一个基于Eclipse的通用Java程序插桩工具

郑晓梅
(南京中医药大学信息技术学院 南京 210046)
(南京大学计算机软件新技术国家重点实验室 南京 210093)

摘 要 插桩技术作为一种有效理解程序动态行为的手段,已经被广泛应用于程序分析、测试和验证中。然而,由于缺少通用的插桩工具,各种具体的应用往往需要从头开发特定的插桩程序,存在着大量的重复性工作。此外,由于在原始程序中插入了大量额外代码,致使调试过程变得更加复杂和困难。针对这些问题,提出了一个基于Eclipse的通用Java代码插桩工具,即通过规则定义匹配程序的执行点,从而定制针对各种分析、测试和验证插桩需求的支持。通过对插桩代码片段的显式/隐式切换实现其可见性管理,从而确保程序的理解和调试过程不受插桩代码影响。通过使用该工具,可以更好地将插桩技术应用于Java程序开发中。

关键词 程序插桩,Java,Eclipse插件

General Java Program Instrumentation Tool Based on Eclipse

ZHENG Xiao-mei
(School of Information Technology, Nanjing University of Chinese Medicine, Nanjing 210046, China)
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

Abstract Program instrumentation technique has been widely used in program analysis, testing and verification as an effective method for understanding the dynamic information of programs. However, for the absence of general instrumentation tools, different applications need to recreate the specific tool meeting their requirements, hence wasting time and energy. In addition, the debugging processes become more and more complicated because of the instrumented code inside the original program. To solve such problems, the paper presented a general instrumentation tool based on Eclipse for Java program. By defining the rules to match the execute points of program, a specific instrumentation tool could be customized to meet the different requirements of Java program analysis, testing and verification. By switching between visibility and non-visibility of the instrumented code, users of our tool can't be disturbed by the planted fragments in comprehending and debugging java program. We believe that through utilizing our tool, the program instrumentation technique can be better applied in the Java program development.

Keywords Program instrumentation, Java, Eclipse plugin

1 引言

随着计算机在人们生活、生产中的普遍应用,软件可信性问题越来越受到人们的关注。测试、分析和验证等技术被认为是可以有效提高程序可信程度的手段,而插桩技术作为一种获取程序动态运行状态的基本方法,已普遍用于此类技术中。

程序插桩是在源程序中植入代码片段,并通过运行程序来获得该程序动态信息的技术。通过插桩可以收集程序在执行过程中某一时刻的系统状态(快照)信息。比如,通过插桩收集某次执行的路径覆盖信息[1];还可以通过插桩收集函数调用图,并以此来验证某个属性(如函数调用关系、调用顺序是否符合一定的规则)。目前利用插桩技术的常用方法是:对特定的应用编制一个特别的插桩程序,将代码直接插入到源程序中。比如文献[2,3]等就是根据具体应用单独各自编写插桩代码。然而这样做往往会导致一次次的重复开发,无谓地浪费了时间和人力;另外,插桩代码充斥在整个逻辑代码中,也非常影响开发人员对程序逻辑的把握。那么,能否有一种较为通用的插桩手段和工具,来支持目前测试、分析及验证等技术对于程序插桩的不同需求,又可以对所植入的插桩代码进行有效管理呢?这正是本文所要解决的问题。

本文提出了一种通用的Java程序插桩方法,并基于Eclipse开发了相应的支撑工具EStrumentJ(Eclipse Instrument for Java),使Java程序的开发人员能够通过该工具为测试、分析以及验证等不同的需求定制相应的插桩实现。同时,该工具还可以对源程序和所植入的代码片段进行显式隔离与合并。用户既可以只关注源程序本身(这样就可以有效屏蔽所植入代码对源程序理解上的干扰),同时可以清晰地看到植入点的位置及相应的提示信息,从而对植入代码如何影响源程序有一个明确的把握。针对不同的插桩需求,用户还可以在

到稿日期:2010-08-15 返修日期:2010-11-22 本文受863国家高技术研究发展计划(2007AA010302),国家自然科学基金(60425204),江苏省自然科学基金(BK2007714),江苏省高校自然科学基金(07KJB10002)资助。

郑晓梅(1978—),女,讲师,主要研究方向为医学信息安全、中医特征信息建模与分析验证等,E-mail:zxm_luck@163.com。

EStrumentJ 中定义匹配规则，用来捕获程序的特定执行点，并在该执行点处插入代码片段。

与其它工具显著的不同点是，EStrumentJ 是基于 Eclipse 官方插件 JDT(Java Development Tool)所提供的 Java 抽象语法树实现程序执行点的定义，因此具有更细的粒度，不但可以匹配方法调用点，甚至可以匹配源程序中循环和分支的内部位置。

