软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(3):633−644 [doi: 10.13328/j.cnki.jos.004985]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 面向方面设计中干涉问题的分析工具*

陈　鑫[1,2], 黄　超[1,2], 张一帆[1,2], 梅一鸣[1,2]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](南京大学 计算机科学与技术系,江苏 南京 210023)

通讯作者: 陈鑫, E-mail: chenxin@nju.edu.cn

**摘　要**: 干涉问题是指基础程序和方面之间或者方面之间发生不需要的相互作用,导致最终程序中产生不想要的功能,危害程序的正确性.很难检测和修正在面向方面设计中存在的干涉,已经成为推广面向方面技术的阻碍.受到技术自身可扩展能力的局限,现有的基于模型验证技术的工作不能有效地处理功能干涉问题.设计开发了基于推理验证技术直接检查和去除面向方面设计中功能干涉的工具,它可以根据类和方面的功能规约自动产生确保不发生干涉的条件,并引入交互式证明工具 PVS 来提高证明过程的自动化程度.证明可以确认设计中无干涉存在或者为修正干涉问题提供线索.

**关键词**: 面向方面的设计;功能干涉;推理验证;设计演算

**中图法分类号**: TP311

中文引用格式: 陈鑫,黄超,张一帆,梅一鸣.面向方面设计中干涉问题的分析工具.软件学报,2016,27(3):633−644. http://www.jos.org.cn/1000-9825/4985.htm

英文引用格式: Chen X, Huang C, Zhang YF, Mei YM. Tool for analyzing interference problems in aspect-oriented designs. Ruan Jian Xue Bao/Journal of Software, 2016,27(3):633−644 (in Chinese). http://www.jos.org.cn/1000-9825/4985.htm

## Tool for Analyzing Interference Problems in Aspect-Oriented Designs

CHEN Xin[1,2], HUANG Chao[1,2], ZHANG Yi-Fan[1,2], Mei Yi-Ming[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of computer science and technology, Nanjing University, Nanjing 210023, China)

**Abstract**: Interference problems refer to the undesired interaction between aspects and base programs or interaction between aspects that results in unexpected functions and is harmful to the correctness of the entire program. The difficulty in detecting and fixing interferences in aspect-oriented designs impedes the widespread application of aspect-oriented programming paradigm. Suffered from the scalability problem, existing researches that use model checking techniques cannot effectively handle functional interferences. The paper designs and implements a tool that employs deduction-based technologies to support direct checking and remove functional interferences in aspect-oriented designs. This tool can automatically generate proof obligations excluding the existence of interference. In addition, the paper introduces the tool PVS to raise the automation level of verification. The proof can either ascertain no interference exists or give clues on how to rectify the design.

**Key words**: aspect-oriented designs; functional interference; deductive verification; design calculus

* 基金项目: 国家自然科学基金(91318301, 61561146394); 国家重点基础研究发展计划(973)(2014CB340703); 教育部高等学校博士学科点专项科研基金(20110091120058); 江苏省产学研项目(BY2014126-03)

Foundation item: National Natural Science Foundation of China (91318301, 61561146394); the National Key Basic Research Program of China (973) (2014CB340703); Specialized Research Fund for the Doctoral Program of Higher Education (20110091120058); Project on the Integration of Industry, Education and Research of Jiangsu Province (BY2014126-03)

收稿时间: 2015-07-15; 修改时间: 2015-10-20; 采用时间: 2015-11-27; jos 在线出版时间: 2016-01-05

CNKI 网络优先出版: 2016-01-05 16:39:59, http://www.cnki.net/kcms/detail/11.2560.TP.20160105.1639.011.html

© 中国科学院软件研究所 http://www.jos.org.cn

近年来,面向方面技术[1]在软件设计与开发中得到了广泛的应用.面向方面技术支持在软件系统的设计和实现过程中运用关注点分离策略,它将软件系统的基本功能和多个横切关注点进行分离,在基础程序中实现系统的基本功能,引入方面将横切关注点进行封装,并依靠编织机制在编译或运行时刻将基础程序与方面进行集成,构造出最终的软件.方面将原先分散在系统各处的同一横切关注点的代码封装在一起,提供了面向横切关注点的模块化封装机制,是对经典的从大功能向多个小功能的纵向模块化分解的有益补充.面向方面技术已经在应用服务器、操作系统内核、分布式中间件、分布式质量服务、对象式数据库、领域相关的可视化建模工具等多个领域中得到了应用,它在降低软件系统的复杂性、提高软件系统的复用性和可维护性方面的效用已经在实践中得到了证明.

面向方面技术的一个固有缺陷是,它易引发干涉问题:方面与基础系统可能相互干扰对方的功能,从而使得编织后的系统中出现不符合规约的功能和行为,破坏系统的正确性.干涉问题产生的根本原因在于:面向方面技术允许方面中的通知代码绕过基础系统模块的接口,直接对模块内部的变量进行操作,改变模块的状态,基础系统模块的封装性被彻底破坏了.干涉问题的存在,使得设计面向方面系统时必须仔细分析和验证系统的功能和性质在编织后是否仍然正确.很多研究者认为:在当前干涉问题没能得到有效处理的时候,若强行向工业界推广面向方面的开发方法与技术,其研发的软件系统中会出现大量无法预期和难以控制的功能和行为,最终将导致软件工业的一场灾难[2].干涉问题已经成为阻碍面向方面的技术进一步发展和应用最大的障碍.

当前,基于形式化技术的干涉问题分析方法和工具存在着很多局限:基于形式推理技术的方法多直接在语句层面构造软件模型和展开分析,模型抽象层次低,手工推理过程非常繁杂,缺乏工具支持,难以在实际系统中得以应用;基于自动模型检验技术的工具,受到其扩展能力的限制,不能直接对系统的功能进行处理,要求使用者手工构造更加抽象的模型,且只能对系统的部分功能和交互行为进行处理.

针对上述问题,本文研究了一个基于形式推理技术对面向方面设计模型中的干涉问题进行分析的工具.该工具支持用户基于 UML 类图构造软件的设计模型,允许用户使用前后置条件对基础系统中类方法和方面中通知代码的规约进行形式化定义.同时,允许通过类不变式对基础系统的重要性质进行形式化描述.基于统一程序设计理论中的设计演算方法,给出了方面中的通知代码不破坏基础系统不变式以及保证系统执行不发散的条件.基于这一条件,分析工具可以从输入的规约和不变式自动生成待证明的条件.分析工具后端集成了交互式证明工具 PVS,通过在 PVS 中嵌入设计演算理论,PVS 可以辅助用户对这些条件的证明.

