面向源代码的软件模型检测及其实现

何恺铎[1,2] 顾 明[1,2] 宋晓宇[3] 李 力[1,2] 李 江[1]
(清华大学软件学院 北京 100084)[1] (清华大学计算机科学与技术系 北京 100084)[2]
(Dept. ECE, Portland State University, Oregon, USA)[3]

摘 要 模型检测应用于检测软件可靠性具有重要意义。介绍了一种基于谓词抽象和反例引导抽象求精技术对源程序进行建模和验证的模型检测方法,并结合自行研发的Jchecker工具详细介绍了该软件模型检测技术的运作过程和关键算法。

关键词 软件模型检测,源程序验证,谓词抽象,抽象求精

Source-oriented Software Model Checking and its Implementation

HE Kai-duo[1,2] GU Ming[1,2] SONG Xiao-yu[3] LI Li[1,2] LI Jiang[1]
(School of Software, Tsinghua University, Beijing 100084, China)[1]
(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)[2]
(Dept. ECE, Portland State University, Oregon, USA)[3]

Abstract Model checking on software reliability is always considered meaningful. This paper studied theory and methodology on source code verification, utilizing predicate abstraction and counterexample-guided abstraction refinement. Detailed verification framework and its implementation of the self-designed Jchecker, a source-oriented software model checker, were also introduced.

Keywords Software model checking, Source verification, Predicate abstraction, Abstraction refinement

1 引言

作为一种有效的形式化方法,模型检测[1-3]自20世纪80年代以来在各领域得到了广泛应用。它通过穷举搜索状态空间来检测计算模型是否满足特定的性质,从而达到验证系统可靠性的目的。近年来,随着相关理论与技术的进一步成熟,软件模型检测成为了形式化领域的研究热点。

区别于传统的手工构造模型,面向源代码的软件模型检测直接基于程序源码提取信息并自动构造模型,具有自动化程度高及模型构造精确的优点。

国外在面向源码模型检测的相关研究方面起步较早,其中较具代表性的是Microsoft公司的SLAM和U. C. Berkeley的BLAST。

SLAM[4,5]项目是微软研究院开发的C语言程序的模型检测软件包。它先将原C语言程序抽象为布尔程序进行验证工作,即抽象后的程序仅剩下布尔变量,然后再进行检测。SLAM中依靠C2bp, Bebop, Newton 3个工具分别负责完成抽象、检测和抽象求精任务。

BLAST[6,7]项目是U. C. Berkeley开发的软件验证工具,它同样是基于反例引导的抽象求精框架对C语言程序进行检测。BLAST创造性地采用了懒惰抽象(lazy abstraction)技术,使得程序在不同的局部按照需要使用不同的谓词集合,从而有效地提高了效率,避免了计算资源的浪费。

Jchecker是清华大学软件学院软件理论与系统研究所自行研发的软件模型检测工具,它基于谓词抽象[8-11]理论,采用基于谓词抽象和反例引导的抽象求精框架[12-15],能够针对C程序源码抽象出模型并完备地搜索其状态空间,以此验证程序的安全属性。与上述两款工具相比,Jchecker在许多细节处理上(例如谓词抽象值域和条件语句后项计算等)存在区别和不同。Jchecker通过在运行时计算插值扩展谓词数量,自适应地不断精化抽象模型到合适的粒度,从而达到最大限度缩减状态空间的目的。

本文将致力于介绍Jchecker所采用的针对源码程序进行建模和验证的模型检测方法及其实现。除引言外,第2节简介相关基本概念;第3节结合程序实例详细讲述验证框架的理论依据、运行流程和计算方法;第4节和第5节分别介绍源码检测工具Jchecker及其实验结果;最后是对研究现状的总结和未来工作的展望。

2 基本概念与理论

2.1 变迁系统与验证目标

模型检测的基本思想是将系统的构造和行为表示为状态迁移系统M。我们可以用一个三元组来抽象该状态变迁系统:模型$M=(S,S_0,R)$,其中S为所有状态的集合,$S_0\subseteq S$是初始状态集,$R\subseteq S\times S$为迁移关系,表示状态间的变迁。

令$\varphi(s_i,s_j)$表示存在从s_i到s_j的变迁路径,则可以给出

到稿日期:2008-02-22 本文及相关研究获国家自然科学基金(No. 60553002, No. 90718039)支持。
何恺铎 硕士,主要研究方向为模型检测、谓词抽象;顾 明 教授,主要研究方向为形式化方法、信息安全、软件中间件技术。

