[19]中华人民共和国国家知识产权局

[51]Int.Cl[7]
G06F 9/44
G06F 1/00

[12]发明专利申请公开说明书

[21]申请号 98806038.8

[43]公开日 2000年7月12日

[11]公开号 CN 1260055A

[22]申请日 1998.6.9 [21]申请号 98806038.8
[30]优先权
[32]1997.6.9 [33]NZ [31]328057
[86]国际申请 PCT/US98/12017 1998.6.9
[87]国际公布 WO99/01815 英 1999.1.14
[85]进入国家阶段日期 1999.12.9
[71]申请人 联信公司
地址 美国加利福尼亚州
[72]发明人 克里斯琴·司温·科尔伯格
克拉克·戴维·汤伯森
道格拉斯·崴·科克·卢
[74]专利代理机构 永新专利商标代理有限公司
代理人 程 伟 余 刚

权利要求书4页 说明书66页 附图页数27页

[54]发明名称 用于提高软件安全性的模糊技术

[57]摘要

本发明提供用于提高软件安全性的模糊技术。在一个实施例 中,一种用于提高软件安全性的模糊技术的方法包括选择代码的 一个子集(例如,应用程序的编译后的源码)来实施模糊,以及 对代码的选中子集进行模糊。模糊包括对选中代码的子集进行模 糊转换。转换后的代码与未转换前的代码有很少的相同之处。应 用转换可以基于要求的安全等级(例如防止反向工程)进行。应用转换还可以包括一种能使用模糊构架产生的控制转换,后者可 以通过使用别名和并发技术构建。因此,代码可以基于要求的安 全等级(例如,基于要求的有效性、弹性和费用),为提高软件的 安全性而进行模糊。

ISSN1008-4274

权 利 要 求 书

1. 一种用于对代码进行模糊的计算机执行方法，其特征包括：

选择待模糊代码的子集；

选择应用的模糊转换；以及

应用转换，其中转换后代码与转换前代码有较弱的相似性。

2. 根据权利要求 1 所述的计算机执行方法，其特征还包括：

识别与源代码对应的一个或多个源代码输入文件，作为待处理的应用程序代码；

选择要求的模糊等级（有效性）；

选择最大执行时间或空间花费（开销）；

阅读并分析输入文件；

提供信息用于识别待处理程序使用的数据类型，数据结构和控制结构；

为源代码对象选择和应用模糊转换，直到获得要求的有效性或超过最大开销；以及

输出应用程序的转换后代码。

3. 根据权利要求 1 所述的方法，其中所述转换包括一个模糊构架，所述模糊构架由别名和并发技术构建。

4. 根据权利要求 1 所述的方法，其特征还包括：

输出应用到模糊后代码上的模糊转换的信息，以及将转换后程序的模糊代码与应用程序的源码联系起来的信息。

5. 根据权利要求 1 所述的方法，其中所述转换被选择用于保存应用程序代码的可观察行为。

6. 根据权利要求 1 所述的方法，其特征还包括：

对代码进行反模糊，代码的反模糊包括，通过使用分片、部分评估、数据流分析或统计分析，从应用程序的模糊后代码中删除所有模糊。

7. 一种在计算机可读媒体中体现，用于对代码进行模糊的计算机程序，其特征在于包括：

选择待模糊代码的子集的逻辑；

选择要应用的模糊转换的逻辑；以及

应用转换的逻辑，其中转换后代码与转换前代码有较弱的相似性。

8. 根据权利要求 7 所述的计算机程序，其特征还包括：

识别与源代码对应的一个或多个源代码输入文件，作为待处理的应用程序代码的逻辑；

选择要求的模糊等级（有效性）的逻辑；

选择最大执行时间或空间花费（开销）的逻辑；

阅读并分析输入文件的链接；

提供信息用于识别待处理程序使用的数据类型、数据结构和控制结构的逻辑；

为源代码对象选择和应用模糊转换，直到获得要求的有效性或超过最大开销的逻辑；及

输出应用程序的转换后代码的逻辑。

9. 根据权利要求 7 所述的计算机程序，其中所述转换包括一个模糊构架，所述模糊构架由别名和并发技术构建。

10. 根据权利要求 7 所述的计算机程序，其特征还包括：

输出应用到模糊后代码上的模糊转换的信息，以及将转换后程序的模糊代码与应用程序的源码联系起来的信息的逻辑。

11. 根据权利要求 7 所述的计算机程序，其中所述转换被选择用于保存应用程序代码的可观察行为。

12. 根据权利要求 7 所述的计算机程序，其特征还包括：

对代码进行反模糊的逻辑，代码的反模糊包括，通过使用分片、部分评估、数据流分析或统计分析，从应用程序的模糊后代码中删除所有模糊。

13. 一种用于对代码进行模糊的设备，其特征在于包括：

选择待模糊代码的子集的设备；

选择要应用的模糊转换的设备；及

应用转换的设备，其中转换后代码与转换前代码有较弱的相似性。

14. 根据权利要求 13 所述的设备，其特征还包括：

识别与源代码对应的一个或多个源代码输入文件，作为待处理的应用程序代码的设备；

选择要求的模糊等级（有效性）的设备；

选择最大执行时间或空间花费（开销）的设备；

阅读并分析输入文件的设备；

提供信息用于识别待处理程序使用的数据类型，数据结构和控制结构的设备；

为源代码对象选择和应用模糊转换，直到获得要求的有效性或超过最大开销的设备；及

输出应用程序的转换后代码的设备。

15. 根据权利要求 13 所述的设备，其中所述转换包括一个模糊构架，所述模糊构架由别名和并发技术构建。

16. 根据权利要求 13 所述的设备，其特征还包括：

输出应用到模糊后代码上的模糊转换的信息，以及将转换后程序的模糊代码与应用程序的源码联系起来的信息的设备。

17. 根据权利要求 13 所述的设备，其中所述转换被选择用于保存应用程序代码的可观察行为。

18. 根据权利要求 13 所述的设备，其特征还包括：

对代码进行反模糊的设备，代码的反模糊包括，通过使用分片、部分评估、数据流分析或统计分析，从应用程序的模糊后代码中删除所有模糊。

19. 根据权利要求 13 所述的设备，其中所述代码由 Java™字节码组成。

20. 根据权利要求 13 所述的设备，其中所述转换提供了一种数据模糊，一种控制模糊，或一种预防性模糊。

说　明　书

用于提高软件安全性的模糊技术

本发明所属技术领域

本发明涉及能防止，或至少能够阻碍对软件的翻译、解码或反向工程的方法和设备。特别地，本发明尽管并非唯一，但涉及通过插入、删除或重组软件的可识别结构或信息，来提高软件的结构和逻辑复杂性，从而增加反编译或反向工程的难度。

本发明背景技术

软件的性质决定了它易被第三方分析和拷贝。人们已经付出了诸多努力来提高软件的安全性，也获得许多成功。这些安全性方面的考虑涉及对防止软件未授权拷贝的需要和隐藏编程技巧的要求，这些技巧本来是可以通过反向工程来发现的。

确立的合法收入，例如版权，提供了一种法律上的保护方式。然而，依法维持在这些体制下建立的合法权益可能是昂贵而费时的。进一步，版权中对软件的保护并没有包括编程技巧。这些技巧（例如，与软件形式相对的功能），很难在法律上予以保护。反向工程就可以通过基于有争议软件的功能的细节知识，从头开始重新编写相关软件，来避开侵犯版权的问题。这些知识可以通过分析数据结构、对代码进行抽象和组织来获得。

软件专利提供了更全面的保护。然而，将软件的法律保护和技术保护结合起来，将会更加有利。

以往保护私有软件的方法或是使用基于加密的硬件解决方案，或是基于源码结构的简单重组。基于硬件的技术是不理想的，因为它们通常很昂贵，并受限于特定的平台或硬件条件。软件解决方案典型地包括细节代码模糊软件，例如用于 Java™的 Crema 模糊软件。一些模糊软件定位于应用程序的词汇结构，典型地删除源码格式和注释，以及对变量重命名。然而，这样的一种模糊技巧并不能提供对恶意反向工程的足够保护：反向工程是一种于软件分发形式无关的问题。进一步说，当软件以独立于硬件的格式分发时，该问题会变得越来越严重，因为这些格式保留了源代码中的许多或全部信息。这些格式的例子有 Java™字节码和体系结构中立分发格式（ANDF）。

软件开发代表着投入的巨大时间和精力，以及编程者的技巧。在商业的环境中，防止竞争对手照搬私有技巧的能力可能是至关重要的。

本发明概要

本发明提供了用于提高软件安全性的模糊技巧的方法和设备，例如计算机执行的降低软件受反向工程的影响程度的方法（或是为公众提供实用的选择）。在一个实施例中，计算机执行的模糊代码的方法，包括测试对代码提供一次或多次模糊转换的过程是否完成，选择代码的子集进行模糊，选择要应用的模糊转换，应用转换，以及返回完成测试步骤。

在可选实施例中，本发明涉及控制计算机的方法，使得该计算机上运行和存储或者控制的软件显示出防反向工程的预定和控制的程度，包括将选定的模糊转换应用到软件的选定部分。其中某一等级的模糊的获得要通过使用选定的模糊转换来提供要求的

防反向工程程度，软件运行的有效性，以及转换后软件的大小，并更新软件以反映出模糊转换。

在优选实施例中，本发明提供了计算机执行的提高软件安全性的方法，包括确定一个或多个对应于待处理应用程序的源软件的源码输入文件，选择要求的模糊等级（例如有效性），选择最大的执行时间或空间限制（例如费用），读取并分析输入文件，是否带有可以被源码直接或间接读取的任何库或补充文件是可选的，提供用于确定被待处理应用程序使用的数据类型、数据结构以及控制结构的信息，构建存储信息的适当的表，预处理应用程序的信息，对应于预处理的步骤，对源码对象选择和应用模糊代码转换，重复模糊代码转换步骤，直到获得要求的有效性或者超过最大开销，然后输出转换后的软件。

优选地，应用程序的信息是通过使用多种静态分析技术和动态分析技术获得的。静态分析技术包括过程间的数据流分析和数据依存分析。动态分析技术包括建立配置文件，可选的是通过用户获取信息。配置文件可以用来确定模糊的等级，后者被用于某一具体的源码对象。转换可以包括使用模糊构架建立的控制转换，其中模糊构架可以是任何的数学对象，该对象被执行时并不费力，用模糊软件建立起来也很简单，而反模糊工具要破解则很困难。优选地，可以使用别名和并发技术构建模糊构架。源程序的信息也可以通过实际的分析获得，后者确定了应用程序包含的语言构架和编程惯用语法的自然属性。

模糊转换的有效性可以通过软件复杂度的标准来估价。模糊代码转换可以应用在任何语言构架：例如模块、类或是被分割或合并的子例程；可以产生新的控制和数据结构；可以修改源控制和数据结构。优选地，加在转换后应用程序的新构架被选择与源应用程序中的那些尽量相似，基于预处理时收集到的实用信息。

这种方法可以产生附加文件，后者包含与使用了哪种模糊转换有关的信息，以及将转换后应用程序的模糊代码与源软件联系起来的信息。

优选地，选择模糊转换是为了保存软件的可观察行为，使得如果 P 是未转换的软件，而 P'是转换后的软件的话，P 和 P'有同样的可观察行为。特别地，如果 P 不能终止运行或者不能在错误发生时终止运行，那么 P'可以终止也可以不终止，否则 P'终止并产生与 P 相同的输出。可观察行为包括用户可以看到的效果，但是 P 和 P'在运行的时候，用户不能观察到它们的不同细节行为。例如，P 和 P'的不同细节行为包括文件创建、内存使用和网络通信。

在一个实施例中，本发明还提供反模糊工具，可以通过使用分片、部分估价、数据流分析或统计分析，从模糊后的应用程序中删除模糊。

附图的简要介绍

下面将仅根据实例和介绍下列附图，对本发明进行描述：

图 1 显示了一种依据本发明的教学的数据处理系统；

图 2 显示了一种包括模糊转换的类别的软件保护分类；

图 3a 和图 3b 显示了通过服务器端执行(a)和部分服务器端执行(b)，来提供软件安全性的技术；

图 4a 和图 4b 显示了通过使用加密(a)和使用带符号本地码(b)来提供软件安全性的技术；

图 5 显示的是通过模糊来提供软件安全性的技术;

图 6 显示的是适于和 Java™应用程序一起使用的模糊工具的例子的结构;

图 7 显示的是列出已知软件复杂性度量的表;

图 8a 和图 8b 显示的是模糊转换的弹性;

图 9 显示的是不同类型的模糊谓词;

图 10a 和图 10b 提供的是琐碎模糊构架(a)和弱模糊构架(b)的实例;

图 11 显示的是计算转换的一个实例（分支插入转换);

图 12a 到图 12d 显示的是一种循环条件插入转换;

图 13 显示的是将可缩减流图转换成非缩减流图的转换过程;

图 14 显示的是如果一段代码不包含数据依靠，它可以被并行处理;

图 15 显示的是通过插入适当的同步原型，一段不包含数据从属的代码可以被分裂为并发的线程;

图 16 显示的是过程 P 和 Q 如何在它们的调用站点嵌入，并从代码中删除;

图 17 显示的是嵌入方法的调用;

图 18 显示的是使同一类中声明的两种方法进行交互的技术;

图 19 显示的是通过对源代码应用不同的模糊转换集，从而产生一个方法的多个不同版本的技术；

图 20a 到图 20c 提供的是循环转换的例子，包括(a)循环阻塞，(b)循环反卷和(c)循环分裂；

图 21 显示的是变量分裂的实例；

图 22 提供了一种被构建用来模糊串“AAA”、“BAAAA”和“CCB”的函数；

图 23 显示的是将 32 位变量 x 和 y 合并成一个 64 位变量 z 的实例；

图 24 显示的是用于数组重构造的数据转换实例；

图 25 显示了对一个继承分层结构的修正；

图 26 显示了从对象和别名构建而成的模糊谓词；

图 27 提供了一个使用线程的模糊构架；

图 28a 到 28d 显示的是模糊与反模糊的对比，其中(a)显示的是包含三个被模糊的语句的源程序 S_{1-3}，(b)显示的是确定“常量”模糊谓词的反模糊工具；(c)显示的是确定语句中公用代码的反模糊工具，(d)显示的是应用某些最终简化和将程序返回为原始形式的反模糊工具；

图 29 显示的是 Java™反模糊工具的体系结构；

图 30 显示的是用于评估的统计分析实例；

图 31a 和 31b 提供了多种模糊转换的概述表；

图 32 提供了对多种模糊构架的概述。

本发明的详细描述

下列描述在 Java™模糊工具的环境中给出，这些工具目前正由申请人在开发。然而，对于本技术领域内的普通技术人员来说，很明显地，本技术也适用于其它编程工具，而且本发明并不能被解释成只适用于 Java™应用程序。本发明在其它编程语言中的实施被认为在有一般技术人员的技能范围之内。为进行清晰的说明，下文的实施例特别针对 Java™的模糊工具。

在下文的描述中，将使用下列指代术语。P 指代模糊的输入应用程序；P'指转换后的应用程序；T 指转换过程，即 T 将 P 转换为 P'。如果 P 和 P'有相同的可观察行为，则 P(T)P'是一种模糊转换。一般将可观察行为定义为用户所经历的行为。那么，P'也可能会有一些例如创建文件之类的附加效果，这些是 P 所不具备的，条件是这些附加效果不会被用户所经历。P 和 P'不必有完全的效率。

示例硬件

图 1 显示的是一个依据本发明教学的数据处理系统。图 1 显示的计算机 100，它包括 3 个主要元件。计算机 100 包括一个输入输出（I/O）电路 120，用于以适当结构化的形式与计算机 100 的其它部分来回交换信息。计算机 100 包括一个与输入输出电路 120 通信的控制处理器（CPU）130 以及一个内存 140（例如挥发性或者非挥发性内存）。这些元件是在最普通的计算机中典型能发现的那些，而且事实上计算机 100 被用于代表数据处理设备的一个大

的分类。显示的光栅显示器 160 与输入输出电路 120 通信，并用于显示 CPU130 产生的图像。任何广为人知的阴极射线管（CRT）显示器或其它形式的显示器都可以被用作显示器 160。显示与输入输出端口 120 的通信中的是传统键盘 150。计算机 100 是较大系统中的一部分，这也会被任何本行业内普通技术人员所赞赏。例如，计算机 100 也能与一个网络进行通信（例如，与局域网（LAN）相连接）。

特别地，计算机 100 可以包括根据本发明教学所示能提高软件安全性的模糊电路，或者会被任何本行业内普通技术人员所赞赏的是，本发明能够在计算机 100 所执行的软件中实施（例如，软件可以存储在内存 140 中并在 CPU130 上执行）。例如，依据本发明的实施例，一个存储在内存 140 中的被反模糊了的程序 P（例如一个应用程序），能够被在 CPU130 上执行的模糊软件模糊，以提供模糊后的程序 P'，并储存在内存 140 中。

细节描述

图 6 显示的是 Java™模糊软件的体系结构。依据发明方法，Java™应用程序类文件与任何库文件一起传递。构建继承树的同时也构建符号表，为所有的符号提供类型信息，为所有的方法提供控制流图表。用户可以有选择地提供 Java™概要工具所产生的概要数据文件。这个信息可以用于指导模糊软件，以确保应用程序经常执行的部分不会被非常昂贵的转换所模糊。使用标准编译器技术，例如过程间数据流分析和数据附加分析，来收集信息。某些信息可以由用户提供，有些由专门的技术提供。这些信息用于选择和应用适当的代码转换。

选择适当的转换。选择最合适转换的首要标准包括要求选中的转换与代码剩余部分能自然地融和。这可以通过赋予转换以较

高的合适值来实现。进一步的要求在于，应该优先选择以较少的执行时间代价产生高级别的模糊的转换。后一点可以通过选择使有效性和弹性最大化，而使开销最小的转换来实现。

模糊的优先级被分配给源代码对象。这会反映出对源代码对象的内容进行模糊是多么重要。例如，如果某个源代码对象包含高敏感性的私有资料，则它的模糊优先级会较高。执行时间的级别由方法决定，如果执行该方法的时间比执行其它方法需要的长，则等于1。

然后，可以通过建立合适的内部数据结构、从每一个源代码对象映射到合适的转换、模糊优先级和执行时间级别上来进行应用程序的模糊。模糊转换被应用，直到达到要求的模糊级别或者超过允许最大的执行时间开销。然后写出转换后的应用程序。

模糊工具的输出是一个新的应用程序，其功能与原来的相同。该工具还可以产生 Java™源文件，带有的注释信息涉及应用了哪种转换以及模糊后的代码如何与源应用程序联系。

下面将要描述一些模糊转换的实例，它们也是在 Java™模糊软件下工作。

模糊转换可以被评估，并根据其质量进行分类。转换的质量根据其有效性、弹性以及开销来表达。转换的有效性与 P'与 P 之间的模糊关系相关。任何一种尺度都会相对模糊，因为它必然依赖于人的认知能力。对本发明来说，将转换的有效性考虑成一种转换可用性的标准就足够了。转换的弹性衡量的是一种转换对抗来自自动反模糊软件的攻击的能力。这是两个因素的组合：程序员的努力和反模糊软件的努力。对弹性进行测量时其比例可以从琐碎到单向。单向转换是极端的，因为它们不能被逆转。第三个

组件是转换的执行开销。这是作为使用转换后应用程序 P'的结果而导致的执行时间和空间开销。转换评估的进一步细节在下文对优选实施例的具体描述中讨论。模糊转换的主要分类在图 2c 中显示，其细节在图 2e 到 2g 中给出。

模糊转换的例子如下所示：模糊转换可以分为以下类别：控制模糊，数据模糊，布局模糊和预防性模糊。下文将讨论这些类别的一些实例。

控制模糊包括聚集转换、排序转换以及计算转换。

计算转换包括：将实际的控制流隐藏在不相关的非功能性语句中；在对象代码级别引入代码序列，因此不会存在相应的高级别语言构架；以及删去实际的控制流抽象或是引入虚假的控制流抽象。

考虑第一个分类（控制流），Cyclomatic 和 Nesing 复杂性量度表示，在一段代码的可见复杂度和它包含的谓词数目之间有很强的联系。模糊谓词使得转换的构建成为可能，后者向程序里引入了新的谓词。

参见图 11a，一个模糊谓词 P^T 词被插入基本块 S 中，其中 S = $S_1 \cdots Sn$。这将 S 平分为两半。谓词 P^T 是无关代码，因为它一直都估计为真。在图 11b 中，S 而被平分为两半，然后被转换到两个不同的模糊后的版本 S^a 和 S^b 中。因此对于反向工程师来说，S^a 和 S^b 都执行相同的功能就不明显了。图 11c 与图 11b 类似，然而，在 S^b 中引入了一个缺陷。P^T 谓词一直选择正确的代码版本 S^a。

