计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20150669
53(1):155-164,2016

基于 CEGAR 的 C 程序空指针解引用检测

段 钊 田 聪 段振华
(西安电子科技大学计算机理论与技术研究所 西安 710071)
(ctian@mail.xidian.edu.cn)

CEGAR Based Null-Pointer Dereference Checking in C Programs

Duan Zhao, Tian Cong, and Duan Zhenhua
(*Institute of Computing Theory and Technology, Xidian University, Xi'an* 710071)

Abstract With the rapid growth of computer software in both scale and complexity, more and more attention has been paid by software developers on the reliability and security issue. Null-pointer dereference is a kind of errors which often occur in programs. This paper proposes a CEGAR based null-pointer dereference verification approach for C programs. With this method, first, a linear temporal logic (LTL) formula is used to specify the null-pointer dereference problem. Then whether null-pointer dereference occuring in a program is checked by a CEGAR based model checking approach. In order to verify null-pointer dereference problem in a total automatic way, this paper also studies how to generate temporal logic formulas automatically with respective to null-pointer dereference problem. Experimental results show that the proposed approach is useful in practice for checking null-pointer dereference in C programs with large scale.

Key words model checking; abstraction refinement; null-pointer dereference; program verification; temporal logic

摘 要 随着计算机软件规模和复杂度的日益增长,软件系统的可靠性和安全性倍受关注.空指针解引用是程序中常见的一类错误.提出了一种基于反例制导抽象精化 CEGAR 的 C 程序空指针解引用检测方法.该方法首先使用线性时序逻辑描述空指针解引用问题,然后通过抽象精化的方法检测待测程序中是否含有空指针解引用错误.为了达到完全自动验证的目标,同时针对空指针解引用问题,研究了该类性质的时序逻辑表达方法,并自动从程序中针对所有的指针变量,形成相应的时序逻辑公式.实验结果表明,所提出的方法在大规模 C 程序的空指针解引用检测方面有着重要的实际应用价值.

关键词 模型检测;抽象精化;空指针解引用;程序验证;时序逻辑

中图法分类号 TP31

计算机软件已深入到国防建设和国计民生的各个领域,如航天航空、工业控制、交通、医疗、教育和社交等,成为当今社会不可或缺的一部分.面对多样化的需求,软件的规模和复杂性不断增大,因而软件的可靠性和安全性更加难以得到保障.目前,大部分软件系统都存在各种各样的错误和缺陷.有的软件虽然可以正常运行,但是其内部仍然存在许多错误隐患,在特定情况下会导致软件性能下降,甚至引起程序崩溃,从而造成巨大的损失.为了提高软件的可信性,研究人员提出大量的技术来检测和预防软件中的错误和缺陷.

C 程序中指针的使用十分广泛,而且语法灵活,

收稿日期:2015-07-20;**修回日期**:2015-10-16

基金项目:国家自然科学基金项目(61322202,61420106004,91418201,61133001,61272117)
This work was supported by the National Natural Science Foundation of China (61322202,61420106004,91418201,61133001,61272117).

但易错.由于指针而引发的内存错误往往难以追踪和分析,空指针解引用是其中常见的错误之一.如果一个指针表达式指向 NULL,或者未被初始化,或者指向一个已被释放的内存,那么,使用该表达式的解引用就会导致系统崩溃或者得到错误的数据.

许多查找软件错误的技术被提出.其中,静态分析是在不执行程序的情况下通过直接分析源程序来检查程序中的错误,主要包括路径敏感/不敏感分析方法[1]和流敏感/不敏感分析方法[2-3]等.程序测试[4-5]是对程序的每一个模块在使用前进行测试,保证该模块的正确性.定理证明[6]将待测问题转化为可满足性问题,通过判断命题逻辑公式的可满足性来检测程序中是否存在错误.模型检测[7-8]主要是把待检测程序描述为状态迁移系统,把待检测的问题用时序逻辑公式表示出来,然后检测状态迁移系统是否满足性质.近年来,许多研究将模型检测、定理证明和静态分析等技术相结合,利用各种方法的优势来检测程序中的错误.

在空指针解引用方面,Dirk 等人开发的自动验证工具 BLAST[9-11],可以验证 C 程序中的安全性质.Dirk 等人将他们的方法进行了扩展,开发了新的工具 CPAChecker[12-13].但是使用 BLAST 和 CPAChecker 检测程序中的空指针解引用需要预先在程序中插桩,如何完全自动地插桩是一个有待解决的实际问题.CMBC[14]是 Edmund 等人开发的一种限界模型检测工具.Charatonik 等学者在此工具的基础上,通过限界的思想来检测程序中是否存在空指针解引用问题[15].另外,还有一些研究针对并行程序中空指针解引用问题[16].在国内,也有一些学者从事这方面的研究,如文献[17]给出了一种检测 C 程序中基于区域内存模型的空指针解引用错误的方法.

总之,现有的程序模型检测方法和工具很多,它们主要是通过对源程序进行插桩,然后再分析插桩后的程序.CBMC 和 BLAST 支持对程序的自动插桩;但是,它们是对程序中每个指针的每一次解引用都进行插桩,因而会大大增加源程序的代码量.CPAChecker 则需要手动地进行插桩,如果给每个位置插桩,同样会增加代码量.

本文提出了一种基于反例制导抽象精化(counter example-guided abstraction refinement, CEGAR)[18]的 C 程序空指针解引用检测方法.该方法首先使用线性时序逻辑(linear temporal logic, LTL)描述空指针解引用问题,然后通过抽象精化的方法检测待测程序中是否含有空指针解引用错误.为了达到完全自动验证的目标,本文同时针对空指针解引用问题,研究了该类性质的时序逻辑表达方法,并自动从程序中针对所有的指针变量形成相应的时序逻辑公式.实验结果表明,该方法可以自动地对程序中的所有指针进行检测,错误覆盖率高,对大规模程序的检测有着很好的实际应用价值.

1 空指针解引用问题的 LTL 描述

1.1 LTL

模型检测中,一般使用时序逻辑公式描述待验证的性质.本文使用 LTL 来描述空指针解引用问题.