下列重要定义：

$$\Phi_{reachable}=\{s\mid \exists s_0\in S_0(\varphi(s_0,s))\}$$

$$\Phi_{unreachable}=\bar{\Phi}_{reachable}=S\backslash\Phi_{reachable}$$

可以看到，系统的状态空间被分成了两部分：可达状态集 $\Phi_{reachable}$ 和不可达状态集 $\Phi_{unreachable}$。那么，若给定一个状态集合 Φ_{bad}，其中包含若干个不希望到达的“坏”的、“错误”的或“危险”的状态，如果能够证明 $\Phi_{bad}\subseteq\Phi_{unreachable}$，这样系统的质量就能够得到保证。类似这样的“坏的状态永不可达”的系统性质被称为安全属性，是我们的验证目标。

2.2 谓词抽象

当模型较复杂、状态数目较多时，常使用抽象技术[16-18]以缩减状态空间。其中谓词抽象是适用于软件模型检测的抽象方法，它提供了自动将无穷状态系统映射到有穷状态系统的方法。若原型系统的状态空间为 $S=V_1\times V_2\times\cdots\times V_n$，选定了谓词集合$\{p_1,p_2,\cdots,p_m\}$，设布尔域为 $B=\{true,false\}$，则状态空间可被抽象为 $S_{abs}=\prod_{i=1}^{m}B$。定义从 S 到 S_{abs} 的狭义抽象映射：

$$f(s)=(sat(s,p_1),\cdots,sat(s,p_m))\in\prod_{i=1}^{m}B$$

其中 $sat(s,p_i)=\begin{cases}true(\text{if } s\Rightarrow p_i)\\ false(\text{if } s\Rightarrow\neg p_i)\end{cases}$

若扩展布尔域 B 为三值域 $B'=\{T,true,false\}$，其中 T 表示“不确定”。抽象后的状态空间为 $S'_{abs}=\prod_{i=1}^{m}B'$。定义与从集合 2^S 到 S'_{abs} 的广义抽象映射 $f_g:2^S\rightarrow\prod_{i=1}^{m}B'$：

$$f_g(con)=(sat_g(con,p_1),\cdots,sat_g(con,p_m))\in\prod_{i=1}^{m}B'$$

其中 $sat_g(con,p_i)=\begin{cases}true(\text{if } con\Rightarrow p_i)\\ false(\text{if } con\Rightarrow\neg p_i)\\ T(\text{otherwise})\end{cases}$

接下来考虑抽象系统的迁移规则。若原系统迁移规则为 $R\subseteq S\times\Sigma\times S$，那么定义新迁移集合 $R'\subseteq S'\times\Sigma\times S'$ 如下：

$$R'=\{(f(s_1),l,f(s_2))\mid s_1,s_2\in S\wedge l\in\Sigma\wedge(s_1,l,s_2)\in R\}$$

至此，抽象状态变迁系统就得到了完整定义。以上的定义说明，只要两个抽象状态中有两个实际状态存在跃迁，那么这两个抽象状态就存在跃迁。这样的抽象系统是过近似[19](overestimated)的。

谓词抽象使得模型规模大大缩小，使我们更容易检验危险状态的可达性。对于“坏”状态 $s_{bad}\in S$，检验其是否可达的问题就可以先考虑相应的抽象状态 $s'_{bad}=f(s_{bad})$ 是否在抽象系统中可达。由于抽象系统是过近似的，因此如果 s'_{bad} 不可达，那么 s_{bad} 在原系统中也一定不可达，系统是安全的；如果 s'_{bad} 可达，那么 s_{bad} 的可达性未知，需作进一步分析。

2.3 反例引导的抽象求精

一般地，状态变迁系统 M 中某一条从初始状态经若干中间节点最终到达危险状态的路径称为该系统的反例或反例路径。上节中提到抽象系统是过近似的，因此即使 s_{bad}' 在 M' 中可达，s_{bad} 在 M 中的可达性也不能确定。抽象系统中存在反例路径而实际系统中不存在对应真实路径的反例称为伪反例。伪反例需要进一步的处理以便消除，真反例则可直接提交给用户解读。

可以想象，为证明某属性而直接构造一个恰好合适的抽象系统的难度是很大的。因此总是首先构造一个相对粗糙的初始抽象模型，然后通过分析由于过度抽象所造成的错误结果（即伪反例）得到精化原抽象模型的方法并消除伪反例。这样的求精过程可能要循环地进行若干次，直至最终产生一个适用的抽象模型为止。这就是反例引导的抽象求精框架，如图 1 所示。