另一种类型的模糊转换是数据转换。数据转换的一个例子是对数组进行反构建，以提高代码的复杂性。一个数组可以被分裂

为多个子数组，两个或多个子数组被组合成一个数组，或是增加（展平）和减少（折叠）一个数组的维数。图 24 显示的是多个数据转换的例子。在语句（1－2）中，数组 A 被分为两个子数组 A1 和 A2。A1 包括带有偶数索引的元件，A2 包括带有奇数索引的元件。声明（3－4）显示的是两个整数数组 B 和 C 是如何被交叉以产生数组 BC 的。来自 B 和 C 的元件可以被均匀扩展到转换后的数组中去。声明（6－7）显示的是将数组 D 折叠成数组 D1。这样的转换引入了以前没有的数据结构或者删除现存的数据结构。这就可以极大地增加程序的模糊性，因为例如在声明一个 2 维数组的时候，一个程序员通常这样做以达到某一目的，而选中的数据结构被映射到相应的数据上去。如果该数组被折叠成为一个 1 维结构，反向工程师就失去了有价值的实用信息。

模糊转换的另一个例子是预防性的转换。与控制转换或者数据转换相对的是，预防性转换的主要目的并非使程序对人类阅读者显得模糊，而是使已知的反模糊技术变得更加困难或者利用现存反模糊软件或反编译器中的已知问题。这些转换分别被人们认为为内在的和要达到的。一个内在预防性转换的例子是对 for-loop 循环重排序以反向运行。如果循环中没有循环携带的数据附件，则这样的重排序是可能的。一个反模糊软件可以执行相同的分析并重新为循环排序，使其向前执行。然而，如果一个向反向的循环中加入假数据附件，就可以阻止对循环的识别和重排序。

下文中将在优选实施例的细节描述中进一步讨论模糊转换的特定实例。

优选实施例的详细描述

以带有原始源文件中存在的多数或全部信息来发布软件的方式已变得越来越普遍。一个重要的例子是 Java 的字节码。因为这

些代码很容易被反编译，所以它们增加了恶意反向工程攻击的危险。

因此，依据本发明的一个实施例，提供了多种对软件秘密进行技术保护的技巧。在优选实施例的详细介绍中，我们认为，目前自动代码模糊是防止反向工程的最有生命力的方法。下面我们描述代码模糊器的设计，这是一种将程序转变成为另一种等同代码的工具，后者会更加难以理解和进行反向工程。模糊软件是基于对代码转换的应用，在许多实例中与那些编译优化器所使用的很相似。我们描述许多这样的转换，对它们分类，并用它们的有效性（例如，人类阅读者被迷惑到何种程度）、弹性（例如，抵制自动反模糊攻击的效果如何）和开销（对应用程序会增加多少性能上的负担）来评价它们。

最后，我们描述多种反模糊技巧（例如程序分片）和模糊工具可能用来对前者进行对抗的措施。

1.介绍

只要有足够的时间、努力和决心，一个能干的程序员肯定能对任何程序实施反向工程。只要对应用程序能进行物理上的访问，一个反向工程师就可以反编译（使用反汇编或反编译器），然后分析其数据结构和控制流。这可以通过手工来实现或者是借助反向工程的工具，例如程序分片器。然而，反向工程直到最近才成为一个问题，这个问题没有被软件的开发者所重视，因为大多数程序很大、集成、发布时是拆卸后的本地码，从而使其对反向工程师来说很困难（尽管从来都不是不可能）。

然而，随着软件的发布形式使反编译和方向工程变得越来越容易，这种情况正在改变。重要的例子包括 Java 字节代码和体系

结构中立分发格式（ANDF）。特别是 Java 应用程序对软件开发人员提出了这个问题。它们是通过 Internet 以 Java 类文件的方式来发布的，是一种与硬件无关的虚拟机代码，虚拟地携带有原 Java 源的所有信息。这样，这些类文件很容易被反编译。进一步说，因为多数计算发生在标准库中，Java 程序通常很小，因此对反向工程师来说更加相对容易一些。

Java 开发人员主要关心的问题并非对整个应用程序进行直接的重新应用。这种行动带来的价值相对较小，因为这显然违反了版权法[29]，也很容易通过法律手段解决。反而，开发人员最害怕的是竞争对手能够从它们的程序中获取私有算法和数据结构，从而将它们集成到自己的程序中去。这不仅会为竞争对手带来商业上的优势（减少了开发时间和费用），而且这在法律上也是难以发现和追踪的。最后一点对小的开发者最有意义，因为他们承担不起与法律预算没有限制的大公司[22]之间旷日持久的法律诉讼。

图 2 中显示的是为软件提供法律保护或安全性的多种形式保护的概述。图 2 提供了(a)防止恶意反向工程的方法的分类，(b)模糊转换的性质，(c)模糊转换针对的信息，(d)布局模糊，(e)数据模糊，(f)控制模糊，以及(g)预防性模糊。

下文讨论的是软件开发人员可以使用的，对知识产权进行技术保护的多种形式。尽管我们的大多数结论可以应用于其它语言和体系结构中立的格式上，我们将讨论的范围限制于在 Internet 上以 Java 类文件的形式发布的 Java 程序。这对熟悉普通技术的人来说是非常明了的。我们认为，活动代码保护的唯一合理方法是代码模糊。我们将进一步提出多种模糊转换，并根据有效性和效率对其进行分类，然后显示如何在自动的模糊工具中使用它们。

优选实施例的其它详细描述在下文中结构化给出。在第 2 部分，我们给出了防止软件偷窃行为的多种形式的技术保护概述，我们认为，现在的代码模糊可以起到最经济的预防作用。在第 3 部分，我们给出了对 Kava 设计的简要概述，这是一种目前正在构建中的 Java 代码模糊软件。第 4 和第 5 部分描述了我们用来分类和评估不同类型模糊转换的标准。第 6，7，8，9 部分给出了模糊转换的目录。在第 10 部分，我们给出了更加详细的模糊算法。在第 11 部分，我们对我们的结果进行总结，对代码模糊的未来发展方向进行讨论来作出结论。

2. 保护知识产权

考虑下列场景。爱丽丝是一个小软件的开发商，她想通过 Internet 以一定的收费向用户发布应用程序。鲍勃是其竞争对手，他知道如果自己了解了爱丽丝的程序中的关键算法和数据结构，就会比爱丽丝更加具有商业上的优势。

这可以被看作是两个对手之间的双人游戏：软件开发商（爱丽丝）试图保护自己的代码免受攻击，而反向工程师（鲍勃）则要分析该程序并将其转化为易读和易懂的形式。注意对鲍勃来说，并非一定要将该程序转化为与爱丽丝的源代码相近的形式；必须完成的是反向工程得出的代码必须能被鲍勃和他的程序员读懂。还需要注意的是，爱丽丝并非要保护整个应用程序免受鲍勃的攻击；该程序中的大部分内容可能都是竞争者毫无兴趣的“面包——黄油代码”。

爱丽丝能使用法律或者技术的手段来保护代码免受鲍勃的攻击，如图 2a 所示，这些在上文中已经讨论过。尽管版权法确实涵盖了软件制品，经济的现实使得爱丽丝拥有的这类小公司很难与强大的竞争对手对簿公堂。对爱丽丝来说，更加有吸引力的解决

方案是，通过使方向工程在技术上变得困难，几乎达到不可能或在经济上不划算的程度，从而保护她的代码。Gosler 描述了许多早期技术上的保护尝试。（James R Gosler. 软件保护：神化还是现实？在 CRYPTO'85——加密学的进展，140—157 页，1985 年 8 月）。

对爱丽丝来说，最安全的方法是根本不销售软件，而是销售其服务。换句话说，用户并不能获得该应用程序本身，而是连接到爱丽丝的站点，如图 3a 远程运行该程序，每次付少量的电子货币。爱丽丝获得的好处在于鲍勃永远不能从物理上获得该应用程序，从而也不能对其进行反向工程。当然，其缺点在于由于网络带宽和其它潜在因素，应用程序执行时比在用户本地站点上执行的效果要差得多。部分的解决方案是将程序分为两个部分：在用户本地站点上执行的公共部分，以及远程运行的私有部分（包含爱丽丝希望保护的算法），例如图 3b 中所示。

爱丽丝可以采取的另一种方法是在将代码发送到用户之前对其进行加密，如图 4a 所示。不幸的是，这只有在整个加密/执行过程在硬件中进行的情况下才有效。Herzberg（Amir herzberg 和 Shlomit S. Pinter. 软件的公共保护。 计算机系统上 ACM 交易，5(4): 371—393，1987 年 11 月）和 Wilhelm（Uwe G. Wilhelm. 密码学上的保护对象。 http://lsewww.epfl.ch/~wilhelm/CryPO.html, 1997 年）描述了这样的系统。如果代码在软件中由虚拟机解释器执行（用 Java 字节码时情况经常是这样），那么鲍勃就一直有可能拦截和反编译加密过的代码。

Java™编程语言已经非常流行，主要是因为它的体系结构中立字节码。虽然这明显地简化了移动代码，但与本地代码相比它确实降低了数量级的性能。可以预测，这会导致对即时编译器的开发，后者能自由地将 Java 字节码翻译成本地代码。爱丽丝可以使

用这样的翻译器来为所有流行的平台产生其应用程序的本地代码版本。下载这些应用程序时，用户的站点将需要确认自己正在运行的体系结构/操作系统的组合，然后相应的版本将会被传输，例如图 4b 所示。只能获得本地代码会使鲍勃的工作变得更加困难，尽管该工作并非完全不可能。传输本地代码还会带来进一步的复杂化。问题在于——与在执行前能进行字节代码验证的 Java 字节代码不同，本地代码在用户的机器上运行时不能保证其完全的安全性。如果爱丽丝是社区中的一个信任成员，用户可以接受她的保证，即应用程序绝不会在用户端做任何坏事。为了确保没有人会试图破坏该应用程序，爱丽丝将不得不在代码传输的时候，以数字方式为代码加上符号，从而向用户证明该代码是她所写的原始代码。

我们最后要考虑的方法是代码模糊，如图 5 中所示。对爱丽丝来说，最基本的观点是通过模糊软件来运行她的应用程序，该模糊软件是一种能将应用程序转换成功能上与原程序等同，但对鲍勃来说更难读懂的程序的软件。我们相信，模糊是一种保护软件交易秘密的有生命力的技术，还有待于应受的重视。

与服务器端执行不同，代码模糊永远不能完全保护一个应用程序免受恶意的反向工程。只要有足够的时间和决心，鲍勃肯定能将爱丽丝的程序分解，从而找出其中的重要算法和数据结构。为帮助这项任务，鲍勃会试图让模糊后的代码通过试图破解模糊转换的自动反模糊软件。

那么，模糊软件为应用程序增加的防范反向工程的安全等级依赖于，例如(a)模糊软件采用的转换的复杂程度，(b)可用的反模糊算法的能力，(c)反模糊软件可用的资源量（时间和空间）。理想状态下，我们希望能模仿现行公钥加密系统中的情况，其中加密（发现大的子素非常容易）和解密（将大的数字分解质因数则非

常困难）所花的费用差别很大。我们看到，事实上如下文谈论，模糊转换可以在多项式的时间内进行，但是反模糊则需要指数级的时间。

3. Java 模糊软件的设计

图 6 显示的是 Java 模糊软件 Kava 的体系结构。该工具的主要输入是一套 Java 类文件，以及用户要求的模糊等级。用户可以选择是否提供由 Java 概要工具产生的概要数据的文件。这些信息能被用于指导模糊软件确保应用程序经常被执行的部分不会被昂贵的转换所模糊。工具的输入是 Java 应用程序，作为一套 Java 类文件给出。用户还可以选择要求的模糊等级（例如，有效性）和模糊软件最大允许为应用程序增加的执行时间/空间的代价（开销）。Kava 读取和分析类文件和直接或间接参照的任何库文件。这样就构建了一个完整的继承树，同时还给出了提供所有符号的类型信息的符号表，以及所有方法的控制流图表。

Kava 包含大量的代码转换池，在下文有描述。然而，在这些被应用之前，一个预处理通道会收集依据实施例的应用程序的信息类型。使用标准的编译器技术，例如过程间数据流分析和数据附加分析，可以收集多种信息，有些可以由用户提供，有些通过专业技术来收集。例如，实用分析对应用程序进行分析，来了解是它包含何种语言构架和编程习语。

在预处理通道中收集到的信息被用于选择和应用合适的代码转换。应用程序终端所有语言构架都可以作为模糊的对象：例如，类可以被分解或合并，方法可以被更改和创建，新的控制和数据结构可以被创建，源控制和数据结构可以被修正。基于预处理通道中收集到的使用信息，可以选择与源应用程序中尽可能类似的那些构架作为应用程序中新增加的构架。

转换过程被重复，直到获得要求的有效性或者超过最大开销。工具的输出是一个新的应用程序——功能上与原来的相同——通常以一组 Java 类文件的形式给出。该工具还可以产生带有注解信息的 Java 源文件，这些信息涉及应用了哪些转换，模糊代码与源码的关系如何。带有注解的源码可用于调试。

4. 模糊转换的分类

在优选实施例的详细描述的剩下部分，我们将介绍、分类和评估不同的模糊转换。首先我们正式给出模糊转换的概念:

定义 1（模糊转换）

令 $P\text{-}^{T}\text{->}P'$为一种合法的模糊转换，其中必须遵守下列条件:

-- 如果 P 不能终止运行或在发生错误时终止运行，则 P'可能或不可能终止。

-- 否则，P'必须终止运行，并产生与 P 相同的输出。

可观察行为可被松散地定义为“用户经历的行为”。这意味着，P'可能会有 P 不具备的其它行为（例如创建文件或通过 Internet 发送信息），条件是这些附加行为不会被用户经历。注意，我们并不要求 P 和 P'完全等效。事实上，我们的许多转换会使 P'比 P 更慢或使用更多的内存。

不同模糊技术之间的主要分界线如图 2c 所示。我们主要依据模糊转换针对的信息种类不同对其进行分类。一些简单的转换是为了改变应用程序的词汇结构（布局），例如源码格式和变量名称。在实施例中，我们感兴趣的更复杂转换针对的是应用程序使用的数据结构或者控制流。

第二，我们根据目标信息上执行的运算不同来划分转换。如图 2d 到 2g 所示，有多种转换来操纵控制或数据的增加。这些转换典型地分解了程序员产生的抽象，和通过将无关数据或控制打包来构建新的假抽象。

类似地，一些转换影响着数据或控制的排序。在许多情况下，声明两个项目或进行两次计算的执行顺序对程序的可观察结果没有任何影响。然而，对写程序的程序员和反向工程师来说，在选择的顺序中可能嵌入了许多有用的信息。两个项目或事件在空间或时间上越接近，它们彼此联系的可能性就越高。对转换排序试图通过使声明或计算的顺序随机化来进行探索。

5. 评估模糊转换

在我们试图设计任何模糊转换之前，应该评估转换的质量如何。在本部分，我们将试图根据下列标准对信息进行分类：它们对程序增加了多少模糊度（例如，有效性），它们多么难以被反模糊工具破解（例如，弹性），以及它们在模糊后的程序中加入了多少计算负担（例如，开销）。

5.1 有效性的测量

首先，我们定义程序 P 比程序 P'更模糊（或称复杂和不可读）意味着什么。定义认为，任何这种量度都是相对模糊的，因为它必须基于（或部分基于）人类的认知能力。

幸运的是，我们可以在软件工程的软件复杂度量度分支中粗略画出该工作的巨大实体。在此领域中，度量被设计用于帮助构建可读、可靠和可维护的软件。量度通常是基于计算源码有多少

种文本属性，并将该计数与复杂度结合起来。虽然这些提出的公式是由对实际程序的实际研究中得出的，其它的纯粹是猜测出的。

在量度文学中发现的详细复杂度公式可被用于得出普遍的陈述，例如："如果程序 P 和 P'唯一的区别在于 P'包含的属性 q 比 P 要多，那么 P'比 P 更复杂。"有了这样的论断，我们就能够试图构建一个转换，为 a 程序增加更多的 q 属性，因为我们知道这样可以增加它的模糊性。

图 7 是列出了更理性的复杂度量度的表，其中 E(x)是软件组件 x 的复杂度，F 是功能或方法，C 是类，P 是程序。在软件构建计划中使用的话，典型目标是使这些量度最小化。与之相对的是，当对程序进行模糊的时候，我们一般都想使量度最大化。

复杂性量度允许我们将有效性的概念正式化，并在下文作为转换可用性的量度。非正式的情况下，如果转换确实通过隐藏爱丽丝的源码的意图，成功地迷惑了鲍勃，则转换是有效的。换句话说，转换的有效性衡量了模糊后代码（对人来说）比源码难理解的程度。在下列定义中对此正式化：

定义 2（转换有效性）

令 T 是保持行为的转换，使得 $P\text{-}^{T}\text{->}P'$将源程序 P 转换为目标程序 P'。令 E(P)是 P 的复杂度，由图 7 中的量度所定义。

Tpot(P)是与程序 P 相关的有效性 T，是 T 改变 P 的复杂性的程度的度量。它被定义为

$$Tpot(P)^{def}=E(P')/E(P)-1$$

如果 Tpot(P)>0，则 T 是有效的模糊转换。

为了达到本处讨论的目的，我们将以三点标准（低，中，高）来衡量有效性。

通过表 1 中的观察，我们可以列出转换 T 所需要的性质。为了使 T 成为一个有效的模糊转换，它应该

--提高整体的程序大小(u_1)并引入新的类和方法($u^a{}_7$)。

--引入新的谓词(u_2)，并且增加条件和循环构架(u_3)的嵌套等级。

--增加方法自变量(u_5)和类间实例变量附件($u^d{}_7$)的数目。

--增加继承树($u^{b,\,c}{}_7$)的高度。

--增大远程变量附件(u_4)。

5.2 弹性的测量

首先，增加的 Tpot(P)似乎是无关紧要的。例如，为了增加 u_2 的值，我们需要做的就是对 P 增加一些判断的 if 语句：

```
main(){                          main(){
S1;                              S1;
S2;          =T=>                if (5= =2) S1;
                                         S2;}
                                         If   (1>2) S2;
                                 }
```

不幸的是，这样的转换事实上是无用的，因为它们可以被简单的自动技术反转换。因此必须引入弹性的概念，来衡量转换对

抗自动反模糊工具的能力如何。例如，转换 T 的弹性可以看作是两种量度的组合：

程序员努力：构建能有效降低 T 的有效性的自动反模糊工具需要的时间量；以及

反模糊工具努力：这样一个自动反模糊工具有效地降低 T 的有效性所要求的执行时间和空间。

区分弹性和有效性是非常重要的。如果转换能迷惑人类阅读者，它就是有效的，但是如果它能够迷惑自动反模糊工具，它就是有弹性的。

我们从琐碎到单向的规模上衡量弹性，如图 8a 所示。单向转换是特殊的，因为它们不可能被反向进行。这非常典型，因为它们删除了对人类程序员有用，但在正确执行程序时并非必要的信息。例子中还包括删除格式化和使变量名混乱的转换。

其它的转换典型地向程序中增加无用的信息，它们不会增加程序的可观察行为，但是会增加人类阅读者的“信息负担”。这些转换被反向进行时的难度各异。

图 8b 显示的是反模糊工具的努力被分为多项式时间或指数级时间。程序员的努力，即使转换 T 的反模糊化自动进行的工作，作为范围 T 的函数被测量。这是基于一种直觉，即构建对抗仅影响小部分程序的模糊转换的措施，比构建可能影响整个程序的措施更容易一些。

使用代码优化理论中借用的术语来定义转换的范围：如果 T 只影响控制流图表（CFG）的一个基本块，则 T 是局部转换；如果 T 影响整个 CFG，则它是全局性的，如果 T 影响过程之间的信

息流动，则它是过程间的转换，如果 T 影响独立执行的控制线程间的相互作用，则它是进程间的转换。

定义 3（转换的弹性）

令 T 是保持行为的转换，使得 $P^{=T=}>P'$将源程序 P 转换为目标程序 P'。$T_{res}(P)$是与程序 P 相关的弹性 T。

如果从 P 中删除信息，使得不能从 P'中重新构建 P，则 $T_{res}(P)$是单向转换。否则

T_{res}^{def}= 弹性（$T_{反模糊工具努力}$，$T_{程序员努力}$），

其中弹性是图 8b 中定义矩阵中的函数。

5.3 执行开销的测量

在图 2b 中，我们看到，有效性和弹性是三个描述转换质量的成分中的两个。第三个成分是转换的开销，即转换为模糊后应用程序带来的执行时间或空间的增加量。我们用 4 点的形式划分开销（免费，便宜，贵，昂贵），下文对对此给出定义：