本文第 1 节介绍设计演算理论.第 2 节研究对干涉问题进行形式化分析的方法.第 3 节给出工具设计与实现的方法.第 4 节进行实例研究.第 5 节比较相关工作.最后是总结和进一步工作的讨论.

## 1 设计演算

设计演算(design calculus)是一种基于指称语义的演算体系.设计(design)是对霍尔逻辑的扩展,它遵从霍尔逻辑的基本思想,即:用前置条件定义程序启动时环境必须满足的条件,再用后置条件定义程序终止时保证的条件.不同于霍尔逻辑仅能提供部分正确性证明的框架,一个设计还包含了对程序成功启动和正常终止的刻画,因此,设计演算可以处理程序执行是否发散与终止的问题.设计演算包含了程序构造算子作用于设计的运算规则、设计的最弱前提条件的计算规则以及设计的精化方法等演算方法.本节介绍设计演算,下面的定义及定理大都来源于文献[3,4].

**定义 1(设计(design))**. 一个定义在输入变量集合 $in\alpha$和输出变量集合 $out\alpha$上的设计是一个谓词,形如 $p(in\alpha)\vdash R(in\alpha,out\alpha')$,它被定义为:$(p\wedge ok)\Rightarrow(R\wedge ok')$.这里,$p(in\alpha)$是关于输入变量集合 $in\alpha$的谓词,$R(in\alpha,out\alpha')$是关于输入变量集合 $in\alpha$和输出变量集合 $out\alpha'$的谓词.一个设计规定:如果该设计被成功启动,即 $ok$=true,并且初始状态满足前置条件 $p$,那么它一定会终止,即 $ok'$=true,并且终止状态满足后置条件 $R$.为区分输出变量的起始和终止状态,对属于输出变量集合 $out\alpha$的变量添加上标撇表示其终止状态.辅助布尔变量 $ok$ 和 $ok'$的值描述设计的正常启动和终止,它不能被任何程序命令所操纵,是对从外部可以观察到的程序行为的描述.

存在几个特殊的设计:

© 中国科学院软件研究所 http://www.jos.org.cn

- 最强的设计 false,记做⊤;
- 最弱的设计 true,记做⊥;
- 设计 *skip* 是这样的一个设计 $skip \overset{def}{=} \text{true} \vdash \wedge_{x\in\alpha} x' = x$,它保持字符表中所有变量的值不变.

文献[4]中指出:设计既可以作为定义规约语言的数学工具,也可以定义程序设计语言中各种命令的语义.文献[3,4]中同时证明了,设计对于程序构造算子是封闭的.考虑这样几种常用的程序构造算子:

- 非确定选择结构 $Cmd_1 \sqcap Cmd_2$:程序即可以选择执行 $Cmd_1$,也可以选择执行 $Cmd_2$,且选择的过程是程序外部无法干预的;
- 条件选择结构 $Cmd_1 \lhd b \rhd Cmd_2$:条件 $b$ 满足则执行 $Cmd_1$,否则执行 $Cmd_2$;
- 顺序组合结构 $Cmd_1;Cmd_2$:先执行 $Cmd_1$,并以 $Cmd_1$ 的结束状态为起始状态执行 $Cmd_2$.

如果预先使用设计定义一种程序设计语言中各种命令的语义,那么使用这种语言开发的程序的语义就可以通过设计关于程序构造算子的演算规则计算出来.

这里以定理的形式给出程序构造算子在设计上的演算规则:

**定理 1**. 设计对于程序构造算子是封闭的:

$$(p_1 \vdash R_1) \sqcap (p_2 \vdash R_2) \equiv (p_1 \wedge p_2) \vdash (R_1 \vee R_2),$$
$$(p_1 \vdash R_1) \lhd b \rhd (p_2 \vdash R_2) \equiv (p_1 \lhd b \rhd p_2) \vdash (R_1 \lhd b \rhd R_2),$$
$$(p_1 \vdash R_1);(p_2 \vdash R_2) \equiv (p_1 \wedge \neg(R_1;\neg p_2)) \vdash (R_1;R_2),$$

其中,谓词的顺序组合 $P(s');Q(s)$定义为$\exists m \cdot P(m) \wedge Q(m)$.

本文基于最弱前置条件演算研究基础系统与方面之间的干涉问题,需采用如下定义设计与最弱前提条件连接起来:

**定义 2(设计的最弱前置条件)**:

$$wp(p \vdash R, q) \overset{def}{=} p \wedge \neg(R;\neg q).$$

定义 2 同时给出了计算一种设计最弱前置条件的方法.对于通过程序构造算子连接的两个设计,可以采用定理 2 描述的规则计算它们的最弱前置条件.

**定理 2**. 使用程序构造算子连接的设计的最弱前置条件可以由如下的规则得到:

$$wp(\text{true} \vdash v' = f(v), q(v)) = q[f(v)/v],$$
$$wp(D_1 \vee D_2, q) = wp(D_1, q) \wedge wp(D_2, q),$$
$$wp(D_1 \lhd b \rhd D_2, q) = wp(D_1, q) \lhd b \rhd wp(D_2, q),$$
$$wp(D_1; D_2, q) = wp(D_1, wp(D_2, q)),$$

其中,$f(x)$是一元函数,$b$ 是布尔表达式,$q$ 是谓词,$D_1=p_1 \vdash R_1$ 和 $D_2=p_2 \vdash R_2$ 是设计,$q[f(v)/v]$表示用 $f(v)$替换变量 $v$ 在谓词 $q$ 中所有的自由出现.

## 2 基础程序与方面干涉问题的分析方法

基础程序与方面不发生干涉是指基础程序的重要性质和行为(是否发散)在基础程序与方面编织而成的代码中依然保持.为实现形式化分析,首先使用设计定义基础程序中模块和方面中通知代码的规约;接着,需要描述编织机制的语义,以便可以从基础程序中模块和方面中通知代码的规约计算出集成后程序模块的规约;最后,运用最弱前置条件给出不产生干涉的条件.

大多数面向方面程序设计语言是从面向对象程序设计语言扩展而来,基于这样语言开发的软件中,基础程序的基本模块结构是类和类方法.因此,首先基于设计给出类方法的形式规约.

**定义 3(类)**. 一个类是一组域变量定义和方法定义的集合 $C=(FDef,MDef)$,其中,

- *FDef* 是一组域变量定义的集合,每个域变量定义形如 $x{:}T$,其中,$x$ 是变量名,$T$ 是类型;
- *MDef* 是一组方法定义的集合,每个形如 $m(x,y)$的定义,定义了一个名为 $m$ 的方法,其输入参数是 $x$,输出

© 中国科学院软件研究所 http://www.jos.org.cn

参数是 $y$.

