一种C程序断言的全自动静态验证方法*)

易晓东　杨学军

(国防科技大学计算机学院　长沙 410073)

摘　要　在软件程序中插入断言是保证软件质量的一个简单但有效的方法,人们常使用测试的方法检验程序中的断言是否满足,但测试很难保证验证的完备性。本文提出了一种可以保证完备性的全自动静态断言验证方法,其基本思想是基于程序切片符号执行程序的所有执行路径,并证明路径上的所有断言都满足。为了尽量减少符号执行的语句的数量,使用了基于反例的抽象精化方法,从一个粗略的切片标准开始迭代地符号执行一条路径,根据验证的反例自动生成下一次迭代过程中使用的精化的切片标准。包含循环的程序可能具有无穷多条程序执行路径,提出的基于符号执行上下文不变式证明的方法可以证明由于循环导致的无穷多条路径中断言都满足,从而使得验证过程可以终止。实验表明,提出的全自动静态断言验证方法不仅可行,而且验证代价较小,具有较强的实用性。

关键词　断言验证,基于程序切片的符号执行,基于反例的抽象精化,静态分析

Full-Automatic Static Assertion Verification for C Programs

YI Xiao-Dong　YANG Xue-Jun

(College of Computer, National Univ. of Defense Technology, Changsha 410073)

Abstract　Inserting assertions into program source codes is a simple but effective way to ensure software quality. One usually checks the assertions through testing, but the testing based verification can't guarantee completeness. We present in this paper a full-automatic static verification method for complete assertion verification. The basic ideal behind our method is symbolically executing every execution trace and proving that all the assertions of the trace hold. We only symbolically execute those statements which are relevant to the assertions and use a slicing criterion to judge which one is relevant. In order to lessen the statements to be executed when symbolically executing a trace, we use the ideal of Counter-Example Guided Abstraction Refinement(CEGAR) to generate a refined slicing criterion from counter-examples and use the new slicing criterion to start a new verification iteration for the same trace until a real counter-example trace found or assertions proved to be hold. There are infinite many traces for programs with loops. We also present a method based on symbolic execution context invariant proving to prove that all assertions of the infinite many traces leading by loops are satisfied and thus make our verification process terminate. It's shown by the experiments that the assertion verification method presented in this paper not only is feasible but also has little verification cost.

Keywords　Assertion verification, Symbolic execution based on program slicing, CEGAR, Static analysis

1　引言

程序员在设计软件程序时,往往在程序中插入断言(assert)语句,以描述程序执行到某点时必须满足的性质。这是保证软件质量的一个简单但有效的方法,已经被广泛地应用于几乎所有的软件开发中。

对于嵌入到软件程序中的断言,人们主要采用测试的方法对其进行验证,即基于各种输入数据运行软件,观察程序中的断言有没有出现不满足的情况。测试是一种动态的方法,它面临的最大问题是很难设计出完备的测试用例,从而很难保证验证的完备性。本文提出的针对C语言的静态验证方法并不运行程序,而是直接分析程序源代码并对其中包含的assert语句进行验证。验证过程中会考虑程序的所有执行路径,从而保证验证的完备性。另外,本文提出的方法无需人工干预,可以全自动地完成对C程序中所有断言的自动验证。

我们提出的静态验证方法的基本思想是考虑程序的所有执行路径,并使用符号执行的方法验证每条可能的程序执行路径上的断言都满足。符号执行(Symbolic execution)[1]使用符号常量作为输入数据并模拟程序中所有路径的执行。在符号执行程序中的每条执行路径时,符号执行每完一条语句,我们就计算出当前程序中的所有变量所满足的条件。如果当前语句的下一条语句是断言语句,我们就判断该条件是否能保证断言语句一定满足。基于符号执行的验证过程会遇到两个问题:一是实际程序中往往存在大量与断言验证无关的语句,符号执行这些语句将大大增加验证的工作量,同时对断言的验证毫无帮助;二是由于程序中循环的存在,导致程序中往往包含无穷多条路径,从而导致基于符号执行的验证过程不能终止。

针对上述第一个问题,我们引入程序切片(Program slicing)[2]的思想,根据一个给定的切片标准自动选择与断言验证相关的语句符号执行。为了生成尽可能宽松的切片标准以排除更多的无关语句,从而尽量减小符号执行的工作量,我们

*)Supported by National 863 Special Software Project unber Grant No. 2002AA1Z2101(863重大软件专项)。**易晓东**　在读博士生,主要研究领域为软件的静态分析与形式化验证;**杨学军**　博士生导师,教授,主要研究领域为并行与分布处理、信息安全、软件工程。