定义 5 （转换的开销）令 T 是保持行为的转换，使得 $T_{cost}(P)\in$｛昂贵，贵，便宜，免费｝，如果执行 P'时要求的资源比 P 要求的多 O(1)，则 $T_{cost}(P)$ = 免费；否则如果执行 P'时要求的资源比 P 要求的多 O(n)，则 $T_{cost}(P)$ = 便宜；否则如果执行 P'时要求资源比 P 要求的多 $O(n^p)$，其中 P>1，则 $T_{cost}(P)$ = 贵；否则 $T_{cost}(P)$ = 昂贵（即，执行 P'需要的资源比 P 需要多得成指数级别）。

需要指出的是，与转换相关的实际开销依赖于其应用的环境。例如，将一个简单的赋值语句 a=5 插入程序的顶级，只会多出一

个常量。在内部循环中插入的同样语句会有高得多的开销。除非特意指出，我们提出的转换开销都是指它处于源程序的最外层嵌套级别。

5.4 质量的测量

我们现在给出模糊转换的质量的正式定义：

定义 6 （转换的质量）

转换的质量 $T_{qual}(P)$被定义为 T 的有效性、弹性和开销的组合：$T_{qual}(P) = (T_{pot}(P), T_{res}(P), T_{cost}(P))$。

5.5 布局转换

在我们探索新的转换之前，我们简单考虑一下琐碎的布局转换，例如，它应用于流行的 Java 模糊工具 Crema。（Hans Peter Van Vliet, Crema -- Java 模糊工具。http://web.inter.nl.net/users/H.P.van.Vliet/crema.html, 1996 年 1 月）。第一个转换删除了 Java 类文件中有时可用的源码格式化信息。这是一种单向的转换，因为一旦源格式化丢失，它就不能被恢复；这是一种有效性很低的转换，因为格式化中的语义内容很少，所以删除格式化时没有引入多大的混乱；最后，因为这是一种免费转换，因为它不影响应用程序的空间和时间复杂度。

搞乱标识符名也是一种单向而且免费的转换。然而，它比删除格式化的有效性要高，因为标识符包含了大量的实用信息。

6. 控制信息

在本部分和以下几个部分，我们将提出模糊转换的目录。有些是从其它领域，例如编译器优化和软件工程的著名转换中借用的，其它是依据本发明的实施例，为模糊的目的而开发的。

在本部分，我们将讨论试图使源应用程序的控制流模糊的转换。如图 2f 所示，我们将这些转换分为控制流的聚集、排序或计算。控制聚集转换可以分离逻辑上组合的计算或合并逻辑上分离的计算。控制排序转换对计算执行的顺序进行随机化。计算转换则插入新的（冗余的或无用的）代码，或者对源应用程序作出算法上的改变。

对于改变控制流的转换来说，不可避免的会产生一定量的计算花费。对于爱丽丝来说，这意味着她可以在高效的程序和高度模糊的程序之间作出抉择。一个模糊工具可以通过让她在便宜和昂贵的转换之间进行选择，来帮助她进行平衡。

6.2 模糊谓词

设计改变控制的转换时真正的挑战在于使它们不仅便宜，而且能抵抗反模糊工具的攻击。为达此目的，许多转换依赖于模糊变量和模糊谓词的存在。非正式情况下，如果预先知道变量 V 有一个模糊工具，但反模糊工具很难推断出的属性 q，则 V 就是模糊的。类似的，如果谓词 P（一个布尔表达式）的结果被模糊工具知道，但反模糊工具却很难推断出来，则谓词 P 也是模糊的。

是否能产生反模糊工具难以破解的模糊变量和谓词，是模糊工具创造者面临的主要挑战，也是高弹性控制转换的关键。我们对模糊变量或谓词的衡量（即它对于反模糊攻击的抵抗能力）也

借用转换弹性同样的标准（即极弱的，弱的，强的，完全的，单向的）。类似地，我们也借用转换开销的衡量标准（免费，便宜，贵，昂贵）来衡量模糊构架的附加开销。

定义 7（模糊构架）

如果一个 V 在程序的 p 点有属性 q，则变量 V 在 p 点是模糊的，此点被称为模糊时刻。如果 p 从上下文来看很清楚，则我们将这记做 V^q_p 或者 V^q。如果谓词 P 的结果在模糊时刻就已知了，则谓词 P 在 P 点模糊。如果 P 在 p 点一直被赋值为假（或真），则我们记做 $P^F_p(P^T_p)$，如果 P 有时估价为真，有时为假，则记做 $P^?_p$。再次，如果 p 可以从上下文看出，则可以将其略去。图 9 显示了不同类型的模糊谓词。实线表示的是有时会采取的路线，而虚线表示从来不被采用的路线。

下面我们给出几个简单模糊构架的例子。这些都很容易被模糊工具构建，也很易被反模糊工具破解。第 8 部分提供的模糊构架带有的弹性要高得多。

6.1.1 极弱和弱的模糊构架

如果反模糊功能能通过静态局部分析来破解（即推知它的值）一个模糊构架，则该构架为极弱的。如果分析局限于控制流图表的单个基本块，则该分析是局部的。图 10a 和 10b 提供了极弱模糊构架(a)和弱模糊构架(b)的例子。

如果一个模糊变量可以通过对带有简单易懂语义的库函数进行调用来计算出来，我们认为该模糊变量为极弱的。对于 Java™ 之类的语言来说，这些语言要求所有的执行过程都支持标准的库类，这些模糊变量很容易构建。一个简单的例子是 int $V^5[1,5]$ =

random (1,5)，其中 random (a,b)是返回 a 到 b 的范围之中的整数的库函数。这些模糊变量同样易于反模糊。反模糊工具设计者所需要做的仅仅是列出所有简单库函数的语义，然后对模糊后代码的函数调用进行模式匹配。

如果反模糊工具能够通过静态全局分析破解一个模糊构架，则该构架是弱构架。如果一个分析仅限于对单个控制流图表进行，则该分析是全局的。

6.2 计算转换

计算转换分为三类：将真正的控制流隐藏在对实际计算没有作用的无关语句后，引入对象代码级别的代码序列，后者没有对应的高级语言构架，或者删除真正的或引入假的控制流抽象。

6.2.1 插入死代码或无关代码

u_2 和 u_3 量度表示一段代码的可觉察复杂性和它带有的谓词数目之间有很强的联系。使用模糊谓词时，我们可以建立能向程序中引入新谓词的转换。

考虑图 11 中的基本块 $S=S_1...S_n$。在图 11a 中，我们向 S 中插入一个模糊谓词 P^T，将其完全分为两半。谓词 P^T 是无关代码，因为它的值一直为真。在图 11b 中，我们再次将 S 分为两半，并继续产生后一半的两个不同的模糊版本 S^a 和 S^b。S^a 和 S^b 是通过向 S 的后半部分应用不同的模糊转换集来产生的。因此，对于一个反向工程师并不明显的是，S^a 和 S^b 事实上执行了同样的功能。

图 11c 与图 11b 类似，但是这次我们向 S^b 引入了错误。P^T 谓词经常选择代码的正确版本 S^a。

6.2.2 扩展循环条件

图 12 显示的是我们如何通过使终止条件更复杂来对循环进行模糊。基本的观点是用一个 P^T 或 P^F 谓词来扩展循环条件，该谓词不会影响循环执行的次数。例如我们在图 12d 中添加的谓词一直都为真，因为 $x^2(x+1)^2 = 0(\mathrm{mod}4)$。

6.2.3 将可推知的流图表转化为不能推知的流图表

编程语言一般被编译为本地或虚拟机代码，后者比语言本身表达得更清楚。如果情况是这样，它允许我们创建破解语言的转换。如果转换引入了和源语言构架没有直接关系的虚拟机（本地代码）指令序列，则该转换是语言破解型的。如果遇到这样的指令序列，反模糊工具将不得不试图去合成一个等效的（但是复杂的）源语言程序或完全放弃。

例如，Java™字节码有一个 goto 指令，但是 Java™语言没有对应的 goto 语句。这意味着 Java™字节码能够表示判断控制流，而 Java™语言仅仅（容易地）能表达结构化的控制流。在技术上我们认为，Java™程序产生的控制流图表一直是可以推导的，但是 Java™字节码可以表示不能推导的流图表。

因为用不带有 goto 语句的语言来表示不能被推导的流图表会非常蹩脚，我们构建了一个能够将可推导的流图表转换成不能推导的流图表的转换。这可以通过将结构化的循环变成带有多个头的循环来实现。例如在图 13a 中我们对一个 while 循环增加一个模糊谓词 P^F，使得看起来在循环的中间有一个跳转。事实上这个分支永远不会被执行。

Java™反编译器必须将一个不能推导的流图表变成一个能复制的代码或包含外部布尔变量的流图表。另一种方法是反模糊工具能够猜测所有不能推导的流图表都是由模糊工具产生的，而且能简单地删除该模糊谓词。对此，我们经常使用图 13b 中显示的另一种转换。如果反模糊工具盲目地删除 P^F，得出的会是不正确的代码。

特别地，图 13a 和 13b 中显示的是一种将可推导的流图表转换成不可推导的流图表的转换。在图 13a 中，我们将循环体 S2 分成两部分（S^a_2 和 S^b_2），然后将一个假跳转插入 S^b_2 的开始部分。在图 13b 中，我们也将 S1 分成两部分，S^a_1 和 S^b_1。S^b_1 被移动到循环中，模糊谓词 P^T 可以确保 S^b_1 在循环体之前执行。另一个谓词 Q^F 确保 S^b_1 只被执行一次。

6.2.4 删除库调用和编程习语

大多数用 Java 编写的程序主要依赖对标准库的调用。因为库函数的语法是众所周知的，这样的调用会为反向工程师提供有用的线索。如果经常通过名称对 Java 库函数进行引用，而这些名称不能被模糊，这个问题就会变得越来越严重。

在许多实例中，模糊工具能通过单独提供自己的标准库版本来解决这个问题。例如，对 Java 字典库（使用杂表来实现）的调用可以被转化为对带有相同行为的类的调用，但是作为一棵红黑树来执行。这种转换的开销对执行时间影响不大，但增大了程序的大小。

同样的问题也发生在习语（或称模式），即在许多应用程序中常见的公共编程习语上。经验丰富的反向工程师会搜索这些模式，以跳跃式地进行对不熟悉的程序的理解。作为一个例子，请考虑

一下 Java™中的链接列表。Java™库没有标准的类来提供常见的操作，例如插入、删除和列举。所以许多 Java™的程序员通过将它们在下一个字段中链接起来，以一种特别的方式构建了对象列表。在 Java™程序中，重复使用这些列表是一种常见的模式。在自动程序识别的领域中发明的技术（参见 Linda Mary Wills，自动程序识别：一个可行性演示。人工智能，45(1－2)：113－172，1990，此处引用作为参考），可以被用于辨别常见模式并用不太明显的模式替换它们。例如在链接列表的实例中，我们可以用不太常见的数据结构来表示标准的列表数据结构，例如将光标换成以数组表示数组的元素。

6.2.5 表解释

一种最有效（也最昂贵）的转换方式是表解释。这种观点是将一段代码（本例中是 Java 的字节码）转变成不同的虚拟机代码。然后这些新代码被模糊后的应用程序所包括的虚拟机解释器执行。显然，某一应用程序可以包括多种解释器，每一种能接受不同的语言并执行模糊后应用程序的不同部分。

因为每一级的解释都有一个数量级的降低过程，这种转换应该对那些组成全部运行时的一小部分代码，及需要非常高级别保护的代码段进行反向。

6.2.6 增加冗余运算符

一旦我们已经构建了一些模糊变量，我们就可以使用代数法则向算术表达式中添加冗余运算符。这将提高 u_1 量度。显然，这种技术对数字精确程度不存在问题的整数表达式效果最好。在下面的模糊后语句(1') 中，我们将使用值为 1 的模糊变量 P。在语句(2')中，我们构建了一个值为 2 的模糊表达式 P/Q。显然。我们能

让 P 和 Q 在程序执行过程中取不同的值，只要在任何需要执行语句(2')的时候它们的商为 2 即可。

(1) X=X+V; $=^{T}=>$ (1') $X=X+V*P^{=1}$;

(2) Z=L+1; (2') $Z=L+(P^{=2Q}/Q^{=P/2})/2$.

6.2.7 并行代码

自动并行是一个在多处理器机器上运行的应用程序性能的重要编译器优化法。我们希望使程序并行的原因当然是不同的。我们希望提高并行程度，并非为了提高性能，而是为了对实际的控制流进行模糊。我们有两个可能的操作方法：

1. 我们可以产生不执行任何实际工作的假进程，以及

2. 我们可以将一段顺序的应用程序代码分为多个并行执行的段。

如果应用程序在多处理器的机器上运行，我们可以预期这些转换会有明显的执行时间上的浪费。在许多情况下，这是可以接受的，因为这些转换的弹性很高：对并行程序的静态分析是非常困难的，因为程序的可能执行路径数随着执行进程的数目而呈指数级增长。并行处理还提供了高级别的有效性：反向工程师会发现并行程序比顺序执行的程序要难懂得多。

如图 14 所示，如果一段代码不包含数据附件，可以很容易地使其并行。例如，如果语句 S^1 和 S^2 的数据是独立的，它们就可以并行运行。在 Java 类没有明显并行结构的编程语言中，可以通过对线程（轻重量级的进程）库的调用来使程序并行化。

如图 15 所示，通过插入适当的同步原语，可以将包含数据附件的一段代码分成并发的线程，例如等待和前进（参见 Michael Wolfe。用于并行计算的高性能编译器。Addison-Wesley，1996. ISBN 0-8053-2730-4，此处引用作为参考）。这样的程序本质上是顺序运行的，但是控制流会从一个线程转到下一个线程。

6.3 聚集转换

程序员通过引入抽象来克服编程的内在复杂性。一个程序的多个级别上都有抽象，但是过程抽象是最重要的。因此，对模糊工具来说，对过程和方法的调用进行模糊是非常重要的。下面，我们将考虑几种对方法和方法实施进行模糊的方法：内联，外联，交织，和复制。这些之后的基本想法是相同的：(1) 程序员加入方法中的代码（想来是因为在逻辑上它们属于同类），应该分解并散开到整个程序中，(2) 看起来并不属于同类的代码应该被聚集成一种方法。

6.3.1 内联和外联方法

当然，内联是一种重要的编译器优化。它也是一种特别有用的模糊转换，因为它删除了程序中的过程抽象。内联是一种弹性很高的转换（它在本质上是单向的），因为一旦程序调用为被调用程序中的主体所替换，程序本身就被删除了，代码中不会留下抽象的痕迹。图 16 显示的是过程 P 和 Q 在它们的调用站点被内联，然后从代码中删除。

外联（将语句序列变为子例程）是内联的一种非常有用的伴随转换。我们通过将 Q 代码的开始部分和 P 代码的尾部提取到过程 R 中，从而创建假的过程抽象。

在 Java™语言等面向对象的语言中，内联事实上并不都是完全单向的转换。考虑一种方法援引 m.P()。实际被调用的程序依赖于 m 的运行时类型。在某一调用站点援引一种以上方法的例子中，我们将所有可能的方法内联起来。（参见 Jeffrey Dean 面向对象语言的全程序优化. 博士论文，华盛顿大学，1996 年，此处引用作为参考），并根据 m 的类型进行分支来选择合适的代码。因此，即使在使用内联和删除方法之后，模糊后的代码还是包括源抽象的一些痕迹。例如，图 17 显示的是内联方法调用。除非我们能够静态地确定 m 的类型，否则所有能引用 m.P()的方法都必须在调用地点进行内联。

6.3.2 插入方法

对插入代码的检测是一个重要而复杂的反向工程任务。

图 18 显示的是我们如何插入同一类中声明的两种方法。这种想法在于将方法的主体和参数列表合并，并且增加多余的参数（或全局变量）来区分对单个方法的调用。理想情况是，方法应该在本质上类似，以允许将公用代码和参数进行合并。这就是图 18 中的实例，M1 和 M2 的第一个参数的类型相同。

6.3.3 复制方法

当反向工程师试图理解子例程的目的时，他自然会检查它的签名和主体。然而，与理解例程的行为同样重要的是，它被调用时的环境是不同的。我们使过程变得困难的方法是，对方法调用的地点进行模糊，使得似乎是在调用不同的例程，而事实上，情况并非如此。

图 19 显示的是我们如何创建一个方法的多个不同版本，方法是将不同的模糊转换集应用到源码上。我们使用方法分发在运行时的不同版本间作出选择。

方法复制与图 11 中的谓词插入转换类似，只是我们使用的是方法分发，而不是模糊谓词，在不同的代码版本之间作出选择。

6.3.4 循环转换

大量的循环转换被设计用于改善（特别是）数值应用程序的性能。参见 Bacon[2]以取得全面的了解。这些转换中的某些对我们是有用的，因为它们也增加复杂性量度，上文中已结合图 7 作了讨论。图 20a 中所示的循环单元化被用于通过分解其重复空间，使得内层的循环正好位于缓存中，来改善循环的高速缓存性能。如图 20b 所示的循环展开，可以一次或多次复制循环的主体。如果循环的边界在编译时已知，则该循环可以完全展开。如图 20c 所示的循环空隙带有复合体的循环转化为多个带有相同重复空间的循环。

所有这三种转换都提高了 u_1 和 u_2 的量度，因为它们增加了原应用程序的总代码大小和条件的数量。循环的单元化转换还引入了多余的嵌套，因此也提高了 u_3 的量度。

单独使用时，这些转换的弹性相当低。反模糊工具要将展开的循环重新卷起，并不需要大量的静态分析。然而，当综合运用这些转换时，弹性就会迅速增加。例如，在图 20b 所示的简单循环中，首先我们可以进行展开，然后应用空隙，最后单元化。将最终的循环恢复到原始形式需要反模糊工具作出大量的分析。

6.4 排序转换

程序员试图组织他们的源代码，使其位置关系最大化。这种想法就是，如果逻辑上相互关连的两个项目在物理上也和源文本相近，则程序易于阅读和理解。这种位置关系在源程序的每一级都起作用：例如，在表达式中的术语，基础模块中的语句，方法中的基础模块，类中的方法以及文件中的类之间都有位置关系。所有种类的空间位置关系都能为反向工程师提供有用的线索。因此，一旦有可能性，我们就将源应用程序中任何项目的位置随机化。对于某些种类的项目（例如类中的方法）来说，这是微乎其微的。在另外一些实例（例如基础模块总的语句）中，执行一个数据附件分析（参见 David F. Bacon，Susan L. Graham 和 Oliver J. Sharp 的《用于高性能计算的编译器转换》。ACM 计算概述，26(4):345-420，1994 年 12 月。http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html。以及 Michael Wolfe 的《用于并行计算的高性能编译器》。Addison-Wesley，1996. ISBN 0-8053-2730-4，此处引用作为参考）以确定哪一种重排序在技术上是有效的。

这些转换的有效性都很低（它们并未为程序增加多少模糊性），但是它们的弹性很高，在许多实例中是单向的。例如，当基础模块中语句的位置已经随机化，在结果代码中将不会留下源代码的痕迹。

排序转换与 6.3.1 节中的“内联-外联”转换一起使用时，会特别有用。提高转换的有效性能够通过(1)在程序 P 中内联几个程序调用，(2)对 P 中对语句的顺序随机化，(3)对 P 中语句的邻接部分进行外联化。这样，原来作为多个不同程序的部分的不相关语句可以合并成假的程序抽象。

在某些实例中，也有可能重新排列循环，例如通过反向运行它们。这样的循环反向转换在高性能编译器（参见 David F. Bacon，Susan L. Graham 和 Oliver J. Sharp 的《用于高性能计算的编译器转换》，ACM 计算概述，26(4):345-420，1994 年 12 月。http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html）中非常普遍。

7. 数据转换

在本节我们将讨论使源应用程序中使用的数据结构变得模糊的转换。如图 2e 中所示，我们将这些转换分为：对存储，编码，聚集或数据排序的影响。

7.1 存储和编码转换

在许多情况下，有一种以“原有”方式来存储程序中的特定数据项。例如，通过重复数组的元素，我们可能将一个适当大小的局部整型变量分配为重复变量。其它的变量类型也有可能，但是它们不是原有的，很可能会降低效率。

进一步，经常会存在对某一变量拥有的位模式的“原有的”解释方式，该方式基于变量的类型。例如，我们通常假定存储位模式 0000000000001100 的 16 位整型变量表示整数值 12。当然，这些仅仅是惯例，而其它解释也是可能的。

模糊存储转换试图为动态和静态数据选择非原有的存储类。类似地，编码转换试图为通常的数据类型选择非原有的编码。存储和编码转换常常一起进行，但是有时它们也可以单独使用。

7.1.1 改变编码

作为一个编码简单例子，我们将用 $i_0 = c_1*i + c_2$ 替换整型变量 i，其中 c_1 和 c_2 是常数。为提高效率，我们选择 c_1 作为两个中的幂。在下面的例子中。我们令 $c_1=8$，$c_2=3$:

```
{                          =T=>          {
int i=1;                                 int i=11;
while (i<1000)                           while (i<8003)
    ...A[i]...;                             ..A[(i-3)/8]...;
    i++;                                    i+=8;
}                                        }
```

