基于抽象-验证-细化范例的软件模型检测*)

刘吉锋　孙吉贵
(吉林大学计算机科学与技术学院　长春 130012)
(吉林大学符号计算与知识工程教育部重点实验室　长春 130012)

摘　要　如何保证软件系统的正确性和可靠性是当前软件开发面临的主要问题之一。模型检测作为一种重要的自动化验证技术在软件的分析与验证中正取得越来越多的成功。本文以微软的SLAM和加州大学伯克利分校的BLAST为例综述性地介绍了基于抽象-验证-细化范例的软件模型检测。

关键词　模型检测，软件模型检测，谓词抽象，反例驱动的细化

Software Model Checking Based on Abstract-verify-refine Paradigm

LIU Ji-Feng　SUN Ji-Gui
(College of Computer Science and Technology, Jilin University, Changchun 130012)
(Key Laboratory of Symbolic Computation and Knowledge Engineer of Ministry of Education, Changchun 130012)

Abstract　How to assure the correctness and reliability of software systems is one of the main problems in software development. Being an important automatic verification technique, Model checking is more and more successful in software analysis and verification. This paper presents a survey to software model checking based on abstract-verify-refine paradigm, using SLAM at Microsoft and BLAST at UC Berkeley as examples.

Keywords　Model checking, Software model checking, Predicate abstraction, Counterexample-driven refinement

1　引言

软件开发所面临的主要问题之一是如何保证系统的正确性和可靠性，形式化方法是解决这个问题的重要途径。在诸多形式化方法中，模型检测作为一种重要的自动化验证技术已经在硬件和协议验证领域取得了巨大的成功，把模型检测应用于软件验证已经成为近年来的一个焦点，并且已经取得了一些成就，可以验证几十万行代码。

模型检测的思想是建立系统的模型，把系统中变量的取值情况作为系统的状态，从指定初始状态开始根据状态转换关系生成状态转换路径，代表系统的运行情况；把被验证属性表示为状态转换路径必须满足的具体要求，如果状态转换路径满足这种要求，则系统满足被验证属性，验证结束，否则，系统不满足被验证属性，生成反例路径证明被验证属性如何被破坏。

模型检测基于对系统状态空间进行穷举搜索，但直接对无限状态系统或巨大的有限状态系统进行穷举搜索是不可行的，另一方面，由于许多状态与被验证属性无关，这样做也是不必要的，所以，模型检测工具通常工作在系统的抽象模型之上。但抽象模型有时与系统不一致，所以，如果在验证过程中发现抽象模型有问题，则需要对它做出相应调整。

软件属于无限状态系统，基于抽象-验证-细化范例的方法[1~3]是一种通常的软件模型检测方法。

(1)抽象：抽象的表示程序的具体状态，建立给定程序的抽象模型；

(2)验证：对于被验证属性自动的验证抽象模型，如果比给定程序粗糙的抽象模型没有破坏属性，那么给定程序也不会破坏属性，于是返回"程序正确"；否则，自动产生一个抽象反例路径，证明模型如何破坏属性；

(3)细化：自动检测抽象反例路径是否对应一个程序中的具体反例，如果存在具体反例，则找到了一个程序错误；否则，表明抽象模型过于粗糙，不包含证明程序满足被验证属性所需的足够信息，于是增加信息，使得步骤(1)可以抽象出更精密的抽象模型；goto (1)。

本文以微软的SLAM[1]和加州大学伯克利分校的BLAST[4]为例介绍基于抽象-验证-细化范例的软件模型检测。SLAM和BLAST是用于验证C语言程序安全性属性的模型检测工具，整个过程可以不依赖使用者提供辅助信息。SLAM把抽象-验证-细化三步循环分别用三个工具来实现，BLAST则把这三步集成在一起。

本文的第2、3、4节分别介绍抽象、验证和细化，第5节介绍BLAST如何把这三步循环集成在一起以及BLAST所使用的基于克雷格插值的反例驱动的细化，第6节是相关工作，最后是结论。

2　抽象

软件模型检测使用的抽象技术是谓词抽象[5,6]，思想是把系统中数量众多的具体状态根据某方面的共性映射为数量少得多的抽象状态，以便于模型检测器处理；这些共性用谓词来表示，抽象状态用谓词的布尔组合来表示。谓词抽象已经被应用于硬件和协议的验证，现在被用于验证软件[4,7~9]。

*)国家自然科学基金(60473003)、教育部"新世纪优秀人才支持计划"、博士点基金(20050183065)资助课题。**刘吉锋**　硕士研究生，主要研究方向是模型检测；**孙吉贵**　博士生导师，主要研究方向是人工智能、约束程序、决策支持系统。