设 AP 是原子命题集合,LTL 公式是由 AP 中的元素和操作符组成.其中,操作符包括逻辑非($\neg$)、逻辑与($\wedge$)、next($\bigcirc$)、until($\cup$)和 release(R).LTL 的语法定义如下:

$$\varphi ::= p \mid \neg\psi \mid \varphi\wedge\psi \mid \bigcirc\psi \mid \varphi\cup\psi \mid \varphi \mathrm{R}\psi, p\in AP.$$

LTL 公式的语义解释在线性结构上.一个线性结构是一个无限序列 $x=x(0),x(1),\cdots$.其中,$x(i)$ 是 AP 到{true, false}的映射.一个 LTL 公式 φ 在线性结构 x 的第 i 个位置满足一个线性结构 x,记作:$x,i\Rightarrow\varphi$.语义定义如下:

$x,i\Rightarrow p$,如果 $x(i)(p)=\text{true}, p\in AP$;

$x,i\Rightarrow\neg\psi$,如果 $x,i\Rightarrow\psi$;

$x,i\Rightarrow\varphi\vee\psi$,如果 $x,i\Rightarrow\varphi$ 或 $x,i\Rightarrow\psi$;

$x,i\Rightarrow\bigcirc\varphi$,如果 $x,i+1\Rightarrow\varphi$;

$x,i\Rightarrow\varphi\cup\psi$,如果存在 $j\geqslant i$,使得对于任意的 $i'\in\{i,i+1,\cdots,j-1\}$,$x,i'\Rightarrow\varphi$,并且 $x,j\Rightarrow\psi$;

$x,i\Rightarrow\varphi \mathrm{R}\psi$,如果对所有的 $j\geqslant i$,$x,j\Rightarrow\psi$,或者存在 $j\geqslant i$,使得对于任意的 $i'\in\{i,i+1,\cdots,j-1\}$,$x,i'\Rightarrow\psi$,并且 $x,j\Rightarrow\varphi$.

其它附加的逻辑操作符定义如下:

$$\text{true}\equiv\varphi\vee\neg\varphi,$$
$$\text{false}\equiv\varphi\wedge\neg\varphi,$$
$$\varphi\rightarrow\psi\equiv\neg\varphi\vee\psi,$$
$$\varphi\leftrightarrow\psi\equiv(\varphi\rightarrow\psi)\wedge(\psi\rightarrow\varphi).$$

另外,时序操作符 sometimes($\Diamond$)和 always($\Box$)的定义如下:

$$\Diamond\varphi\equiv\text{true}\cup\varphi,$$
$$\Box\varphi\equiv\neg\Diamond\neg\varphi.$$

LTL 公式可以表示为范式.对于一个 LTL 公式 φ,它的范式定义为

$$\varphi=\bigvee_i\beta_i\wedge\bigcirc\varphi_i,$$

其中,β_i 是状态公式,即只包含逻辑操作符的公式;

φ_i 是一个任意的主操作符不为析取操作的 LTL 公式.在范式中,每一个析取项都代表一个迁移关系.直观地讲,范式表示存在一个整数 i,β_i 在当前状态可满足,φ_i 在下一状态可满足.已证明,所有的 LTL 公式都可以等价地转化为它的范式[19].

1.2　LTL 到 TGBA 的转换

任何一个 LTL 公式可以被等价地转换为一个基于迁移的广义 Büchi 自动机(transition based generalized Büchi automata, TGBA)[20].为了构造 LTL 公式 φ 的 TGBA,首先创建初始状态$[\varphi]$,然后将 φ 转换为范式.假设:

$$\phi \equiv (p \wedge \neg r \wedge \neg p \wedge \neg r) \wedge \bigcirc(\phi_1 \wedge \phi_2) \vee p \wedge \neg r \wedge \bigcirc \phi_3 \vee p \wedge \neg r \wedge \bigcirc(\phi_4 \wedge \phi_5),$$

那么可以生成 3 个新结点,即$[\phi_1 \wedge \phi_2]$,$[\phi_3]$和$[\phi_4 \wedge \phi_5]$,如图 1 所示:

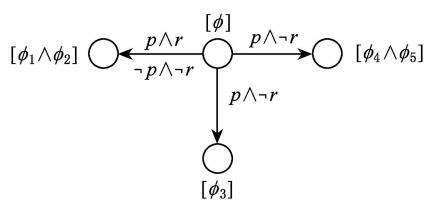


Fig. 1　Constructing TGBA.

图 1　构造 TGBA

同时,生成迁移关系为

$$\tau([\phi], p \wedge r) = [\phi_1 \wedge \phi_2],$$
$$\tau([\phi], p \wedge \neg r) = [\phi_4 \wedge \phi_5],$$
$$\tau([\phi], \neg p \wedge \neg r) = [\phi_1 \wedge \phi_2],$$
$$\tau([\phi], p \wedge \neg r) = [\phi_3].$$

为了完成整个 TGBA 的构造,需要针对每个新生成的结点,通过将表示该结点的公式转换成范式,从而不断地产生新的结点,直到没有新的结点可以被生成.这样的一个图我们称为 φ 的范式图.

在此过程中,必须考虑一个问题:对于形如 $p \cup q$ 的公式,q 最终要发生,即 q 不能被无限地延迟.但是,在范式图中没有显式地体现出 q 有没有发生.

因此,为了准确地刻画 $\cup$ 操作的语义,本文进一步在范式图的边上增加 $Flag$ 集合标记.我们关注公式中不同的 $\cup$ 操作.对于每一个主操作符为 $\cup$ 的子式,使用一个唯一整数下标 i 来区分,这里 $i>0$.这样,第 i 个 $\cup$ 操作,记为 $\cup_i$.一个 LTL 公式的范式图中,一个状态$[\varphi]$中的所有主操作符为 $\cup$ 的子式所对应的下标构成的整数集合称为 $Until_\varphi$.整个范式图中,所有主操作符为 $\cup$ 的子式所对应的下标构成的整数集合称为 $AllUntil$.据此,指向状态$[\varphi]$的迁移上的标记为 $Flag = AllUntil / Until_\varphi$.