显然，还需要考虑溢出（而且，在浮点变量中还有精确性）问题。我们可以确定，因为被讨论变量有一个范围（该范围可以使用静态分析技术或询问用户来决定），所以不会发生溢出，或者我们可以将其变为较大的变量类型。

在弹性与有效性的组合和开销之间会有一个平衡。上例中使用例如 $i_0=c_1+i+c_2$ 的简单编码函数，增加的额外执行时间很少，但是可能因使用常见的编译器分析技术（参见 Michael Wolfe 的《用于并行计算的高性能编译器》。Addison-Wesley，1996. ISBN 0-8053-2730-4，以及 David F. Bacon，Susan L. Graham 和 Oliver J. Sharp 的《用于高性能计算的编译器转换》，ACM 计算概述，26(4): 345-420，1994 年 12 月。http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html）反模糊化。

7.1.2 提升变量

有多个简单的存储转换，可以将专门化的存储类中的变量提升为更加普遍的类。它们的有效性和弹性通常很低，但是在与其它的转换结合运用时会相当有效。例如，在 Java 中，一个整型变量会被提升为一个整型对象。对于其它拥有对应“打包”类的标准类型也是这样。因为 Java™支持垃圾收集，所以一旦对象不被引用，就会被自动删除。以下是一例：

```
{                                    {
int i=1;                             int i=new int(1);
while (i<9)          =T=>            while (i.value<9)
...A[i]...;                          ..A[i.value]...;
i++;                                 i.value++;
}                                    }
```

还可以改变一个变量的生存时间。最简单的这类转换将局部变量转化为独立的程序调用之间共享的全局变量。例如，如果程序 P 和 Q 都参考一个局部整型变量，则 P 和 Q 不能同时活动（除非程序包括线程，这可以通过检查静态调用表来决定），然后变量被全局化，并在它们之间共享：

```
void P() {                           int C;
int i;...I...                       void P(){
}                                    ...C...
                                     }
void Q(){            =T=>            while {i.value<9}
int k;...k...                        ...C...
}                                    }
```

这种转换提高了 u_5 量度，因为被 P 和 Q 参考的全局数据结构的数量增加了。

7.1.3 分离变量

布尔变量和其它有限定范围的变量可以被分离为 2 个或多个变量。我们将被分离为 k 个变量 p_1，...p_k 的变量 V 写作 V=[p_1，...，p_k]。典型地，这种转换的有效性会随着 k 增加而增加。不幸的是，转换的开销也会增加，所以我们通常将 k 限定为 2 或 3。

要让 T 类型的变量 V 被分离成 U 类型的两个变量 p 和 q，就要求我们提供三种信息：（1）将 p 和 q 的值映射为相应的 V 的值的函数 f (p; q)，（2）将 V 的值映射为相应的 p 和 q 的值的函数 g(V)，以及（3）通过 p 和 q 的运算符产生的新运算符（对应于根据 T 类型的值的原运算）。在本节的剩余部分我们将假定 V 的类型为布尔，p 和 q 是小的整型变量。

图 21a 显示的是表示分离布尔变量的可能选择。表中显示，如果 V 被分离为 p 和 q，并且在程序的某一点，p=q=0 或者 p=q=1，则 V 对应的为假。类似地，p=0，q=1 或者 p=1，q=0 对应的是真。

有了这种新的表示方法，我们就必须为多个内置的布尔运算符（例如&，or）创建代替的方法。一种方法是为每个运算符提供一个运行时查询表。“AND”和“OR”的表分别在图 21c 和 21d 中显示。如果给出两个布尔变量 V_1=[p,q]和 V_2=[r,s]，则可以用 AND[2p+q，2r+s]来计算 V_1&V_2。

在图 21e 中，我们显示分离的结果，即三个布尔变量 A=[a1, a2], B=[b1, b2], 和 c=[c1, c2]。我们选择的表达方式有一个有趣的方面，即同一个布尔表达式有多种可能的计算方法。例如图 21e 中的语

句（3’）和（4’），尽管它们为变量赋的值都为假，但看起来是不同的。类似的，虽然语句（5’）和（6’）完全不同，但它们都计算A&B 的值。

转换的有效性、弹性和开销都随原变量被分为的变量数增多而增加。还可以通过选择运行时编码来提高弹性。换句话说，图 21b 到 21d 中的运行时查询表并不是在编译时构建的（这样它们不能对抗静态分析），而是通过包含在模糊应用程序中的算法构建的。当然，这会阻止我们使用内联代码来计算原始的运算符，如图 21e 中所示的(6’)语句。

7.1.4 将静态数据转化为程序数据

静态数据，特别是字符串，包含有对反向工程师特别有用的众多实用信息。对静态串进行模糊的技术是将它转化为能生成串的程序。该程序－－可能是 DFA 或 Trie 横贯程序－－可能生成别的串。

作为一例，请考虑图 22 中的函数 G，它就是被构建用于对串”AAA”、”BAAAA”和”CCB”进行模糊的。G 产生的值为 G(1)=”AAA”; G(2)=”BAAAA”, G(3)=G(5)=”CCB”和 G(4)=”XCB”（程序中实际上并不常用）。对于其它的自变量值，G 可能会终止运行也可能不会。

将所有静态数据的计算聚集到一个函数中去，当然是绝不受欢迎的。如果 G 函数被分解成能嵌入到源应用程序的“普通”控制流中的较小部件，则可以获得更高的有效性和弹性。

值得指出的是，我们能将这项技术和 6.2.5 中的表解释转换结合起来实用，模糊的目的在于将一段 Java 字节码转化为另一种虚

拟机的代码。新代码典型地在模糊后的程序中作为静态串数据存储。然而，要达到更高等级的有效性和弹性，可以将这些串转换为能生成它们的程序，如上文所述。

7.2 聚集转换

与命令式的和功能的语言相比，面向对象的语言比面向控制的语言对数据的针对性更强。换句话说，在面向对象的程序中，是根据数据结构组织控制，反之而不然。这意味着，对面向对象的程序进行反向工程时，一个重要的部分是试图恢复程序的数据结构。反过来，对模糊工具来说，试图隐藏这些数据结构是非常重要的。

在大多数面向对象的语言中，有两种方法聚集数据：数组和对象。在以下三节中，我们将考察模糊数据结构的方式。

7.2.1 合并等级变量

两个或多个等级变量 $V_1...Vk$ 可以被合并为一个变量 V_M，条件是 $V_1...Vk$ 的组合范围在 V_M 的精确度之内。例如，两个 32 位的整型变量可以被合并成一个 64 位的变量。单个变量上的算术知识将被转换为 V_M 上的算术知识。作为一个简单的例子，请考虑将两个 32 位的整型变量 X 和 Y 合并成一个 64 位的变量 Z。实用以下合并公式：

$$Z(X,Y) = 2^{32}*Y+X$$

我们在图 23a 中可得算术衡等式。图 23b 中也给出了一些简单的例子。

特别地，图 23 显示将两个 32 位的变量 X 和 Y 合并成一个 64 位的变量 Z。Y 占据 Z 的高 32 位，X 占据低 32 位。如果 X 或 Y 的实际范围可以从程序推出就可以实用直觉性更弱的合并。图 23a 给出了 X 和 Y 的加法和乘法的规则。图 23b 显示了一些简单的例子。例子还可以进一步被模糊，例如通过将(2')和(3')合并成 Z+=47244640261。

合并变量的弹性非常低。反模糊工具只需考察应用到特定变量上的算术运算符集，就可以猜出它实际上包含了两个合并的变量。我们可以通过引入假运算符来增加弹性，该运算符并不对应于单个变量的任何一个合理运算符。在图 23b 中的例子中，我们可以插入看起来是将 Z 的两半合并起来的运算符，例如，通过移位：如果 (P^F) 则 Z = rotate(Z,5)。

该转换的一个变体是将 V_1...Vk 合并成有合适类型的数组：

$$V_A = \begin{matrix} 1...k \\ V_1...Vk \end{matrix}$$

例如，如果 V_1...Vk 是对象参照的变量，则 VA 的元素类型可以是在继承等级上比 V_1...Vk 中的任何类型都要高的类。

7.2.2 重新构建数组

可以有多种方法对数组要运行的运算符进行模糊，例如，我们可以将数组分离为多个子数组，将两个或多个数组合并成一个数组，折叠一个数组（提高维的数目），或展平一个数组（减少维的数目）。

图 24 显示的是数组重构建的一些例子。在语句（1-2）中，数组 A 被分离为两个子数组 A1 和 A2。A1 包含 A 中带有偶索引的元素，而 A2 包含 A 中带有奇索引的元素。

图 24 的语句（3-4）中，显示的是两个整型数组 B 和 C 是怎样被交叉成为结果数组 BC 的。B 和 C 中的元素均匀分布到结果数组中。

语句（6-7）中显示的是一维数组 D 是怎样被折叠成为二维数组 D1 的。最后，语句（8-9）中显示的是反向转换：二维数组 E 被展平成为一维数组 E1。

数组分裂和折叠提高了 u^6 的数据复杂性度量。另一方面，数组合并和展平似乎降低了这个量度。尽管这似乎表明这些转换只有最低限度的或甚至负面的有效性，事实上，这是具有欺骗性的。问题在于图 7 的复杂性量度没有抓住数据结构转换的一个重要方面：它们引入了原来没有的结构或是从源程序中删除了结构。这可以极大地提高程序的模糊性。例如，程序员声明一个二维数组的目的是：选中的结构清晰地映射到要操纵的数据上。如果数组被折叠成一维的结构，反向工程师就不能获得许多有用的实用信息。

7.2.3 修改继承关系

在 Java™语言等流行的面向对象的语言中，主要的模块化和抽象的概念是类。类在本质上是将数据（实例变量）和控制（方法）进行封装的抽象数据类型。我们将类记做 C = (V,M)，其中 V 是 C 的实例变量集，M 是 C 的方法。

与传统的抽象数据类型的概念相比，两个类 C1 和 C2 可以通过聚集（C_2 有一个 C_1 类型的实例变量）或继承（C_2 通过增加新的方法和实例变量对 C_1 进行扩展）获得。我们将继承记为：$C_2=C_1UC_2$。我们说，C_2 是从其上类或父类 C_1 继承得到的。U 运算符的功能是将父类与 C'_2 定义的新属性结合起来。U 的确切语义依赖于特定的编程语言。在例如 Java 的语言中，U 被应用到实例变量时经常被解释成结合，而被应用到方法上时被解释成覆盖。

依据度量 u_7，类 C1 的复杂度随其在继承级别中的深度（与根的距离）和直接继承类的数目增加而增加。例如，我们可以利用两种基本方法来提高复杂度：我们能如图 25a 所示分离（分解）一个类，或者如图 25b 所示插入一个新的假类。

分解类时存在的问题是它的弹性较低；没有什么能够阻止反模糊工具简单地合并分解后的类。为防止这类情况，通常如图 25d 所示综合运用分解和插入。另一种增加这种类型转换的弹性的方法是，确保新类是由所有的引入的类产生的。

图 25c 显示的是类插入的一个变体，称为假重分解。重分解是重新构建结构已被破坏的面向对象程序的（有时是自动的）一种技术（参见 William F. Opdyke 和 Ralph E. Johnson 的《利用重分解创建抽象上类》。 Stan C. Kwasny 和 JohnF. Buck 编写的《计算机科学 21 届年会进展》，66-73，纽约，纽约州，美国，1993 年 2 月。ACM 出版社。ftp://st.cs.uiuc.edu/pub/papers/refactoring/refactoring-superclasses.ps，此处引用作为参考）。重分解分两步进行。首先，检测到两个表面上独立的类事实上实施同样的行为。其次，两个类共有的特征被转移到一个新的（可能是抽象的）父类中。假重分解的过程非常类似，它是在没有共有行为的两个类 C_1 和 C_2 上执行的。如果两个类都有同样类型的实例变量，则它们

可以被转移到新的父类 C_3 中。C_3 的方法可以是来自 C_1 和 C_2 的某些方法的带缺陷版本。

7.3 排序转换

在 6.4 节中，我们显示了对计算执行的顺序进行随机化是一种有用的模糊手段。类似地，对源应用程序中声明的顺序进行随机化也很有用。

特别地，我们打乱类中的方法和实例变量以及方法内的正式变量的顺序。在后一种情况中，相应的实际值当然会被重排序。这些转化的有效性很低，而弹性是单向的。

在许多实例中，数组内的元素重排序也是可能的。简而言之，我们提供一个模糊编码函数 f(i)，将源数组中的 i:th 元素映射到它在排序后数组中的新位置：

```
{                                   {
int i=1,A[1000];                    int i=1,A[1000];
while (i<1000)      =T=>            while (i<1000)
...A[i]...;                         ..A[f(i)]...;
i++;                                i++;
}                                   }
```

8. 模糊值和谓词

正如我们所看到的，模糊谓词在能模糊控制流的转换的设计过程中是重要构造的模块。事实上，大部分控制转换的质量直接依赖于这些谓词的质量。

在节 6.1 中，我们给出的例子是带有极弱和弱的弹性的简单谓词。这意味着可以用局部或全局静态分析来破解（一个自动反模糊工具可以确定它的值）模糊谓词。显然，我们通常要求的抗攻击能力要高得多。理想情况是，我们希望构建的模糊谓词需要指数级的时间（对程序的大小而言）来破解，而只需多项式级别的时间来构建。在本节中，我们将提出两种这样的技术。第一个基于别名，第二个基于轻量级的处理。

8.1 使用对象和别名的模糊构架

只要存在别名的可能性，程序间具体分析就是非常复杂的。事实上，精确的、对流敏感的别名分析在带有动态分配、循环和 if 语句的语言中是不能确定的。

在本节中，我们将利用别名分析的复杂性来构建对于自动反模糊攻击开销小且有弹性的模糊谓词。

8.2 使用线程的模糊构架

并行程序比顺序程序更难以静态分析。原因在于它们相互交叉的语义：如果并行区 PAR 中有 n 个语句 $S_1,S_2,\ldots,Sn$，则 ENDPAR 可以以 n!种不同的方法被执行。尽管这样，并行程序上的一些静态分析可能以多项式级别的时间来执行[18]，而其它的分析则需要考虑所有的 n!个交叉。

在 Java 中，使用被称为线程的轻量级进程来构建并行区。Java 的线程有两个非常实用的属性（从我们的观点来看）：(1) 它们的调度策略没有被语言规则所严格指定，所以依赖于执行，(2) 线程的实际调度依赖于例如由用户交互和网络流量产生的异步事件。

与并行区域内部的价值语义结合起来，就意味着对线程很难进行静态分析。

我们将利用这些分析来创建破解时需要指数级时间的模糊谓词（参见图 32）。基本的观点与 8.2 节中用到的相似：创建全局数据结构 V，并不时对其进行更新，但保持其状态，使得可以进行模糊查询。不同点在于由并发执行的线程来对 V 进行更新。

显然，V 可以是如图 26 中创建的动态数据结构。这些线程通过异步执行调用来实现移动和插入，从而使全局指针 g 和 h 在它们各自的组件中随机移动。这么做的优势在于能将数据类与交织和别名效果相结合，从而达到高水平的弹性。

在图 27 中，我们用简单得多的例子展示了这些观点，其中 V 是两个全局整型变量 X 和 Y。这是基于基础数论中的著名事实：对于任何整数 x 和 y，$7y^2-1$ 不会等于 x^2。

9. 反模糊和预防性转换

我们的许多反模糊转换（特别是节 6.2 中的控制转换）可以说就是在真实程序中嵌入假程序。换句话说，一个模糊后的应用程序事实上包含了合为一体的两个程序：执行有用任务的真程序和计算无用信息的假程序。假程序的唯一用途是，通过将真程序藏在无关代码后，来迷惑反向工程师。

模糊谓词是模糊工具的主要工具，可以用来防止假程序被轻易地识别和删除。例如，在图 28a 中，一个模糊工具在真程序的三个语句中嵌入被模糊谓词保护的假代码。反模糊工具的任务是检测模糊后的应用程序，并自动识别和删除内部假程序。为完成

此任务，反模糊工具首先必须识别并评估模糊构架。该过程在图 28b 到图 28d 中描述。

图 29 显示了半自动反模糊工具的结构。它集成了多种在反向工程界众所周知的技术。在本节的剩余部分，我们将简单介绍这样一些技术，并讨论多种对抗措施，它们被模糊工具用来使反模糊过程变得更困难（即所谓的预防性转换）。

9.1 预防性转换

上文讨论的与图 2g 相关的预防性转换，与控制或数据转换在风格上是完全不同的。与这些转换相比，它们的主要目的不是使程序对人类阅读者更模糊。它们被设计用于使已知的自动反模糊技术更加困难（内在的预防性转换），或者用于探索当前反模糊工具和反编译器中的已知问题（预期的预防性转换）。

9.1.1 内在的预防性转换

内在的预防性转换一般有效性很低而弹性较高。最重要的是，它们能迅速提高其它转换的弹性。例如，假定我们如 6.4 节中所提到的那样，对一个向后运行的 for 循环重新排序。只要因为我们能确定循环中没有循环携带的数据附件，我们就能够应用这种转换。当然，以上工作并不能阻止反模糊工具执行相同的分析并且将循环转向为向前执行。为防止这类事情的发生，我们可以在反向循环加一个假数据附件。

```
{                                    {
for (i=1;i<=10;i++)=T=>                 int B[50];
    A[i]=i;                             for (i=10;i<=1;i--)
}                                       A[i]=i;
                                        B[i]+=B[i*i/2]
                                    }
```

内在预防性转换为循环重排序转换增加的弹性，取决于假附件的复杂性和附件分析[36]中的技术状态。

9.1.2 预期的预防性转换

作为预期预防性转换的一个例子，考虑 HoseMocha 程序（Mark D. LaDue. HoseMocha. http://www.xynyx.demon.nl/java/HoseMocha.java，1997 年 1 月）。它被特别设计用于探索 Mocha 编译器(Hans Peter Van Vliet. Mocha---《Java 编译器》，http://web.inter.nl.net/users/H.P.van.Vliet/mocha.html，1996 年 1 月)的弱点。HoseMocha 在源程序每种方法中的每一个返回语句后插入额外的指令。这种转换对应用程序的行为没有影响，但是它足以使 Mocha 崩溃。

9.2 识别和评估模糊构架

反模糊工作中最困难部分是识别和评估模糊构架。注意，识别和评估是不同的行为。一个模糊构架可以是局部的（包含在一个基础模块中），全局的（包含在一个程序中），或者程序间（分布在整个程序中）。例如，if (x*x == (7^F*y*y-1))是一个局部模糊谓词，而 R=X*X;...; S=7*y*y-1; ...; if (R= =S^F)...都是全局变量。如果在不同的程序中执行 R 和 S 的运算，构架就是程序间的。显然，识别局部模糊谓词比确定程序间的模糊谓词要容易。

9.3 通过模式匹配进行识别

反模糊工具可以利用对已知模糊工具使用的策略的知识来识别模糊谓词。一个反模糊工具的设计者可以研究模糊工具（通过反编译或是简单地研究它生成的模糊后代码），和构建可以识别广泛使用的模糊谓词的模式匹配规则。这种方法对简单的局部谓词效果很好，例如 x*x = =(7*y*y-1))和 random(1^F,5)<0。

为了挫败模式匹配的企图，模糊工具应该避免使用千篇一律的模糊构架。同样重要的是，选择在句法上与实际应用程序中使用的构架类似的模糊构架。

9.4 程序分片识别

程序员通常会发现程序被模糊后，比源程序会更难理解和进行反向工程。主要原因在于在模糊后的程序中，(a)活动的“真”程序中散布着死的假代码，(b)逻辑上关连的代码段被分解并散布到程序中。反向工程师可以使用程序分片工具来对抗模糊。这些工具可以交互式地帮助工程师将程序分解为被称为片的可管理块。程序 P 中与点 p 和变量 v 有关的片包括在 p 点能得出 v 值所有 P 中的语句。因此，即使模糊工具将计算模糊变量 v 的算法的语句分散在整个程序中，程序分片器也可以从模糊后的程序中分离出这些语句。

模糊工具有多种策略来破坏分片这种识别工具的效果：增加参数别名。一个参数别名是两个指向同一内存区域的正式参数（或一个正式的参数和一个全局变量）。精确程序间分片的开销会随着程序中潜在别名的数量增加而增长，而别名的数量随正式参数的数量增加呈指数级增长。因此，如果模糊工具在程序中加入了有别名的假参数，它会明显降低分片器的速度（如果要求进行精确

分片的话），或是强迫分片器产生不精确的片（如果要求快速分片）。

如果增加变量附件，例如 Unravel（James R. Lyle，Dolorres R. Wallace，James R. Graham，Keith B. Gallagher，Joseph P. Poole 和 David W Binkley 的《Unravel：一种有助于评估高集成软件的软件工程工具》，卷 1：要求和设计。技术报告 NIS—TIR 5691，美国商业部，1995 年 8 月）的流行分片工具运用在小片上效果还可以，但是计算大片时有时会要求额外的时间。例如，对 4000 行的 C 程序进行分析时，某些实例中的 Unravel 计算 1 个片需要 30 分钟以上。为了强化这种行为，模糊工具应该试图通过增加假变量附件，来增加片的大小。在下面的例子中，我们通过增加两个似乎对计算 x 的值有用而实际无用的语句，使片变大。

```
main() {                    =T=>      main() {
int x=1;                              int x=1;
x=x*3;                                  if (P^F) x++;
}                                     x=x+V0;
                                      x=x*3;
                                        }
```

9.5 统计分析

反模糊工具能利用模糊后的程序，来分析出所有谓词的结果。我们可以使用任何用这种方法，即统计分析，来检查模糊后程序的运行时特征的反模糊方法。反模糊工具会提醒反向工程师注意任何在多次运行过程中一直返回“真”值的谓词，因为它们很可能就是模糊 $P^T(P^F)$谓词。反模糊工具不能盲目地用“真”或“假”来替换这些谓词，因为这样太危险了。许多应用程序会包含对例

外环境中发生的条件进行检验的“真实”谓词，对反模糊工具来说，它们看起来和模糊谓词是一样的。作为一例，请考虑 if（Leap Year）....