SLAM 中用工具 C2BP[8,10] 来进行谓词抽象。C2BP 把给定的 C 语言程序 P 根据谓词集合 E 抽象为布尔程序[11] BP(P,E)。E 可以人为给定,但在细化阶段相应的工具会向 E 中增加谓词,所以 E 初始也可以为空。E 中的谓词是 P 中变量和常量构成的 C 语言布尔表达式。P 中任何可行的执行路径在 BP(P,E)中仍是可行的执行路径。BP(P,E)拥有与 P 相同的控制结构,但只包含布尔变量,一个布尔变量代表一个谓词。例如在图 1 中。对于(a)中的程序 P 和谓词集合 E={x=='x, x=='x+1, x<4, a==2, b==3},可以生成图 1(b)中的布尔程序 BP(P,E)。

在 BP(P,E)中,{a==2}是代表谓词(a==2)的布尔变量,它的取值等于对应谓词的取值,表示 P 中语句的执行对谓词(a==2)取值情况的影响,有 true、false 和 * 三种情况,* 表示不确定。之所以会出现不确定,是因为根据当前谓词集合有时不知道一条语句的执行对相关谓词有什么影响。例如对一条语句"curr=nextCurr;",其中 curr 和 nextCurr 是指针变量,当前谓词集合中包含一个谓词(curr==NULL),但不包含任何涉及 nextCurr 的谓词,则在这条语句执行后,由于当前谓词集合不包含任何关于 nextCurr 的信息,所以谓词(curr==NULL)的真值无法判定,所以把对应布尔变量赋值为 * 。对于取值为 * 的谓词,随机地认为它的真值为 true 或 false。

```
int inc(int x){
x=x+1;
  return x;
}

void main(){
  int a,b,c;
  a=1;
  c=0;
  if(a!=2){
a=2;
  }
  b=inc(a);
  return;
}
```

(a) C 语言程序

```
bool,bool,bool inc(bool {x<4}){
   bool {x=='x},{x=='x+1}
   {x=='x}=true;                    //初始化绑定
                                      谓词
   {x=='x},{x=='x+1}=false,true;    //x=x+1;
  return {x=='x},{x=='x+1},{x<4}; //return x;
}
void main(){
  bool {a==2},{b==3};
  bool prm,rct1,rct2,rct3;
  {a==2}=false;                     //a=1;
  skip;                             //c=0;
  if(*){                            //if 语句
    assume(!{a==2});                //a!=2 的情况
      {a==2}=true;                    //a=2;
  }
  prm=true;                        //计算实参值
  rct1,rct2,rct3=inc(prm);         // b=inc(a);
   {b==3}=true;                     //更新相关谓词
  return;
}
```

(b) 布尔程序

图 1

BP(P,E)中"{x=='x}, {x=='x+1}=false, true;"表示对两个布尔变量的连续赋值。

有些语句不对任何谓词的取值产生影响,比如语句"c=0;",在 BP(P,E)中抽象为 skip。

对于影响谓词取值的语句,如语句"a=0;",在 BP(P,E)中抽象为对相关布尔变量赋值的语句。一个谓词在一条语句执行后的取值取决于它的最弱前置条件。如果一个谓词(或它的非)的最弱前置条件在语句执行前被蕴含,则这个谓词在语句执行后真值为 true(或 false)。例如语句"a=0;",在 BP(P,E)中对应语句"{a==2}=true;"。因为一个指针的间接引用可能是一个变量的别名,所以计算最弱前置条件时要考虑这种情况。例如计算 WP(x=5, * y>6),要考虑 y 是否是指向 x 的指针,所以此时 WP(x=5, * y>6)=((&x=y)∧(5>6))∨((&x!=y)∧(*y>6))。

对于 C 语言程序中的 skip 语句、goto 语句和语句标号,在布尔程序中保持不变。

对于需要条件判断的语句,如 if 语句、while 语句等,在布尔程序中转变成结合 * 和 assume 语句的形式。例如,if 语句 if (C){…} else {…}被转变为 if (*) {assume(C');…} else {assume(¬C');…}的形式,if(*)不进行条件判断,由 assume 语句来进行。如果 C 恰好就是一个谓词(或谓词的非),那么 C'就是 C 对应的布尔变量(或布尔变量的非),否则 C'是被 C 蕴涵的最强的谓词(或谓词的非)对应的布尔变量(或布尔变量的非)。例如,C=(y==3),谓词集合中不包含(y==3)这个谓词,但包含谓词(y<6),于是 C'={y<6}。

SLAM 中的谓词抽象称为多态谓词抽象[10]。抽象一个函数时,引入象征常量保存调用传入的实参值,在函数内涉及到的实参值用象征常量代替,与具体的实参变量和实参值无关,以此来建立与具体调用无关的抽象,使得对不同实参值的调用都可以重用同一个抽象。谓词的作用域取决于谓词中变量的作用域。包含全局变量的谓词是全局谓词,在抽象程序的每条语句时都要考虑对全局谓词取值的影响;只包含某个函数的局部变量的谓词是这个函数的局部谓词,只在抽象这个函数时考虑对它取值的影响。对图 1 中的例子,抽象函数 inc 时只需考虑谓词集合 E_{inc}={x=='x, x=='x+1, x<4},其中的谓词都是局部谓词。