使用了基于反倒的抽象精化(Counter-Example Guided Abstract Refinement,CEGAR)[3]思想,从一个宽松的(甚至可以是空的)初始切片标准开始对程序进行切片,然后对切片后的程序基于符号执行的方法进行验证。如果所有 assert 语句在切片后的程序中都满足,则我们可以保证它们在原程序中也满足;如果某个 assert 语句在切片后的程序中不满足并给出了导致其不满足的一条反例路径,我们就在原程序中检验该反例路径是否可行(feasible),即检验反例路径在程序实际执行时是否存在。如果可行,就找到了一个真实的反例,否则反例路径为伪反例路径。我们通过它自动推断出一个精化的切片标准并基于新的切片标准,重新符号执行原程序。

针对上述第二个问题,我们引入了符号执行上下文不变式(Context invariant)的概念,符号执行的上下文描述了基于某切片标准符号执行到路径中的某点时程序中的所有变量所满足的条件。如果不论一条路径中任意循环语句的循环体被执行多少次,路径中某点的上下文都相同,我们称该点的上下文为上下文不变式。对于一条路径 p,如果我们能够证明:(1)p 中所有断言语句对应的上下文都能保证断言一定满足;(2)p 中每点的上下文都是上下文不变式,那么我们就可以保证不论 p 中循环语句的循环体被执行多少次,所得到的路径中的断言都一定满足。如果不能证明上述两点,我们就推断出新的切片标准并重新符号执行路径 p。实际证明时,我们先找到程序中所有不包含循环的执行路径,它们的数量一定是有限的,再基于切片符号执行每条不包含循环的执行路径,并证明这些路径中的每一点的上下文都是不变式。

我们提出的全自动静态断言验证方法综合使用了多种思想和技术,因此取得了比较好的效果。实验证明,我们的方法不仅可行,而且验证的代价比同类方法要小很多,因而具有良好的实用性。

2 控制流图与程序路径

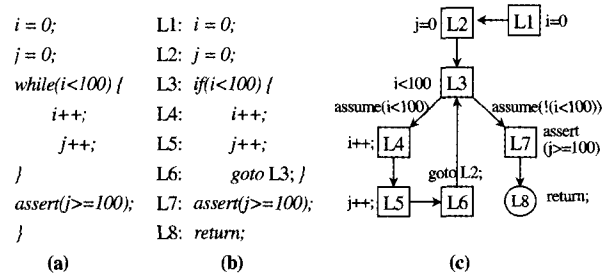

图 1 示例程序及其控制流图

与 BLAST[4]和 MAGIC[5]等工具一样,我们通过将函数体的代码内嵌到函数的调用点而得到一个具有单一函数的 C 程序。接下来,我们对程序中的循环进行变换,将之改写为使用 if 和 goto 语句的等价形式。经过上述变换,我们可以假设程序中只包括单个函数,函数中只包括赋值语句(Assignment)、if-then-else 分支语句、goto 语句、return 语句和 assert 语句(文[5]有类似的假设),其中 return 语句表示程序控制流的结束。如果需要,我们在程序中适当的地方插入 return 语句,以保证所有情况下函数返回前执行的最后一条语句是 return 语句。图 1(a)是一段简单的示例程序,图 1(b)是变换后的程序,图 1(c)则是该程序对应的控制流图。

定义 2.1(执行路径) 程序的一条执行路径定义为程序的控制流图中的一条路径,即,$p=s_1,s_2,\cdots,s_n$,其中 s_1 语句为程序的入口点,s_n 语句为 return 语句,这些语句在一定的条件下将按顺序被执行。程序的执行路径中不含分支语句,任何"if(c)then A else B;"分支语句根据程序执行路径所采用的分支被替换为"assume(c); A;"或"assume(! c); B;",其中"assume(c)"或"assume(! c)"语句使用"if(c)"语句的标号。

定义 2.2(无循环执行路径) 程序的无循环执行路径是指程序中所有同一标号的语句最多只出现一次的执行路径。

定义 2.3(循环路径) 程序的循环路径是指程序控制流图中的圈,对循环路径 $p=s_1,s_2,\cdots,s_n$ 而言,要求 s_1 为 assume 语句,且 $\forall I\leqslant i<j\leqslant n$。$L(s_i)\neq L(s_j)$、语句 s_1 为 s_n 的后续语句之一,其中 $L(s)$为语句 s 的标号。