统计分析还可以用于赋值。一旦识别出程序 M 中的潜在模糊谓词（例如 P^T），我们就可以猜测其值（为真），然后产生模糊后程序的版本 M'，其中模糊谓词被猜测出的值所代替。然后我们对同一个输入并行运行 M 和 M'，并比较它们的输出是否相同。如果输出相同，我们可以确定该谓词是假程序的一部分，而不是真程序部分，如图 30。

我们必须确信，我们选中的输入完全涵盖了程序中的所有路径。然而，如果程序包括很少采用的路径（if (Leap Year)），这就会很困难。进一步，产生大量的正确输入数据是非常困难的，特别是当应用程序的内部结构未知，或者输入是通过复杂的图形用户界面进行的时候。

为了避免被统计分析所识别，模糊工具可以优先选择插入 $P^?$ 谓词的转换（如图 11b 所示），而不是插入 P^T 或 P^F 谓词的转换。

另一个对抗统计分析的可能措施是设计模糊谓词时，使得必须同时破解多个谓词。这样做的一种方法是，让模糊谓词有附加效果。在下例中，模糊工具（通过某种静态流分析）确定，S_1 和 S_2 语句必须执行相同的次数。通过引入调用函数 Q_1 和 Q_2 的模糊谓词，对语句进行模糊。Q_1 和 Q_2 增加和减少了全局变量 k:

```
{                    =T=>        {
S1;                              int k=0;
S2;                              bool Q1 (x) {
}                                k+=2^31; return (P^T_1)}
```

bool Q_2 (x) {

k-=2^{31}; return (P^T_2)}

{

if (Q_1 (j) T) S_1;

…

if (Q_2 (k) T) S_2;

}

如果反模糊工具试图用真来替换一个（而不是两个）谓词，则 k 会溢出。结果是，反模糊后的程序会在出错的条件下终止。

9.6 使用数据流分析的评估

反模糊与许多种代码优化是类似的。删除 if (False)就是除去死亡代码，从 if 语句的分支移动相同的代码就是代码提升，这两者都是常见的代码优化技术。

一旦模糊构架被识别，我们可以试图对其进行评估。在简单的例子中，使用到达定义的数据流分析的常数传播可能是足够的：x=5; …; y=7; …; if (x*x= =(7*y*y-1))….

9.7 使用定理证明的评估

如果数据流分析不足以强大到破解模糊谓词，反模糊工具可以试图使用定理证明器。这是否可行完全取决于现有的定理证明器的威力（这是很难确定的），以及需要证明的定理的复杂程度。当然，能被推导（例如 $x^2(x+1)^2=0 \pmod 4$）来证明的定理都完全在目前的定理证明器的能力范围之内。

为了使任务变得困难，我们可以使用被知道是很难证明的定理，或者没有任何已知证据存在的定理。在下面的例子中，反模糊工具必须证明，假循环经常终止，目的是为了确定 S_2 是活代码：

```
{                    =T=>        {
S1;                              S1;
S2;                              n=randon(1,2^32);
}                                do
                                        n=((n%2)!=0)?3*n+1:n/2
                                 S2;
                                 }
```

这被称为 Collatz 问题。一种猜测说，循环一直都可以终止。尽管这种猜测没有任何证据，但对于比 $7*10^{11}$ 这样大的数下的所有数，我们知道代码会终止。因此，这种模糊是安全的（源代码和模糊后代码的表现相同），但是很难被反模糊。

9.8 反模糊的部分评估

反模糊也模仿部分评估。部分评估器将程序分为两部分：能被部分评估器预先计算的静态部分，和在运行时执行的动态部分。动态部分与我们原来的模糊前的程序对应。而静态部分与我们的假内部程序对应，如果它被识别，则可以在反模糊时被评估和删除。

和所有其它的静态程序间分析方法一样，部分评估对别名很敏感。因此，与分片结合讨论的同样的预防性转换，也可以应用到部分评估上。

10. 模糊算法

在第 3 节的模糊工具体系结构，第 5 节中的模糊质量的定义以及第 6 到第 9 节中对不同模糊转换的讨论的基础上，我们现在依据本发明的实施例，提出更详细的算法。

模糊工具的顶级循环可能有这种常见的结构:

```
WHILE NOT Done(A) DO
        S:=SelectCode(A);
        T:=SelectTransform(S);
        A:=Appy(T,S);
End;
```

SelectCode 返回待模糊的下一个源代码对象。SelectTransform 返回应该用于对特定的源代码对象进行模糊的转换。Apply 将转换应用到源代码对象上并据此更新应用程序。Done 确定何时获得需要的模糊等级。这些函数的复杂性依赖于模糊工具的复杂性。如果模糊工具最简单，则 SelectCode 和 SelectTransform 简单地返回随机源代码对象/转换，而 Done 可以在应用程序的大小超过一定的限制时终止循环。通常，这种行为是不够的。

算法 1 给出了带有更复杂选择和终止行为的代码模糊工具的描述。在一个实施例中，算法使用了算法 5，6，7 构建的多种数据结构。

每一种源代码对象 S 的 P_S，$P_S(S)$是程序员在 S 中使用的语言构架集。$P_S(S)$被用于为 S 找出合适的模糊转换。

每一种源代码对象 S 的 A，A(S)={Ti-->V_1;...;Tn-->Vn}是转换 T 与值 Vi 之间的映射，描述的是将 Ti 应用到 S 上的合适程度如何。想法是某些转换对于特定源代码对象 S 可能是不合适的，因为它们引入了对于 S 来说“不自然”的新代码。新代码在 S 中显得不合适，因此很容易被反向工程师定位。合适值 Vi 越大，则转换 Ti 引入的代码就越适合。

每一种源代码对象 S 的 I，I(S)是 S 的模糊优先级。I(S)描述的是它对于模糊 S 从内容是多么重要。如果 S 包含重要的商业秘密，I(S)会很高，如果它包含的只是“面包—黄油”代码，则 I(S)会很低。

每个例程 M 的 R，R(M)是 M 的执行时间级别。如果执行 M 花费的时间比执行其它例程的时间要长，则 R(M)=1。

算法 1 的主要输入是应用程序 A 和一套模糊转换{T_1;T_2;...}。算法还要求与每种转换相关的信息，特别是三个质量函数 $T_{res}(S)$，$T_{pot}(S)$，$T_{cost}(S)$（名称与第 5 节中相同，但返回数字值）和一个函数 P_t:

$T_{res}(S)$返回的是转换 T 被应用到源码对象 S 上时的弹性量度（即 T 对抗自动反模糊工具的效果如何）。

$T_{pot}(S)$返回的是转换 T 被应用到源码对象 S 上时的有效性量度（即 T 被 T 模糊后，对人来说理解的难度增加了多少）。

$T_{res}(S)$返回的是 T 在 S 上增加的执行时间和空间花费的量度。

P_t将每一个转换 T 映射到 T 为应用程序增加的语言构架集上。

算法 1 的点 1 到点 3 载入待模糊的应用程序，并建立适当的内部数据结构。点 4 建立 $P_s(S)$，A(S)，I(S)和 R(M)。点 5 应用模糊转换，直到达到要求的模糊级别或超过了最大时间花费。最后，点 6 重新写出新的应用程序 A'。

算法 1 （代码模糊）

输入：

a) 由源代码或对象代码文件 C1； C2. . . 组成的应用程序 A。

b) 有语言定义的标准库 L1； L2; . . .

c) 模糊转换集{L1； L2; . . . }。

d) 映射 P_t，对每一个转换 T 给出 T 将会在应用程序加入的一套语言构架。

e) 三个表示与源码对象 S 有关的转换 T 的质量的函数 $T_{res}(S)$，$T_{pot}(S)$，$T_{cost}(S)$。

f) A 的一组输入数据 I={I1; I2;. . . }

g) 两个数字值 AceptCost>0 且 ReqObf>0。AceptCost 是用户可以接受的最大额外执行时间/空间花费的量度。ReqObf 是用户要求达到的模糊程度的量度。

输出：由源代码或对象代码文件组成的模糊后应用程序 A'。

1. 载入待模糊的应用程序 $C_1;C_2;. . .$。模糊工具可以：(a) 载入源码文件，其中模糊工具必须包含完整的前端，可以执行语义、

句法和语法分析（仅限于单纯句法转换的威力较小的模糊工具，也可以在不进行语义分析的情况下工作），或者

b) 载入对象代码文件。如果对象代码中保存着源代码中的大部分或全部信息，则优先选择这种方法。

2. 载入应用程序直接或间接引用的库代码文件 L1; L2;...

3. 建立应用程序的一个内部表达。对内部表达的选择取决于源文件的结构和模糊工具所实施转换的复杂度。典型的数据结构组可能包括:

a) A 中每一个例程的控制流图表。

b) A 中例程的调用图表。

c) A 中类的集成图表。

4. 构建映射 R(M)和 P_s(S)(使用算法 5)，I(S)(使用算法 6)和 A(S)(使用算法 7)。

5. 将模糊转换应用到应用程序中。每一步，我们选择一个源码对象 S 进行模糊，以及一个合适的转换来应用到 S 上。一旦获得要求的模糊级别或者超过可接受的执行时间花费，进程终止。

```
REPEAT
    S:=SelectCode(I);
    I:=SelectTransform(S,A);
    对 S 应用 T，并从点 3 更新相关数据结构;
UNTIL Done(ReqObf，AcceptCost，S，T，I)
```

6. 将模糊后源代码对象重建到新的模糊后程序 A'中。

算法 2 （SelectCode）

输入：算法 6 计算的模糊优先级映射 I

输出：源码对象 S

I 将每个源码对象 S 映射到 I(S)，后者是 I 对 S 的重要程度的量度。为了选择下一个要模糊的源码对象，我们将 I 作为优先级队列处理。换句话说，我们选择 S 的原则是使 I(S)最大。

算法 3 （SelectTransform）

输入：

a) 源码对象 S。

b) 算法 7 计算的合适度映射 A

输出：T 转换

可以尝试多次，以选择最合适的转换应用到特定源码对象 S 上。然而，还需要考虑两个重要的问题。首先，选中的转换必须与 S 中的其它代码自然融和。这可以通过在 A(S)中给转换提供高的合适度值来实现。第二，我们希望优先选择那些产生较高的'性价比'（即模糊等级较高而执行时间花费较少）的转换。这可以通过选择那些有效性和弹性最大而开销最小的转换来实现。这些探试可以由下列代码捕获，其中 w1，w2，w3 是执行时定义的常量：

返回一个转换 T，使得 T-->V 在 A(S)的范围内，而且$(w1*T_{pot}(S)+w2*T_{res}(S)+W3*V)/T_{cost}(S)$最大化。

算法 4 （Done）

输入：

a) ReqObf，模糊的剩余等级。

b) AcceptCost，剩余的可接受执行时间花费。

c) 源代码对象 S。

d) T 转换。

e) 模糊优先级映射(I)。

输出：

a) 更新后的 ReqObf。

b) 更新后的 AcceptCost。

c) 更新后的模糊优先级映射(I)。

d) 布尔返回值，当满足终止条件时为真。

Done 函数有两个作用。它对优先级队列 I 进行更新，以反映出源码对象 S 已被模糊化，且应该接受一个减小的优先级值的事实。该减小过程在将转换的弹性和有效性结合起来的基础上进行。Done 还更新 ReqObf 和 AcceptCost，并确定终止条件是否满足。w_1，w_2，w_3，w_4是执行时定义的常量：

$I(S):=I(S)-(W_2T_{pos}(S)+W_2T_{res}(S))$;

$ReqObf:=ReqObf-(W_2T_{pos}(S)+W_2T_{res}(S))$;

$AcceptCost:=AcceptCost-T_{cost}(S))$;

RETURN AcceptCost<=0 OR ReqObf<=0.

算法 5 （实用信息）

输入:

a) 应用程序 A。

b) 输入 A 的一组数据 I={I1; I2; . . . }。

输出:

a) 映射 R(M)，对于 A 中的每一个例程 M，给出 M 的执行时间等级。

b) 映射 $P_s(S)$，对于 A 中的每一个源代码对象 S，给出 S 中使用的一组语言构架。

计算实用信息。该信息将被用来为每个特定的源代码对象选择合适的转换类型。

1. 计算动态实用信息（例如，在用户提供的输入数据组 I 上的概要文件下运行应用程序）。为每一个例行/基础模块计算 R(M)（M 的执行时间等级），指出应用程序的大部分时间花费在哪里。

2. 计算静态实用信息 $P_s(S)$。$P_s(S)$提供的是程序员在 S 中使用的多种语言构架的统计数字。

FOR S: = A 中的每一个源代码对象 DO

O: = S 使用的一组运算符;

C: = S 使用的一组高级语言构架（WHILE 语句，例外，线程等）;

L: = S 参考的一组库类/例程;

Ps(S): =O U C U L

END FOR

算法 6 （模糊优先级）

输入:

a) 应用程序 A。

b) R(M)，M 的等级。

输出： 映射 I(S)，对于 A 中的每一个源代码对象 S，给出 S 的模糊优先级。

I(S)可以由用户明确给出，或利用基于算法 5 中收集的统计数据的探试法来计算得出。可能的探试法为:

1. 对于 A 中任何例程 M，令 I(M)与 M 的等级 R(M)成反比。即，观点是，“如果执行例程 M 花费了许多时间，那么 M 可能是需要着重模糊的重要程序”。

2. 令 I(S)为 S 的复杂度，如表 1 中的一个软件度量所定义的那样。然后，（可能有缺陷的）直觉是，复杂的代码比简单代码更可能包含重要的商业机密。

算法 7（模糊适合度）

输入:

a) 应用程序 A。

b) 映射 Pt，对每一个转换 T，给出 T 为应用程序增加的一组语言构架。

c) 映射 Ps(S)，对于 A 中的每一个源代码对象 S，给出 S 中使用的一组语言构架。

输出： 映射 A(S)，对于 A 中的每一个源代码对象 S 和每个转换 T，给出与 S 相关的 T 的适合度。

计算每一个源代码对象 S 的合适度组 A(S)。映射主要基于算法 5 中计算的静态实用信息。

```
FOR S: = A 中的每一个源代码对象 DO
    FOR T: =每个转换 DO
        V: = Pt(T)和 Ps(S)之间的类似程度;
        A(S): = A(S) U{T-->V};
    END FOR
END FOR
```

11. 总结和讨论

我们已经观察到，在许多情况下，模糊后的程序和源程序表现不同是可以接受的。特别是，我们的大多数模糊转换使目标程序与源程序相比较慢或者较大。在特殊的实例中，我们甚至允许目标程序有与源程序不同的附加效果，或者即使源程序在错误条件下终止，目标程序也可以不终止。我们唯一要求是两个程序的可观察行为（用户所经历的行为）应该相同。

允许源程序和模糊后程序的相同点如此不明显，是一个新奇而令人兴奋的想法。尽管上文提供和描述了多种转换，了解一般技术的人员可以清楚地知道有其它的转换，而且这些转换可以提供依据本发明的模糊，以提高软件的安全性。

在未来的研究中，识别未知的转换还大有可为。特别地，我们希望看到下列领域被大家研究：

1. 应该识别新的模糊转换。

2. 应该研究不同转换间的相互作用和排序。这和代码优化中进行的工作是一样的，对优化转换序列的排序在代码优化中也经常是一个困难的问题。

3. 应该研究有效性和开销之间的关系。对于特定类型的代码，我们希望知道哪一种转化将给出最好的“性价比”（即以最低的执行花费取得最高的有效性）。

欲获得对以上讨论中所有转换的整体看法，请参见图 31。欲获得对以上讨论中模糊构架的整体看法，请参见图 32。然而，本发明不应局限于上文讨论的示例转换和模糊构架。

11.1 模糊的效果

加密和程序模糊有很大的相似性。两者不仅都试图隐藏信息以避开那些探索的眼睛，而且使用它们也只是为了使信息隐藏有限的时间。加密后文档的保存期限是有限的：只要加密算法本身能对抗攻击，而且硬件发展的速度并不允许选中代码长度的信息被机械地解密，它就是安全的。对于模糊后的应用程序也是这样；只要没有建立足够强大的反模糊工具，它就是安全的。

对于发展中的应用程序来说，只要发布需要的时间比反模糊工具跟上模糊工具所花费的时间短，这就不会成为问题。如果这是事实，那么到应用程序可以被自动反模糊的时候，它就已经过时了并且竞争者对它也失去了兴趣。

然而，如果一个应用程序包含了在多个发布版本中都存在的商业秘密，那么可以采用模糊以外的其它方法来进行保护。似乎应该选择部分服务器端执行（如图 2(b)所示），但是它的缺陷在于：应用程序会执行得较慢或（当网络连接中断时）完全不执行。

11.2 模糊的其它用途

说起来很有趣，模糊除了上文讨论的应用之外，还有很多其它的潜在应用。一种可能性是利用模糊跟踪软件盗版。例如，厂商为每一个新用户产生一个应用程序的新模糊版本（我们可以通过将随机因素引入 SelectTransform 算法（算法 3）中，为同一个程序产生不同的模糊版本。随机数生成器的不同种子可以产生不同的版本。），并且记录下每个版本的购买者。很可能只有当软件在网上销售和发布的时候，这才是合理的。如果厂商发现他的应用程序被盗版了，他所需要的做的是获得一份盗版的拷贝，将它与数据库中的记录进行比较，看出谁购买了原始程序。事实上储存每一个卖出的模糊版本的拷贝并非必要。只要保存售出的随机数的种子就可以了。

软件盗版自身也可以（非法）利用模糊。因为我们上文描述的 Java 模糊器在字节码的级别进行起作用，所以不能阻止盗版者对合法购买的 Java 程序进行模糊。模糊版本可以被转售。面对诉讼的时候，盗版者辩驳说，实际上他并没有转售最初购买的应用程序（毕竟，代码是完全不同的！），而是在销售合法的重新编写的版本。

结论

总之，本发明提供了计算机实施的方法和设备，用于防止或至少阻碍对软件的反向工程。尽管这是以执行需要的时间或程序

的大小为代价来实现的，同时生成的转换后程序在细节上表现有所不同，但我们相信，本技术在适当的环境中可以产生巨大的效果。在实施例中，转换后的程序与未转换的程序有相同的可观察行为。因此，本发明允许原始程序和模糊后程序之间有较小的相同点。

尽管本讨论主要是涉及阻碍对软件的反向工程，但是可以想到它还有其他用途，例如为软件对象（包括应用程序）打上标记。这就利用了任何单个模糊过程潜在的不同性质。厂商可以为每一个用户生成应用程序的不同模糊版本。如果发现盗版拷贝，厂商只需对比原始的模糊信息数据库，就能跟踪原始程序。