针对定义 3 描述的每一个类,我们用下面的定义给出其形式规约的描述.

**定义 4(类的规约)**. 一个类的形式规约是四元组 $CSpec=(C,Init,Spec,Inv)$,其中,

- $C$ 是一个类;
- $Init$ 是一个谓词,定义类中域变量的初值;
- $Spec$ 是一个函数,它将类中的每个方法映射到它的形式规约上去,形如 $p(x,C.FDef)\vdash R(x,y',FDef')$的一个设计;
- $Inv$ 是一个谓词,定义了该类必须满足的性质.

定义 4 中,$Init$ 对应于构造函数为对象的域变量赋初值;一个类方法形式规约是关于域变量和该方法输入输出参数的一个设计.

采用类似的方法,我们可以定义方面和方面的形式规约.

**定义 5(方面)**. 一个方面是三元组 $A=(IT,PCut,Advs)$,其中,

- $IT$ 是一个跨类类型说明的集合,集合中每个形如 $x{:}T$ 的元素定义了类型为 $T$ 的跨类变量 $x$;
- $PCut$ 是一个切入点,它定义了基础系统上联结点的特征,方面中的通知代码将在基础程序中所有满足这样特征的联结点上与基础代码编织在一起;
- $Advs$ 是一组通知代码的集合,集合中的元素是 *before*,*after* 和 *around* 中的一种.

为简明起见,本文将联结点类型限定为类方法的被调用处,将通知代码的类型限定为 *before*,*after* 或 *around*. 其中,类型为 *before* 的通知代码会在类方法本体被执行前执行,类型为 *after* 的通知代码会在类方发执行结束后执行,类型为 *around* 的通知代码会直接替换类方法原来的实现代码.

**定义 6(方面的规约)**. 一个方面的形式规约是三元组 $ASpec=(A,Init,Spec)$,其中,

- $A$ 是一个方面;
- $Init$ 是一个谓词,定义方面中跨类变量的初值;
- $Spec$ 是一个函数,它用设计定义了方面中的每个通知的形式规约,形如:

$$p(A.IT,Context.(C.m))\vdash R(A.IT',Context.(C.m)');$$

这里, $Context.(C.m)\overset{def}{=}C.FDef\cup C.m.x$ 和 $Context.(C.m)'\overset{def}{=}C.FDef\cup C.FDef'\cup C.m.x\cup C.m.y'$ 是方法 $C.m$ 调用前后能存取的变量的集合,包括了类的域变量和方法的输入输出参数.

面向方面技术定义了编织机制,在编译时或运行时将基础程序和方面集成在一起.在编织过程中,编织器会将方面中切点的定义与基础程序中类和方法的型构相匹配.如果匹配成功,则在相应的联结点上将通知代码与基础程序的类方法集成在一起.引入集合 $Match(C,A)$记录方面与类匹配的结果.集合 $Match(C,A)$中的一个元素$(m,ad)$表示类 $C$ 中的方法 $m$ 可以与通知代码 $ad$ 所需的切点定义相匹配.

通过定义编织所得增强类的规约来描述编织过程的语义:

**定义 7(增强类的规约)**. 设 $C$ 是基础程序中的一个类,$A$ 是一个方面,且方面中的切点可以匹配类中的某个方法,那么通过编织得到的增强类 $C\oplus A$ 的规约由下面几条规则得到:

- $C\oplus A.FDef=C.FDef\cup A.IT$;
- $C\oplus A.MDef=C.MDef$;
- $C\oplus A.Init=C.Init\wedge A.Init$;
- $C\oplus A.Spec=\Phi$;
- $C\oplus A.Inv=C.Inv$.

其中,函数$\Phi$是这样定义的:

© 中国科学院软件研究所 http://www.jos.org.cn

$$\Phi=\begin{cases}Spec(m), & \text{if } (m.ad)\notin Match(C,A)\\ Spec(ad)[*/C.*];Spec(m), & \text{if } ad=before\wedge(m.ad)\in Match(C,A)\\ Spec(m);Spec(ad)[*/C.*], & \text{if } ad=after\wedge(m.ad)\in Match(C,A)\\ Spec(ad)[*/C.*], & \text{if } ad=around\wedge(m.ad)\in Match(C,A)\end{cases}.$$

上面的定义表明:增强类中的域变量是由类的域变量和方面的跨类变量共同组成的;增强类的方法与基础类方法相同;增强类的初始化由类和方面的初始化共同完成;增强类的不变式就是基础类的不变式.增强类中方法的规约就是基础类中对应方法的规约,如果此方法不是匹配成功的联结点;否则,根据通知代码不同的类型,将基础类方法的规约和通知代码的规约进行顺序组合或是替换,可以得到增强类方法的规约.

考虑一个关于增强类的运行环境的假设.由于编织的过程只会将满足切点定义的基础程序与方面进行集成,其余的类和方法保持不变,因此编织得到的增强软件系统通常是由基础类和增强类共同组成的.为明确增强类方法被调用的环境,不妨假设增强类会在对应的基础类相同的环境下被调用.考虑到基础类方法的前置条件描述了该方法被调用时环境能给予该方法最强的保证,以下关于干涉的讨论都假设增强类中的方法在对应的基础类规约中前置条件描述的状态下被调用.

根据关注的行为和性质的不同,方面与基础程序是否产生干涉可以有不同的定义.关于干涉问题,一些保守的定义要求方面不得修改基础类域变量的值,否则即认为有干涉.另一些稍为宽泛的定义允许方面修改基础类的域变量,但增强类中对应方法的后置条件不能被改变.上述两种定义都禁止方面修改基础类方法的功能,在具体开发中,对横切关注点的定义和限制太大.

本文采用了一种更加灵活的视角来定义干涉,不仅允许通知代码修改基础类域变量,而且允许通知代码增强基础类方法的功能.定义方面与基础程序之间没有发生干涉,只要基础类中的类不变式在增强类中仍然保持.这样的定义给予方面开发者最大的设计空间,他们可以按照横切关注点的需求,自由地对基础类的功能进行增强.同时,对于基础程序的设计者来说,也不需要关心未来哪些方面会编织进来,他们只需要使用类不变式描述自己的设计中哪些重要的性质在未来的变化中仍然需要保持.这一关于基础类中不变式的需求可以用最弱前置条件加以形式化的定义:

$$(C.I\wedge pre(C.Spec(m)))\Rightarrow wp(C\oplus A.Spec(m),C.I),$$

其中,$pre(C.Spec(m))$表示方法 $C.m$ 规约的前置条件.