对于一个 LTL 公式 $\phi = p \cup (q \cup r)$,它对应的 TGBA 如图 2 所示,在图 2 中"{}"是每一个状态所包含的主操作符为 $\cup$ 的子式的整数标记集合,"[]"是每一条迁移上的 $Flag$ 集合.

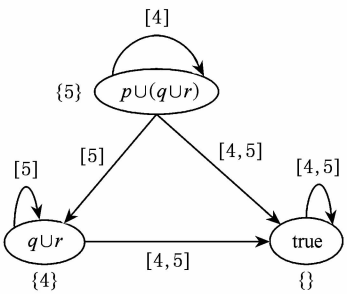


Fig. 2　TGBA of the formula ϕ.

图 2　公式 ϕ 的 TGBA.

从图 2 可以看出,"$q \cup r$",记为 ϕ 中的主操作符 $\cup$,被标记为 4;"$p \cup (q \cup r)$",记为 ϕ_1 中的主操作符 $\cup$,被标记为 5;"true",记为 ϕ_2,不包含 $\cup$.所以,整个 TGBA 的 $AllUntil$ 集合为 $\{4,5\}$,$Until_\varphi$ 为 $\{4\}$,$Until_{\varphi_1}$ 为 $\{5\}$,$Until_{\varphi_2}$ 为 $\{\}$.

范式图以及 $Flag$ 标记构成了与给定 LTL 公式等价的 TGBA.TGBA 上一个无穷路径是指从初始结点出发的一个结点与边相交互的无穷序列,$\pi = \langle n_0, e_0, n_1, e_1, \cdots, (n_i, e_i, \cdots, n_j, e_j)^\omega \rangle$,$0 \leqslant i \leqslant j$.为了方便,我们用 $inf_n(\pi)$ 表示 π 的循环部分的结点集合,$inf_e(\pi)$ 表示循环部分的边集合.一个路径 π 是可接收的,当且仅当:

$$AllUntil = \{Flag(e) \mid e \in inf_e(e)\}.$$

1.3　空指针解引用的 LTL 描述

空指针解引用问题指:当程序中使用某个指针变量的解引用形式时,该指针变量为空.在该情况下会导致程序出错甚至崩溃.为了使用 LTL 公式表示空指针解引用问题,我们首先定义 2 个命题:

命题 1. d:表示指针 $p \neq$ null.d 为真当且仅当在当前状态下指针变量 $p \neq$ null.

命题 2. r:表示使用了指针 p 的解引用形式,即 $^*p$ 或 $p \rightarrow$.r 为真当且仅当当前状态使用了指针的解引用形式.

因此,指针变量 p 的空指针解引用可以描述为:程序中某一时刻 $p =$ null 且出现了对 p 的解引用形式.在模型检测中,一般描述期望的性质:即该程序中不存在空指针解引用.该期望的性质用 LTL 公式描述为 $\square(r \rightarrow d)$.

为了检测待测程序是否满足性质 $\square(r \rightarrow d)$,理论上我们需要将 $\square(r \rightarrow d)$ 的否定,即 $\neg(\square(r \rightarrow d))$

转换为 TGBA;然后检测程序中是否存在可能的执行路径与所得到的 TGBA 的接收路径相吻合.

对于公式 $\neg(\square(r\rightarrow d))$,我们将其等价的表达为 $\text{true}\cup(\neg d\wedge r)$. 它的 TGBA 如图 3 所示,边的 $Flag$ 集合如表 1 所示. 图 3 中,只有一个 $\cup$ 操作符,它对应的整数为 1,所以,$AllUntil=\{1\}$.

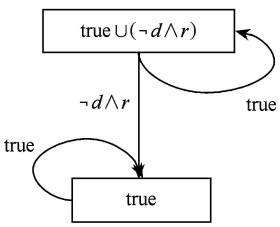


Fig. 3 TGBA of formula $\text{true}\cup(\neg d\wedge r)$.

图 3 公式 $\text{true}\cup(\neg d\wedge r)$的 TGBA

Table 1 Set of *Flag* for Transitions

表 1 迁移的 *Flag* 集合

Transition			*Flag*
predecessor	edge	successor	
$\text{true}\cup(\neg d\wedge r)$	$\neg d\wedge r$	true	{1}
$\text{true}\cup(\neg d\wedge r)$	true	$\text{true}\cup(\neg d\wedge r)$	{}
true	true	true	{1}

2 空指针解引用性质的 LTL 公式生成

如 1.3 节所述,程序中不存在空指针解引用的性质可以通过 LTL 公式 $\square(r\rightarrow d)$来描述. 因此,我们可以看出,对于不同的指针变量公式是不变的. 但是命题 d 和 r 的具体定义不同,它们要对应具体的指针变量. 因此,基于这种模式,本节给出了针对任意指针变量的 LTL 公式形式方法.

在程序中,指针变量的定义存在 2 种情况:1) 声明语句,可以以全局或局部变量的形式出现;2) 函数的形参,出现在函数的定义中. 为了对所有的指针变量形成相应的 LTL 公式,我们首先通过分析程序的语法树,提取所有的指针变量,并记录相关的变量名称、类型和作用域(通过所在函数的名称来体现,其中全局变量略去函数名称);然后针对不同的指针变量,例如变量 v,定义命题 d 和 r:

$$d:\text{函数名}::v!=0$$

$$r:\text{函数名}::*v$$

具体的指针变量的提取算法如算法 1 所示:

算法 1. 指针变量的提取.

输入:控制流自动机 cfa、保存变量信息的集合 $varInfo$、保存函数信息的集合 $funcInfo$、保存类型信息的集合 $typeInfo$;

输出:$varInfo$.