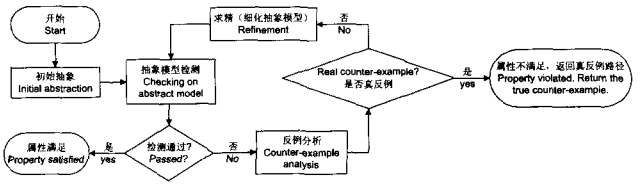

图 1 反例引导的谓词抽象求精框架

反例引导抽象求精的特点在于“运行时”改变抽象策略，面向验证目标逐步地自动修正抽象系统。利用反例的信息进行抽象模型精化有助于使模型在必要的地方足够精密，在不必要的地方足够简略，从而提高了验证效率。

3 面向源码软件模型检测

面向源代码的软件模型检测是指直接从源代码角度提取信息，自动构造和修正模型的方法。它既方便用户省却了传统方法中手工建模的步骤，又能保证模型的准确性和精度，因此具有较高的实用性和研究价值。

本节将主要介绍一种针对源码程序进行建模和验证的模型检测方法，尤其是如何将谓词抽象理论和反例引导抽象求精理论应用到面向源码的程序可达性检测中。我们将首先引入一种程序的形式化模型，并在此基础上详细论述具体的 4 个模型检测步骤：初始抽象、前项搜索、后向搜索以及抽象求精。

3.1 程序模型

一般地，一个程序的运行结果是由源代码决定的。因此，直接从程序源代码入手进行分析验证更接近程序本质，有助于提高软件质量。而基于源码验证的首要步骤是把程序理解为一个状态迁移系统，它应当具有两个要素：状态集合和迁移规则。

我们认为，程序模型的状态是由其运行时各变量取值和当前运行位置所共同决定的。所谓当前运行位置是指向下条待执行语句的指针 PC，它说明了程序执行到了哪里。设程序的变量集为$\{v_1,v_2,\cdots,v_n\}$，每一个变量 v_i 的值域为 D_i，PC 的取值范围为 D_{PC}，则程序的实际状态空间为 $D_{PC}\times D_1\times D_2\times\cdots\times D_n$。

我们知道，变量的值随着程序语句的执行一直发生变化。对应到状态变迁系统，那就是程序语句的语义决定着状态与状态之间的迁移关系：变量的值发生改变后，系统也就从一个状态到达了另一个状态。为保证变迁系统和原程序的等价性，在构建状态变迁系统时，我们必须确保每一条语句的含义都纳入系统的状态迁移规则。注意到程序语句的语义可以与谓词相对应，故程序模型的变迁规则可以被解释为谓词集合上的推理和演算。用下面的 Hoare 三元组表示：

$$\{\phi\}prog\{\psi\}$$

其中 ϕ 为前置条件，ψ 为后置条件（亦可记为 $\phi\xrightarrow{prog}\psi$）。这表示如果输入状态满足公式 ϕ，那么在 $prog$ 执行后，输出状态将会满足 ψ。此处 ϕ 和 ψ 亦可作状态集合解。程序状态空间

的变迁规则正是以这样的一种方式得到定义的。

一般地，模型检测检查的属性是由用户给出的。面向源码的模型检测中，一般通过特殊的程序语句（如图 2 中的 Error 语句）来进行危险状态的人工标注。对应到程序模型中，问题就转化为了确保所有可达状态的 PC 值不处于危险 PC 值的集合中。

3.2 程序的初始抽象

初始抽象是基于程序语句结构生成的。初始抽象非常简单，因为我们只考虑最基本的语句的顺序和分支，就可以得到一个简单的 CFA（Control Flow Automation，控制流自动机）。考虑图 2 中的一段源程序，其右侧即为其对应的初始抽象图 CFA（实心箭头代表赋值语句，空心箭头代表条件判断语句）。

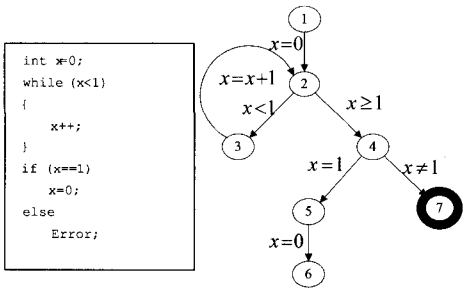

图 2 初始抽象

CFA 可以理解为程序执行流程图，同时本身也是一个抽象状态迁移系统：此时没有添加任何谓词，抽象映射的形式为

$f: D_{PC} \times D_1 \times D_2 \times \cdots \times D_n \rightarrow D_{PC}$

图 2 中的节点 7 对应着程序中的错误语句 Error，是一个错误状态。在这个图中节点 7 是可达的，这是一条明显的反例路径，是由于当前的谓词集合为空，状态过于抽象而导致迁移不受限所造成的。因此，为了精化粗糙的初始抽象系统，我们需要加入新的谓词。一般发现新谓词的方法是利用反例分析计算插值。但对于尚无谓词的初始抽象系统，我们可以略过反例分析步骤而直接选取一个或多个初始谓词。一般地，我们可以使用程序中第一条赋值语句或条件语句作为初始谓词。