定义 2.4(带循环的分支语句与带循环的 assume 语句) 设路径中的某 assume 语句 s 对应的 C 程序中的分支语句为 $\tilde{s}$,则显然 $L(s)=L(\tilde{s})$。我们考查是否存在循环路径 $p=s_1,s_2,\cdots,s_m$,使得 $L(s_1)=L(\tilde{s})$。如果存在一条或多条这样的循环路径,说明 $\tilde{s}$ 是 while、for 等语句或与 goto 形成的循环的 if-then-else 语句,我们称 $\tilde{s}$ 是带循环的分支语句,称 s 为带循环的 assume 语句。

定义 2.5(循环扩展路径) 无循环执行路径 p 中的带循环的 assume 语句对应于程序中的循环。实际执行过程中,当程序执行到这些循环时,循环的循环体常会被执行多次,相当于在 p 中的这些 assume 语句之前插入了多条循环路径,我们称这样得到的路径为无循环执行路径 p 的循环扩展路径。

例如对于图 1 所示的程序,无循环执行路径只有一条,为 L1→L2→L3→L7→L8,对应的语句序列为"$i=0$; $j=0$; assume(! ($i<100$)); assert($j>=100$); return;",其中"assume(! ($i<100$));"语句为带循环的 *assume* 语句。循环路径也只有一条,为 L3*→L4→L5→L6,对应的语句序列为"assume($i<100$); $i++$; $j++$; goto L2;"(我们使用 L3* 表示循环路径中的语句,而用 L3 表示无循环执行路径中的语句,它们的标号相同,但对应的 assume 语句的条件不同)。路径 $L1\rightarrow L2\rightarrow\{L3^*\rightarrow L4\rightarrow L5\rightarrow L6\}^n\rightarrow L3\rightarrow L7\rightarrow L8$ 都是上述无循环路径的循环扩展路径,其中 $\{p\}^n$ 表示将路径 p 重复 n 次,重复不同的次数将得到不同的循环扩展路径。

3 基于切片的最强后置条件

3.1 最强后置条件

程序利用一系列的语句对输入数据集进行操作,最终得到输出数据集。设在执行某语句 s 前的数据集为 D,执行完语句 s 后的数据集为 D'。数据集的约束条件 φ 相对于程序语句 s 的最强后置条件(Strongest post-condition)[6],记为 $SP(s)(\varphi)$,描述了 φ 在执行 s 后对应的最强条件,它满足如下两个性质:

1)如果数据集 D 满足 φ,则执行语句 s 后的数据集 D' 必然满足 $SP(s)(\varphi)$;

2)如果数据集 D' 不满足 $SP(s)(\varphi)$,则可以断定在执行 s 语句前的数据集 D 不满足 φ。

赋值语句和 *assume* 语句的最强后置条件定义如下[6,7]:

$$SP(x=e)=\lambda f.\ \exists x'.\ f[x'/x]\wedge(x=e[x'/x]) \quad (3.1)$$

$$SP(assurm(e))=\lambda f.\ f\wedge e \quad (3.2)$$

其中 $f[x'/x]$与 $e[x'/x]$是指将公式 f 与 e 中的所有自由变量 x 替换为 x'后得到的公式。为了去掉计算过程中赋值语句的最强后置条件中的存在量词(∃),我们使用文[7]中

的方法，首先对路径 p 进行标注，对于路径 p 的每个赋值语句 $x=e$ 和 assume 语句 assume(e)，如果 e 中引用了某个变量 x，但 x 在之前没有定义，则我们在该赋值语句或 assume 语句之前插入一条赋值语句 $x=\theta_x$，其中 θ_x 为一个 Skolem 常量。对路径 p 标注后，设路径 p 的所有语句中包含的所有变量的集合为 $Vars(p)$，则 $Vars(p)$ 中的所有变量都有确定的初值(为数学常量与符号常量组成的表达式)，我们再使用上下文$\langle\Omega,\Phi\rangle$来表示和计算最强后置条件，$\Omega$ 是一个部分函数 $Vars(p)\rightarrow Exp$，用于存放赋值语句对变量的赋值，其中 Exp 为所有 C 语言表达式的集合；Φ 为 Exp 中的布尔表达式集合，用于存放 assume 语句中的约束。基于$\langle\Omega,\Phi\rangle$的赋值语句 $x=e$ 和 assume 语句 assume(e)的最强后置条件计算方法如下[7]：

$$SP(x=e)=\lambda\langle\Omega,\Phi\rangle\cdot\langle\Omega[x\rightarrow\Omega(e)],\Phi\rangle \tag{3.3}$$

$$SP(\text{assumse}(e))=\lambda\langle\Omega,\Phi\rangle\cdot\langle\Omega,\Phi\cup\Omega(e)\rangle \tag{3.4}$$