① for $node$ in cfa do /* 对 cfa 中每一个结点 */

② if $node$ 是函数入口结点 then /* 提取函数参数 */

③ $paras=extractFuncInfo(node, funcInfo)$;

④ for $para$ in $paras$ do /* 对每一个参数 */

⑤ if $para$ 是一个指针变量 then

⑥ $extractVarInfo(para,varInfo)$; /* 提取参数 */

⑦ end if

⑧ end for

⑨ end if

⑩ $edges=getLeavingEdges(node)$; /* 得到出边集合 */

⑪ while $edge\in edges$ do

⑫ if $edge$ 是变量声明边且声明的变量为指针变量 then

⑬ $extractVarInfo(edge,varInfo)$; /* 提取变量 */

⑭ else if $edge$ 是类型声明边 then

⑮ $extractTypeInfo(edge,typeInfo)$; /* 提取类型 */

⑯ end if

⑰ end while

⑱ end for

⑲ return $varInfo$.

在算法 1 中,为了方便分析,提取指针变量时,同时记录了所在函数和复杂类型的信息. 在得到所有变量信息后,可以任意选择一个变量,生成对应的 LTL 公式进行检测,也可以选择对所有变量自动地逐个形成公式并检测.

3 基于 CEGAR 的空指针解引用模型检测

本节首先描述了模型检测中使用的 2 种数据结构:控制流自动机(control flow antomata, CFA)和抽象可达图(abstract reachability graph, ARG);然后,详细地介绍了基于 CEGAR 的空指针解引用模型检测方法.

3.1　CFA 和 ARG

CFA 是一个有向图. 其中,结点代表程序的位置,通过一个程序计数器的值来编号;边表示的是连接 2 个结点的操作,对应 C 程序的语句,包括赋值操作、分支语句、函数调用和 return 语句等.

一个程序的 CFA 可以定义为一个四元组 $A=\{L,O,l_{\text{entry}},l_{\text{exit}}\}$,其中,$L$ 是程序位置的集合;$O\subseteq L\times ops\times L$ 是控制流边的结合,表示控制流 2 个相邻结点之间执行的操作;$l_{\text{entry}}\in L$ 是程序的起始位置;$l_{\text{exit}}\in L$ 是程序的终止位置.

对于 CFA 中的一个程序位置 $l\in L$,l 的前驱集合定义为 $Pre(l)=\{l'\mid(l',ops,l)\in O\}$,$l$ 的后继集合定义为 $Succ(l)=\{l'\mid(l,ops,l')\in O\}$. 以下面代码为例,它的 CFA 如图 4 所示.

```
① void main(){
②       int i;
③       int *p;
④       if i<2
⑤          p=&i;
⑥       end if
⑦       *p=1;
⑧    }
```

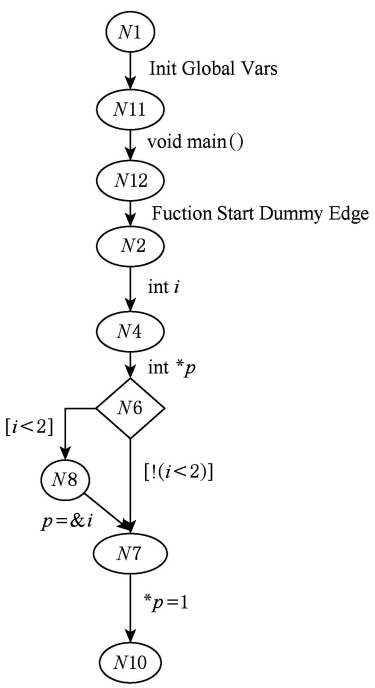


Fig. 4　CFA.

图 4　控制流自动机

ARG 是在考虑 CFA 中每个位置的谓词真值的情况下,通过展开 CFA 而得到的. 它表示的是程序可达的状态空间.

给定一个程序的控制流图 $A=\{L,O,l_{\text{entry}},l_{\text{exit}}\}$ 和相关的谓词集合 $Pred$. 该程序的 ARG 为一个四元组 $G=\{N,E,N_{\text{init}},N_{\text{f}}\}$. 其中,$N$ 是结点集合,对于任意的 $n\in N$,$n=(l,Pred(l))$,$l\in L$,$Pred:L\to 2^{Pred}$ 称为可达域,表示位置 l 与 $Pred$ 的映射关系;$E\subseteq N\times ops\times N$ 是控制流边的子集;N_{init} 和 N_{f} 是初始结点集合和终止结点集合.

为了方便,对于一个结点 $n=(l,Pred(l))\in N$,我们用 $n.l$ 表示 l;用 $n.r$ 表示 $Pred(l)$. 如果 $n\in N_{\text{init}}(N_{\text{f}})$,则 $n.l=l_{\text{entry}}(l_{\text{exit}})$.

实际上,ARG 是程序的抽象模型,其中一条路径对应于程序的一次执行. 一个结点 n 的可达域 $(n.r)$ 表示假设程序沿着一个操作序列执行,从起始结点到结点 n 得到的可达状态的近似集合. 因此,考虑的谓词越少,ARG 就越小,验证的效率越高. 一个 CFA 可以看成一个特殊的 ARG,其中,所有结点的可达域都为空. 但是在 ARG 中,一条不满足期望性质的路径可能在实际程序中不存在. 然而,基于 CEGAR 的懒惰抽象(lazy abstraction)[21] 方法可以通过插值得到更多的谓词,从而有效地消除虚假反例.

3.2　基于 CEGAR 的 C 程序空指针解引用检测

基于 CEGAR 的 C 程序空指针解引用模型检测的整体框架如图 5 所示. 1)使用 LTL 公式 P 描述期望的性质,即程序中不存在空指针解引用问题;2)检测程序是否可以满足 P 的否定. 如果可以满足,那么程序中存在可能的执行路径与期望的性质相违背,否则程序满足期望的性质. 程序的状态空间往往非常大,甚至无穷大,因此,本文使用谓词抽象的方法得到程序的一个抽象模型. 在初始的抽象模型中,仅关心与性质 P 相关的所有谓词. 如图 5 所示,通过抽象我们从待测程序 Pro 得到它的抽象模型 M';然后检测 M' 是否可以满足性质的否定,即 $\neg P$. 若 M' 的所有路径都不能满足 $\neg P$,可以给出结论:程序 Pro 中没有空指针解引用发生;反之,如果在 M' 中找到一条路径可以满足 $\neg P$,需要进一步检测该路径的真实性,即该路径是否在源程序中存在.