这一定义要求增强类在与基础类相同的环境中执行时,可以保持基础类的不变式.基础类方法的前置条件描述了该方法被调用时,环境能够给予该方法最强的保证.上述定义描述了基础类方法的运行环境可以满足增强类对应方法执行后满足不变式所需的最弱前置条件.

不发生干涉的定义还需要考虑增强类的行为.为保证增强软件的执行不发散,增强类在对应基础类相同的环境中执行时,也不可以发散.这样的要求可以形式化的定义为

$$(C.I\wedge pre(C.Spec(m)))\Rightarrow pre(C\oplus A.Spec(m)).$$

这里,$C.I\wedge pre(C.Spec(m))$描述了基础类对应方法被调用时环境提供的最强保证,$pre(C\oplus A.Spec(m))$描述了增强类方法执行不发散所需的最弱条件.

行为上的干涉还需要研究增强类方法的执行对其后执行方法的影响.假设在基础类中,类方法 $C.n$ 可以在类方法 $C.m$ 执行结束立即被调用,那么在增强类中,也应该对对应的类方法作相同的要求.如果这一要求得不到满足,那么增强类就可能导致系统的执行发散.例如,基础类中原来存在 $C.m;C.n$ 的执行序列,在增强类中执行这一序列将直接导致系统发散.这一要求可以形式化定义为

$$(sp(C.I\wedge pre(C.Spec(m)))\Rightarrow pre(C.Spec(n)))\Rightarrow(sp(C.I\wedge pre(C\oplus A.Spec(m)))\Rightarrow pre(C\oplus A.Spec(n))),$$

其中,$sp(q,p\vdash R)\overset{def}{=}(\exists x_0\cdot q\wedge p[x_0/x]\wedge R(x,x')[x_0/x])[x'/x]$ 是在前置条件 $q$ 下启动设计 $p\vdash R$,执行结束后可以得到的最强后置条件.

合并上述关于干涉问题的形式定义,就可以得到分析基础程序和方面间是否发生干涉的形式化分析方法.

**定义 8(基础系统与方面不发生干涉)**. 设 $C$ 是基础程序中的一个类,$A$ 是一个方面,且方面中的切点可以匹

© 中国科学院软件研究所 http://www.jos.org.cn

配类中的某个方法.在增强类 $C\oplus A$ 中不会发生基础系统与方面间的干涉,如果以下几个条件得到满足:

- 增强类中所有方法都保持类不变式,即 $\forall m\in C\oplus A.MDef$,有:

$$(C.I\wedge pre(C.Spec(m)))\Rightarrow wp(C\oplus A.Spec(m),C.I).$$

- 增强类中所有方法可以在对应基础类方法的相同条件下启动并结束执行,即 $\forall(m,ad)\in Match(C,A)$,有:
  - 如果 *ad* 是 *before* 或者 *around*,则:

$$(C.I\wedge pre(C.Spec(m)))\Rightarrow pre(A.Spec(ad));$$

  - 如果 *ad* 是 after,则:

$$(C.I\wedge pre(C.Spec(m)))\Rightarrow wp(C.Spec(m),pre(A.Spec(ad)));$$

- 基础类中方法可行的执行序列在增强类中仍然可行,即 $\forall m,n\in C\oplus A.MDef$,有:

$$(sp(C.I\wedge pre(C.Spec(m)))\Rightarrow pre(C.Spec(n)))\Rightarrow(sp(C.I\wedge pre(C\oplus A.Spec(m)))\Rightarrow pre(C\oplus A.Spec(n))).$$

这里的第 2 个条件是根据定义 6 将增强类的规约展开后,化简原有的条件得到的.

## 3 干涉问题形式化分析工具的设计与实现

为方便设计人员在开发过程中应用上述形式化分析技术,我们设计和开发了针对面向方面设计中干涉问题的形式化分析工具.该工具提供了图形化设计界面,支持用户以UML类图为建模手段应用面向方面技术进行系统设计.用户可以直接在类图中的类和方面上添加形式规约.设计完成后,工具根据定义 7 的规则,自动生成验证基础系统和方面间是否发生干涉所需证明的条件.这些待证明的条件被传递给后端的交互式证明工具 PVS,在 PVS 的帮助下,用户可以方便地验证这些条件是否满足.如果所有条件都满足,则可以证明没有干涉发生;否则,可以在证明过程中分析仍缺少的条件,通过对方面中通知代码的规约进行修改和加强.最终证明所需的条件,保证设计模型中不会发生干涉.

### 3.1 工具总体结构

分析工具的总体结构如图 1 所示.工具包含了 3 个模块:用户界面模块、语义表示生成模块和 PVS 工具.用户界面模块的主要功能是提供了支持面向方面设计的类图,这一类图中除了包含标准 UML 类图中的建模元素以外,还扩展了方面、通知代码和联结点的图形元素.更为关键的是,用户界面模块提供了直接输入类和方面形式规约的手段.为提高开发效率,用户界面模块是通过扩展开源插件 *Topcased* 实现上述功能的.

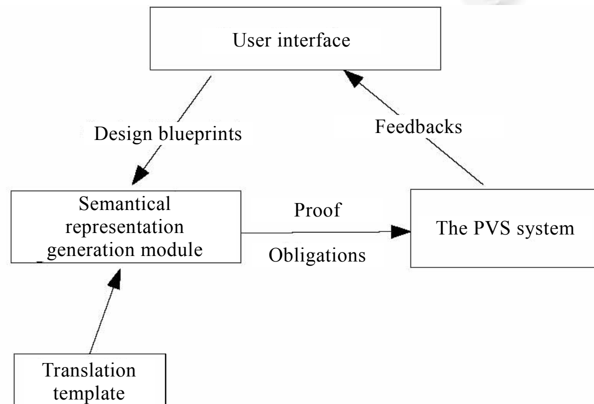


Fig.1 Structure of tool

图 1 工具总体结构图

语义表示生成模块的主要功能是生成验证干涉问题所需证明的条件,它接收从用户界面传来的设计蓝图,并根据文中的方法自动生成所需证明的条件,传递给交互式证明工具 PVS.PVS 支持基于一阶谓词逻辑的推导和证明,并不能直接识别和处理设计演算的推理规则,因此需要将设计演算的推理规则做成 PVS 的定理系统,提

© 中国科学院软件研究所 http://www.jos.org.cn

交给 PVS.用户界面传来的形式规约,也需要按照这一定理系统的模板,进行结构化翻译后再提交给 PVS 处理.