C2BP 抽象 C 语言程序时,先生成每个函数的接口,然后再抽象每个函数内的语句。假设函数 R 中只有一条返回语句"return r;",函数 R 的接口是六元组(F_R, r, S_R, E_f, E_r, B_R)。F_R 是形参变量集合,r 是函数的返回变量,S_R 是象征常量的集合。象征常量'f 保存传递给形参变量 f 的实参值,图 1(b)布尔程序中的'x 就是对应于 x 的象征常量;如果 f 是指针,对于所有合法的 $*^k$f(k∈N, $*^k$ 表示 k 个 *),引入象征常量'$*^k$f 保存函数开始时 $*^k$f 的值。E_f 中包含的谓词是包含形参变量而不包含 R 的局部变量和象征常量的谓词,称为形参谓词。E_r 中包含的谓词是包含返回变量或象征常量而不包含除 r 以外任何局部变量和形参变量的谓词,称为返回谓词。对于形参变量 f 和象征常量'f,在 R 的谓词集合中还要加入谓词(f=='f),表示在函数开始时形参变量和象征常量相等,这样的谓词称为绑定谓词,构成集合 B_R。如果 f 是

指针，对于所有合法的 $*^k f$，B_k 中还包括绑定谓词（$*^k{}' f == *^k f$）。绑定谓词对应的布尔变量在函数开始处被赋值为 true。

对于函数调用语句 $v=R(a_1,\cdots,a_j)$，它的抽象分三步计算：计算实参值，生成布尔程序中对应的调用，更新调用函数布尔变量的取值。

首先，C2BP 计算布尔程序中需要传递的实参值。因为布尔程序中考虑的是对谓词取值的影响，所以布尔程序中的形参变量是形参谓词对应的布尔变量，数量不一定等于形参的数量。计算需要传递的实参值实际上是根据实参变量计算形参谓词的取值。具体做法是把形参谓词中的形参变量替换为实参变量，在调用函数中为每个这样得到的新谓词 α_i 引入一个新的局部布尔变量 prm_i 来保存其真值，如果 α_i（或它的非）在调用前被蕴涵，那么 prm_i 取值为 true（或 false），对应的返回谓词的布尔变量在调用时被赋值为 prm_i 的值。例如在图 1(b)中，语句“prm=true;”来源于根据实参变量 a 计算出形参谓词（x<4）在调用前被蕴涵。

其次，生成布尔程序中的调用语句。为了反映被调函数对调用函数中谓词取值的影响，布尔程序中被调函数返回的不是 r 的值，而是所有返回谓词的取值情况，用于更新调用函数中相关谓词的取值。具体做法是在被调函数中把“return r;”转变为“return $\{\varphi_1\},\cdots,\{\varphi_k\}$;”（$\varphi_i$ 为返回谓词），在调用函数中为每个返回谓词 φ_i 引入一个新的局部布尔变量 ret_i 来保存其真值。这样，在布尔程序中函数调用语句被转换为 $ret_1,\cdots,ret_k=R(prm_1,\cdots,prm_n)$ 的形式。

最后，更新调用函数中相关谓词的取值。在调用函数中，所有包含 v 的谓词都需要被更新。一些全局变量可能被 R 更改，所以所有包含这些全局变量的谓词也需要被更新。因为调用函数中可能有指针指向 v 或全局变量，所以包含这些指针的谓词也需要被更新。另外，调用时可能传递指向调用函数局部变量的指针，那么这些局部变量也可能被更改，所以相关谓词也需要被更新。C2BP 使用可能别名分析[12]来生成需要更新的谓词集合的上近似。更新的具体做法是把返回谓词中出现的 r 替换为 v，象征常量 $'f$ 替换为对应的实参变量 a_i，$'*^k f$ 替换为 $*^k a_i$，这样返回谓词就被翻译成了调用函数中的形式，然后结合调用函数中原有的谓词取值情况，判断哪些新的取值情况被蕴涵，据此来更新相关布尔变量的取值。

这样，一个 C 语言程序 P 就被转化为对应的布尔程序 BP(P,E)，BP(P,E)反映了 P 中每条语句的执行对谓词取值的影响，即对某些共性的影响。在 BP(P,E)的某一点，这些影响的累积就是 P 的抽象状态，这些影响就是抽象状态的转换关系。因为可以根据 BP(P,E)得到 P 的抽象状态和抽象状态的转换关系，所以可以生成 P 的抽象状态转换路径，对 P 进行验证。

3 验证

验证的过程就是判定状态转换路径是否满足特定要求的过程。SLAM 和 BLAST 利用判定一条带有特定标号的语句在抽象状态转换路径中是否可到达来实现软件时态安全性属性的验证[13,4]。如果从程序的一个初始状态开始最终可以执行某条语句，则这条语句是可到达的。如果到达某条语句意味着破坏被验证属性，则可以根据这条语句是否可到达来判定被验证属性是否成立。