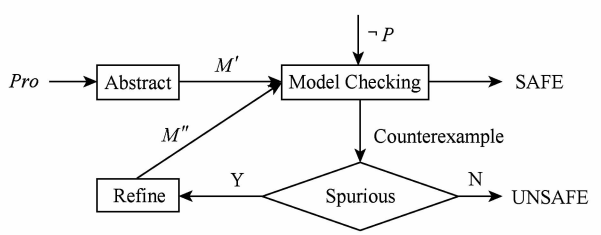


Fig. 5　Framework of CEGAR based model checking.

图 5　基于 CEGAR 的模型检测整体框架

若该路径是真实存在的路径,那么我们找到了程序中的一个空指针解引用错误;否则,如果该路径是虚假的,需要进一步通过添加新的谓词来对 M' 进行细化. 重复以上过程,直到发现一个错误,或者证明程序中没有错误为止. 在 CPAChecker 的 CEGAR 算法中,细化操作是通过调用 SMT 求解器来完成的,它将一条反例路径转化为 SMT 求解器可处理的公式序列,然后通过求解器来求解. 新的谓词是通过 Craig 差值[22]算法得到的.

本文的算法是在现有的软件模型检测工具 CPAChecker 的基础上实现. CPAChecker 自身不能支持时序逻辑性质的检测,它需要首先在程序中对待检测的问题进行插桩,然后使用上述 CEGAR 循环进行检测. 通过插桩的方式,只需要对能到达插桩的位置的路径进行检测,从这个角度看,插桩的方法隐式地缩小检测范围,在一定程度上对检测路径进行了筛选,而且,如果程序中有错误,那么错误肯定在插桩的位置,方便定位错误. CEGAR 算法检测流程图如图 6 所示. 在图 6 中,首先读入插桩后的 C 程序,并生成程序的 CFA;然后根据插桩信息,展开 CFA,提取谓词,生成 ARG. 如果错误标签不可达,即不存在反例路径(counterexample, CE),说明程序安全;否则需判断反例路径是否虚假. 如果反例路径是虚假的,说明考虑的谓词太少,需要提取新的谓词进一步细化,然后继续检测;如果反例路径不是虚假的,则说明程序不安全,输出反例路径.

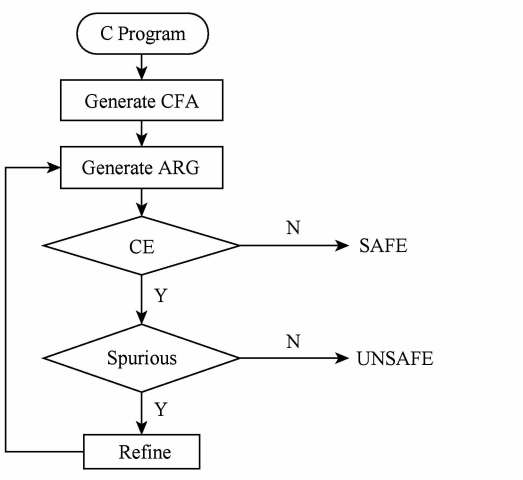


Fig. 6 Flow diagram of the CEGAR algorithm in CPAChecker.

图 6 CPAChecker 中 CEGAR 算法的流程图

3.3 on-the-fly 检测

为了缓解验证过程中的状态空间,本文结合 CFA 和 LTL 性质,采用 on-the-fly 的方式逐步生成 ARG,同时对性质进行检测.

算法 2 给出了基于 CEGAR 的 C 程序 LTL 模型检测的算法. 该算法的输入为待检测的程序 M 和所检查的指针变量 V. 其中,变量 *target* 表示目标状态,即如果存在反例路径,则 *target* 为该反例路径最后一个状态;否则 *target* 为空. 行③根据变量 V 生成原子命题集合 AP,再根据 AP 生成性质 P. 行⑦生成性质 P 对应的范式图,并计算 *AllUntil* 集合. 行⑧~⑱是基于反例-抽象-细化模型检测的主要过程. 如果布尔变量 *success* 为 true,则表示反例路径是虚假的,并且已经细化成功,生成细化的 ARG;反之,*success* 为 false,则说明反例路径为真,程序不安全. 行⑨是本文提出的算法的核心部分,算法 3 给出了该部分的详细过程.

算法 2. *LTLModelCheck*(M,V).

输入:带验证的程序 M、待检测的指针变量 V;

输出:true 或 false.

① *target*=null, *success*=false;
② $M\rightarrow cfa$; /* 生成 CFA */
③ $AP=generate(V)$, $P=Property(AP)$;
④ $S_0=initARGState(cfa,AP,P)$; /* 初始状态 */
⑤ $reached=\{S_0\}$; /* 可达状态集合 */
⑥ $waitlist=\{S_0\}$; /* 一个栈,存放未处理的状态 */
⑦ $AllUntil=LTL2NF(P)$;
⑧ do{
⑨ *target*=*generateARG* (*cfa*,*reached*, *waitlist*,*AllUntil*);
⑩ if *target*!=null then
⑪ *success*=*refine*(*reached*); /* 细化 */
⑫ if !*success* then
⑬ *printCounterExample*(*reached*); /* 输出反例路径 */
⑭ return false;
⑮ end if
⑯ *target*=null;
⑰ end if
⑱ } while(*success*);
⑲ return true.

算法 3. *generateARG*(*cfa*,*reached*,*waitlist*, *AllUntil*).

输入:控制流自动机 *cfa*、可达状态集合 *reached*、用来存放未处理状态的一个栈 *waitlist*、待验证性质对应的 *AlLUntil* 集合 *AllUntil*;

输出:1 个 *ARGState*.