交互式证明工具 PVS 以两种方式被用户界面调用:自动证明和交互式证明.对于一些简单的设计模型,PVS 可以无需人工干预而自行完成整个证明过程,并将证明过程和结果返回到用户界面进行显示.对于复杂的例子,采用交互式方式进行证明,即:在证明过程中的关键步骤上,要求用户输入证明命令,驱动 PVS 向前推进证明的过程.PVS 也允许用户将自己的经验写成推理规则嵌入其中,并在随后的证明中自动应用它.PVS 的运用极大地提高了证明的效率,使得处理实际系统成为可能.

### 3.2 从设计规约到待证明PVS定理的自动转换

PVS 是一个交互式推理证明辅助工具,它的内部定义了一组面向一阶谓词逻辑和高阶谓词逻辑的推理规则,用户可以通过简单的命令调用这些规则推进定理的证明过程.同时,通过机械地循环应用推理规则,它具备了有限的自动推理证明能力,可以完成对一些定理的完全自动证明.为辅助对用户自定义逻辑系统的证明,PVS 提供了一套有效的逻辑系统构建方法.它允许用户定义新的类型和作用于新类型上的运算规则和逻辑推理规则.其中,新类型的运算规则和逻辑推理规则都是用 PVS 内置的运算规则和推理规则进行语义定义的.PVS 称一个新的类型及其上运算和推理规则为一个理论(theory).对于复杂的逻辑系统,可以采用分层技术,逐层定义所需的理论,最终在顶层构造出整个逻辑系统.由于新的逻辑系统是通过 PVS 内置的逻辑系统进行语义定义的,调用 PVS 提供的扩展命令 extend,就可以将自定义逻辑系统中待证明的假设,直接转换到一阶或者高阶谓词逻辑上的假设进行证明.

我们的工具充分利用了 PVS 提供的这一扩展机制,使得推理过程可以在 PVS 的辅助下完成.在第 2 节中,我们用谓词和设计定义了类与方面的性质和功能规约,为使 PVS 可以识别和转换这些谓词和设计,就需要在 PVS 中定义关于谓词和设计的理论.可知:实现 PVS 辅助证明的核心工作,就是如何基于 PVS 内置一阶谓词逻辑定义出谓词和设计的理论.

在 PVS 的 prelude 目录下的 booleans.pvs 中,我们找到了其内置一阶谓词逻辑的定义.PVS 定义了基本类型 boolean,定义了其上的运算符:合取、析取、取反和蕴含以及两个常量.对照我们在第 2 节引入的谓词和设计演算可以发现,PVS 内置的一阶谓词逻辑缺少了运算符:顺序组合和选择,因此需要在 PVS 中构造一个新的理论.它以一阶谓词为基本类型,除了支持一阶谓词上已有的合取、析取、取反和蕴含算子外,还需要支持顺序组合算子和选择算子.由于设计演算是以一阶谓词演算为基础的,因此可以在新构造的谓词理论上直接定义出设计理论.

- 一阶谓词理论的定义

扩展后的一阶谓词理论的基本类型为 *Predicate*,它是布尔函数,定义为 $P$:$X$→{true,false},其中,$X$ 为对应谓词表达式中出现的变量的集合,即字母表.扩展一阶谓词理论上的算符包括合取、析取、取反、蕴含、选择和顺序组合.这里,所有的运算符都定义成为形如 *f*:(*Predicate*,*Predicate*)→*Predicate* 的二元函数,此函数从两个输入的 *Predicate* 产生一个 *Predicate* 输出.例如两个谓词的顺序组合算子@@,按照定理 1 中的语义描述,在此谓词系统中表示为

@@=(*p*:*Predicate*,*q*:*Predicate*):*Predicate*=

*LAMBDA*(*vin*:*Type*;*vout*:*Type*):(*EXISTS*(*v*0:*Type*):*p*(*vin*;*v*0) and *q*(*v*0;*vout*)),

其中,*vin* 表示输入变量的集合,*vout* 表示输出变量的集合,它们有共同的类型 *type*.

上述定义使用了 PVS 中的匿名函数 *LAMBDA*.*LAMBDA* 是这样一种函数:它接收任意数量和任意类型的变量作为输入,并根据函数体的定义计算出函数的返回值.上面顺序组合算子的定义中,*LAMBDA* 函数接受两个类型为 *Predicate* 的变量作为输入,返回一个类型为 *Predicate* 的表达式.

- 设计理论的定义

基于扩展后的一阶谓词理论,可以在 PVS 中进一步定义设计演算理论.定义 1 给出了设计的谓词定义形式,使用这个定义,就可以在 PVS 中将设计演算理论与扩展后的谓词演算理论连接起来,从而在一阶谓词理论的基础上定义设计理论.设计被定义成为 PVS 中的二元组:

© 中国科学院软件研究所 http://www.jos.org.cn

$$Design:TYPE=[\#pre:Predicate,post:Predicate\#],$$

其中,*pre* 和 *post* 的类型都是 *Predicate*,分别表示设计中的前后置条件.

基于定理 1,可以在扩展后的一阶谓词理论上直接定义设计演算的基本算子:非确定性选择算子、条件选择算子和顺序组合算子.例如,设计演算理论中的顺序组合算子可以定义为

@@=(*d*1:*Design*;*d*2:*Design*):*Design*=
[#*pre*:=*pre*(*d*1) and *not*(*pre*(*d*1)@@*not*(*pre*(*d*2))),*post*:=*post*(*d*1)@@*post*(*d*2)#].

根据定义 2,可以定义出最弱前置条件的解析式:

*wp*(*d*:*Design*,*p*:*Predicate*):*Predicate*=*pre*(*d*) and *not*(*post*(*d*)@@*not p*).

最后需要说明的是类型转换.规约中出现的简单类型如 *string*,*int*,*float* 等可以直接翻译成 PVS 内置类型系统中对应的类型.结构类型(struct)可以转换到 PVS 中的记录类型,而枚举类型可以用 PVS 内置的枚举类型直接表示.例如:

- 结构类型:*struct strName*{*field*1:*Type*1,*field*2:*Type*2}可以在 PVS 中表示为
  *strName*:*TYPE*=[#*field*1:*Type*1,*field*2:*Type*2#];
- 枚举类型 *enum enuName*{*field*1,*field*2}可以在 PVS 中表示为
  *enuName*:*TYPE*=[#*field*1,*field*2#].

需要指出的是,所有类型相关的转换都是工具中专门开发的代码自动完成的.

综上所述,设计规约的转换过程是分两个步骤完成的:第 1 步,由专门开发的代码完成规约中类型的转换;第 2 步,将类型转换后的规约和预先完成的一阶谓词理论和设计理论定义文件提交给 PVS,再调用‘extend’命令,就可以直接得到待证明的 3 个条件.