本文首先简要介绍 EStrumentJ 的设计思想，然后给出底层插桩引擎的系统结构，同时说明插桩引擎的实现方法，之后进一步介绍基于底层插桩引擎的上层手工插桩和基于规则匹配的批量自动插桩插件的实现，最后将我们的工作与其他相关工作进行了比较，并指出本文的贡献及未来工作的方向。

2 设计思想

2.1 关注点分离

在传统的编程情况下，代码中经常会充斥着很多与业务功能无关的代码，比如安全控制、记录日志等。其实这些关注点与软件的业务功能并没有直接的关系，它们可以说是正交于业务逻辑功能的。这些与业务功能无关的代码数量很多，经常会“淹没”了业务功能代码，使维护人员不易清晰地看出原开发人员的设计意图。AOP[4]改变了这一切，它的思想就是要把独立于功能的软件模块从软件开发中独立出来，使程序员不必关心业务逻辑以外的事情，比如安全需求、日志要求等。这使得程序看上去很“干净”。

我们把这种“干净”的思想融入到我们的设计中。对于有着如此广泛需求并且与业务无关的插桩技术，也应该像安全控制一样从软件开发中剥离出来，成为一个独立的关注点。对于这样一个关注点，需要提供一个基础设施，使这个关注点能够得到发挥。我们的做法是将这个关注点实现为一个 Eclipse 插件，作为 Eclipse 的一个基础设施，随时能够供程序员使用。我们这个关注点相对于安全控制、日志等关注点又是有区别的，其所插入的代码都会被编译进发布的软件中。而这个关注点的代码只出现在开发阶段，发布出去的软件不包括这些插入的代码。

2.2 无痕

我们认为，计算机应该像文字一样无痕地嵌入到生活的方方面面，无需刻意地注意那些计算机的存在。事实上，我们确实淹没在其中，无时无刻不在使用它。它是我们手上随时可用的工具，不管你是否需要，它就在那里。我们将无痕的想法也融入到我们的设计思想中。要做到无痕，在插桩的时候，不应该给程序员造成不必要的困扰，插桩的代码对程序员来说应该不可见。同时，要使程序员能够感知一个插桩点的存在，程序员必须能够确切知道在什么地方会有什么样的插桩动作将会进行，这才不会使程序员感到意外。我们选择了 Eclipse 提供的标记(marker)设施，来达到这个目的。标记是 Eclipse 为插件开发者提供的一种可以出现在编辑器左侧标尺上的控件，这个控件与一个文件的某一行关联，并能携带一些可持久化的属性，这些属性可用来传递程序员的插桩意图。

2.3 强扩展性

如同很多软件(比如 vim[5]，emacs[6])一样，Eclipse 有很好的扩展机制。这使得 Eclipse 不会局限于开始的一些功能，不会局限于某一种编程语言，也不会局限于一种编程方式；这使得 Eclipse 可以不断成长，它可以利用第三方开发的插件来扩充自身的功能。

我们开发的插桩引擎也采用了同样的思想：只将一个最小的核放在引擎里面。这个核只提供编译执行插桩程序的基础设施。利用 Eclipse 提供的插件机制，我们对外提供了一个扩展点，这个扩展点定义了插桩动作，该动作被插桩引擎调用。这样，更多的动作细节通过提供扩展点的方式，由外来的插件完成。这样插件通过扩展这个扩展点，增加不同的插桩动作。下一部分开始介绍这个插桩引擎的系统结构。

3 系统结构

我们设计的系统结构如图 1 所示。

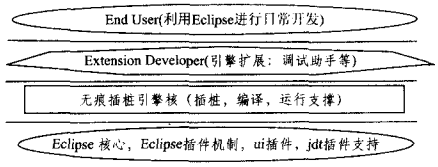

图 1 无痕插桩系统框架结构图

利用 Eclipse 插件机制，构建了一个多层次的系统结构。利用 Eclipse 的现有插件，构建了一个引擎核，这个引擎核并不进行特定的插桩动作，只提供一个编译执行的基础设施。在这个引擎核之上，可以定义一些插桩插件，这些插件通过扩展引擎核提供的一个扩展点来实现自己特定的插桩需求。这些扩展了引擎核的插件同样可以继续被扩展，也可以直接与最终用 Eclipse 进行日常开发的程序员交互。插桩引擎核可以同时支持多种插桩需求，当对一个文件执行完所有的插桩以后，再将不同的插桩动作通过 diff-patch 的方式合并到一起。用户也可以通过配置，使插桩引擎只处理一种(或若干种)插桩动作，如图 2 所示。