其中，函数 $\Omega[x\rightarrow e]$定义为：

$$\Omega[x\rightarrow e](y)=\begin{cases}\Omega(y) & \text{if } y\neq x\\ e & \text{if } y=x\end{cases} \tag{3.5}$$

对于一条路径 $p=s_1,s_2,\cdots,s_n$，定义 $SP(p)=SP(s_1)\circ SP(s_2)\circ\cdots\circ SP(s_n)$，函数组合符号“$\circ$”将函数按右结合的方式组合，定义为 $g\circ h=\lambda x.\ g(h(x))$。如果 $SP(p)(True)=$ False，那么对执行路径 p 前的任意数据集 D(都满足条件 True)，由于执行完路径 p 后的最强后置条件都不满足，说明 p 是一条不可行路径。设使用上述最强后置条件计算方法计算完路径 p 的所有语句后得到的上下文为$\langle\Omega,\Phi\rangle$，根据最强后置条件的定义及式(3.1)和式(3.2)易知：

$$SP(p)(\text{True})=\exists\theta_1,\cdots,\theta_m.\ \left(\left(\bigwedge_{\omega\in\Omega}e2b(\omega)\right)\wedge\left(\bigwedge_{\varphi\in\Phi}\varphi\right)\right) \tag{3.6}$$

其中，$\theta_1,\cdots,\theta_m$ 为计算过程引入的所有 Skolem 常量，函数 $e2b(\omega)$将 Ω 中的赋值等式 ω 转化为布尔公式，如 $e2b(x:=1)=(x==1)$。实际应用中，我们一般使用与 $SP(p)(\text{True})=\text{False}$ 等价的式(3.7)判断路径的可行性[7]。如果基于约束条件 True 计算出的路径 p 的最强后置条件的上下文$\langle\Omega,\Phi\rangle$满足式(3.7)，则路径 p 不可行，否则路径 p 是可行路径。

$$\left(\bigwedge_{\varphi\in\Phi}\varphi\right)=\text{False} \tag{3.7}$$

3.2 程序切片

我们并不符号执行所有的语句，而是利用程序切片去除程序中的无关语句，以大量减小符号执行的复杂度。切片的任务是根据给定的切片标准，即一个变量的集合 V，判定一个赋值语句或一个 assume 语句是否是需要符号执行的相关语句。

定义 3.1(赋值语句的相关性) 对于一个作为切片标准的变量集合 V 和一个赋值语句 $x=e$，其中 x 与 e 都是 C 语言的表达式，我们定义该赋值语句的相关性如下：

- 如果 $vars(x)\subseteq V$ 且 $vars(e)\subseteq V$，则我们称赋值语句 $x=e$ 是切片标准 V 下的完全相关语句；
- 如果 $vars(x)\subseteq V$ 但 $vars(e)\not\subseteq V$，则我们称赋值语句 $x=e$ 是切片标准 V 下的部分相关语句；
- 如果 $vars(x)\not\subseteq V$，则我们称赋值语句 $x=e$ 是切片标准 V 下的无关语句。

定义 3.2(assume 语句的相关性) 对于一个作为切片标准的变量集合 V 和一个 assume 语句 assume(e)，其中 e 为 C 语言的布尔表达式，我们定义该 assume 语句的相关性如下：

- 如果 $vars(e)\subseteq V$，则我们称 assume 语句 assume(e)是切片标准 V 下的相关语句；
- 如果 $vars(e)\not\subseteq V$，则我们称 assume 语句 assume(e)是切片标准 V 下的无关语句。

要注意的是，本文提出的程序切片与传统的程序切片并非同一个概念，尽管它们的思想相似。由于不需要计算程序语句的相关性，本文提出的程序切片的计算开销要小得多。

定义 3.3(程序切片下的最强后置条件计算) 设在符号执行某语句 s 前的最强后置条件为$\langle\Omega,\Phi\rangle$，那么执行 s 后的最强后置条件 $SP(s)(\langle\Omega,\Phi\rangle)$定义如下：

- 如果 s 是无关语句，则 $SP(s)(\langle\Omega,\Phi\rangle)=\langle\Omega,\Phi\rangle$；
- 如果 s 是完全相关的赋值语句或相关的 assume 语句，则按式(3.3)和式(3.4)计算 $SP(s)(\langle\Omega,\Phi\rangle)$；
- 如果 s 是部分相关的赋值语句 $x=e$，由于 $vars(e)\not\subseteq V$，因此我们不能确知 e 的值，从而我们引入一个 Skolem 变量 θ_e 来表示 e 的符号值，并定义 $SP(s)(\langle\Omega,\Phi\rangle)=SP(x=\theta_e)(\langle\Omega,\Phi\rangle)$。