文中介绍的特定模糊转换并未穷尽。进一步的模糊体制可以被识别，并在目前的新模糊工具体系结构中使用。

在前述的介绍中，参照了带有已知同等性质的元素和整数，然后可以包括这些等价性质，就象它们被单独提出一样。

尽管本发明是通过实例和参照特定的实施例来描述的。需要理解的是，只要不脱离本发明的范围，还可以进行修正和改进。

说 明 书 附 图

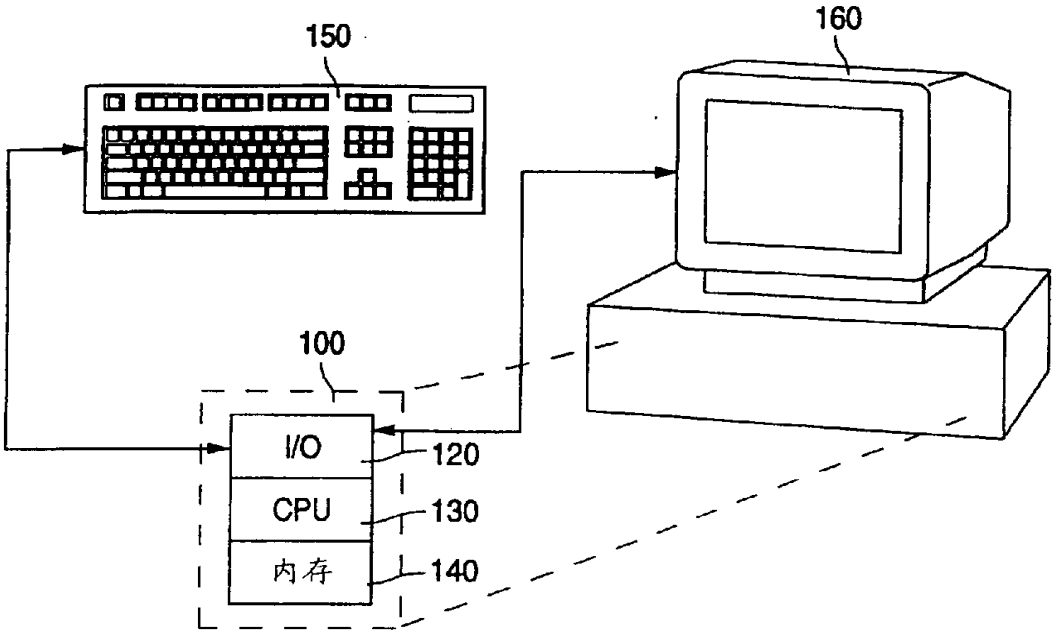


图 1

图 2a

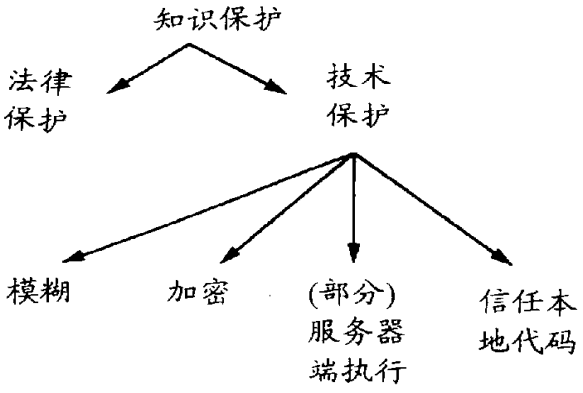


图 2b

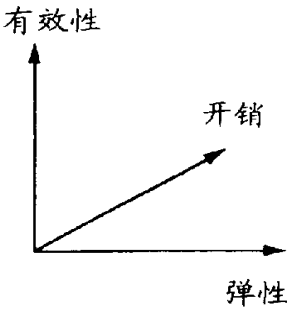


图 2c

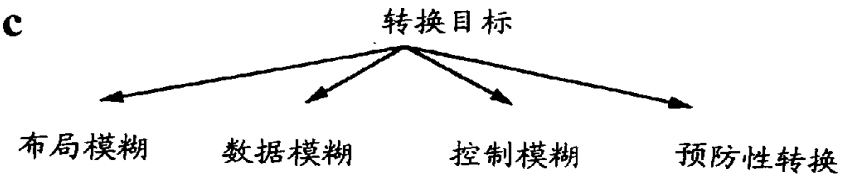


图 2e

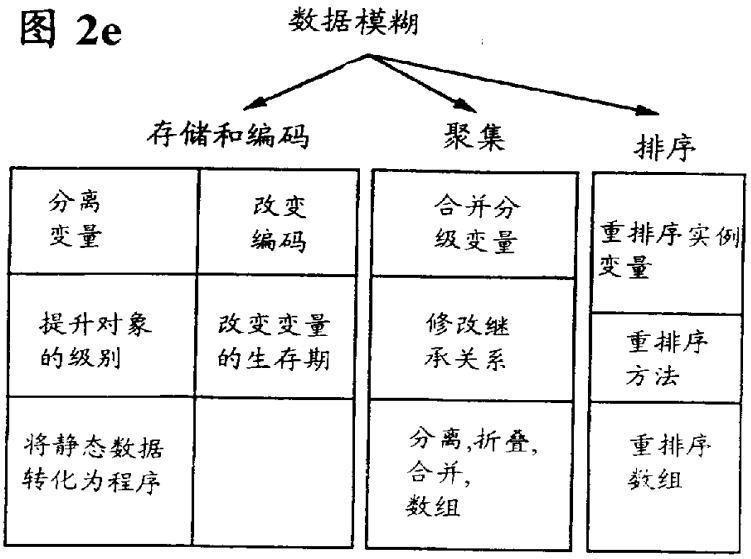
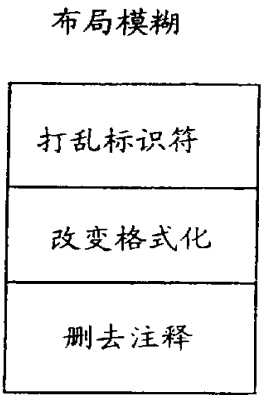


图 2d



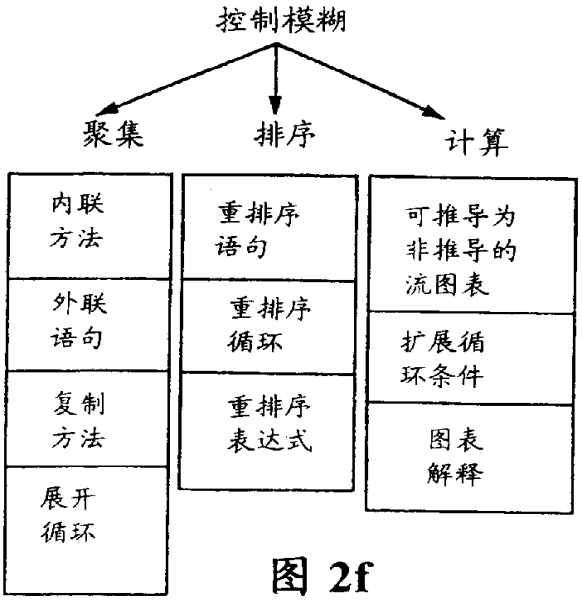


图 2f

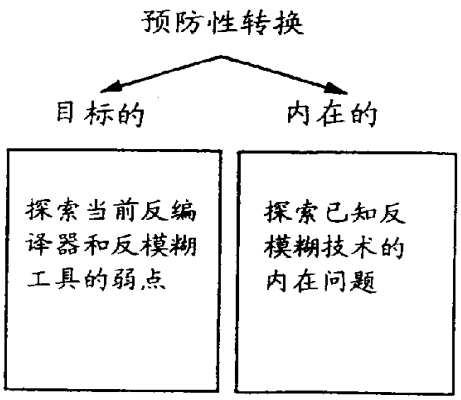


图 2g

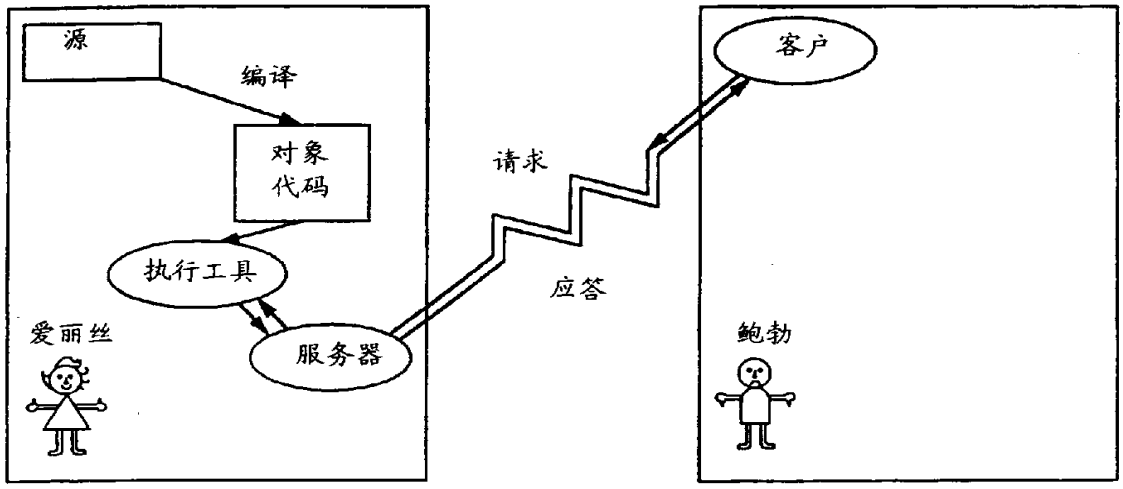

源
编译
对象
代码
执行工具
爱丽丝
服务器
请求
应答
客户
鲍勃

图 3a

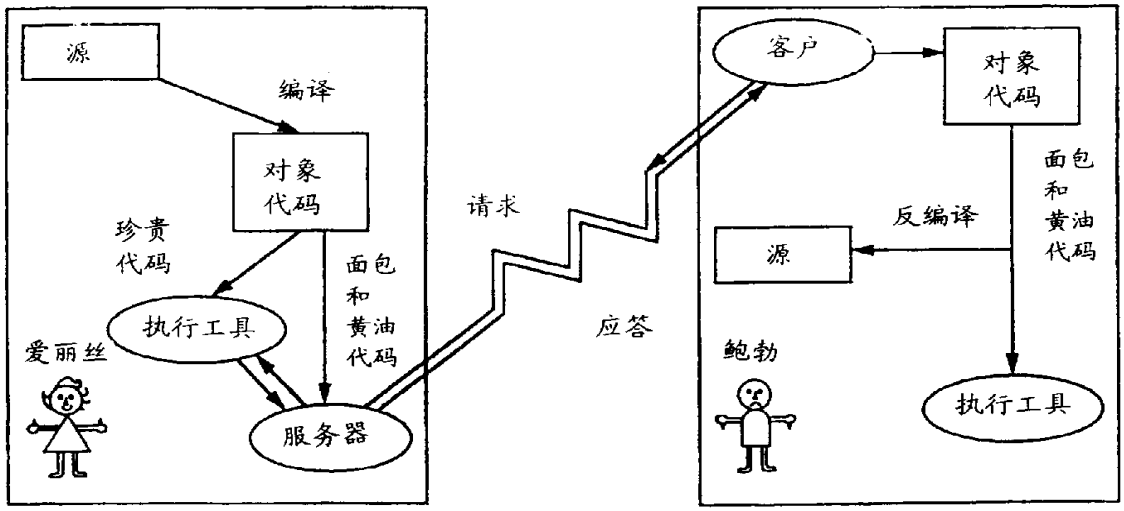

源
编译
对象
代码
珍贵
代码
执行工具
爱丽丝
面包
和
黄油
代码
服务器
请求
应答
客户
对象
代码
面包
和
黄油
代码
反编译
源
鲍勃
执行工具

图 3b

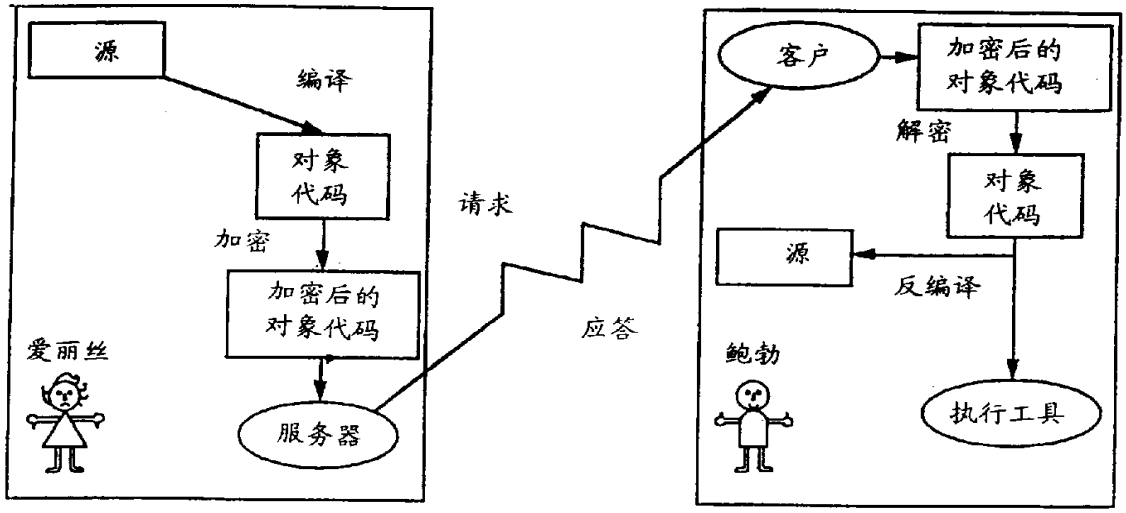

源
编译
对象
代码
加密
加密后的
对象代码
爱丽丝
服务器
请求
应答
客户
加密后的
对象代码
解密
对象
代码
源
反编译
鲍勃
执行工具

图 4a

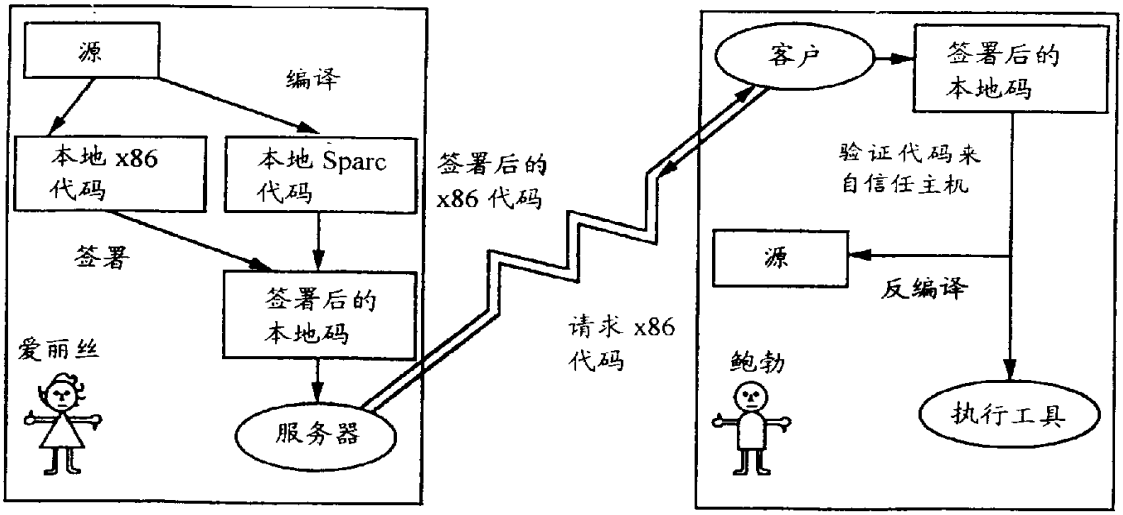

源
编译
本地 x86
代码
本地 Sparc
代码
签署
签署后的
本地码
爱丽丝
服务器
签署后的
x86 代码
请求 x86
代码
客户
签署后的
本地码
验证代码来
自信任主机
源
反编译
鲍勃
执行工具

图 4b

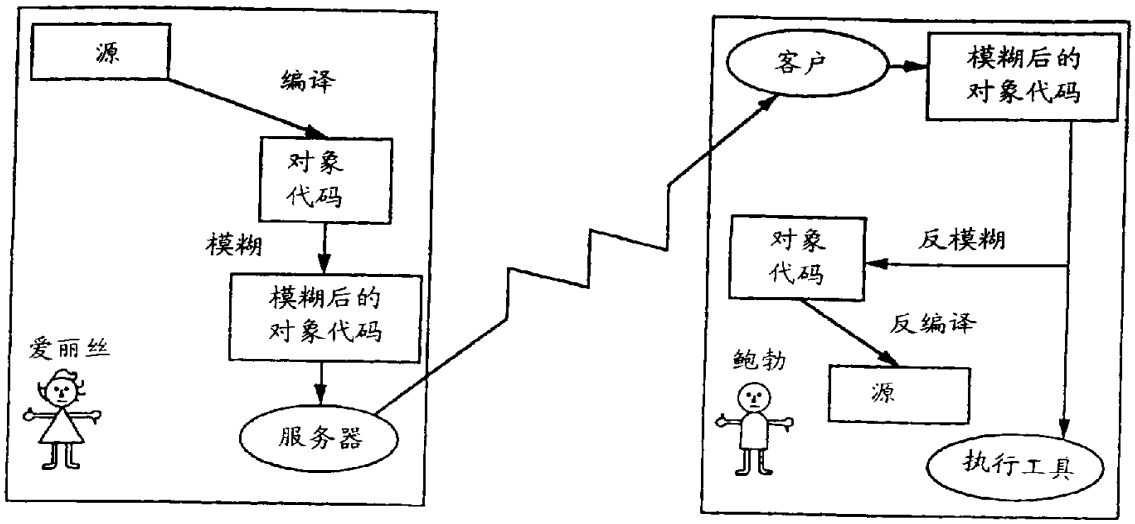

源
编译
对象
代码
模糊
模糊后的
对象代码
爱丽丝
服务器
客户
模糊后的
对象代码
对象
代码
反模糊
反编译
鲍勃
源
执行工具

图 5

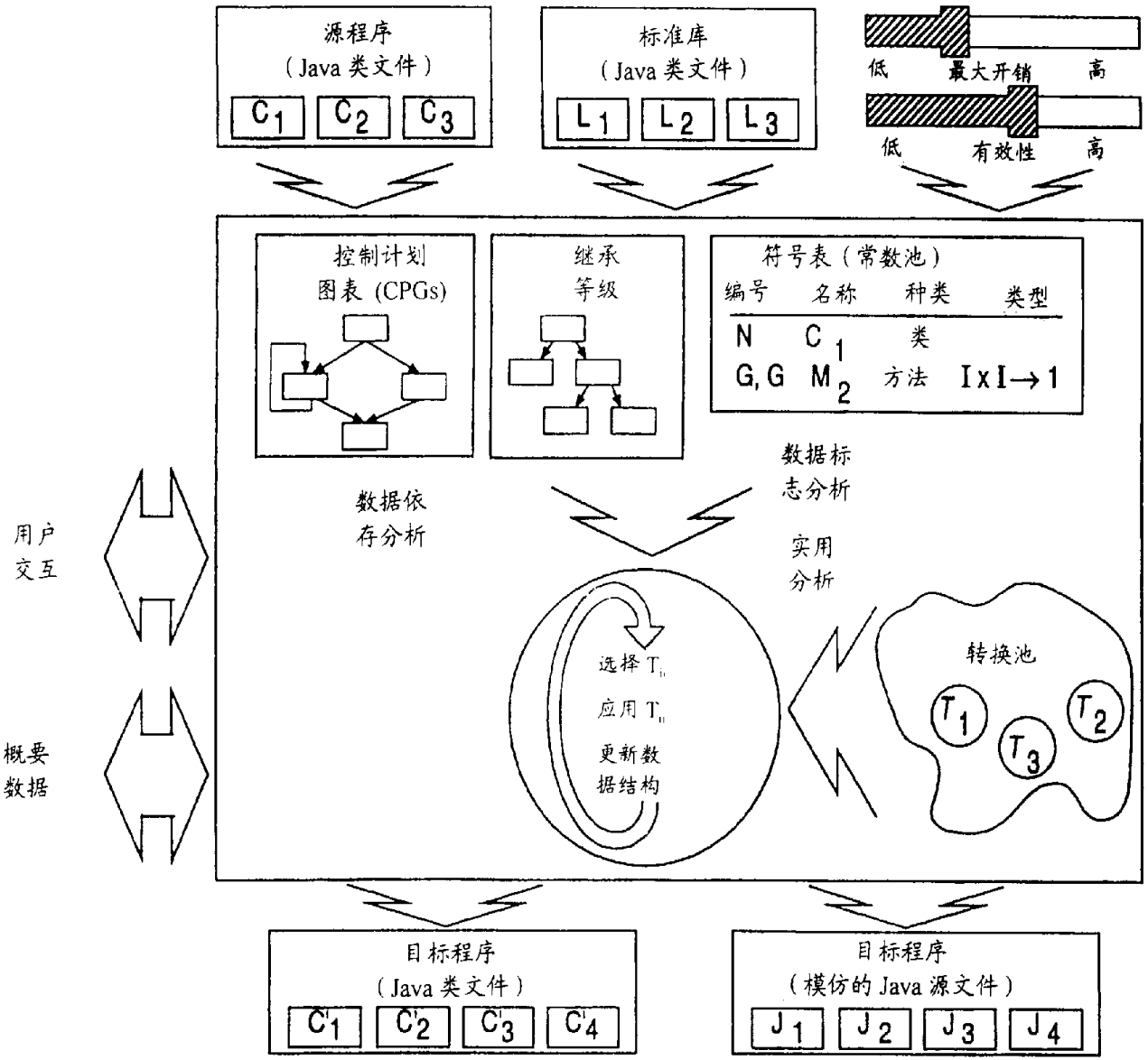

源程序
（Java 类文件）
C1 C2 C3
标准库
（Java 类文件）
L1 L2 L3
低 最大开销 高
低 有效性 高
控制计划
图表 (CPGs)
继承
等级
符号表（常数池）
编号 名称 种类 类型
N C1 类
G,G M2 方法 IxI→1
数据依
存分析
数据标
志分析
实用
分析
用户
交互
概要
数据
选择 Tu
应用 Tu
更新数
据结构
转换池
T1
T2
T3
目标程序
（Java 类文件）
C'1 C'2 C'3 C'4
目标程序
（模仿的 Java 源文件）
J1 J2 J3 J4