3.3 前向搜索

当选择了新谓词后，我们需要重新构造或精化抽象系统。由于谓词集合是在原有基础上增加，因此我们的新抽象系统一定只比前者更精细、表达能力更强。因此，以前由于过度抽象所造成的伪路径可能随着精化而切断，这也正是我们进行精化的原因所在。

对于从语句序列构造的程序模型，在前一阶段使用的 CFA 图仍然可以用以构造、更新和表达抽象模型。事实上，我们将基于原有 CFA 使用新谓词集合沿执行路径计算和表达各节点的新的抽象状态空间，这一过程称为“前向搜索”（forward search）。

面向源码的模型的特点在于保留了所有有关程序当前运行位置（即 PC）的信息。因此当谓词集合为 $\{p_1, p_2, \cdots, p_m\}$ 时，狭义和广义抽象映射关系分别为：

$$f: D_{PC} \times D_1 \times D_2 \times \cdots \times D_n \rightarrow D_{PC} \times \prod_{i=1}^{m} B$$

$$f_g: D_{PC} \times 2^{D_1 \times D_2 \times \cdots \times D_n} \rightarrow D_{PC} \times \prod_{i=1}^{m} B'$$

前向搜索的主要内容是通过前一节点的状态空间 S_{pre} 和语句 st 的语义来计算得出后一个节点的状态空间 S_{post}。必须注意，此时状态空间是基于谓词集合进行了抽象的，只能通过当前的谓词集合做粗糙描述。所以计算后继节点状态的方法分为两步：(1) 计算 S_{pre} 执行 st 语义后的精确状态空间 S_{mid}（最强后置条件）；(2) 将 S_{mid} 通过广义抽象映射 f_g 映射到抽象状态空间 $D_{PC} \times \prod_{i=1}^{m} B'$ 上。

下面介绍具体计算后继节点状态的方法。若前一节点状态空间为 con，st 为赋值语句，对应谓词 p_i 的分量 b_i（其中 $WP(p_i, st)$ 表示经过 st 满足 p_i 的最弱前置条件）：

$b_i = sat_g(con, WP(p_i, st)) \in B'$；

st 为条件判断语句时，若 $con \wedge st$ 不可满足，则说明后继节点状态为空集，不再沿此路径进行前向搜索；若 $con \wedge st$ 可满足，对应谓词 p_i 的分量 b_i 的计算方法为（其中 $project(con, p_i)$ 指 con 中 p_i 分量的值）：

$$b_i = \begin{cases} true(\text{if } st \Rightarrow p_i) \\ false(\text{if } st \Rightarrow \neg p_i) \\ project(con, p_i) \end{cases}$$

对于前面的源程序实例，若初始谓词设为 $x=0$，则按照上述规则前向搜索的结果如图 3(a) 所示。

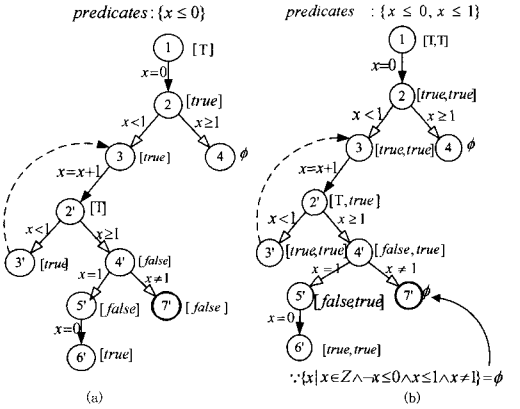

图 3 前向搜索

显然，图 3(a) 在前向搜索中出现了一条反例路径，即“1-2-3-2'-4'-7'”，其实这是一条伪反例路径（如何对该反例进行分析以确定其真伪，将在下一节详细讲述）。这说明仅有一个谓词 $x \leqslant 0$ 时我们的系统过于抽象了。