BEBOP[13,14]是 SLAM 中的模型检测器。对于布尔程序 BP(P,E)和一条带有特定标号的语句 s_{error}，BEBOP 判定 s_{error} 在布尔程序中是否可到达。如果 s_{error} 可到达，BEBOP 生成一条从初始状态对应语句 s_0 到 s_{error} 的错误轨迹，代表抽象反例路径。BEBOP 的算法根据 RHS[15,16]算法修改而成。

BEBOP 根据 BP(P,E)构建一个控制流图，表示 BP(P,E)的控制结构。控制流图的每个节点 v 对应于 BP(P,E)中的一条语句 s_v，v 的后继节点对应于 s_v 的后继语句。BEBOP 从 s_0 对应的初始节点 v_0 开始为每个节点计算路径边，如果一个节点对应于函数调用语句，则为它计算扼要边。

对于节点 v，它的一个路径边为$\langle\Omega_e,\Omega_v\rangle$，$s_e$ 是包含 s_v 的函数的第一条语句，Ω_e 和 Ω_v 分别表示 s_e 和 s_v 执行前 BP(P,E)中可见的布尔变量的取值情况。对于不在当前作用域内的谓词，考虑它们的取值情况对验证没有意义，所以考虑可见布尔变量的取值情况就可以表示抽象状态。$\langle\Omega_e,\Omega_v\rangle$表示存在两段执行路径，一段从 s_0 到 s_e，另一段从 s_e 到 s_v。本质上 v 的一个路径边代表了一种达到 s_v 的可能执行情况。

如果 v 对应于函数调用语句，那么它的一个扼要边为$\langle\Omega_1,\Omega_2\rangle$，$\Omega_1$ 和 Ω_2 分别表示被调函数执行前后调用函数中可见布尔变量的取值情况。本质上 v 的一个扼要边代表了一种可能的函数调用和相应的影响。

BEBOP 在计算路径边和扼要边时维持一个工作表，初始其中只有 v_0。每当为工作表中的一个节点 v 计算了一个路径边（或扼要边）就把 v 从工作表中删除，然后根据 v 的路径边（或扼要边）判断可以为 v 的哪些后继节点计算路径边（或扼要边），等价于根据当前抽象状态判断在布尔程序中可以执行哪些后继语句。把这些可以处理的节点加入工作表，继续处理。如此这样不断扩展，有可能再次遇到 v，这意味着一种新的达到 s_v 的执行情况，需要为 v 计算新的路径边（或扼要边）。BEBOP 保存所有节点的路径边集合（或扼要边集合）。如果算法结束时 s_{error} 对应节点 v_{error} 的路径边集合（或扼要边集合）为空，则 s_{error} 是不可到达的，P 满足被验证属性，结束验证；否则，s_{error} 是可到达的，生成错误轨迹。

BEBOP 生成从 s_0 到 s_{error} 的最短执行轨迹作为错误轨迹，这条最短执行轨迹由对应的抽象反例路径上每条语句的唯一编号和每条语句执行前可见布尔变量的取值情况组成。为了找到最短执行轨迹，对于 v 的每个路径边（或扼要边），BEBOP 还保存对应的从 v_0 到 v 的路径长度，据此把路径边集合（或扼要边集合）划分为不同的等价类。算法结束时可以得知到达 v_{error} 的最短路径长度，然后从对应的等价类中任取一个路径边（或扼要边），计算对应的轨迹。具体计算方法是先生成 s_{error} 的编号和对应的可见布尔变量取值情况，然后寻找 v_{error} 的一个前驱节点 u，u 的一个路径边（或扼要边）对应于一条长度为 r−1 的路径，生成 u 对应语句的编号和可见布尔变量取值情况，如此不断逆向处理，最终可以生成整条轨迹。

4 细化

对于一条抽象反例路径，如果对应的具体路径是可行的，则意味着找到了一个真正的系统错误。但是，由于抽象模型比具体系统粗糙，所以有可能引入虚假反例路径。例如在第 2 节中曾提到，程序 P 中的条件（y==3）在布尔程序 BP(P,E)中被抽象为（y<6），这样就使得在 BP(P,E)中本来只有在（y==3）时才能执行的语句在其他一些情况下也可以执行，可能导致出现 P 中不会出现的错误。虚假反例路径对应于具体系统中的不可行执行路径。如果经判定反例路径是虚假

的，则应该把它从抽象模型中排除，但是根据当前信息进行抽象得到的抽象模型必然导致这条虚假反例路径的出现，所以必须适当的增加信息，使得可以抽象出更加精细的排除了这条虚假反例路径的抽象模型。这些用于排除虚假反例路径的信息是通过对不可行路径进行分析而得到的，所以这种细化被称为反例驱动的细化[17,18,13,19]。