图 6

量度	量度名称	引用
μ_1	程序长度	Halstead
	*E(P)*随 P 中运算符和运算符数目的增加而增加	
μ_2	圈数复杂性	McCabe
	*E(F)*随 F 中的谓词数增加而增加	
μ_3	嵌套复杂性	Harrison
	*E(F)*随 F 中的条件嵌套等级的增加而增加	
μ_4	数据流的复杂性	Oviedo
	*E(F)*随 F 中的基础间模块的变量参考数增加而增加	
μ_5	扇形-入/出复杂性	Henry
	*E(F)*随 F 正式参数数目，以及 F 读取或更新的全局数据结构的数目增加而增加	
μ_6	数据结构复杂性	Munson
	*E(P)*随 P 中声明的静态数据结构的复杂性增加而增加。等级变量的复杂性是恒定的。数组的复杂性随维数和元素类型的增加而增加。记录的复杂性随其字段的数目和复杂性增加而增加。	
μ_7	OO 量度	Chidamber
	*E(C)*随 C_1 中的方法数(μ^a_7)，继承树中 C 的深度（与根的距离）(μ^b_7)，C_1 的直接子类数目(μ^c_7)，C 匹配的其它类的数目(μ^d_7)，对应于发往对象 C_1 的信息而执行的方法的数目(μ^e_7)，C 的方法没有参考相同实例变量组的等级(μ^f_7)的增加而增加。注意：μ^f_7 测量凝聚力；即模块的元素之间关系多紧密。	

* 两个类被能配对的条件是，其中一个使用另一个的方法或实例变量。

图 7

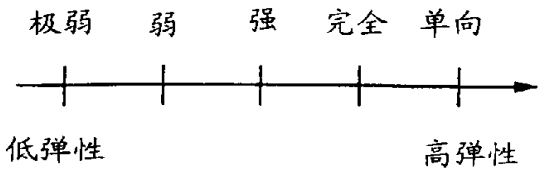


图 8a

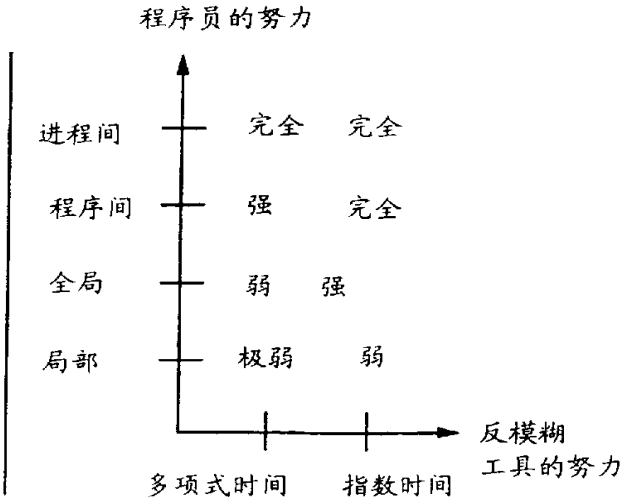


图 8b

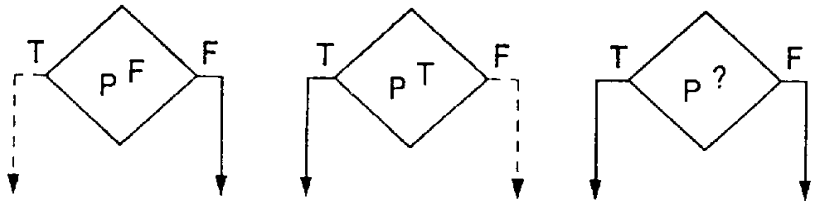


图 9

```
{    int v, a=5; b=6;
     v=11= a + b;
     if (b > 6) T ...
     if (random (1,5) < 0) F...
}
```

图 10a

```
{    int v, a=5; b=6;
     if (...) ...
          ⋮      (b is unchanged)
     if (b < 7) T  a++1
     v=11= (a > 5) 7v=b=b; v=b
}
```

图 10b

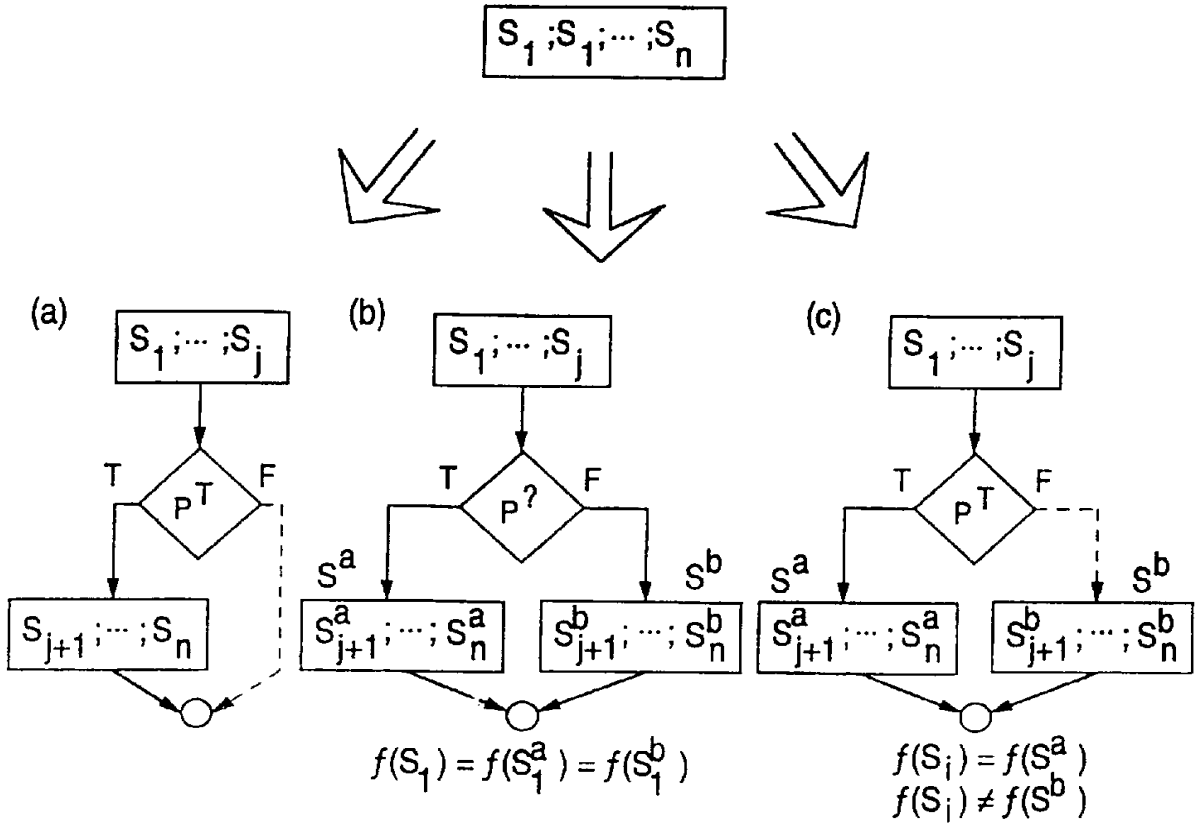

$S_1;S_1;\cdots;S_n$
(a)
$S_1;\cdots;S_j$
T
P T
F
$S_{j+1};\cdots;S_n$
(b)
$S_1;\cdots;S_j$
T
P ?
F
S^a
S^b
$S^a_{j+1};\cdots;S^a_n$
$S^b_{j+1};\cdots;S^b_n$
$f(S_1) = f(S^a_1) = f(S^b_1)$
(c)
$S_1;\cdots;S_j$
T
P T
F
S^a
S^b
$S^a_{j+1};\cdots;S^a_n$
$S^b_{j+1};\cdots;S^b_n$
$f(S_i) = f(S^a)$
$f(S_i) \neq f(S^b)$

图 11

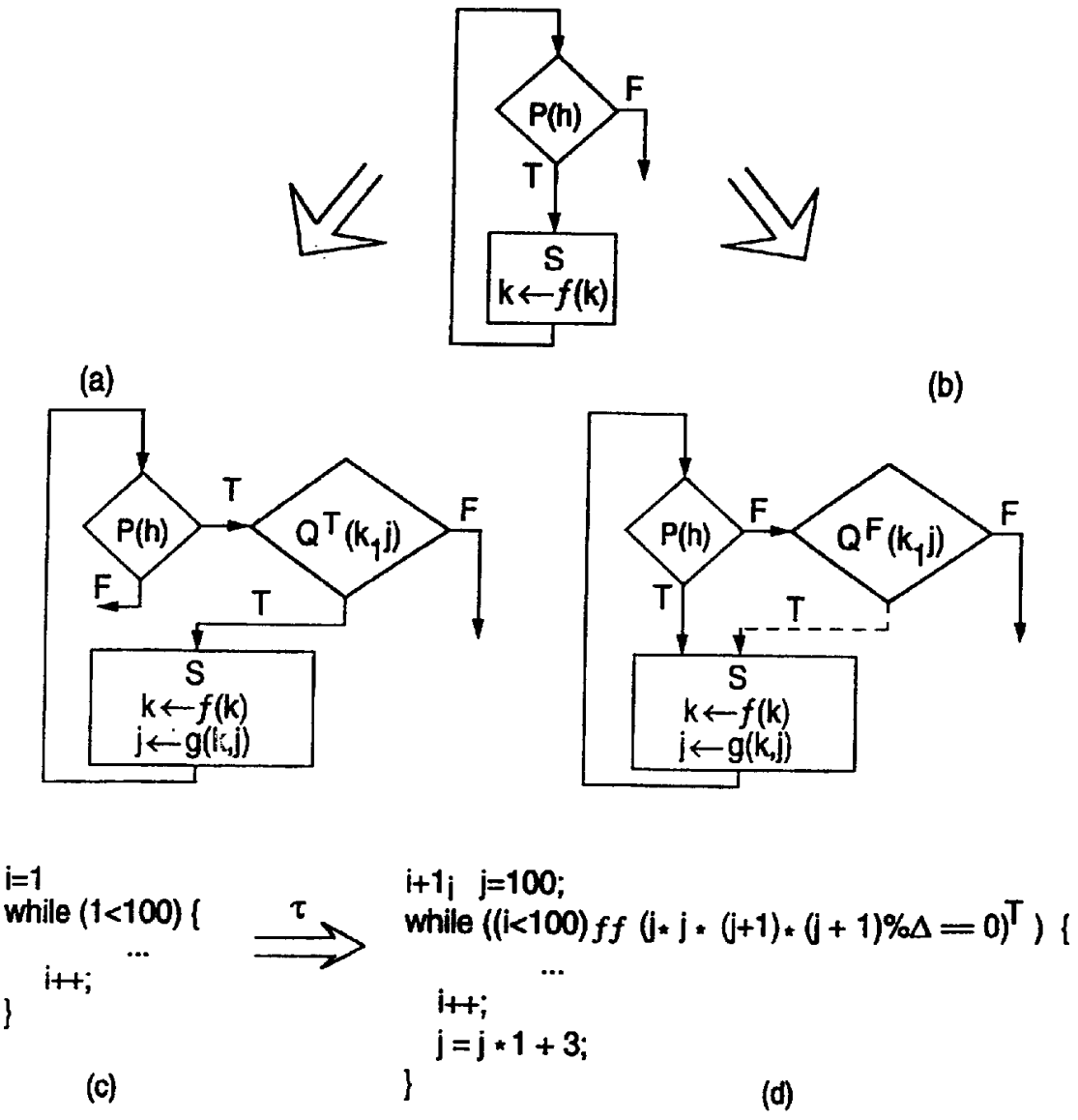

P(h)
F
T
S
k←f(k)
(a)
(b)
T
F
T
F
T
F
T
S
k←f(k)
j←g(k,j)
i=1
while (1<100) {
...
i++;
}
τ
i+1; j=100;
while ((i<100) ff (j* j * (j+1)* (j + 1)%Δ == 0)^T) {
...
i++;
j = j * 1 + 3;
}
(c)
(d)

图 12

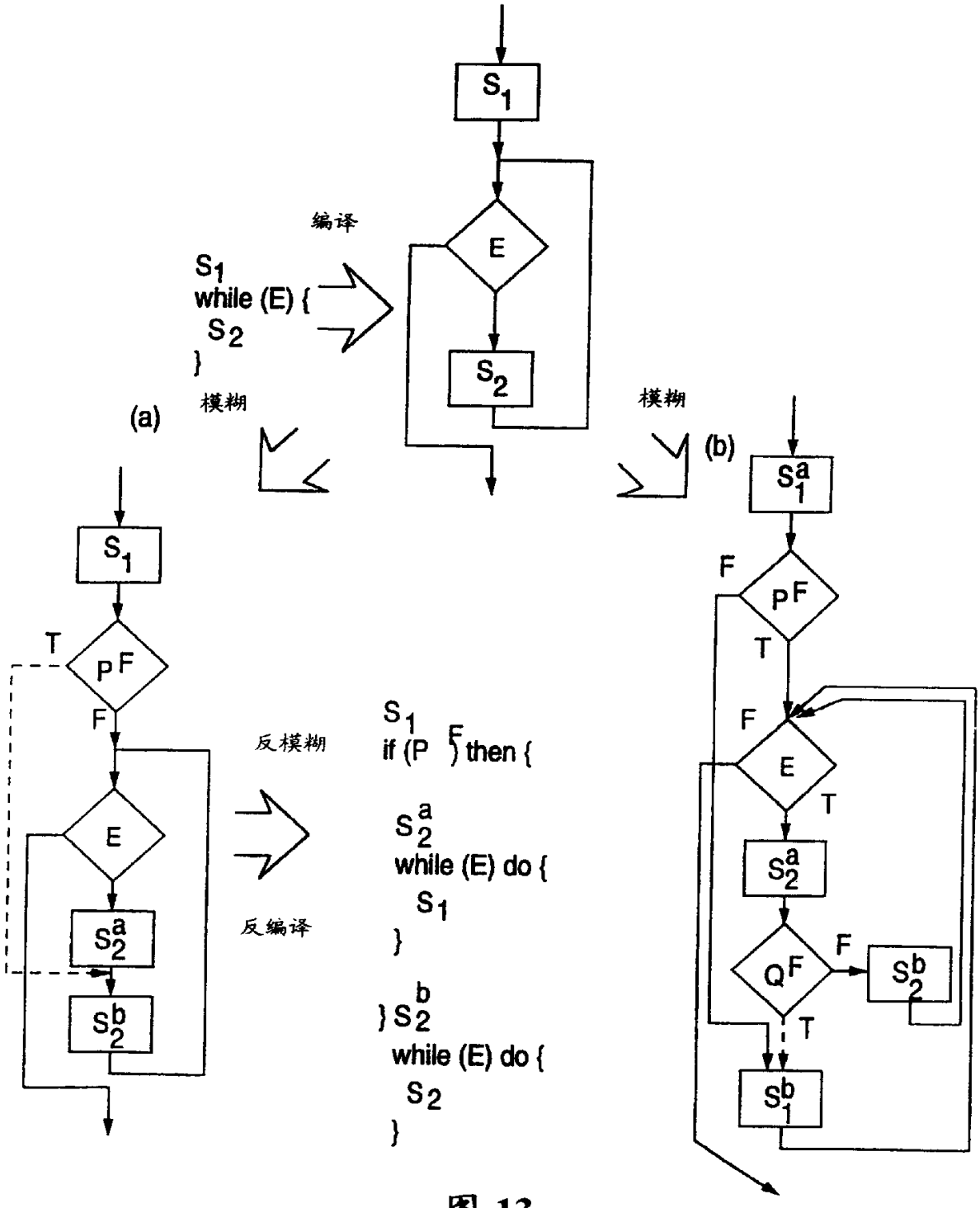

编译
S1
while (E) {
S2
}
模糊
模糊
(a)
(b)
反模糊
反编译
if (P F) then {
while (E) do {
}
} S2b
while (E) do {
}

图 13

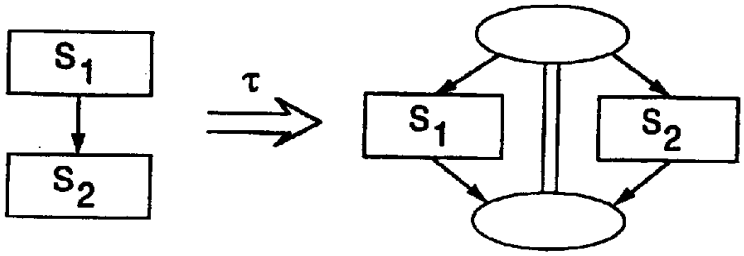


图 14

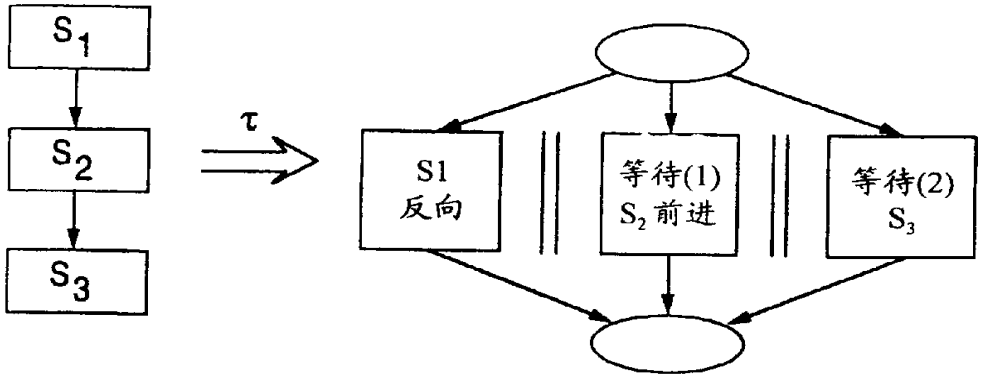


图 15

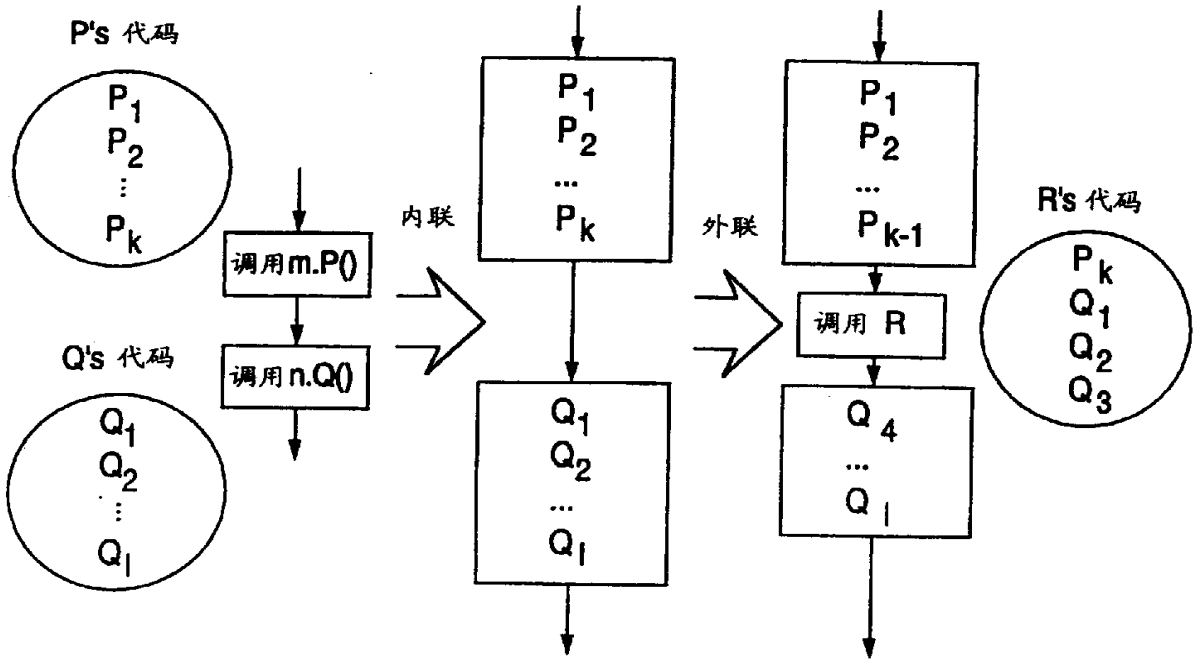

P's 代码
P1
P2
Pk
Q's 代码
Q1
Q2
Ql
调用m.P()
调用n.Q()
内联
P1
P2
Pk
Q1
Q2
Ql
外联
P1
P2
Pk-1
调用 R
Q4
Ql
R's 代码
Pk
Q1
Q2
Q3