图 3(b) 所进行的前向搜索添加了 $x \leqslant 1$ 作为额外的谓词，因此系统的表述能力更强了。可以看到，原来可达的错误节点 7' 在新谓词参与表达抽象系统的情况下，已不可达了（即节点对应的状态空间为 ϕ）。这样的情况正是我们希望的：伪反例路径被消除，该 Error 节点在实际系统中不可达，程序的安全属性得到了保障。

如果在前向搜索过程完成后没有任何错误节点可达，那么程序的安全性就已经得到了证明，整个检测过程结束。若系统中还存在可达的错误节点，则将此条路径记录下来并提交给反例分析相关模块。

另外，在此值得一提的是前向搜索时处理循环的技巧。由于循环语句的存在，可能导致 PC 值变小，即向回跳转到执行过的语句之前。为避免无限地展开循环下去，模型检测器需要保存展开过的节点相应的状态空间。若某节点被重复访问，且新的状态空间 S_{new} 被包含于以前的空间 S_{old} 中（即 $S_{new} \subseteq S_{old}$），则此节点就无需再次展开了。图 3 中节点 3' 就是这样

的情况，图中的虚线表示此状态空间已经在先前被搜索过了。

3.4 后向搜索

前向搜索中发现的反例路径需要进行分析，以确认它是真反例还是伪反例。在以 CFA 为框架的程序模型框架中，我们采用“后向搜索”(backward search)技术来确认反例真伪。所谓后向搜索，具体是指从反例路径末端的错误节点出发，逆向地沿着反例路径向上计算可达状态空间的路径检验算法。

后向搜索的初始状态空间尽可能大地设为$\{PC_{Error}, T, \cdots, T\}$，然后向上逆向计算前一个节点的最大可能的状态空间。显然，若此节点的状态空间为 ψ，前一条语句为 st，那么前一节点的最大可能的状态空间正是其最弱前置条件 $WP(\psi, st)$。

每次通过计算最弱前置条件得到前一节点的状态空间 ψ_{back}，需要和前向搜索时得到的状态空间 $\psi_{forward}$ 求交。若 $\psi_{back} \wedge \psi_{forward} = \phi$，则说明此反例路径其实是不可达的，是伪反例。此时停止后向搜索，进入插值计算步骤。若 $\psi_{back} \wedge \psi_{forward} \neq \phi$，则再计算上一个节点的状态空间，继续后向搜索过程。若此时已是系统的开始节点，且两个状态空间的交不为空的话，则可断定此条反例路径是存在于原系统的真反例，应当将此错误报告给用户后退出。

现在来看图 3(a)中发现的反例。我们使用后向搜索对其进行分析，结果如图 4 所示。

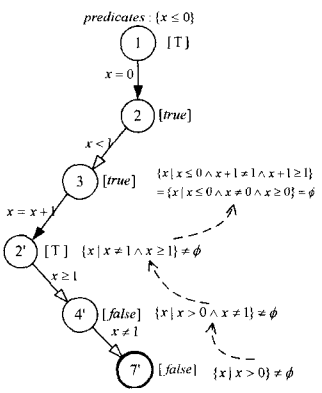

图 4 反向搜索

可以看到，图中每逆向求得一步新的状态空间，就会和前向搜索时得到的状态空间求交。若交集不为空，则继续上溯。可以看到，在节点 3 前后搜索的状态空间的交为空，由此我们可以断定此反例是一个伪反例。下一步需要根据此反例路径挖掘新谓词，以便精化抽象模型，消除反例。

3.5 插值与新谓词发现

在后向搜索中发现的伪反例路径是当前抽象系统过于粗糙导致的结果。基于反例引导的抽象求精框架的思想，我们试图从伪反例路径本身寻找隐藏其中的信息来帮助寻找新谓词，从而对当前抽象系统进行合适和必要的精化。

Jchecker 是基于 Craig 插值理论[22,23]来获取新谓词的，这也是当前世界上谓词抽象求精的主流方法之一。考虑相互矛盾的两个公式 A 和 B，即 $A \wedge B$ 不可满足，若有公式 I 符合以下条件：

(1) $I \wedge B$ 不可满足；

(2) $A \Rightarrow I$ 成立；

(3) I 中只引用同时出现在 A 和 B 中的变量。

则称 I 为 A 和 B 的一个 Craig 插值。三者关系如图 5 所示。

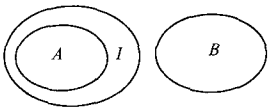

图 5 Craig 插值的状态空间表述

Craig 插值 I 是一个比 A 更大但仍不和 B 相交的集合。当 A 中状态如果在一次抽象后对 B 可达(这样就有可能造成伪反例的出现)时，插值 I 因为更“大”、更“弱”，所以其抽象很可能对 B 不可达，这样反例就有可能被消除。因此，插值 I 可以作为一个新谓词来增强抽象系统的表达能力，达到抽象细化的目的。这就是使用 Craig 插值来进行谓词发现的原理。