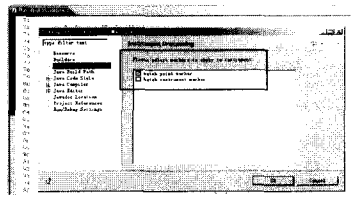

图 2 插桩引擎配置

被选中的标记类型所表示的插桩需求将被满足。未被选中的需求，即使在程序中添加了相应的插桩标记，也不会起作用。当用户试图执行该被插桩程序时，插桩引擎就收集用户对原程序的插桩意图(标记)，根据用户对原程序的插桩及配置来对原程序进行插桩，然后保存 Eclipse 为原程序生成的 class 文件，编译插桩后的代码。执行插桩后的代码，待该被插桩程序执行完毕后，再将保存的 class 文件恢复到原先的地方。如果编译出错，则直接将保存的 class 文件恢复到原处，并在编辑器中向程序员展示编译出错的地方。下面一部分就对该插桩引擎的实现做一个详细的介绍。

4 插桩引擎的实现

4.1 插桩引擎的工作流

该引擎的工作分成 4 部分：1)插桩点的定义与显示；2)实现插桩动作；3)被插桩程序的编译与执行；4)被插桩程序编译出错时的处理。下面逐一介绍这 4 个方面的功能。

我们要求插桩点的定义是，它们不应该被硬编码到引擎中，引擎扩展者(ED)可以自定义自己的插桩要求；同时，引擎

必须能够感知到引擎扩展者的插桩意图，这样引擎才能在编译的时候执行相应的处理动作。另外，考虑到最终 Eclipse 使用者(EU)可能希望在下次打开工程时，上次所设置的插桩效果依然存在，不需要再次重新设置插桩，我们需要保存它们所设置的插桩点，因此用一个标记(marker)来标识一个插桩点。标记是 Eclipse 开发环境提供的一种资源，可与一个文件的某一行进行关联，标记的类型决定了不同的插桩类型。插桩引擎通过查找资源方式获取一个文件中所有的标记，进而得到所有的插桩点。标记将会出现在集成开发环境编辑区左侧的标尺上，为不同的插桩标记赋予不同的提示信息，以使使用者能够了解当时设置插桩点时的意图。将该标记设置为可持久化的，使插桩信息能够在不同的会话之间传递。引擎工作流如图 3 所示。

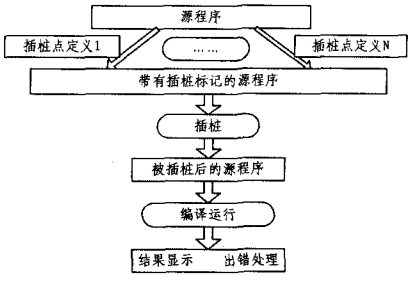

图 3　引擎工作流

与插桩点的定义一样，我们也希望插桩动作不要固化在引擎中，这个动作应该交由引擎的扩展者来实现。为了达到这样的效果，我们在引擎中声明了一个新的扩展点。不同类型的插桩动作都需要扩展这个扩展点。引擎的扩展者在扩展引擎时，必须实现一个接口 IInstrumenter，该接口声明了如下 3 个方法：

```
public String getMarkerLabel();
public String getUtilities(IProject project);
public void instrumentFile(String fromFile, String toFile, IMarker[]
    markers);
```

其中的 getMarkerLabel 方法返回图 2 中显示的标记类型名字。instrumentFile 方法利用文件中包含的标记(markers)对 fromFile 进行插桩，插桩结果被输出到 toFile 文件中。具体的插桩动作留给了扩展者。不同的扩展者可能需要定义这样一些方法，即能够在被插桩的程序中使用。getUtilities 方法为他们提供了一个机会，让其能够提供一些方法，作为该函数的返回值，在后面的实例中将看到这个方法的使用。