SLAM 中用工具 NEWTON 来进行反例驱动的细化[20]。如果错误轨迹对应的具体反例路径 p 是可行的，则输出这条错误轨迹，指示一个真正的错误；如果 p 不可行，则这个反例是虚假的，生成用于排除虚假反例路径的谓词，添加到谓词抽象的谓词集合中。即使谓词抽象的初始谓词集合为空，NEWTON 也会找到验证所需的足够谓词，所以 SLAM 不需要人为提供辅助信息。

NEWTON 把 p 预处理为只包含赋值语句、assume 语句、函数调用语句和返回语句的等价路径 p'。具体反例路径是程序 P 中被执行的语句的序列，等价于一个顺序语句序列。在路径中，skip 可以省略；goto 语句转向的语句就在 goto 语句之后，所以 goto 语句也可以省略；对于需要条件判断的语句，由于具体反例路径已经表明在执行时作了什么判断，所以只保留这种判断对应的 assume 语句即可；除函数调用语句以外影响变量取值（即影响系统状态）的语句等价于赋值语句，所以可以替换为赋值语句。因此，经过这样预处理得到的路径 p' 与原来的具体反例路径 p 对系统状态的影响是等价的。模型检测关注的是系统的状态和状态的转变，所以可以用 p' 代表 p。

在程序 P 中，正是条件判断决定了语句执行的方向，决定了是否能执行导致错误出现的语句块，而在一个语句块中，语句只是被顺序执行而已。所以，如果一条具体反例路径是可行的，那么路径中所有条件判断做出的选择都应该是符合条件的合理选择；如果具体反例路径不可行，必然在验证时做出了在 BP(P,E) 中符合条件而在 P 中不符合条件的不合理选择，导致执行了不该执行的语句。对于 p'，NEWTON 参考变量的取值情况构建一个表示 p 中条件判断做出的选择的公式 e，p 是否可行对应于 e 是否可满足。对于不可行路径，NEWTON 计算它的解释，用于生成排除对应虚假路径的谓词。

NEWTON 用象征常量代表变量的取值。当 p' 中语句 s_i 最先使用一个变量 x 的值并且没有任何象征常量可以表示 x 的值的时候，NEWTON 引入一个象征 θ_x 来代表 x 的值，并在 s_i 前插入语句"$x=\theta_x$;"，这样的语句被称为注释；对于可以根据其他变量的值来确定取值的变量，用相关的象征常量表式它们的取值。p' 中用 assume 语句来表示条件判断做出的选择，如果某条 assume 语句的条件根据条件中变量的取值情况被前面所有 assume 语句的条件的合取所蕴涵，则为这个条件中没有自己的象征常量的变量引入新的象征常量，并在这条 assume 语句前插入新的注释，利用象征常量的差异消除蕴涵，使得这个条件在构建公式 e 时也有意义。在调用函数时，为了避免用实参的象征常量表示被调函数中变量的取值，与谓词抽象中一样，为每个形参引入一个象征常量，其值等于实参值。对于指针，除了引入象征常量表式指针的值，还引入象征常量表式可以根据指针访问的值。

语境是三元组$\langle\Omega,\Phi,\Pi\rangle$，其中 Ω 是一条语句执行前各变量用象征常量表示的取值情况的集合，称为存储；Φ 是用象征常量表示的所有 assume 语句的条件的集合，称为条件；Π 是各个变量曾经的取值的集合，称为历史。

NEWTON 的具体做法是把语境作为最强后置条件，在 p' 的起点以空语境开始不断计算新的语境。对于除 assume 语句外的语句，更新 Ω 和 Π；对于 assume 语句，更新 Φ。在更新时，还要标出 Ω 和 Φ 中的元素依赖 Ω 和 Π 中的哪些元素。$\wedge_{c\in\Phi}c$ 就是公式 e。如果处理了某条 assume 语句后出现 $\wedge_{c\in\Phi}c$ 不可满足的情况，则路径是不可行的。如果 p 是可行的，则根据变量的取值情况 p 中所有条件判断做出的选择都是符合条件的，此时公式 e 根据变量的取值情况是可满足的；如果 p 不可行，则根据变量的取值情况 p 中有的条件判断做出的选择不符合条件，此时公式 e 根据变量的取值情况是不可满足的。不符合条件的选择来源于抽象模型没有关注某些信息，如果关注了必要的信息就不会做出不符合条件的选择。例如，p' 为语句序列"$x=\theta_x$; $y=2x$; assume($y=x$); ERROR:;"，表示($y=x$)的公式 c 为($\theta_x=2\theta_x$)，显然 e 不可满足，如果在抽象时关注了 y 是否等于 x，那么在验证时就不会认为($y=x$)的分支可以执行。当 e 不可满足时，NEWTON 从 Φ 中删除一个条件，如果 $\wedge_{c\in\Phi}c$ 仍旧不可满足，则这个条件与虚假反例无关，可以被忽略；否则，保留这个条件。这样不断处理，会得到最小的解释 p 不可行的 Φ，表示不符合条件的选择和相关的选择，如果谓词抽象时考虑这些信息就可以排除这条虚假反例路径。根据 Ω 和 Φ 中元素与 Ω 和 Π 中元素的依赖关系和插入的注释，可以把最小 Φ 中的条件转换为变量表示的形式，这就是谓词抽象需要增加的谓词。