## 4 实例研究

我们使用这一工具分析一个自动交换机仿真软件的设计模型.如图 2 所示,基础程序包括两个类:

- 类 *Device* 表示参与通话的两个终端设备;
- 类 *Connection* 负责管理通话的两个设备,它实现了 3 个方法:*locate* 方法可以根据电话号码定位对应的终端设备;*complete* 方法负责连接域变量 *origin* 和 *dest* 指向的两个终端设备;*drop* 方法负责中断当前的连接,结束两个设备的通话.类 *Connection* 需要满足性质:已经建立连接的两个终端不能是同一个设备,即,终端不能自己连接自己.

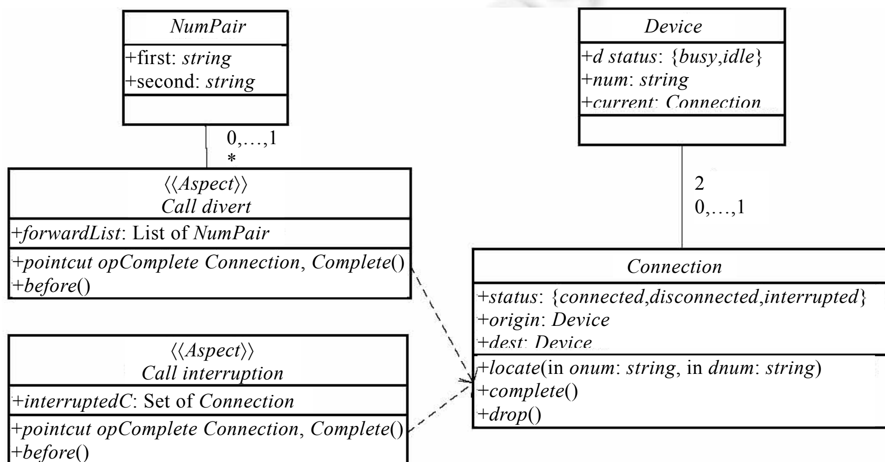


Fig.2 Class diagram of the telecommunication system

图 2 自动交换机仿真软件的类图

考虑基于面向方面的技术对基础程序的功能进行扩展.方面 *Call divert* 实现了遇忙转移的功能,即:当前被呼叫的终端忙时,可以根据跨类声明 *forwardList* 中的配置,将呼叫转移到第 3 台终端上.方面 *Call interruption* 在被呼叫的终端忙时,可以直接中断当前的会话,并建立被呼叫方与当前呼叫方的会话连接.这两个方面中,切点定义都指向了基础类 *Connection* 的方法 *complete*,并定义了类型为 *before* 的通知代码完成相应的功能.

- 基础程序中类 *Connection* 中的方法 *complete* 的规约为

$$\begin{aligned}&status = disconnected \wedge origin \neq null \wedge dest \neq null \wedge origin \neq dest\\ &\vdash\\ &status' = connected\\ &\wedge origin.d_status' = busy\\ &\wedge origin.current' = this\\ &\wedge dest.d_status' = busy\\ &\wedge dest.current' = this.\end{aligned}$$

- 基础程序中类 *Connection* 的类不变式为

$$status = connected \Rightarrow origin.num \neq dest.num.$$

- 方面 *Call divert* 中的通知代码 *before* 的规约为

$$\begin{aligned}&\text{true} \vdash\\ &\exists (f;t) \in forwardList, de: Device \cdot c.dest' = de \wedge c.dest.num = f \wedge de.num = t\\ &\lhd c.dest.d_status = busy \wedge \exists (f;t) \in forwardList, de: Device \cdot c.dest' = de \wedge c.dest.num = f \wedge de.num = t \rhd.\\ &c.dest' = c.dest\end{aligned}$$

将上述规约输入工具中的设计模型,并进行自动化分析.工具报告保持不变式不变这条性质无法得到证明.在证明过程中,PVS 将与这条性质有关的定理证明拆分成两条待证明的目标.图 3 是其中一条无法证明的目标.

```
Theorem 2.1:
{-1}    disconnected?(status!1)
{-2}    disconnected?(status0!1)
{-3}    [origin!1=de!2]
{-4}    busy?(dest_d_status!1)
{-5}    forwardList(fo!1)
{-6}    dest_num!1=f(fo!2)
{-7}    num(de!1)=t(fo!1)
{-8}    forwardList(fo!2)
{-9}    dest0!1=de!2
{-10}   f(fo!1)=f(fo!2)
{-11}   num(de!2)=t(fo!2)
{-12}   (origin0!1=de!2)
|...
{1}     (de!2=null)
{2}     (dest!1=null)
{3}     (de!2=dest!1)
```

(a)

```
Theorem 2.1:
{-1}    disconnected?(status)
{-2}    disconnected?(status0)
{-3}    busy?(dest_d_status)
{-4}    forwardList(fo)
{-5}    dest_num=f(fo!2)
{-6}    num(de)=t(fo)
{-7}    forwardList(fo!2)
{-8}    dest0=de!2
{-9}    f(fo)=f(fo!2)
{-10}   num(de!2)=t(fo!2)
{-11}   (origin0=de!2)
{-12}   (de!2≠null)
{-13}   (dest≠null)
{-14}   (de!2≠dest)
|...
{1}     [origin!1≠de!2]
```

(b)

Fig.3 An unproved sub goal about the class invariant property

图 3 关于类不变式的一个无法证明的目标

在显示证明结果时,PVS 会自动将肯定条件作为证明目标的前项,否定条件作为后项.图 3(a)显示的条件中,出现了很多形如“变量名+!+数字”的变量,这是 PVS 在自动量词消去过程中新产生的变量.在我们的分析方法中,存在量词是由顺序组合算子引入的.这一定义被随后的最弱前置条件定义,最强后置条件定义和增强类规约定义所引用.在这个例子中,方面 *Call divert* 中通知代码 *before* 的规约中含有存在量词,也会引发量词消去过程.基于这一原理,可以方便地通过变量名中!之前名字找到其在规约中对应的变量.

按照规约和待证明的性质,图 3(a)中的证明目标可以改写成图 3(b)中的形式.为使得证明目标成立,需要在

© 中国科学院软件研究所 http://www.jos.org.cn

前项部分增加条件 *origin*.≠*de*!2.对照输入的规约可知,*de*!2 对应于方面 *Call divert* 的通知代码 *before* 规约中的变量 *de*.据此我们修改此规约,向其中出现 *de* 的地方填加条件 *origin*.≠*de*,并再次提交给工具进行分析.此时,工具可以自动证明所有待证明的条件都成立,即,方面与类编织后不会发生干涉.