插桩引擎现在只支持 Java Application 的编译与运行。通过扩展 org. eclipse. debug. core. launchConfigurationTypes，增加了一种新的执行类型，该扩展点的代理类直接继承 org. eclipse. jdt. launching. JavaLaunchDelegate。在 launch 方法中，先将原程序的 class 文件保存起来，以避免以后因为再次编译而花费额外的时间。编译好被插桩后的程序以后，再调用父类的 launch 方法运行程序。为了编译被插桩后的程序，我们使用了 Eclipse 的 jdt 插件的内部编译器(org. eclipse. jdt. internal. compiler. batch. Main[1])。为了配置运行时的参数，我们使用了与普通 java application 同样的参数配置界面。另外，为了使被插桩程序执行完毕后不会影响原程序，达到无痕的目的，我们增加了一个回调函数。该回调函数在被插桩程序运行完后，将原先保存的原程序的 class 文件再复制回去。

程序在编译时难免会出错，我们为出错的编译提供了两种信息输出方式：直接显示和记录日志。为了捕获编译出错的信息，我们在调用 org. eclipse. jdt. internal. compiler. batch. Main 的编译方法(boolean compile(String commandLine, PrintWriter outWriter, PrintWriter errWriter))时给了一个 errWriter 参数，该参数指示编译器将出错信息定向到该流(stream)。为了将编译出错的信息立刻显示在用户界面上，我们定义了一种新的标记，使其继承 org. eclipse. jdt. core. problem，并在每个标记上提供一些简单信息，以提示开发者(EU)编译错误的位置。在标记上标示的信息是有限的，因为不太适合将过于详细的信息(比如被插桩的代码)显示在标记上，而且程序员有时也需要完整的信息来发现问题，所以需要另外的方式向他们展现完整的出错信息，这些信息就被记录在日志文件中。在插桩过程中，插桩后的文件的行数可能会发生变化，这就需要解决一个问题：如何正确地定位错误。这里，我们通过标记的属性(extra_lines)来记录该标记所代表的插桩代码需要占用的额外行数。在出错时，通过计算得到被插桩后的程序中的行所对应的源程序的行数。引擎扩展者负责为自己的标记赋予 extra_lines 属性，如果没有赋值，则认为该插桩动作没有占用额外的行。

4.2　插桩引擎提供的扩展点

为了使引擎的扩展者能够定义一种插桩动作，我们的插桩引擎提供了一个扩展点 instrument：

```
〈! ELEMENT extension(instrumenter)〉
〈! ATTLIST extension
    point CDATA #REQUIRED
    id CDATA #IMPLIED
    name CDATA #IMPLIED〉
〈! ELEMENT instrumenter EMPTY〉
〈! ATTLIST instrumenter
  class CDATA #REQUIRED
  marker_type CDATA #REQUIRED〉
```

从上面的定义可以看出，最重要的是 instrumenter 项，该项提供了两个域：class 和 marker_type。class 域需要引擎扩展者提供一个实现 IInstrumenter 接口的类，而 marker_type 是一种标记类型的 id，该标记是扩展者用来标识插桩点的，因此该标记就使引擎能够查找一个文件里面的所有插桩点。一个新的插桩需求就可以通过扩展该扩展点来实现。下面两部分介绍两个实例，这两个实例利用这个插桩引擎开发了两个实用工具。

5　基于插桩引擎的手动与自动插桩

5.1　手动插桩

随着计算机技术的日益发展，对称多处理、多核等新技术不断涌现。为了充分利用这些新技术，多线程、多进程等程序范性也经常被人们使用。多线程程序一般逻辑比较复杂，很难一次性开发出没有逻辑错误的软件，这就需要对程序进行调试。但是普通的调试办法(设置断点-执行到断点-检查状

1)　虽然在 Eclipse 插件中使用内部类是不被建议的，但是作为一个编译主程序，其接口应该是不会随便改变的。另外，jdt 插件的文档中也明确地指出编译 java 程序应该使用该类。

态)并不能适应多线程程序:多线程程序的本质是多个执行体同时执行,如果让其中的某一个线程暂且挂起,不能反映真实的执行环境。另外,由多个线程组成的进程,每次执行都是一个特殊的各个线程之间的交错实例。一旦让线程因为非程序自身的原因而挂起,它就不能反映当时那一次真实的执行轨迹。调试时最好的办法是不要挂起线程,而是使其一直执行。可以使用最原始的调试手段:插桩调试,即在需要检查状态的地方插入一段代码,用以显示或捕捉那个时刻的系统状态。然而不断地向工程里面"加入代码-调试-修改代码-删除代码"是一件相当费时、费力的事情,如果需要监测状态的地方太多,也会导致管理很混乱,程序员自己有可能都弄不清楚在哪些地方增加了一些代码来监测状态。为了减轻程序员的工作量和维护效率,可以利用本文涉及的无痕插桩引擎来提供一个调试助手。通过这个调试助手,程序员不必显示向程序中添加代码,而只要在某些感兴趣的地方设置一些标记(就如同设置断点一样),并赋予这些标记一定的属性,来表达程序员监测系统状态的意图。