把这些谓词加入谓词集合 E，根据新的谓词集合进行抽象，因为考虑了新增谓词的取值情况，所以抽象模型中不包含这条虚假反例路径，在验证阶段不会走向这条虚假反例路径。例如对上段中的例子，在谓词集合中加入谓词($y=x$)，对语句"$y=2x$;"进行谓词抽象后布尔变量{$y=x$}的值为 false，这就排除了转入($y=x$)为 true 的分支的可能性，在验证时不会因为执行了只有在($y=x$)为 true 时才能执行的语句而导致错误出现。

如此这样按照抽象-验证-细化的范例继续进行，就可以继续验证程序 P 是否真正满足被验证属性。

5 BLAST

抽象-验证-细化这三步循环在步骤 1 和步骤 2 都是计算难题。BLAST 采用被称作 lazy abstraction 的方法，集成并优化了这三步循环，极大的提升了性能[4]。Lazy abstraction 的思想是由验证驱动，不断按需求抽象和细化一个抽象模型，把抽象、验证和细化结合在一起进行。因为对程序的某一点只按需求细化到足够的精度，所以模型的不同部分有不同的精度。

BLAST 把每个函数预处理为控制流自动机。自动机的节点代表程序的特定位置，自动机的边被标上由若干条语句组成的程序片断或条件判断的条件的真值，表示如何从一个位置执行到另一个位置。

然后，BLAST 根据各个控制流自动机和谓词集合从程序的初始位置出发，深度优先的计算程序在各个位置的抽象状态，建立一棵搜索树，判断是否能执行意味着错误的语句，达到错误状态。搜索树的节点对应于抽象状态，标有一个谓词集合和其中谓词的布尔组合，谓词集合表示在此处关注的事实和抽象精度，布尔组合表示此处的抽象状态。

抽象状态的计算是根据一个谓词的最弱前置条件在前一

个状态是否被蕴涵来判断这个谓词在当前的取值。计算了一个抽象状态后，根据这个抽象状态判断搜索树可以如何扩展，扩展搜索树并为扩展出的节点计算抽象状态。这样边计算抽象状态边搜索即是同时进行抽象与验证。如果一个节点的抽象状态是所有访问过节点的抽象状态集合的子集，这个节点称为被覆盖节点，可以在这个节点停止继续搜索，因为从这个节点可以进行的扩展在前面的节点也可以进行。每当遇到被覆盖节点就回溯，按深度优先顺序选择没有扩展的分支继续搜索。如果达到了错误状态，则转入细化阶段，否则，当搜索树无法再扩展时就找到了所有可到达的状态，可以认定程序没有错误。

如果在构建搜索树时达到了错误状态，那么在搜索树的根节点到错误状态节点的路径上的语句就构成了具体反例路径。为了判定这条路径是否可行，构建一个路径公式，具体反例路径是否可行对应于路径公式是否可满足。为了构建路径公式，要为每条语句生成变量值的约束，表示每条语句对变量值的影响。BLAST 也用赋值语句、assume 语句、函数调用语句和返回语句来表示路径，所以只为这四种语句生成约束。赋值语句的约束是把赋值语句中的变量 x 替换为常量⟨x,i⟩，其中 i 是 x 的不同取值的编号。如果 x 是指针变量，用⟨*⟨x,i⟩,j⟩表示 *x。生成包含指针变量的约束时可能还要生成关于指针的间接引用的约束，一条语句的所有约束用合取符号连接。assume 语句的约束就是它的条件，其中的变量也作替换。对于函数调用语句，引入象征常量代表实参值，函数调用语句的约束是把实参值赋值给象征常量的约束，并且认为在被调函数的第一条语句前存在 assume 语句，其条件就是形参变量等于象征常量。调用函数时遇到的指针变量用与 SLAM 类似的方法处理。返回语句的约束是把返回值赋值给调用函数中获取返回值的变量的约束。具体反例路径上所有约束的合取就是路径公式，如果具体反例路径是可行的，则根据变量的取值在条件判断时做出的选择是合理的，路径公式是可满足的。

如果路径公式不可满足，BLAST 应用克雷格插值来计算排除这条虚假反例路径所需的谓词，然后在搜索树中重新扩展这条路径[21]。子句集 A 和 B 的克雷格插值式[22]是满足如下要求的公式 ϕ：(1)A 蕴涵 ϕ；(2)ϕ 与 B 不一致；(3)ϕ 只包含 A 和 B 的公共变量。可以从 A 和 B 不一致的归结证明在线性时间内生成 A 和 B 的克雷格插值式[23]。