① while *waitlist* !=null do
② $s=getTopWaitlist()$, $p=P(s)$;
③ $TR=nextStateNF(s,p)$;
④ if TR=null then goto ③;
⑤ for T' in TR do
⑥ $sucs=computeSucs(s,cfa)$; /* 后继状态集合 */
⑦ if $sucs$==null then
⑧ if $existCE(reached,s,AllUntil)$ then
⑨ return s;
⑩ end if
⑪ for suc in $sucs$ do
⑫ $CP(s,suc)$; /* 计算后继满足的命题 */
⑬ $CF(s,AllUntil,suc)$; /* 计算 *Flag* 集合 */
⑭ $sucPro=getSucProperty(suc,T')$; /* 计算后继的性质 */
⑮ if $sucPro$==null then
⑯ continue;
⑰ end if
⑱ $reached=reached\cup\{suc\}$;
⑲ if !$stop(suc,reached)$ then
⑳ $waitlist=waitlist\cup\{suc\}$;
㉑ else if $existCE(reached,suc,AllUntil)$ then
㉒ return suc;
㉓ end if
㉔ end for
㉕ end if
㉖ end for
㉗ end while
㉘ return null.

在算法 3 中,语句 $p=P(s)$ 计算状态 s 的性质集合;函数 *stop* 判断后继状态 *suc* 是否被覆盖;函数 *getSucProperty* 计算后继状态满足的性质;函数 *CP* 计算后继状态需要满足的命题;函数 *existCE* 判断是否存在反例路径. 函数 *getSucProperty* 的计算过程如算法 4 所示.

算法 4. *getSucProperty*(S,T).

输入:1 个 *ARG* 状态 S、性质的 1 条迁移 T;

输出:S 的后继状态满足的性质.

① $M=S.getPropositions()$; /* 状态 S 满足的命题集合 */
② *boolean bool*=true;
③ $T:P\xrightarrow{\varepsilon'}P'$;
④ for m in M do /* 对 M 中每一个命题 */
⑤ if $m\wedge\varepsilon'$!=false then
⑥ continue;
⑦ else
⑧ *bool*=false;
⑨ break;
⑩ end if
⑪ end for
⑫ if *bool*==true then
⑬ return P';
⑭ else
⑮ return null;
⑯ end if

算法 4 中,T 为迁移关系,ε'代表的是迁移关系 T 的接受条件. 如果命题集合 M 中的每一个命题与 ε'的交集都不为空,则说明状态 S 满足性质 P'.

函数 *CP* 的计算方法如下:1) 求出初始状态到 S 的路径,记作 *path*. 2)将 *path* 转为 SMT 能够接受的形式,记作 *path formula*. 3)得到前驱状态的命题集合 *pp*. 4) 对集合 *pp* 中的命题 d,做如下操作:将命题 d 转为 SMT 公式 f,加入 *path formula*,并用 SMT 求解. 如果结果为 true,则说明 S 状态满足命题 d;否则,S 状态不满足命题 d,即满足 d 的否定. 对于命题 r,则通过分析对应的 c 表达式来判断其是否可满足.

函数 *existCE* 的计算过程如算法 5 所示. 其中,*path* 为从初始状态到达输入状态 E 的路径. 反例路径是根据 *path* 的循环部分来判断.

算法 5. *existCE*$(reached,E,AllUntil)$.

输入:可达状态集合 *reached*、一个 *ARG* 状态 E、待验证性质对应的 *AlLUntil* 集合 *AllUntil*;

输出:true 或 false.

① $path=getPath(reached,E)$;
② for e in *path* do
③ $Flag=getFlag(e)$; /* 计算 *Flag* 集合 */
④ for i in *Flag* do
⑤ $F_i=F_i\cup\{e\}$;
⑥ end for
⑦ end for
⑧ for i in *AllUntil* do
⑨ if $F_i\wedge path$!=null then
⑩ continue;
⑪ else

⑫ return false;
⑬ end if
⑭ end for
⑮ return true.

4 实验结果分析

为了验证本文提出的方法,我们在 CPAChecker 的基础上开发了 C 程序空指针解引用检测工具 NULPChecker,并通过程序验证 Benchmark 程序[①]对工具的效果进行评估.

4.1 工具实现

本文在现有 CPAChecker 的基础上,利用谓词抽象、CEGAR、on-the-fly 以及时序逻辑等开发了 C 程序空指针解引用检测工具:NULPChecker. 图 7 给出了该工具的主要框架. CFA 模块是基于 Eclipse 平台的插件,它根据 C 程序的语法树生成程序的 CFA,同时,变量提取模块通过分析 CFA 提取指针变量. LTL 模块根据变量提取模块提取的变量,生成 LTL 公式,将 LTL 公式转为与其等价的 TGBA,然后和 ARG 模块进行交互. ARG 模块利用 on-the-fly 的思想,逐步展开 CFA 和 LTL 公式,在展开的同时,检查 CFA 当前状态的命题与公式范式的当前状态是否冲突. 重复此过程,直到不再产生新的状态. 在没有冲突出现时产生的一条完整路径对应一条反例路径,该路径是否真实存在还需进一步判断. 细化模块首先利用 SMT 求解器判断反例的真实性,如果反例路径是虚假的,则采用插值的方法提取新的谓词,重复上述过程直到给出真反例或证明程序正确. 输出模块输出程序的验证结果,验证过程中,可以选择"单个指针"和"所有指针".

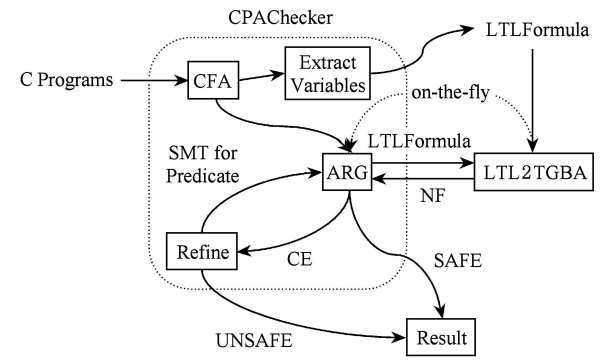


Fig. 7 Core of NULPChecker
图 7 NULPChecker 的中心结构

我们使用 NULPChecker 检测第 3 节中的程序中是否存在空指针解引用问题. 在检测过程中,工具提示程序中存在一个指针变量,如图 8 所示,我们选择检测指针变量 p 是否存在空指针解引用现象. 检测结束后,工具提示存在错误路径,如图 9 所示.