5.1.1 定义扩展

通过扩展上一节中描述的插桩框架提供的扩展点(instrument)来实现上面的调试助手所需要的插桩要求,扩展定义如下:

```
〈extension point="cn. edu. nju. seg. debug. assist. ui. engine. instrument"〉
    〈instrumenter
        class="cn. edu. nju. seg. debug. assist. ui. watchpoint. WatchPointInstrumenter"
        marker_type="cn. edu. nju. seg. debug. assist. ui. watchpointmarker"〉
    〈/instrumenter〉
〈/extension〉
```

在这个定义中,WatchpointInstrumenter 类实现了接口 IInstrumenter。而 watchpointmarker 标记提供了另外一些设施:插桩点的显示、插桩信息的保存和持久化。在该调试助手中,我们提供了两种插桩设施:表达式求值与原生代码。程序员(EU)可以在原程序的某一行(在该行之前)检查某一个表达式的值(包括简单变量),表达式之间用分号作为分隔符。

有时候,程序员可能有更复杂的需求。这时候,他们还可以通过直接在某一行(同样,在该行之前)插入一段 java 代码,该代码会被原封不动地插入,我们称之为原生代码。

5.1.2 标签的生成

本插件扩展了 org. eclipse. ui. popupMenus 扩展点,以 #CompilationUnitRulerContext 为 TargetID,即右击编辑器左侧的标尺,在出现的菜单中选择"EyeWatch…"对插桩点进行编辑,如图 4 所示。

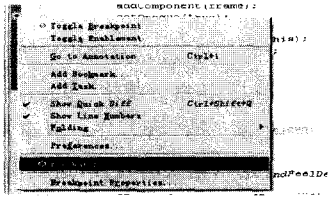

图 4 调试助手示意图

插入的表达式与原生代码将在程序运行之前被插入到原文件中,与原文件一起进行编译。

5.2 基于规则的批量自动插桩

在测试和验证中,要插桩的地方非常多,不可能像上面的手动插桩一样,手工地在需要监测系统状态的地方插入插桩点。这需要一个能够理解程序员(EU)的意图,从而进行批量插桩的设施。为此,可以利用前面的插桩引擎开发一个基于规则的批量插桩工具。程序员提供一定的规则,然后由这个插桩工具根据这些规则对程序员希望插桩的代码进行插桩。

5.2.1 规则定义

我们的批量插桩工具是基于规则的,就是说由程序员给出插桩规则,然后我们的插桩工具完成插桩,最后交由引擎实现插桩要求。规则的 BNF 如下:

```
rules::=EMPTY | rule '\n' rules
rule::=property_rule | init_rule | control_rule
property_rule::=property ':' properties
properties::=EMPTY | property;properties
property=IDENTIFIER '(' TYPE ')'
init_rule::=INIT ':' ACTION
control_rule::=controls ':' ACTION
controls::=control | control '|' controls
control::=METHOD_ENTRY | METHOD_EXIT | METHOD_RETURN | IF | FOR | WHILE | BREAK | CONTINUE | FOR_BREAK | WHILE_BREAK | FOR_CONTINUE | WHILE_CONTINUE | SWITCH | CASE | SWITCH_BREAK
```

在上面的 BNF 中,IDENTIFIER 表示合法的 java 变量名,TYPE 表示合法的 java 类型名,ACTION 表示合法的 java 语句。

从上面的 BNF 可以看出,一共定义了 3 类规则:属性(property)、初始化(init)、控制结构规则。每条规则由两部分组成:第一部分为关键字,表明了是一条什么样的规则;另一部分是应用于该规则的必要动作。在关键字与动作之间必须用冒号隔开,每条规则之间必须用一个空行隔开。下面是几个实例:

程序员可以按如下方式定义一种属性:property:numOfMethodCalled(int),其中,property 是属性关键字,numOfMethodCalled 是属性名,int 是该属性的类型。

按如下方式定义初始化:init:numOfMethodCalled=0。这里,init 是初始化关键字,冒号后面为 java 代码,用来完成一些初始化工作。

控制结构规则应该如下定义:

```
METHOD_ENTRY:
System. out. println(type+". "+method+" is called at line "+line);
numOfMethodCalled++;

METHOD_RETURN|METHOD_EXIT:
System. out. println("The "+numOfMethodCalled+"the method("+method+") returned");
```