把路径公式以某个抽象状态为界分成分别表示前后两段路径的子句集 A 和 B，由此得到的克雷格插值式 ϕ 被 A 蕴涵，而且由 A 和 B 的公共变量构成，所以 ϕ 表示的是分割点处具体状态集合的上近似，即前半段路径的可到达状态的上近似，又因为 ϕ 与 B 不一致，所以从 ϕ 表示的状态出发无法执行后半段路径的所有语句，达到错误状态。把 ϕ 翻译成若干谓词，加入分割点处的谓词集合，在重新扩展这一点时根据新的谓词集合计算抽象状态，再根据得到的抽象状态决定如何继续扩展。由于原有谓词集合可能为空，也可能不为空，所以新的抽象状态表示的具体状态集合可能等于 ϕ 表示的具体状态集合，也可能是它的子集，但从其中的任何状态出发都无法执行后半段路径。在具体反例路径上每个抽象状态的位置都这样计算克雷格插值式并用获得的谓词补充该处的谓词集合，则在重新扩展这条路径时必然在某一点转向其他方向。

BLAST 边抽象边验证，且只抽象到能够排除虚假反例的精度为止，这样就避免了独立的代价高昂的抽象模型构建阶段。另外，BLAST 只对抽象模型的当前部分进行验证，这样就避免了对无关部分进行不必要的搜索。所以，BLAST 减低了抽象和验证阶段的计算困难。根据克雷格插值式为某一点得到的谓词只对这一点有意义，在其他地方不必考虑这些谓词，所以每个抽象状态都有自己的谓词集合，这样就避免了考虑无关谓词的取值所花费的巨大代价。

6 相关工作

MAGIC 是另一个基于抽象-验证-细化范例的软件模型检测工具，使用了组合推理，用于验证组件的规格说明和 C 语言实现是否一致[24]。MAGIC 把程序抽象为 LTS(labeled transition systems) M_{Imp}，然后验证描述规格说明的 LTS M_{Spec} 是否是 M_{Imp} 的安全抽象。如果答案是肯定的，则程序满足属性，验证结束；否则，判断反例是否是真实的，如果反例是虚假的，则计算一个改进的 M_{Imp}，继续验证。ESC/Java 是使用谓词抽象验证 Java 程序的模型检测工具[25,26]，它把 Java 程序和规格说明翻译为称作验证条件的逻辑公式，根据验证条件是否可满足来判定程序是否正确，由于抽象的原因 ESC/Java 可能报告虚假反例。Visser 等人描述了用于 C++ 的谓词抽象技术及用于 Java 语言的实现[7]，抽象的结果是新的 Java 程序，其中去掉了一些变量，加入了一些表示谓词取值情况的布尔变量。这个新的 Java 程序可以用 Java PathFinder[27]进行验证。Bandera 是另一个 Java 程序抽象工具[28,29]，它根据用户的规格说明对程序进行切片，去除无关的部分，然后对程序进行数据抽象，把得到的结果翻译为某些现有模型检测工具的输入。

结论 在诸多验证方法中，模型检测因为高度自动化、对使用者要求不高、覆盖系统全部状态和可以生成反例等原因而受到越来越多的关注。本文以 SLAM 和 BLAST 为例介绍了基于抽象-验证-细化范例的软件模型检测。目前软件模型检测还处在起步阶段，距离真正成熟还有许多工作要做。

参考文献

1 Ball T, Rajamani S K. Automatically validating temporal safety properties of interfaces. The 8th International SPIN Workshop on Model Checking of Software, Toronto, Canada, 2001

2 Clarke E, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement. The 13th Conference on Computer Aided Verification, Chicago, USA, 2000

3 Saidi H. Model checking guided abstraction and analysis. The 7th International Static Analysis Symposium, Santa Barbara, USA, 2000

4 Henzinger T A, Jhala R, Majumdar R, et al. Lazy abstraction. The 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages, Portland, USA, 2002

5 Graf S, Saidi H. Construction of abstract state graphs with PVS. The 10th Conference on Computer Aided Verification, Haifa, Israel, 1997

6 Das S, Dill D L, Park S. Experience with predicate abstraction. The 12th Conference on Computer Aided Verification, Trento, Italy, 1999

7 Visser W, Park S, Penix J. Using predicate abstraction to reduce object-oriented programs for model checking. The 3rd Workshop on Formal Methods in Software Practice, Portland, USA, 2000

8 Ball T, Majumdar R, Millstein T, et al. Automatic predicate abstraction of C programs. The 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, USA, 2001