4 断言证明

4.1 基于切片的符号执行

我们使用一个集合 SPC 来描述基于切片符号执行时的上下文(Context)，SPC 是一系列最强后置条件(Strongest Post-Condition)的集合，$SPC=\{spc_1,spc_2,\cdots,spc_n\}$，$SPC$ 的每个元素定义为 $spc_i=\langle\Omega_i,\Phi_i\rangle$，$\Omega_i$ 与 Φ_i 为 3.1 节定义的最强后置条件的上下文。程序的入口语句对应的上下文 SPC 包含一个元素$\langle\Omega,\Phi\rangle$，其中 Ω 与 Φ 都为空。

基于切片的符号执行考虑程序中的每条无循环执行路径 $p=s_1,s_2,\cdots,s_n$，计算路径的每个语句对应的上下文。设语句 s_i 的上下文为 SPC_i，则符号执行完 s_i 后的上下文 SPC_{i+1}按如下方式计算：

如果 s_i 为赋值语句，则对每个$\langle\Omega,\Phi\rangle\in SPC_i$ 按定义 3.3 计算其在 SPC_{i+1}中的对应元素；

如果 s_i 为不带循环的 assume 语句，则对每个$\langle\Omega,\Phi\rangle\in SPC_i$ 按定义 3.3 计算其在 SPC_{i+1}中的对应元素；

如果 s_i 是带循环的 assume 语句 assume(e)，设包含语句 s_i 的所有循环路径为 $p^1,p^2,\cdots,p^c$(其中 $p^k=s_1^k,s_2^k,\cdots,s_{tk}^k$)，它们满足 $\forall(1\leqslant k\leqslant c).\ L(s_1^k)=L(s_i)$。为了保证验证的完备性，我们必须考查 p 的所有的循环扩展路径。为此，我们以迭代的方式符号执行这些循环路径，每次迭代过程中将所有的循环路径都符号执行一遍。设上一次迭代得到的符号执行上下文为 SPC^t，设本次迭代过程中对循环路径 p^k 基于上下文 SPC^t 符号执行一次得到的上下文为 SPC_k^{t+1}，则本次迭代得到的新上下文 SPC^{t+1}定义为：

$$SPC^{t+1}=\bigcup_{1\leqslant k\leqslant c}SPC_k^{t+1} \tag{4.1}$$

为了判断何时能够终止迭代，我们先定义一个将上下文 SPC 映射到一个一阶逻辑公式的函数，如式(4.2)所示。$H(SPC)$实际上是上下文 SPC 记录的所有循环扩展路径的最强后置条件的并($\vee$)，式(4.2)可以由式(3.6)直接推导得出。

$$H(SPC)=\bigvee_{\langle\Omega_j,\Phi_j\rangle\in SPC}\left(\exists\theta_1,\cdots,\theta_n\left(\left(\bigwedge_{\omega\in\Omega_j}e2b(\omega)\right)\wedge\left(\bigwedge_{\varphi\in\Phi_j}\varphi\right)\right)\right) \tag{4.2}$$

如果公式 $H(SPC^{t+1})\Rightarrow H(SPC^t)$满足，那么无论再执行循环路径多少次，都不会产生新的上下文，因此我们可以终止对循环路径的迭代执行。上下文 SPC^t 可以表示所有循环扩

展路径的最强后置条件，我们称上下文 SPC' 是一个上下文不变式(Context invariant)。

每当符号执行完一条语句 s_i 后，我们都要判断上下文 SPC_i 对应的所有循环扩展路径是否可行(feasible)。如果所有的循环扩展路径都不可行(infeasible)，我们就终止对路径 p 的执行。如果 SPC_i 使得式(4.3)成立，则基于当前无循环路径 $P_t=S_1,S_2,\cdots,S_i$ 的所有循环扩展路径都不可行。

$$\bigvee_{\langle\Omega_j,\Phi_j\rangle\in SPC_i}\left(\bigwedge_{\varphi\in\Phi_j}\varphi\right)=\text{False} \tag{4.3}$$

4.2 断言验证与切片标准精化

对程序中某条包含 assert 语句的无循环执行路径 $p=s_1,s_2,\cdots,s_n$，设其包含 m 条 assert 语句，分别为 assert(e_1)，assert(e_2)，…，assert(e_m)，则作为初始切片标准的变量集合 V_0 定义如下：

$$V_0=\bigcup_{1\leqslant k\leqslant m}vars(e_k) \tag{4.4}$$

如果符号执行到某语句 s_i，且 s_i 为 assert 语句 assert(e)，那么我们必须判断当前的最强后置 SPC_i 条件是否使表达式 e 满足，式(4.5)给出了判别条件。

$$\left(\bigvee_{\langle\Omega_j,\Phi_j\rangle\in SPC_i}\left(\left(\bigwedge_{\varphi\in\Phi_j}\varphi\right)\wedge\Omega_j(\neg e)\right)\right)=\text{False} \tag{4.5}$$