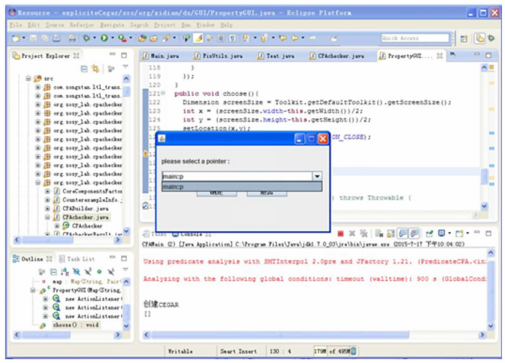


Fig. 8 Choosing pointer.
图 8 指针选择

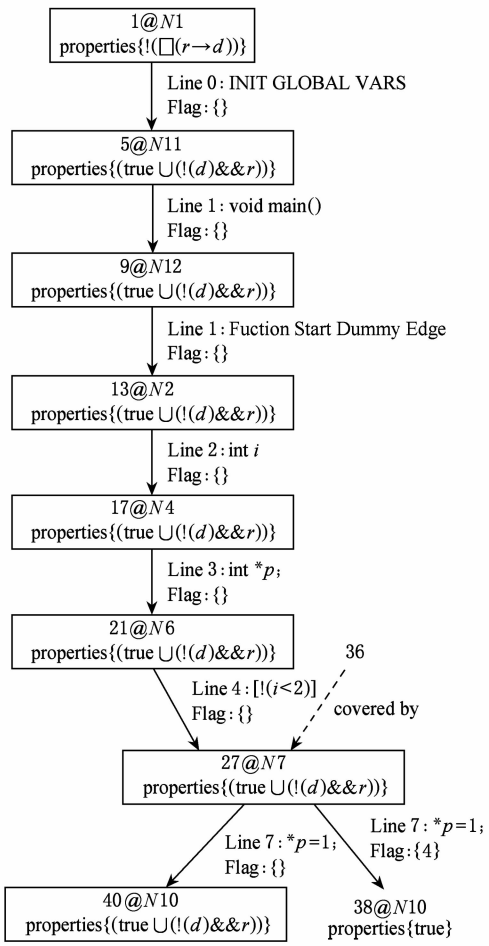


Fig. 9 ARG.
图 9 抽象可达图

① https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp14

其中,到达状态 38@$N10$ 的路径是反例路径.从图 9 可以看出,沿着反例路径,指针 p 为空,且反例路径最后一条迁移关系上使用了指针 p 的解引用形式.

4.2　实验结果

本文通过 NULPChecker 检测了程序验证 Benchmark 程序中的空指针解引用问题,检测结果如表 2 所示.表 2 中,列 1 显示的是测试程序名,列 2 是程序的行数,列 3 是程序中指针变量的总数,列 4 是验证结果安全的指针变量总数,列 5 是出现空指针解引用问题的指针变量总数.

Table 2　Experimental Data

表 2　实验数据

Program	Line	VarNums	Number of SAFE Pointers	Number of UNSAFE Pointers
test-0102_false-valid-memtrack. c	127	17	8	9
test-0236_true. c	182	23	8	15
cs_dekker_true. c	512	1	1	0
merge_sort_false. cil. c	837	54	50	4
43_1a_cilled_true_ok_nondet_linux-43_1a-drivers--clocksource--cs5535-clockevt. ko-ldv_main0_sequence_infinite_withcheck_stateful. cil. out. c	1 107	19	16	3
s3_clnt. blast. 01_false. i. cil	1644	8	7	1
32_1_cilled_true_ok_nondet_linux-3. 4-32_1-drivers--w1--slaves--w1_bq27000. ko-ldv_main0_sequence_infinite_withcheck_stateful. cil. out. c	1 723	30	26	4
m0_false_drivers-staging-comedi-drivers-ni_670x-ko--107_1a--adbbc36-1. c	2 915	29	22	7
m0_false_drivers-staging-comedi-drivers-ni_6527-ko--107_1a--adbbc36-1. c	3 215	32	24	8
m0_true_drivers-hwmon-ibmpex-ko--130_7a--d631323. c	3 397	53	49	4
m0_false_drivers-staging-comedi-drivers-ni_65xx-ko--107_1a--adbbc36-1. c	3 959	110	100	10
m0_true_drivers-staging-comedi-drivers-ni_65xx-ko--107_1a--adbbc36. c	3 989	109	100	9
email_spec1_product12_true. cil. c	4 185	78	78	0
m0_false_drivers-block-virtio_blk-ko--101_1a--39a1d13. c	5 743	60	52	8
usb_urb-drivers-hid-usbhid-usbmouse. ko_false. cil. out. i. pp. cil. c	6 444	144	122	22
module_get_put-drivers-net-pppox. ko_true. cil. out. i. pp. cil. c	6 839	27	25	2

这里需要指出的是,CPAChecker 只支持对加错误标签的位置进行检查,因此,要检查一个指针的所有使用情况,则需要手动地加入很多标签.如果要检测所有的指针,不能保证能够准确地在所有的位置加上合适的标签,而本文的方法无需手动对程序作任何修改.NULPChecker 支持对任意 C 程序中所有空指针解引用的全自动检测,无需人工干预.