9 Flanagan C, Qadeer S. Predicate abstraction for software verification. The 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages, Portland, USA, 2002
10 Ball T, Millstein T, Rajamani S K. Polymorphic predicate abstraction. ACM Transactions on Programming Languages and Systems, 2005,27(2): 314～343
11 Ball T, Rajamani S K. Boolean programs: A model and process for software analysis:[Technical Report]. Redmond, USA: Microsoft Research, 2000
12 Das M. Unification-based pointer analysis with directional assignments. The 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, Canada, 2000
13 Ball T, Rajamani S K. Bebop: A symbolic model checker for Boolean programs. The 7th International SPIN Workshop on Model Checking of Software, Stanford, USA, 2000
14 Ball T, Rajamani S K. Bebop: A path-sensitive interprocedural dataflow engine. The 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, USA, 2001
15 Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, USA, 1995
16 Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science, 1996,167: 131～170
17 Kurshan R P. Computer-aided Verification of Coordinating Processes. Princeton, USA: Princeton University Press, 1994
18 Rusu V, Singerman E. On proving safety properties by integrating static analysis, theorem proving and abstraction. The 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Amsterdam, the Netherlands, 1999
19 Lakhnech Y, Bensalem S, Berezin S, et al. Incremental verification by abstraction. The 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems, Genova, Italy, 2001
20 Ball T, Rajamani S K. Generating abstract explanations of spurious counterexamples in C programs:[Technical Report]. Redmond, USA: Microsoft Research, 2002
21 Henzinger T A, Jhala R, Majumdar R, et al. Abstractions from proofs. The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy, 2004
22 Craig W. Linear reasoning: A new form of the Herbrand-Gentzen theorem. Journal of Symbolic Logic, 1957, 22(3):250～268
23 Pudlák P. Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic, 1997,62(2):981～998
24 Chaki S, Clarke E, Groce A, et al. Modular Verification of Software Components in C. ACM Transactions on Software Engineering,2004,30(6):388～402
25 Flanagan C, Leino K R M, Lillibridge M, et al. Extended static checking for java. The 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, 2002
26 Flanagan C, Qadeer S. Predicate abstraction for software verification. The 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages, Portland, USA, 2002
27 Havelund K, Pressburger T. Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer, 2000,2(4):366～381
28 Corbett J, Dwyer M, Hatcliff J, et al. Bandera: Extracting finite-state models from Java source code. The 22nd International Conference on Software Engineering, Limerick, Ireland, 2000
29 Dwyer M, Hatcliff J, Joehanes R, et al. Tool-supported program abstraction for finite-state verification. The 23nd International Conference on Software Engineering, Toronto, Canada, 2001

(上接第 188 页)

下,发现基于实例特征的位于不同 Ontology 分类树的两个概念对的相似性,同时判断一个概念在另一个 ontology 中的相对位置,获取基于实例特征的概念相似性矩阵及位置特征矩阵。

匹配标记器(如:Hidden Markov Model Labeler)在分析树结构、领域公理约束、共有知识的基础上,利用概念相似性矩阵及位置特征矩阵标记映射关系,获得参考匹配结果。

利用概念相容性检测、实例一致性检测推理机(如:Tableau Algorithms)检测参考匹配结果,最后获得可以信赖的结果。当然,半自动方式下进行的有条件映射标记,也是可以接受的(如:Protege PROMPT)。

总结 语义服务是下一代 WebWeb 服务必须解决的难点问题,语义网为实现广泛的语义服务提供了可能,Ontology 是语义网体系结构的核心。分布式协作系统的重要问题是本体的匹配和集成,针对协作的分布式系统需要语义互联的问题,本文分析了造成语义互联困难的主要因素是本体的匹配和集成,提出了一个基于机器学习的 Ontology 集成的框架模型。该框架的四个关键步骤是:学习分类算法及策略、相似性评估、标记算法、相容性一致性检测。

基于机器学习的本体概念匹配研究才刚刚起步,本文提出的这个框架,结构完整,每一步工作的目标明确,我们将在之后的研究论文中逐一讨论。

参 考 文 献

1 Berners-Lee T, Hendler J, Lassila O. The Semantic Web, Scientific American, 2001,284(5):34～43
2 Foster I, Kesselman C, Nick J M, et al. The Physiology of the Grid - An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/ogsa/, Feb. 2002
3 Noy F N. What Do We Need for Ontology Integration on the Semantic Web, Position Statement. In: Proc. of the Workshop on Semantic Integration, the 2nd International Semantic Web Conference, Sanibal Island, Florida, USA, Oct. 2003
4 Studer R, Benjamins V R, Fensel D. Knowledge Engineering, Principles and Methods. Data and Knowledge Engineering, 1998,25(122):161～197
5 Jurisica I, Mylopoulos J, Yu E. Ontologies for Knowledge Management: An Information Systems Perspective. Knowledge and Information Systems, 2004,6(4):380～401
6 Baader F, McGuinness D, Nardi D, Schneider P P. The description logic handbook: theory, implementation and applications [Z]. Cambridge: Cambridge University Press, 2002
7 Naing Myo-Myo, Limy Ee-Peng, Hoe-Lian Dion Goh. Ontology-based Web Annotation Framework for HyperLink Structures. In: Proceedings of the Third International Conference on Web Information Systems Engineering (WISE), Singapore, 2002
8 李善平,尹奇,胡玉杰,等. 本体论研究综述. 计算机研究与发展, 2004,41(7)