如果式(4.5)满足，说明 SPC_i 中的所有元素都使得表达式 e 满足，从而 SPC_i 对应的所有循环扩展路径中 assert 语句都满足。但是另一方面，如果式(4.5)不满足，并不意味着该 assert 语句在原程序中不满足，这是因为基于切片的符号执行是一个保守的过程，即它所遍历的执行路径有可能在原程序中并不存在(但它并不会漏掉任何一条程序中存在的执行路径)。

我们将检验 assert 语句在实际程序中是否满足的问题转化为路径的可行性验证问题，并基于式(3.7)对路径的可行性进行验证。对于路径 $p=s_1,s_2,\cdots,s_n$，设语句 s_{i+1} 是 assert(e) 语句，则语句 s_{i+1} 在路径 p 中是一定满足的等价于路径 $\bar{p}_{i+1}=s_1,s_2,\cdots,s_i,\bar{s}_{i+1}$ 是不可行路径，其中 $\bar{s}_{i+1}$ 为语句 assume($\neg e$)。这是因为，根据符号执行的结果，我们知道路径 $p_i=s_1,s_2,\cdots,s_i$ 是可行的，同时路径 $\bar{p}_{i+1}$ 又不可行，显然这是因为 $\bar{s}_{i+1}$ 语句中的布尔表达式($\neg e$)为假，从而 $assert(e)$ 是一定满足的。如果路径 $\bar{p}_{i+1}$ 是不可行路径，我们根据它生成一个切片标准 V'，并使用新的切片标准 $V_{i+1}=V_i\cup V'$ 重新符号执行路径 p。

在计算路径 p 的最强后置条件时，集合 Φ 实际上记录了 p 中所有 assume 语句所描述的约束。一条路径是否可行，取决于该路径上的所有 assume 语句是否能够同时满足。如果式(3.7)为真，说明路径 p 上的 assume 语句存在冲突，因此 p 不可行。实际中的不可行路径往往只有少数 assume 语句存在冲突，因此我们可以进一步只选择那些确实存在冲突的 assume 语句中变量的集合作为切片标准进行程序的动态切片[8]，找出与该变量集合相关的所有变量和语句，将这些相关变量加入到原来的变量集合中，就得到了精化的切片标准。

例如由四条语句组成的不可行路径“assume($x>0$); $y=x$; assume($z>0$); assume($y<0$);”中，计算完该路径的最强后置条件后的上下文$\langle\Omega,\Phi\rangle$中，$\Omega=\{x=\theta_x,y=\theta_x,z=\theta_z\}$、$\Phi=\{\theta_x>0,\theta_z>0,\theta_x<0\}$，显然 Φ 的子集$\{\theta_x>0,\theta_x<0\}$满足式(3.7)，因此我们将两个表达式 $x>0$ 和 $y<0$ 中的变量，即 $\{x,y\}$作为精化的切片标准，就可以判断出上述路径为不可行路径。

5 验证工具原型与实验结果

我们基于 MAGIC[5]实现了一个 C 语言断言验证工具原型，该验证工具原型使用 Simplify[9]作为定理证明工具，能够全自动地对输入的标准 C 程序中的断言进行验证，工具原型当前还暂时不支持变量别名。MAGIC 已经实现了大部分我们需要的服务，包括对标准 C 程序的解析、最弱前置条件(Weakest precondition)[6]计算(我们基于其实现最强后置条件及符号执行上下文的计算)、定理证明工具的调用封装等。

我们使用断言验证工具原型验证了 3 个 C 程序，其中后两个程序来自于相关文献。我们对验证结果进行了比较，如表 1 所示。我们主要与基于谓词抽象[10]的验证方法进行了比较，这是因为后者的研究比较活跃，也比较成熟和实用。表 1 中 Vars、Thm Calls 和 Time3 列中包含了两组数据，上面一组数据是使用我们的验证工具的验证结果，下面一组是从相关文献中引用过来的，我们将在讨论每个 C 程序时详细讨论。表中的 Name 是指 C 程序的名字，LOC 是指 C 程序的行数(未预处理前)，Iters 是指验证过程中符号执行所有无循环执行路径的总迭代次数。Vars/Preds 的上面一组数据是指使用我们的方法生成的切片标准变量集合中的变量数量，下面的一组数据是指基于谓词抽象生成的谓词集合中谓词的数量。Thm Calls 是指调用定理证明工具的次数。由于定理证明工具的调用往往是验证过程中的瓶颈，因此减少调用次数将大大缩短验证时间，正如 Time 一列中的数据所示。我们所有的实验数据均在 AMD Athlon 1.12G 的 CPU 和 256M 内存的机器上得到，软件运行环境是 Windows 2000 和 CygWin 2.427。