这两条一般规则集中体现了如何为一个控制结构书写一条规则。其中,METHOD_ENTRY,METHOD_RETURN,METHOD_EXIT 是一些与控制结构相关的关键字,每个关键字都代表一种控制结构。后面的动作就是在原程序中当遇到前面关键字所标示的控制结构时应该采取的动作,都必须是合法的 java 代码。不同的控制结构中一条规则的关键字之间可以通过"|"隔开,这是一种"或"的关系,表示这些关键字的动作都一样,并在后面给出。在这些控制结构规则的动

作(ACTION)中,可以使用 3 个预定义的参数:type(用来表示当前插入点所处于的类的全名,String 类型)、method(用来表示当前插入点所在的方法名,String 类型)、line(当前插入点在原程序中的行号,int 类型)。另外,对于 BREAK,FOR_BREAK,WHILE_BREAK,CONTINUE,FOR_CONTINUE,WHILE_CONTINUE,还有一个参数可以使用:label。该参数显示了 break 语句跳出或者 continue 继续循环的标签(如果没有标签,则为空)。

5.2.2 定义扩展

同样,本插桩工具也扩展了 instrument 扩展点,如下所示:

```
〈extension point="cn. edu. nju. seg. debug. assist. ui. engine. instrument"〉
  〈instrumenter
      class="cn. edu. nju. seg. debug. assist. ui. batchinstrument. BatchInstrumenter"
      marker_type="cn. edu. nju. seg. debug. assist. batchinstrument. marker"〉
      〈/instrumenter〉
  〈/extension〉
```

其中,BatchInstrumenter 实现了 IInstrumenter 接口。标记类型与前面提到的调试助手差不多,也是主要用来指示插桩点的。在插桩时,先调用 ConfigParser 的静态方法 parse,分析程序员(EU)提供的规则,将解析出来的信息存放在 BatchInstrumentConfig 的类实例中。BatchInstrumentConfig 采用了单件(singleton)模式,因为我们任一时刻只需要一个这样的实例,并且该实例需要在不同的方法和类之间共享。使用单件模式就避免了错综复杂的参数传递过程。BatchInstrumenter 还实现了 getUtilities 方法,通过该方法生成了一个工具类(Utilities)。在这个类里面,实现了插桩时在被插桩程序中需要用到的方法。在这个类中,所有的方法都是静态(static)方法,这是为了方便插桩程序的调用。在这个类里面,还有一个静态初始化区,用来满足扩展者(ED)的一些初始化需求。getUtilities 方法返回的内容将会被编译成一个 class 文件。

5.2.3 标签的生成

标签的生成过程就是程序员表达插桩意图的过程。为了能使 EU 表达插桩意图,本插件同样扩展了 popupMenus 扩展点,不过是以 IJavaElement 为 TargetID,即右击 Package Explorer 中的一个元素,在出现的菜单中选择编辑/删除插桩规则,如图 5 所示。

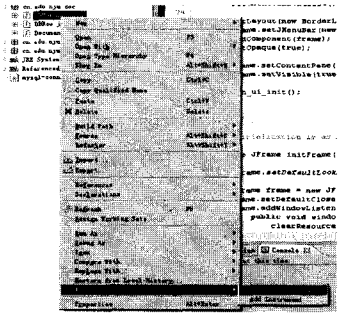

图 5 插桩示意图

可以对工程、包、java 文件、类、函数等进行插桩,可以定制插桩的粒度。但是,一次插桩动作使用的所有插桩规则都必须相同。根据程序员写好的规则,使用访问者(visitor)模式遍历 EU 所选的单元的抽象语法树(AST)。在遍历过程中增加(或删除)标记。

6 相关工作

对插桩工具的研究与开发一直是人们的关注点。对于 java 语言而言,插桩主要发生在对源代码和字节码之上,当然也有直接对 JVM 插桩的。BIT[7],BCEL[8],JIAPI[9] 等都是对字节码进行插桩的工具。我们采用的是对源代码进行插桩。Eclipse 插桩框架[10] 允许开发者收集用户在使用 Eclipse 时的一些配置信息,并周期性地将其发送到一个中央服务器。但这不是一个通用的框架,也不能收集程序员写的程序中数据。Instr[11] 是基于源代码插桩的库,也提供了很多有用的插桩方法,但是它没有集成到开发环境中,使用起来非常不方便。InsECTJ[12] 是一个基于 Eclipse 的通用插桩框架,它在源代码的基础上进行插桩,也被集成到了 Eclipse 开发环境中,可以扩展新的需要收集的信息。它是基于规则的,只能够在满足某个规则的地方插桩。它完成的功能相当于我们的基于规则的批量插桩工具。而我们的框架是基于代码行的,可以在没有任何规则的某一行上进行插桩,就像调试助手那样。