文献[12]中利用 Alloy analyzer 工具也对这一实例进行了分析.当检测出方面与基础程序的干涉后,由于 Alloy analyzer 是借助模型验证工具进行分析,它仅能给出一种反例,一组对象相互连接组成的对象网络,描述了可以引起干涉发生的一种对象的配置.设计人员需要仔细分析反例才能找到引发干涉的原因,进而对设计进行修正.与之相比,我们的工具给出的如何修正规约的提示要直接得多.

## 5 相关工作

干涉问题的检测与消除一直是面向方面研究的热点问题.Xu 等人[13]使用带标记的转换系统为基础程序、方面和编织机制建模.在验证时,这些基于状态的模型先被转换成为有限状态进程,再使用模型验证工具 LTSA 验证编织后的模型是否满足给定的性质.基础系统与方面之间的干涉问题,可以通过验证给定的性质在编织后的程序中是否仍然满足进行判定.

基于模型验证技术,Katz 等人[7]研究了他们称其为叠加技术的方面重用.类似于前后置条件,当满足叠加规约的一个方面被编织到一个基础程序上时,只要基础程序满足叠加规约定义的假设条件,那么编织后的程序就可以满足特定的性质,并且符合基础程序的规约.他们实现了一个原型工具[2],并调用 Bandera 工具从 Java 程序中自动生成 SPIN 或者是 SMV 的输入.赵建军等人[18]扩展规约语言 JML,提出了 AspectJ 的规约语言 Pipa,支持对基础系统和方面的规约描述,并利用 JML 工具进行性质分析.以上的工作都是直接对编织后的代码进行整体验证,受制于模型验证技术的可扩展性,它们都不能直接对功能性质进行验证,只能处理相对简单的行为特性,而且很难扩展到比较大的程序上去.

为解决可扩展性的问题,学者们提出了增量验证的方法.Krishnamurthi 等人[11]提出:在一个方面编织入基础程序时,应立即对基础程序的不变式进行验证.这样一来,验证就可以不必扩展到整个基础程序上.这一方法在验证大的基础程序与多个方面之间的干涉时特别有效.Goldman 等人[3]提出了基于 assume-guarantee 规约的模块化验证方法.在这一方法中,一个方面被构造成由基于 LTL 描述的假设、关于切点的描述和定义通知代码行为的自动机组成的单自动机.通过基于 LTL 的假设对方面依赖的基本程序进行充分的抽象,方面能够保证的性质可以在没有基本程序的情况下,使用模型验证工具独立进行验证.在方面编织入基础代码之前,再使用工具对基础程序的状态机是否满足方面的假设进行验证.这些工作的目标仍然是与程序行为有关的性质,而且当干涉出现后,他们也不提供指导消除干涉的方法.

Aksit 等人[1]提出了基于图转换的方法来检测位于同一切点上多个方面之间的干涉问题.他们用一套图的转换和产生规则为面向方面的语言定义了一个运行时刻的语义.通过对方面的执行进行仿真,可以自动产生出运行状态空间,并在此状态空间上对性质进行分析和验证.为降低复杂性,他们开发了无需对整个系统进行仿真运行的方法.与验证方法类似,仿真运行也需要大量的计算能力和空间,可扩展性有限

Mostefaoui[12]开发了基于形式化规约语言 Alloy 检测 Aspect-UML 模型中干涉问题的工具.Aspect-UML 模型允许使用前后置条件定义类和方面的规约.验证时刻,这些规约被自动转换成为 Alloy 的结构体,再依靠模型验证工具 Alloy analyzer 进行验证.与我们工具不同的是:当干涉发生时,模型验证工具会给出一个反例描述干涉存在的一种情况.用户需要仔细分析干涉发生的各种反例,再逐一针对这些反例对规约进行修正.同时,受模型验证工具底层求解器能力的限制,这一工具的可扩展性也比较有限.

Khatchadourian 等人[8]提出了运用 Rely-guarantee 技术分析干涉问题的方法.他们将基础程序和方面看成是独立的进程,并引入四元组(*pre*;*rely*;*guar*;*post*)作为进程的规约,其中,(*rely*;*guar*)定义了允许被植入的进程的规约.他们认为,面向方面技术中的干涉问题与并发进程的分析问题是类似的.通过他们的方法,就可以直接将并发程序的分析方法[14]运用在面向方面程序的分析上.然而,如何计算(*rely*;*guar*)是一个十分困难的问题,他们在文中并没有给出具体可行的方法.

© 中国科学院软件研究所 http://www.jos.org.cn

Dantas 等人[2]将基础系统与方面之间不发生干涉定义为:除了改变基础程序的中止性和进行输入/输出操作之外,方面不改变程序任何其他的功能和行为.为分析这样的行为,他们为一个小型的面向方面语言定义了操作语义,并给出了判定不发生干涉的推理规则.Oliveira 等人[14]用函数式语言给出了类似的非干涉定义和形式推理方法.与本文的非干涉定义相比较,这两个工作处理和识别的非干涉是非常受限的.同时,他们的语义模型都是定义在程序设计的每条语句之上,模型抽象层次较低,在分析干涉问题时推理过程比较复杂,且缺少辅助工具的支撑.

Selim 等人[17]将方面间不发生干涉定义为:多个方面编织在同一个基础系统上之后,执行序列中方法的执行顺序仍然满足通过基础系统不变式描述的时态逻辑公式.他们为 UML 模型定义了操作语义,并基于此语义将一个设计所有可能的执行序列表示为一张图,然后,利用模型验证工具验证不变式是否满足.

上述干涉分析方法和工具主要是应用模型验证技术分析面向方面设计中的干涉问题.受模型验证技术可扩展性的限制,这些方法都要求用户构造抽象模型以控制待验证问题的规模和复杂度.因此,行为模型以及行为有关的性质成为这类工作主要关注的内容.相对于行为干涉,面向方面设计的功能上的相互干涉,是开发人员首先关注的问题.我们的工具支持用户在设计模型上直接使用前后置条件定义基础程序和方面的功能规约,用户无需额外构造模型,这样的方式更容易被用户掌握和运用.同时,交互式证明工具的引入减轻了用户证明时的负担,提高了证明效率和处理大型程序的能力.更重要的是:证明过程的中间结果可以直接指导用户对规约进行修正,有效地排除设计中存在的干涉.这种方式比反例制导的规约修正方法更加直接和高效.最后,本文的形式化模型定义在模块的层次,相对于定义在语句级别上的形式模型,在处理干涉问题时,抽象层次更好、更加有效.