当后向搜索状态空间 ψ_{back} 和前向搜索时的 $\psi_{forward}$ 在某节点交集为空时，设从它到错误节点的路径上的语句序列为 $st_1, st_2, \cdots, st_m$，则必有 $\psi_{forward} \wedge st_1 \wedge st_2 \wedge \cdots \wedge st_m$ 不可满足。因此可以从此式中计算出若干个插值并加入到新的谓词集合中。需要注意的是，每遇到一次赋值语句，此式左端被赋值的变量及其后续的出现必须更换名字。

让我们回过头来看图 4 中的例子。上溯返回到节点 3 时交集为空，故我们需要从路径“3-2'-4'-7'”中寻找新谓词。整理后的语句序列如图 6，这样我们共得到了如图 6 的 3 个插值。

$$x \leqslant 0 \wedge x' = x + 1 \wedge x' \geqslant 1 \wedge x' \neq 1$$

$x \leqslant 0$　　$x' \leqslant 1$　$x' = 1$

图 6 计算插值

把插值中的别名还原后，我们得到了 3 个不同的插值：$x \leqslant 0$，$x \leqslant 1$，$x = 1$。其中 $x \leqslant 0$ 已在谓词集合中，我们可以选择其余两者。第 3.3 节的图 3(b)显示了当 $x \leqslant 1$ 加入谓词集合后的情形：伪反例被成功地消除，程序安全属性得到验证。

可以看到，新谓词发现的关键在于生成合适的插值。高质量 Craig 插值的自动计算是一个具有挑战性的课题，我们也正在对它进行深入的研究。

4 软件模型检测器 Jchecker

基于上述理论和方法，Jchecker 采用 Java 语言开发，具体地实现了检测所依赖的理论框架，包括谓词抽象和程序模型；也实现了面向源码软件模型检测的每一个重要步骤和流程，包括源码解析、CFA 生成、前向搜索、反向搜索等，实现了可以自动分析 C 程序源码，并对用户指定的安全属性进行可达性验证。

Jchecker 具体实现模型检测和抽象求精的算法和步骤可用下列伪代码表示：

```
1. sourceParser(); // get the CFA Graph
2. beginForwardSearch (n); // initial and compute the root
3. forwardSearch () // forward explore
4.     currNode () = n':e; // read a node n'
5.     if (currNode = isErrorNode) goto 12 with the error trace "t";
6.       else if (currNode = leafNode) currNode = fatherNode;
7.            else ns' = getNextStateSpace (n,e);
8.                 if (ns' = ⊥) goto 5;
9.                   else t = t ∪ n':e';
```

```
10.                    curNode = sonNode; goto 3;
11. curNode = FatherNode; return "the system is safe";
12. backTrace () // backward check
13.     errorNode = currNode;
14.     if (errorNode = root) return (errorTrace (t));
15.       else errorNode = n′:e′;
16.            backStateSpace () = pre(n′,e′);
17.            if (pre (n′,e) ∧ ns′ = ∅) errorNode = refineNode;
18.              goto 20 with exp(e);
19.              else exp(e) = exp(e) ∪ e′,goto 13;
20. refineFromHead () //  add new predicates
21.   getInterpolation (exp(e));
22.   if (getInterpolaton () = Predicate()) goto 5;
23.     else addPredicate ();
24.          root = refineNode;
25.          goto 4;
```

Jchecker 的工程框架如图 7 所示。

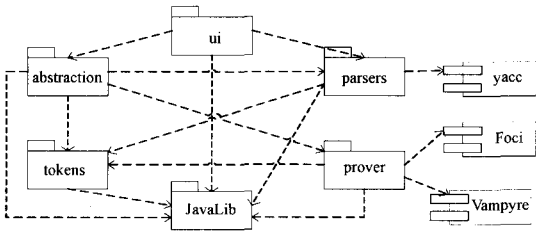

图 7　Jchecker 工程架构

图 7 中 tokens 包为基本数据结构的实现，包括变量、表达式、条件、语句等；parser 包为解析器，负责对输入的 C 程序源码进行解析，其实现借助了 BYACC/J 程序(Berkeley Yacc for Java)生成代码；prover 包通过对 Foci[20] 和 Vampyre[21] 的调用实现了蕴涵判断、可满足性判断和计算插值的功能；abstraction 包实现了核心的谓词抽象、反例分析和模型求精功能；ui 包负责提供用户界面。