Log4J[13] 可以作为调试工具使用,同样它所生成的 debug 代码最终也不会被记录任何信息,但是它会在代码中充斥着很多业务无关代码。AspectJ[14] 作为 AOP 的先行者,已经在 AOP 方面做了很多的工作,它可以以 aspect 的方式向代码中加入日志,但是 AspectJ 为支持的某个方面(Aspect)生成的代码最终会被编译进发布程序。它不支持那些只出现在开发中而在发布时不需要也不能够出现的方面。

结束语 面对越来越复杂的软件逻辑与多线程等程序范型的出现,程序的调试、测试、验证等都面临着越来越多的挑战。传统技术的失效、程序员工作量的激增在插桩这一技术需求上表现得尤为明显。多线程范型的出现给调试技术带来了新的问题:如何在保证基本不影响原程序执行路径的前提下捕捉系统状态。原始的调试技术——插桩技术再次被用到了调试中。测试、验证技术的广泛使用使它们的支撑——插桩技术被运用得越来越多。

然而,随着分析、测试、验证以及调试等技术的广泛应用,传统的为每次插桩编制一个特殊的插桩程序的方式已不再适合,因为它们不能满足复用的想法。本文提出了一种无痕插桩的方法,并实现了一个插桩工具。这个工具的核心是一个插桩、编译、运行引擎,并在此引擎上实现了两个具体的实例,通过这两个实例介绍了如何利用插桩引擎实现不同目的的插桩需求,并展现了整个框架的结构。

现阶段,我们开发的这个插桩引擎原型还只支持 java application 程序。以后的工作可以增加对 java applet,甚至是 Eclipse 插件的支持。除了 java 语言之外,还可以增加对 c++ 的开发环境包 cdt 也提供相应的插桩支持。在基于规则的批量插桩工具中,还需要进一步做的工作有:对不同的单元使用不同的规则、支持基于数据流的规则,还可以把 InsECTJ 中的规则引进来。

参 考 文 献

[1] 戴清涵,李宣东,赵建华,等. 面向设计流图的代码支撑工具[J]. 计算机科学,2005(11):203-206

(下转第 169 页)

的算法要变换的只是那些含有某个 I_{sc} 的非空子集的数据项，所以完全可以在第(3)步中第一次扫描时用一些存储空间来存储需要做变换的项的 TID 即可。但是这样做虽然节省了时间，却占用了更多空间。对于海量的数据仓库，在变换时可能空间需求会更难满足。但是当设备十分先进时，可能更看重的是时间因素。所以，可以针对实际情况灵活运用该方法。

此外，该算法并非十全十美。注意到在 Procedure_Transformation 子过程中，步骤(16)的表述为"find the nearest j that satisfies $i<j$ and $T_j \cap \{I_1, I_2, \cdots, I_k\} = \emptyset$;"，事实上，这句话所假设的前提有可能为假，也就是说事物数据库 D 中已经不存在相对于 $I_1 \cup I_{sc}$ 的零事务了。这种情况当然有可能发生，但是我们在第 2 节关于数据项闭包的定义已经指出，由于是大型数据库，因此存在着海量的数据，相对于某一类型的商品来说，不存在相对于它们的零事务的概率是微乎其微的。但是一旦这种情况真的发生了，可能要进行更加复杂的变换，而这也是进一步要研究的课题。

结束语 本文提出的基于数据项闭包的保密数据挖掘方法，能够有效解决这样的问题：(1)数据提供方不完全信任挖掘请求方，但是又希望通过提供变换过的数据库给对方挖掘来获得一些利益；(2)明确对方在挖掘过程中使用的是类 Apriori 算法，且已方为水平格式(horizontal data format)的事务数据库。与此同时，基于数据项闭包这一思想是否可以应用在更广阔的数据挖掘领域，仍未可知。比如，能否对垂直数据格式的数据库进行闭包变换，能否对序列模式数据库进行闭包变换[17,18]等，都是有待解决的问题。另一方面，从该算法能够针对的挖掘方法来说，是否也能通过该算法针对其他数据挖掘算法(如 FP 增长算法)来进行保护呢？同样没有答案，而新的发现有待进一步研究。

参 考 文 献

[1] Agrawal R, Imieliński T, et al. Mining asso-ciation rules between sets of items in large databases[C]// Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. Washion D. C., USA, May 1993: 207-216

