013: 1-17.

[30] BEYER D, LOWE S. Explicit-State software model checking based on CEGAR and interpolation[C]//The International Conference on Fundamental Approaches to Software Engineering. 2013: 146-162.
[31] ALBARGHOUTHI A, GURFINKEL A, CHECHIK M. From Under-Approximations to Over-Approximations and Back[C]//The International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2012: 157-172.
[32] KROENING D, STRICHMAN O. Decision procedures: an algorithmic point of view[M]. Springer Publishing Company, 2008.
[33] SINZ C, MERZ F, FALKE S. LLBMC: a bounded model checker for LLVM's intermediate representation[C]//The International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2012: 542-544.
[34] BEYER D. Status report on software verification[C]//The International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2014: 373-388.

[作者简介]



王舜（1988-），男，陕西西安人，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。

杜晔（1978-），男，黑龙江哈尔滨人，博士，北京交通大学副教授、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。

韩臻（1962-），男，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。

刘吉强（1973-），男，山东海阳人，北京交通大学教授、博士生导师，主要研究方向为密码学、可信计算、隐私保护技术等。