5　结束语

本文提出了全自动的基于 CEGAR 的 C 程序空指针解引用检测方法,并开发了相应的支撑工具.本文的贡献在于指出了程序中常见错误的基于 LTL 的验证模式.本文针对的是 C 程序中的空指针解引用问题,但是,所提出的方法也可以用来检测其它问题,如数组越界、除零问题等,这也是本文的后期研究工作.本文给出的模型检测算法目前已经可以对空指针解引用问题进行检测,但是还存在许多缺陷.如果程序存在循环结构,可能会展开多次,最终系统内存不足使程序停止.在分析结构体指针或数组等涉及内存操作的变量时,对变量信息的提取和分析不够完整,会导致程序误报或漏报.程序对 C 语言的一些复杂的语法结构的支持还不够完善,而且调用的系统函数不能识别其作用,也会导致程序分析结果错误.检查的程序规模虽然有所提高,但是跟待测程序的设计结构有很大关系.有时程序的规模不是很大,但函数之间的调用关系可能会很复杂,如出现递归调用或者循环体的循环次数太大而导致程序堆栈和堆不够用.算法本身会保存很多信息来分析程序,这会占用大量内存.这些问题在后续的研究中都要进行分析.

参　考　文　献

[1] Xiao Qing, Gong Yunzhan, Yang Chaohong, et al. Path sensitive static defect detecting method [J]. Journal of Software, 2010, 21(2): 209-217 (in Chinese)
(肖庆,宫云战,杨朝红,等. 一种路径敏感的静态缺陷检测方法[J]. 软件学报, 2010, 21(2): 209-217)

[2] Shapiro M, Horwitz S. Fast and accurate flow-insensitive points-to analysis [C] //Pror of the 24th ACM Symp on Principles of Programming Languages. New York: ACM, 1997: 1-14

[3] Ben H, Lin C. Semi-sparse flow-sensitive pointer analysis [J]. SIGPLAN Notices, 2009, 44(1): 226-238

[4] Dalley James L. The art of software testing [C] //Proc of National Aerospace and Electronics Conf. Piscataway, NJ: IEEE, 1991: 757-760

[5] Repasi T. Software testing-state of the art and current research challenges [C] //Pror of the 5th Int Symp on Applied Computational Intelligence and Informatics. Piscataway, NJ: IEEE, 2009: 47-50

[6] Ellis B J, Ireland A. An integration of program analysis and automated theorem proving [C] //Proc of the 4th Int Conf on Integrated Formal Methods. Berlin: Springer, 2004: 67-86

[7] Clarke E M, Grumberg O, Long D E. Model checking and abstraction [J]. ACM Trans on Programming Languages and Systems, 1994, 16(5): 1512-1542

[8] Jhala R, Rupak M. Software model checking [J]. ACM Computing Surveys, 2009, 41(4): 1-21

[9] Beyer D, Henzinger T A, Jhala R, et al. The software model checker Blast: Applications to software engineering [J]. International Journal on Software Tools for Technology Transfer, 2007, 9(5/6): 505-525

[10] Bayer D, Henzinger T A, Jhala R, et al. Checking memory safety with Blast [C] //Pror of the 8th Int Conf on FASE. Berlin: Springer, 2005: 2-18

[11] Condit J, Harren M, McPeak S, et al. CCURED in the real world [C] //Proc of SIGPLAN Notices. New York: ACM, 2003: 232-244

[12] Beyer D, Keremoglu M E. CPACHECKER: A tool for configurable software verification [C] //Proc of the 23rd Int Conf on Computer Aided Verification. Berlin: Springer, 2011: 184-190

[13] Bayer D, Henzinger T A, Theoduloz G. Configurable software verification: Concretizing the convergence of model checking and program analysis [C] //Proc of the 19th Int Conf on Computer Aided Verification. Berlin: Springer, 2007: 504-518

[14] Clarke E M, Kroening D, Lerda F. A tool for checking ANSI-C programs [C] //Proc of the 10th Int Conf on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2004: 168-176

[15] Charatonik W, Georgieva L, Maier P. Bounded model checking of pointer programs [C] //Proc of the 19th Int Workshop on Computer Science Logic. Berlin: Springer, 2005: 397-412

[16] Frazan A, Madhusudan P, Razavi N, et al. Predicting null-pointer dereferences in concurrent programs [C] //Proc of the 20th ACM SIGSOFT Int Symp on the Foundations of Software Engineering. New York: ACM, 2012: 47-57

[17] Dong Yukun, Gong Yunzhan, Jin Dahai. Null pointer dereference defect detecting based on region-based memory model [J]. Acta Electronica Sinica, 2014, 42(9): 1744-1752 (in Chinese)
(董玉坤, 宫云战, 金大海. 基于区域内存模型的空指针引用缺陷检测[J]. 电子学报, 2014, 42(9): 1744-1752)

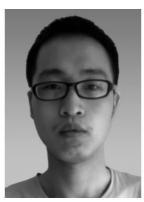
[18] Clarke E. Counterexample-guided abstraction refinement [C] //Proc of the 10th Int Symp on Temporal Representation and Reasoning and the 4th Int Conf on Temporal Logic. Piscataway, NJ: IEEE, 2003: 154-169

[19] Tian Cong, Duan Zhenhua. A note on stutter-invariant PLTL [J]. Information Processing Letters, 2009, 109(13): 663-667

[20] Gastin P, Oddoux D. Fast LTL to Büchi automata translation [C] //Proc of the 13th Int Conf on Computer Aided Verification. Berlin: Springer, 2012: 53-65

[21] McMillan K L. Lazy abstraction with interpolants [C] //Proc of the 18th Int Conf on Computer Aided Verification. Berlin: Springer, 2006: 123-136

[22] Henzinger T A, Jhala R, Majumdar R. Abstractions from proofs [C] //Pror of the 31st ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 2004: 232-244



Duan Zhao, born in 1990. Received his MS degree from the Institute of Computing Theory and Technology, Xidian Universiy, Xi'an, in 2015. PhD candidate. His main research interests include software model checking and program analysis.

Tian Cong, born in 1981. Professor and PhD supervisor in the Institute of Computing Theory and Technology, Xidian University. Member of IEEE. Her main research interests include theories in model checking, temporal logics and automata, formal verification of software systems, and software engineering.

Duan Zhenhua, born in 1948. Professor and PhD supervisor in the Institute of Computing Theory and Technology, Xidian University. Senior member of IEEE. His main research interests include model checking, temporal logics, formal verification of software systems, and temporal logic programming.