5　实验结果

我们使用 Jchecker 对实际源码进行了检测实验。实验的硬件环境为一个使用 Athlon64 3000+ (2.0GHz)处理器和 1024M 内存的计算机，软件环境为 2.6 内核版本的 Linux 和 1.6.0 的 Sun JVM。验证对象分别是：来源于文献[7]的 Foo.c 和 Loop.c，文献[6]中的 Funlock.c 以及一个三位计数器程序 Counter.c。实验的结果如表 1 所示。

表 1　Jchecker 源码验证实验

程序名	程序规模	路径数目	谓词数量	TP 调用	检测时间(s)
Foo.c	8	1	1	5	1
Loop.c	11	13	3	206	5
Counter.c	14	15	5	191	4
Funlock.c	65	298	11	2915	23

其中，“程序规模”一栏是指源程序的代码行数，“路径数目”代表 Jchecker 在验证此程序时所搜索的路径数目，“谓词数量”一栏代表验证过程中同时存在的最大谓词数量，“TP 调用”表示调用定理证明器的次数，“时间”即完成验证所需的时间。可以看到，基于我们的框架和工程实现，Jchecker 都能够顺利地对这些源码进行安全属性的检测。可以从实验结果中得到的重要结论是，相对于程序规模的扩大，定理证明器调用次数和时间空间复杂度的增长呈接近线性的关系，属于稳定可控的范围。

客观地说，同国外工具相比，我们的模型检测器尚不够成熟，尤其在性能上有待优化。但 Jchecker 的研发无疑使我们积累了宝贵的实践经验，也填补了国内相关领域的空白。此外，在深刻理解相关理论的基础上，我们已经提出了一系列优化验证过程的算法和思想[24-26]。实验表明它们能够有效地提高检测速度。这些改进已被或即将应用于 Jchecker 中。

结束语　模型检测作为一种重要的形式化方法已在越来越多的领域得到重视和应用。软件模型检测的难点在于其复杂度很高、抽象难度大，容易造成状态爆炸问题。谓词抽象和反例引导的抽象求精技术能够有效地帮助进行软件抽象模型的构造和修正，是进行程序自动验证的有力手段。

本文阐述了一种面向源码的运行时程序建模方法，并在此基础上结合实例详细地说明了如何自动验证程序的安全属性。

Jchecker 的实现反映了我们验证框架的正确性和有效性。其主要特点在于：(1)基于源码进行抽象保证了模型的正确性；(2)反例引导的抽象求精在保持必要细节的前提下最大限度地降低了抽象模型的复杂程度；(3)验证过程完全自动化，无需人工干预。

虽然软件模型检测技术近年来取得了一定的进展，但距离大规模进入工业界应用尚有距离。面向源代码的软件模型检测仍然具有很大的发展空间；而 Jchecker 工具作为原型系统，也有不少可改进之处。我们将来的工作将围绕着两条主线展开：(1)研究并实现软件模型检测器依赖的关键基础架构，如定理证明器和插值生成器，(2)优化算法和设计，提升工具性能。

参 考 文 献

[1] Clarke E M, Grumberg O, Peled D A. Model Checking. London, England: The MIT Press, 1999

[2] 林惠民，张文辉. 模型检测：理论方法与应用. 电子学报，2002 (12A)：1907-1912

[3] Huth M, RyanM. Logic in Computer Science. 北京：机械工业出版社，2005

[4] Ball T, Rajamani S K. The SLAM Project : Debugging System Software via Static Analysis. POPL 2002, January 2002: 1-3

[5] SLAM. http://research.microsoft.com/slam/

[6] Henzinger T A, Jhala R, Majumdar R, et al. Lazy abstraction // Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Program Languages. ACM Press, 2002: 58-70

[7] http://mtc.epfl.ch/software-tools/blast/

[8] Graf S, Saidi H. Construction of abstract state graphs with PVS // Proc. 9th International Conference on Computer Aided Verification (CAV 97). Vol 1254 of LNCS. Springer-Verlag, 1997: 72-83

[9] Ball T, Cook B, Das S, et al. Refining approximations in software predicate abstraction // Tools and Algorithms for the Construction and Analysis of Systems (TACAS 04). LNCS 2988. Springer-Verlag, 2004: 388-403

[10] Das S, Dill D L, Park S. Experience with predicate abstraction // Computer-Aided Verification, Vol 1633 of LNCS. Springer-Verlag, July 1999: 160-171