除了上述基于形式化技术的干涉分析工具,很多工作基于基础程序和方面对域变量的读写行为分析干涉是否存在.Clifton 等人[15]最先提出将方面分为观察者和协助者两种类型:观察者只读取基础程序变量的值,协助者可以改变基础程序变量的值,观察者与基础程序之间一定不发生干涉.Rinard 等人[16]进一步根据方面和基础程序对共享变量的读写关系,将方面与基础程序之间的关系分为正交、独立、观察、驱动和干涉 5 种类型.他们开发了自动分析工具,实现对方面的自动分类.这些工作的优势在于:将干涉问题的分析转化为程序对变量存取行为的分析,比较容易实现自动化,且分析速度较快.同时可以看到:由于缺少精确的语义分析,因此上述工作对干涉的定义和分析都是相当粗糙的,远远不能满足面向方面技术对功能干涉进行精确分析的要求.

## 6 总 结

分析和排除面向方面设计中可能存在的干涉,是保证面向方面设计正确性不可缺少的步骤.本文基于推理验证技术设计和实现了一个分析面向方面设计模型中干涉问题的软件工具.根据基础程序和方面的功能规约,工具可以自动地生成证明无干涉存在所需的条件,并调用后端的交互式证明工具 PVS 辅助对这些条件的证明.成功的证明可以确认设计模型中不存在功能干涉,失败的证明结果可以指导开发人员通过修正规约消除功能干涉.进一步的工作将重点考虑基于类似的技术研发分析编织在同一基础程序上的多个方面之间的干涉问题的工具;研究根据基础程序的设计规约计算出非干涉方面规约的方法,进一步降低验证过程的复杂度;同时设计更多的推理规则,进一步提高推理证明过程的自动化程度.

**References**:


[1] Filman R, Elrad T, Clarke S, Aksit M. Aspect-Oriented Software Development. Addison-Wesley Professional, 2004.

[2] Dantas DS, Walker D. Harmless advice. In: Morrisett JG, Jones SLP, eds. Proc. of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 2006. 383−396. [doi: 10.1145/1111037.1111071]

[3] He JF, Li XS, Liu ZM. rCOS: A refinement calculus of object systems. Theoretical Computer Science, 2006,365(2):109−142. [doi: 10.1016/j.tcs.2006.07.034]

[4] Hoare CAR, He JF. Unifying Theories of Programming. London: Prentice-Hall, 1998.

[5] Xu DX, Alsmadi I, Xu WF. Model checking aspect-oriented design specification. In: Proc. of the 31st Annual Int'l Computer Software and Applications Conf. New York: IEEE Computer Society, 2007. 491−500. [doi: 10.1109/COMPSAC.2007.152]


© 中国科学院软件研究所 http://www.jos.org.cn

[6] Katz S, Sihman M. Aspect validation using model checking. In: Dershowitz N, ed. Proc. of the Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday. LNCS 2772, Berlin: Springer-Verlag, 2003. 373−394. [doi: 10.1007/978-3-540-39910-0_18]

[7] Corbett JC, Dwyer MB, Hatcliff J, Robby. Bandera: A source-level interface for model checking Java programs. In: Ghezzi C, Jazayeri M, Wolf AL, eds. Proc. of the 22nd Int'l Conf. on Software Engineering. New York: ACM Press, 2000. 762−765. [doi: 10.1145/337180.337625]

[8] Krishnamurthi S, Fisler K, Greenberg M. Verifying aspect advice modularly. In: Taylor RN, Dwyer MB, eds. Proc. of the 12th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2004. 137−146. [doi: 10.1145/1029894.1029916]

[9] Goldman M, Katz S. Maven: Modular aspect verification. In: Grumberg O, Huth M, eds. Proc. of the 13th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer-Verlag, 2007. 308−322. [doi: 10.1007/978-3-540-71209-1_24]

[10] Aksit M, Rensink A, Staijen T. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: Sullivan KJ, Moreira A, Schwanninger C, Gray J, eds. Proc. of the 8th Int'l Conf. on Aspect-Oriented Software Development. New York: ACM Press, 2009. 39−50. [doi: 10.1145/1509239.1509247]

[11] Mostefaoui F, Vachon J. Design-Level detection of interactions in aspect-uml models using alloy. Journal of Object Technology, 2007,6(7):137−165. [doi: 10.5381/jot.2007.6.7.a6]

[12] Khatchadourian R, Soundarajan N. Rely-Guarantee approach to reasoning about aspect-oriented programs. In: Bergmans L, Brichau J, Ernst E, Gybels K, eds. Proc. of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies. New York: ACM Press, 2007. 5. [doi: 10.1145/1233843.1233848]

[13] Xu QW, de Roever WP, He JF. The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspect of Computing, 1997,9(2):149−174. [doi: 10.1007/BF01211617]

[14] Oliveira BCdS, Schrijvers T, Cook WR. EffectiveAdvice: Disciplined advice with explicit effects. In: Jézéquel JM, Südholt M, eds. Proc. of the 9th Int'l Conf. on Aspect-Oriented Software Development. New York: ACM Press, 2010. 109−120. [doi: 10.1145/1739230.1739244]

[15] Clifton C, Leavens GT. Observers and assistants: A proposal for modular aspect-oriented reasoning. In: Proc. of the 1st Workshop on Foundations of Aspect-Oriented Languages. 2002. 33−44.

[16] Rinard M, Salcianu A, Bugrara S. A classification system and analysis for aspect-oriented programs. In: Taylor RN, Dwyer MB, eds. Proc. of the 12th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2004. 147−158. [doi: 10.1145/1029894.1029917]

[17] Ciraci S, Havinga W, Aksit M, Bockisch C, van den Broek P. A graph-based aspect interference detection approach for UML-based aspect-oriented models. Trans. on Aspect-Oriented Software Development VII, 2010,(7):321−374. [doi: 10.1007/978-3-642-16086-8_9]

[18] Zhao JJ, Rinard M. Pipa: A behavioral interface specification language for AspectJ. In: Pezzè M, ed. Proc. of the 6th Int'l Conf. on Fundamental Approaches to Software Engineering. Berlin: Springer-Verlag, 2003. 150−165. [doi: 10.1007/3-540-36578-8_11]

**陈鑫**(1975－),男,辽宁大连人,博士,讲师,主要研究领域为软件工程,软件测试和验证技术.

**张一帆**(1989－),男,主要研究领域为软件工程,软件开发,设备驱动程序.

**黄超**(1988－),男,博士,主要研究领域为信息物理融合系统的建模、验证、仿真和控制.

**梅一鸣**(1992－),男,主要研究领域为软件工程,软件开发,软件测试.


© 中国科学院软件研究所 http://www.jos.org.cn