表 1 实验结果与比较

Name	LOC	Iters	Vars/Preds	Thm Calls	Time
Example	18	2	2	603	9s
			n/a	n/a	n/a
Driver	95	9	2	46	<10ms
			3	260	60ms
Partition	55	4	2	68	<10ms
			4	263	9s

第一个程序 Example 是图 1 所示的示例程序。由于我们需要对其基于两种切片标准各符号执行 100 次，因此调用了定理证明工具 600 多次，并花费了 9s 的时间。我们使用了 BLAST 和 MAGIC 两个基于谓词抽象的验证工具对 Example 程序进行验证，但二者均报告无法验证。但我们可以对基于谓词抽象的验证过程中需要调用定理证明工具的次数做一个保守的估计。为了排除基于图 1 中的无循环扩展路径 L1→L2→L3→L7→L8 的所有循环扩展路径，基于谓词抽象的验证方法必须进行 100 次迭代(我们的验证工具是 2 次)，并推导出 100 个谓词(分别为 $i=0,i=1,\cdots,i=100$)，每次迭代过程由抽象模型生成、模型检验和模型精化 3 个步骤组成，我们只考虑模型精化这一步中的反例路径的可行性检验过程，详细的讨论请参阅文[11]的 7.4 节“Problematic C Programs”。基于谓词抽象的验证方法的整个迭代过程中仅反例分析这个子过程，就要调用至少 5000 次定理证明工具，所以我们相信验证时间将大大超过 9s。

(下转第 273 页)

重于伺服对象的服务器端控制。连接管理则解决构件间网络连接的使用问题以及相关的可伸缩性问题。线程管理讨论在CORBA应用程序中使用多个线程，侧重于在CORBA服务器中使用多线程的方法。

(6)系统集成

与EJB集成：CCM规范几乎是在EJB的基础上建模的。与EJB不同，CCM使用CORBA对象模型作为底层的对象互操作框架，从而不局限于特定的编程语言。鉴于这两种技术非常相似，CCM专门定义了这两个标准之间的标准映射。因此，通过使用适当的桥接技术，CCM构件可以表现为对于EJB来说的EJB Beans，EJB Beans也表现为CCM构件。EJB也支持CORBA IIOP作为其通讯框架。CCM和EJB是互补的。有关CCM和EJB互操作的详细信息见文[14]。

与遗产系统集成：应用服务器使用包装器来完成企业遗留系统的集成。我们采用与构件相同的规则来描述包装器，通过在软件包描述器定义一个特殊的实现标记，提供一个链结到可执行应用。这种机制使得遗产系统能够被描述使用，包装器确保了应用能够通过IIOP被访问。包装器可以是一个分布式的管理体系结构的管理对象，可以灵活使用管理工具来完成遗产系统的集成。

3.3 应用服务器的特点

应用服务器框架集成了CORBA构件模型诸多优点，具有以下主要特点：(1)位置透明。构件对象可以相互作用，无论宿主位置如何；(2)多语言支持。构件对象可以使用任何语言实现，映射规则由CORBA规范定义；(3)实现独立。构件对象的通信由标准的接口实现，这样一个对象实现的改变对其它部分不会产生影响；(4)结构和操作系统独立；(5)完整的构件开发、部署机制；(6)集成了CORBA的基本服务，支持持久服务、事务服务、通知服务、安全服务等；(7)支持容错和负载平衡；(8)支持企业数据集成。

结束语 本文在深入研究CCM后提出了一个基于CORBA构件模型的应用服务器框架，并就应用服务器构件的定义与类型、构件的实现框架、构件容器、应用服务器安全模型、资源管理及系统集成等方面的内容进行了详细讨论。由于CORBA构件模型是一个开放的标准，本身仍在不断地完善和发展，而且至今还没有一个完整的商业实现，所以需要进一步研究的工作还有许多。以后的工作主要集中在应用服务器的安全、构件模型的优化、与WebServices集成等方面，并需要进一步实现和完善应用服务器的核心技术和功能。