[11] 陈炜. 基于谓词划分的完备谓词抽象研究. 硕士学位论文. 清华大学,2007
[12] Clarke E M, Grumberg O, Jha S. Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM,2003,50(5):752-794
[13] Clarke E M, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement// Computer Aided Verification, CAV00. Springer-Verlag,2000:154-169
[14] 周旻. 基于惰性抽象的软件模型检验算法研究. 学士学位论文. 清华大学,2007
[15] 杨柳春. 基于反例的抽象细化研究. 硕士学位论文. 清华大学,2007
[16] Clarke E M, Grumberg O, Long D E. Model checking and abstraction. ACM Transactions on Programming Languages and System(TOPLAS),1994,16(5):1512-1542
[17] Dams D, Gerth R, Grumberg O. Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and System(TOPLAS),1997,19(2):253-291
[18] 袁志斌,徐正权,王能超. 软件模型检测中的抽象. 计算机科学,2006,33(7):276-279
[19] Lee W, Pardo A, Jang J, et al. Tearing based abstraction for CTL model checking// International Conference of Computer-Aided Design. 1996:76-81
[20] http://www.kenmcmil.com/foci.html
[21] http://www.cs.ucla.edu/~rupak/Vampyre/
[22] Henzinger T A, Jhala R, Majumdar R, et al. Abstraction from proofs// Proc. of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Program Languages. ACM Press,2004:232-244
[23] McMillan K L. An interpolating theorem prover// Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. volume 2988 of LNCS. Springer,2004:16-30
[24] 李力. On the Improvements of Abstract-Check-Refinement Verification. 技术报告
[25] 李力. Fast Finding Abstract Counter-example. 技术报告
[26] 李江. Game Theory-based Optimization of Predicates Abstraction Model Checking. 技术报告

(上接第 246 页)

得出了与 RSR 算法相同的结论。体现了简单高效并保持简化集的错误检测能力的优点。

结束语 本文中基于新颖测试覆盖的测试集简化方法的特点在于:(1)提出了一种新颖的测试覆盖准则,二级变量串联准则;(2)运用上述准则与分支测试覆盖准则相组合,形成了新的测试覆盖需求;(3)在所形成的新的测试覆盖需求的基础上,运用 HGS 算法,针对特定的被测程序取得了较好的测试集简化效果,过程简便并保持了简化后的测试集的错误检测能力。

该准则的扩充是今后要进一步研究的工作,当程序中存在变量繁多、关系复杂的情况时,变量间多重并联关系的表示以及冗余关系的剔除都是研究的难点。

参 考 文 献

[1] Chvatal V. A greedy heuristic for the Set-Covering problem. Mathematics of Operations Research,1979,4(3):233-235
[2] Harrold M J, Gupta R, Soffa M L. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology,1993,2(3):270-285
[3] Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. IEEE Transactions on Software Engineering,2007,33(2):108-123
[4] Jones J A, Harrold M J. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Trans. Software Eng.,2003,29(3):195-209
[5] Tallam S, Gupta N. A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization. PASTE,2005:35-42
[6] McMaster S, Memon A M. Call Stack Coverage for Test Suite Reduction// icsm'05. Vol. 00, Sept. 2005:25-30
[7] Rapps S, Weyuker E J. Selecting Software Test Data Using Data Flow Information. IEEE Trans. Software Eng.. 1985,11(4):367-375
[8] Frankl P G, Iakounenko O. Further Empirical Studies of Test Effectiveness// Proc. Sixth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,Nov. 1998:153-162
[9] Frankl P G, Weiss S N. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. IEEE Trans. Software Eng.,1993,19(8):774-787
[10] Hutchins M, Froster H, Goradia T, et al. Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria// Proc. 16th IEEE Int'l Conf. Software Eng.. May 1994:191-200
[11] Frankl P G, Weiss S N. An Experimental Comparison of the Effectiveness of the All-Uses and All-Edges Adequacy Criteria// Proc. Fourth Symp. Testing, Analysis, and Verification. 1991:154-164
[12] Huang Chenliang, Gao Jianhua. Research of tracing dependency based on scenario-driven approach. Jisuanji Gongcheng/Computer Engineering(Language:Chinese),2005,31(19):102-104
[13] Zhu Yuan, Gao Jianhua. Method of improving software reliability by controlling logical complexity. Jisuanji Gongcheng/Computer Engineering(Language:Chinese),2004,30(1):73
[14] Elbaum S, Kallakuri P, Malishevsky A, et al. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Software testing, verification and reliability, 2003(13):65-83
[15] Mansour N, Bahsoon R. Reduction-based methods and metrics for selective regression testing. Information and Software Technology,2002,44:431-443