[2] Agrawal R, Srikant R. Fast Algorithms for Mining Association Rules[C]// Proceedings of the 20th International Conference on Very Large Databases. Santiago, Chile, Sept. 1994: 487-499

[3] Han Jia-wei, Micheline K. 数据挖掘：概念与技术(第二版)[M]. 范明，孟小峰，译. 北京：机械工业出版社，2007

[4] Tavani H T. Information privacy, data mining, and the internet [J]. Ethics and Information Technology, 1999, 1(2): 137-145

[5] Lindell Y, Pinkas B. Privacy preserving data mining[J]. Journal of Cryptology, 2002, 15(3): 177-206

[6] Agrawal R, Evfimievski A, Srikant R. Information sharing across private databases[C]// Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2003: 86-97

[7] Agrawal R, Srikant R. Privacy-preserving data mining [C]// Proceedings of the 2000 ACM SIGMOD Conference on Management of Data. 2000: 439-450

[8] Kantarcioglu M, Clifton C. Privacy-preserving distributed mining of association rules on horizontally partitioned data[J]. IEEE Transactions on Knowledge and Data Engineering, 2004, 16(9): 1026-1037

[9] Du Wen-liang, Zhan Zhi-jun. Using randomized response techniques for privacy-preserving data mining[C]// The 9th ACM SIGKDD Int'l Conf Knowledge Discovery in Databases and Data Mining. Washington D. C., USA, August 2003: 24-27

[10] 葛伟平，汪卫，周皓峰，等. 基于隐私保护的分类挖掘[J]. 计算机研究与发展，2006，43(01)：39-45

[11] 罗永龙，黄刘生，等. 一个保护私有信息的布尔关联规则挖掘算法[J]. 电子学报，2005，33(5)：900-903

[12] 张锋，常会友. 基于分布式数据的隐私保持协同过滤推荐研究[J]. 计算机学报，2006，29(8)：1487-1495

[13] 周水庚，李丰，陶宇飞，等. 面向数据库应用的隐私保护研究综述[J]. 计算机学报，2009，32(5)：847-861

[14] Chris C, Donald M. Security and privacy implications of data mining[C]// Proceedings of the ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery. 1996: 15-19

[15] Brin S, Motwani R, Silverstein C. Beyond Market Baskets: Generalizing Association Rules to Correlations[C]// Proceedings of SIGMOD'97. AZ, June 1997: 265-276

[16] Srikant R, Vu Q, Agrawal R. Mining association rules with item constraints[C]// KDD-97. 1997: 67-73

[17] Aggarwal C C, Yu P S. Privacy-preserving Data Mining: Models and Algorithms[M]. New York: Springer-Verlag, 2008

[18] 王虎，丁世飞. 序列模式挖掘研究与发展[J]. 计算机科学，2009，36(12)：14-17

(上接第 143 页)

[2] Jeffrey D, Gupta N. Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction[J]. IEEE Transactions on Software Engineering, 2007, 33(2): 108-123

[3] Beydeda S, Gruhn V. BINTEST-Binary Search-based Test Case Generation[C]// Proceedings of the 27th Annual International Conference on Computer Software and Applications. IEEE Computer Society, 2003

[4] Kiczales G, et al. Aspect-oriented Programming [C]// Proceedings of the European Conference on Object-oriented Programming. Finland: Springer-Verlag, 1997

[5] vim: http://www.vim.org/

[6] emacs: http://www.gnu.org/software/emacs/

[7] Lee H B, Zorn B G. BIT: A Tool for Instrumenting Java Bytecodes

[8] BCEL: The Byte Code Engineering Library[EB/OL]. http://jakarta.apache.org/bcel/

[9] Java Instrumentation API(JIAPI) [EB/OL]. http://jiapi.sourceforge.net/

[10] Eclipse Instrumentation Framework [EB/OL]. http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-ui-home/instrumentation/index.html

[11] Java Source Code Instrumentation[EB/OL]. http://www.glenmccl.com/instr/instr.htm

[12] Seesing A, Orso A. InsECTJ: a generic instrumentation framework for collecting dynamic information within Eclipse[C]// ETX. 2005: 45-49

[13] Log4J project[EB/OL]. http://logging.apache.org/log4j/1.2/index.html

[14] Kiczales G, Hilsdale E, Hugunin J, et al. An Overview of AspectJ[C]// Proceedings of the 15th European Conference on Object-oriented Programming. London, UK: Springer-Verlag, 2001: 327-353