参 考 文 献

1 Enterprise Application Server. http://dev2dev.bea.com
2 OMG CORBA Specification. http://www.omg.org/corba/whatiscorba.html
3 Ben-Natan R. CORBA--A Guide to the Common Object Request Bröker Architecture. New York:McGraw-Hill,1995
4 OMG CORBA Components. http://cgi.omg.org/cgi-bin/doc?orbos/02-06-33
5 OMG CORBA Component Scripting. http://cgi.omg.org/cgi-bin/doc? orbos/01-12-05
6 Overview of the CORBA Component Model. http://www.ditec.um.es/~dsevilla/ccm
7 SUN. http://develoner.iava.sun.com/developer
8 Sun Microsystem, JavaBeans API Specification Vision 1.01, July 24,1997
9 EJB specification. http://WWW.sun.com
10 Microsoft. http://www.microsoft.com/isapi
11 Microsoft. Distributed Component Object Model Protocol DCOM/1.0 Specification. May,1996
12 Brocjschmidt K. Inside OLE2. Microsoft Press,1994
13 CORBA Secutity Specification. http://www.omg.org/corba/
14 OMG CORBA Components, Integrating with Enterprise JavaBeans. http://www.omg.org/corba

(上接第256页)

第二个验证程序"Driver.c"来自文[12]，BLAST在文[13]中也使用其作为实验对象，它是从Windows的PCI设备驱动程序中摘出来的一段C语言代码，用于在锁的保护下处理中断请求包(IRP)，我们要验证的性质是该程序中锁的使用符合规范(即加、解锁要交替进行)。我们验证工具仅调用了Simplify定理证明工具46次。与我们相比，BLAST调用了定理证明工具260次，并花费了0.06s的时间[13]。第3个C程序"Partition.c"来自于文[14]，它的功能是遍历一个链表，根据链表中每个节点的value域的值是否大于一个给定的值，将节点链到两个不同的链表中，待验证的性质是在程序的循环中的某处指向前趋节点的指针和指向当前节点的指针不相等。我们的方法调用了Simplify定理证明工具68次，而SLAM则调用了Simplify 263次，并花费了9s的时间[14]。

结束语 我们提出了一种C程序断言的全自动静态验证方法，该方法综合使用了程序切片、符号执行、谓词抽象、基于反例的抽象精化等多种思想和技术，从而取得了较好的验证效果。

下一阶段我们的主要任务是提高切片的精度并支持基于变量别名的切片，然后通过对实际程序的断言验证评估我们的方法的效果。

参 考 文 献

1 King J C. Symbolic execution and program testing. Communications of the ACM,1976,19(7):385～394
2 Weiser M. Program slicing. IEEE Transactions on Software Engineering(TSE)1982,SE-10(4):352～357
3 Clarke E M, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement. In:Proceedings of CAV 1855,2000. 154～169
4 Henzinger T A, Jhala R, Majumdar R, et al. Software Verification with BLAST. 10th Int SPIN Workshop (SPIN'2003)2648 of Lecture Notes in Computer Science,2003. 235～239
5 Chaki S, Clarke E, Groce A. Modular Verification of Software Components in C. In: ACM-SIGSOFT Distinguished Paper in the 25th International Conference on Software Engineering (ICSE), 2003. 385～395
6 Gries D. The Science of Programming. Springer-Verlag,1981
7 Ball T, Rajamani S K. Generating abstract explanations of spurious counterexamples in C programs:[Technical Report]. MSR-TR-2002-09, Microsoft Research, Microsoft Corporation, 2002
8 Zhang Xiangyu, Gupta R, Zhang Youtao. Precise Dynamic Slicing Algorithms. IEEE/ACM International Conference on Software Engineering (ICSE), 2003. 319～329
9 Detlefs D, Nelson G, Saxe J. Simplify: A theorem prover for program checking. http://research compaq com/src/esc/simplify html. 2003
10 Graf S, Saidi H. Construction of abstract state graphs with PVS. In:CAV 97:Computer-aided Verification, LNCS 1254, Springer-Verlag,1997. 72～83
11 Weibenbacher G. An abstraction/refinement scheme for model checking C programs:[PhD Dissertation]. Technischen University of Graz (TUG),2003
12 Ball T, Rajarnani S K. Automatically Validating Temporal Safety Properties of Interfaces. PLDI2001,2001
13 Henzinger T A, Jhala R, Majumdar R, et al. Lazy Abstraction. In:Proceedings of the 29th Annual Symposium on Principles of Programming Languages (POPL),2002. 58～70 ACM
14 Ball T, Majumdar R, Millstein T, et al. Automatic predicate abstraction of C programs. PLDI2001: Programming Language Design and Implementation, 2001