图 16

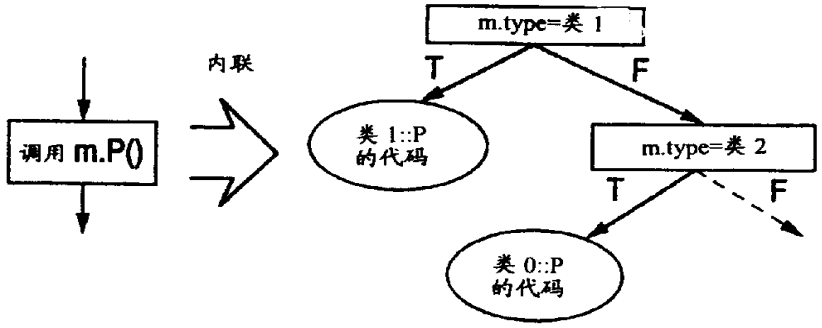

调用 m.P()
内联
m.type=类 1
T
F
类 1::P
的代码
m.type=类 2
T
F
类 0::P
的代码

图 17

```
class C {
  method M1 (T1 a) {
    S1^M1;... Sk^M1;
  }
  method M2 (T1 b; T2 c) {
    S1^k1;... Sm^k2 ;
  }
}

{ C x=new C;
  x.M1(a); x.M2(b, c); }
```

$\xrightarrow{\tau}$

```
class C' {
  method M (T1 a; T2 c; int V) {
    if (V==p) {S1^M1;...Sk^M1;}
    else      {S1^M1;...Sm^k2;}
  }
}

{ C' x=new C';
  x.M(a, c, V=p);
  x.M(b, c, V=g); }
```

图 18

```
class C {
  method m (int x)
    {S1 ... Sk}
}

{ C x = new C;
  x.m(8); ... x.m(7);
}
```

$\xrightarrow{\tau}$

```
class C1 {
  method m (int x)
    {S1^a ;...Sn^a }
  method m1 (int x)
    {S1^c ;...Sn^c }
}
class 02 inherits C1 {
  method M (int x)
    {S1^b ... Sk^b }
}

{ C1 x ;
  if (P7) x=new C1 else x=new C2;
  x.m(5); ...; x.m1(7);
}
```

图 19

图 20a

```
for (i=1,i<=n,i++)                    τ      for(I=1, I<=n, I+=64)
  for (j=1, j<=n,j++)          ⟹              for (J=1,J<=n,J+=64)
    a[1,j]=b[j,I]                                for (i=I,i<=min(I+63,n),i++)
                                                   for (j_J,j<=min(J+65,n),j++)
                                                     a[i,j]=b[j,i]
```

图 20b

```
for(i=2,i<(n-1),i++)                  τ      for (i=2,i<(n-2),i+=2) {
  a[i] +=a[1-i]==[i+1]         ⟹                a[i] += n[i-1]=a[i+1];
                                                 a[i+1] += a[i]=a[i+2];
                                               };
                                               if (((n-2) % 2) == 1)
                                                 x[n-1] += a[n-2]=a[n]
```

图 20c

```
for(i=1,i<n,i++) {                    τ      for (i=1,i<n,i++)
  a[i] += c;                   ⟹                a[i] += c;
  x[i+1]=d+x[i+1]=a[i]                         for (i=1, i<n, i++)
}                                                x[i+i] <d+x[i+1]=a[i]
```

图 21a

g(V) p	g(V) q	f(p,q) V	2p + q
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

图 21b

VAL[p,q]		p = 0	p = 1
q	0	0	1
	1	1	0

图 21c

AND[A,B]		A = 0	A = 1	A = 2	A = 3
B	0	3	0	0	0
	1	3	1	2	3
	2	0	2	1	3
	3	3	0	0	3

图 21d

OR[A,B]		A = 0	A = 1	A = 2	A = 3
B	0	3	1	2	3
	1	1	1	2	2
	2	2	2	1	1
	3	0	1	2	0

```
(1)  bool A,B,C;                      T      (1')  short a1,a2,b1,b2,c1,c2;
(2)  A = True;                               (2')  a1=0; a2=1;
(3)  B = False;                              (3')  b1=0; b2=0;
(4)  C = False;                              (4')  c1=1; c2=1;
(5)  C = A & B;                              (5')  x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(6)  C = A & B;                              (6')  c1=(a1 ^ a2) & (b1 ^ b2); c2=0
(7)  C = A | B;                              (7')  x=OR[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(8)  if (A) ...;                             (8')  x=2*a1+a2; if ((x==1) || (x==2) ...;
(9)  if (B) ...;                             (9')  if (b1 ^ b2) ...;
(10) if (C) ...;                             (10') if (VAL[c1,c2]) ...;
```

图 21e

```
String G (int n) {
    int i=0,k;
    String B;
    while (i) {
        L1: if (n==1) {S[i++]="A";k=0;goto L6};
        L2: if (n==2) {S[i++]="B";k= -2 ;goto L6};
        L3: if (n==3) {S[i++]="C";goto L8};
        L4: if (n==4) {S[i++]="K";goto L9};
        L5: if (n==5) {S[i++]="C";goto L11};
              if (n>12) goto L1;
        L6: if (k++<=2) {S[i++]="A";goto L6} else goto L8;
        L8: return S;
        L9: S[i++]="C"; goto L10;
        L10: S[i++]="B"; goto L8:
        L11: S[i++]="C"; goto L12;
        L12: goto L10;
    }
}
```

图 22

图 23a

$$Z(X + r,Y) = 2^{32} \cdot Y + (r + X) = Z(X,Y) + r$$
$$Z(X,Y + r) = 2^{32} \cdot (Y + r) + X = Z(X,Y) + r \cdot 2^{32}$$
$$Z(X \cdot r,Y) = 2^{32} \cdot Y + X + r = Z(X,Y) + (r - 1) \cdot X$$
$$Z(X,Y \cdot r) = 2^{32} \cdot Y \cdot r + X = Z(X,Y) + (r - 1) \cdot 2^{32} \cdot Y$$

图 23b

```
(1) int X=45, Y=95;                  (1') long Z=167759066119551045;
(2) X += 5;          τ               (2') Z += 5;
(3) Y += 11;         ⟹              (3') Z += 47244640256;
(4) X *= c;                          (4') z += (c-1)* (Z & 4294967295);
(5) Y *= d;                          (5') Z += (d-1)* (Z & 18446744069414584320);
```

```
(1)  int A[9];
(2)  A[i] = ...;
       ...
       ...
(3)  int B[8],C[19];
(4)  B[i] = ...;
(5)  C[i] = ...;
       ...
(6)  int D[9]
(7)  for(i=o;i<=B;i++)
       D[i]=2 * D[i+1];
       ...
       ...
       ...
(8)  int E[2,2];
(9)  for(i=Q;i<=2;i ++)
       for(j=0;i<=2;i++)
         swap(E(i,j], E[j,i]);
```

τ ⟹

```
(1')  int A1[4],A2[4];
(2')  if ((i%2)==0) A1[i/2] =...
      else A2[i/2]=...;
        ...
(3')  int BC[20];
(4')  BC[3*i] = ...;
(5')  BC[i/2*3+1+i%2] = ...;
(6')  int D1[1,4];
(7')  for(j=0;j<=1;j++)
        for(k=0;k<=4;k++)
          if (k==4)
            D1[j,k]=2*D1[j+1,0];
          else
            D1[j,k]=2*D1[j,k+1];
        ...
(8')  int E1[8]
(9')  for(i=0;1<=8;i++)
        swap(E[i], E[3=(i%3)+i/3]);
```

图 24a

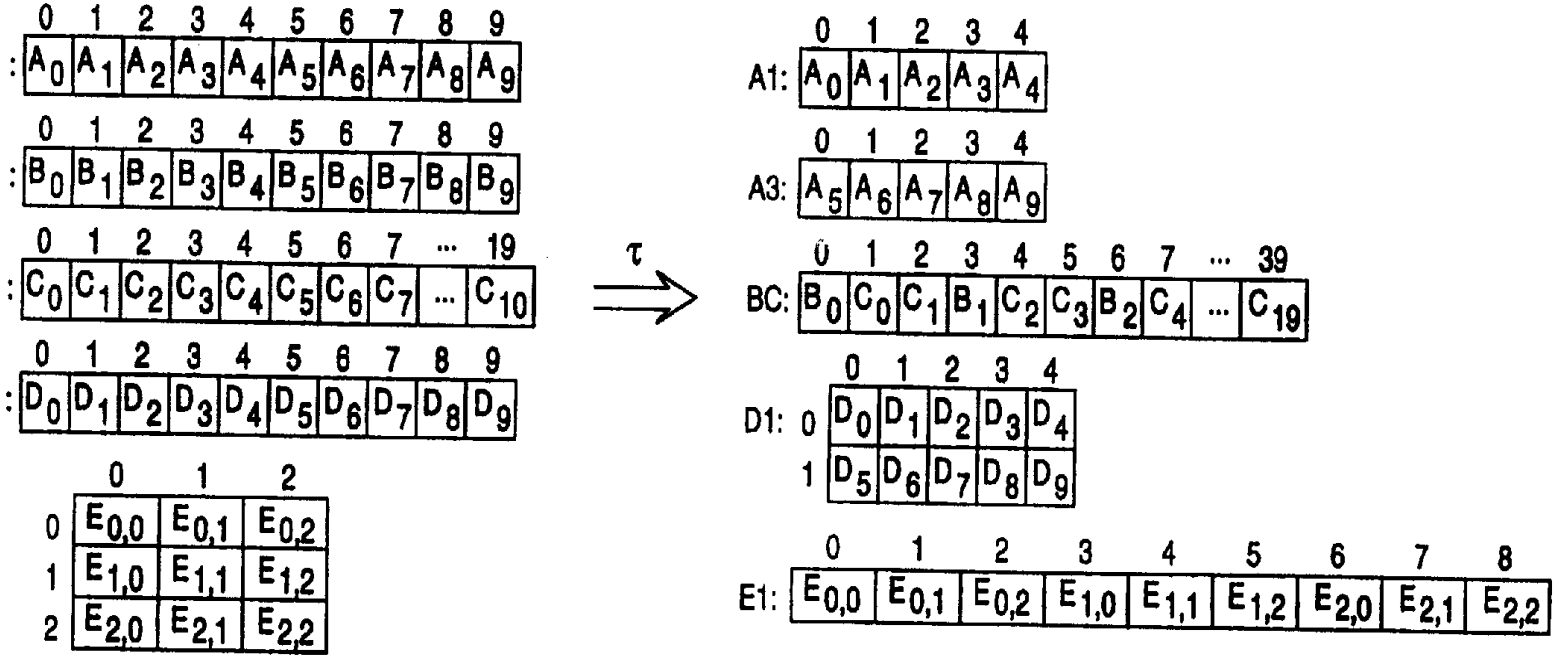

0 1 2 3 4 5 6 7 8 9
: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9
0 1 2 3 4 5 6 7 8 9
: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9
0 1 2 3 4 5 6 7 ... 19
: C0 C1 C2 C3 C4 C5 C6 C7 ... C10
0 1 2 3 4 5 6 7 8 9
: D0 D1 D2 D3 D4 D5 D6 D7 D8 D9
0 1 2
0 E0,0 E0,1 E0,2
1 E1,0 E1,1 E1,2
2 E2,0 E2,1 E2,2
τ
A1: 0 1 2 3 4 A0 A1 A2 A3 A4
A3: 0 1 2 3 4 A5 A6 A7 A8 A9
BC: 0 1 2 3 4 5 6 7 ... 39 B0 C0 C1 B1 C2 C3 B2 C4 ... C19
D1: 0 1 2 3 4 0 D0 D1 D2 D3 D4 1 D5 D6 D7 D8 D9
E1: 0 1 2 3 4 5 6 7 8 E0,0 E0,1 E0,2 E1,0 E1,1 E1,2 E2,0 E2,1 E2,2

图 24b

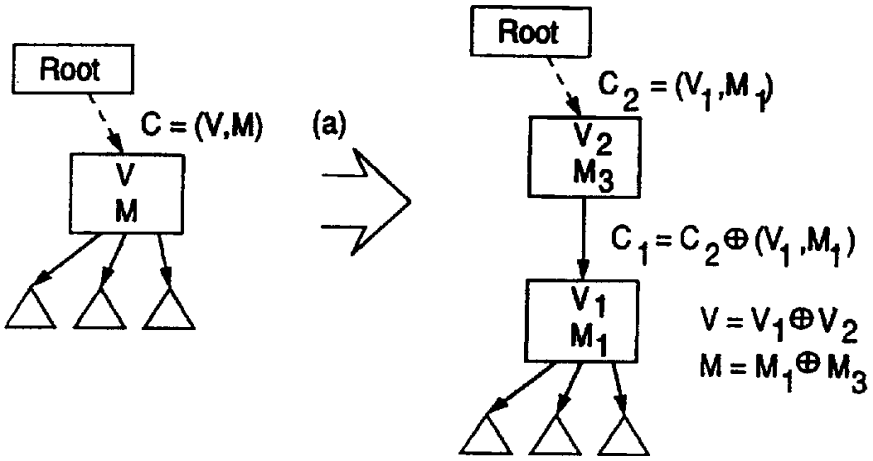

Root
C = (V,M)
V
M
(a)
Root
C2 = (V1,M1)
V2
M3
C1 = C2⊕(V1,M1)
V1
M1
V = V1⊕V2
M = M1⊕M3

图 25a

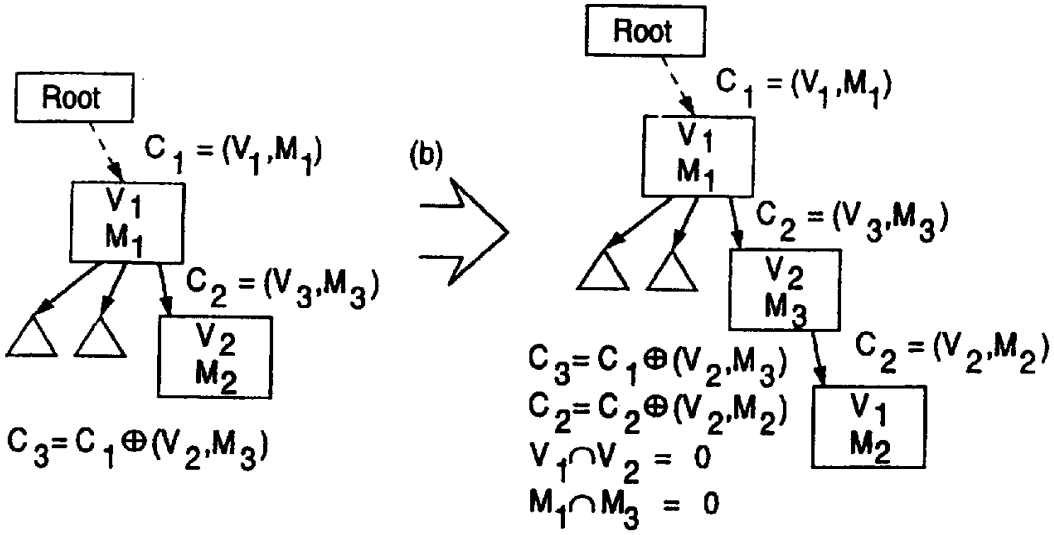

Root
C1 = (V1,M1)
V1
M1
C2 = (V3,M3)
V2
M2
C3= C1⊕(V2,M3)
(b)
Root
C1 = (V1,M1)
V1
M1
C2 = (V3,M3)
V2
M3
C2 = (V2,M2)
V1
M2
C3= C1⊕(V2,M3)
C2= C2⊕(V2,M2)
V1∩V2 = 0
M1∩M3 = 0

图 25b

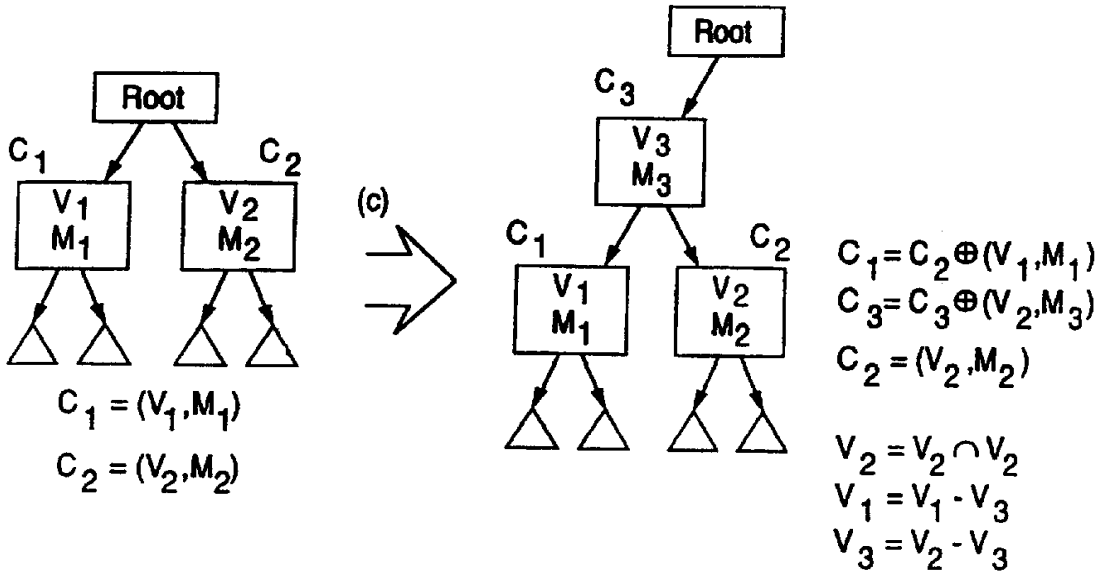

Root
C1
C2
V1
M1
V2
M2
(c)
C1 = (V1,M1)
C2 = (V2,M2)
Root
C3
V3
M3
C1
C2
V1
M1
V2
M2
C1 = C2 ⊕ (V1,M1)
C3 = C3 ⊕ (V2,M3)
C2 = (V2,M2)
V2 = V2 ∩ V2
V1 = V1 - V3
V3 = V2 - V3

图 25c

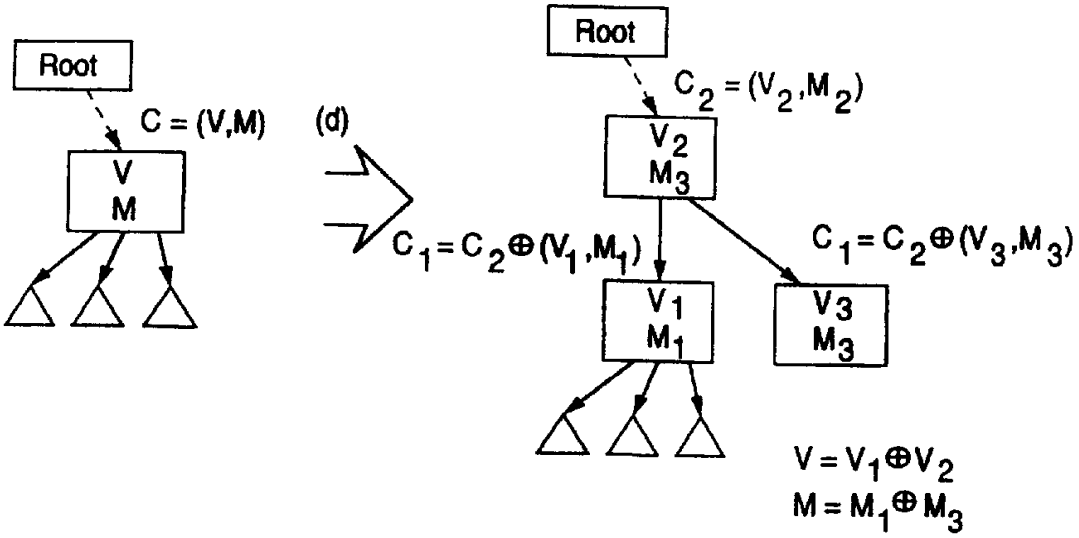
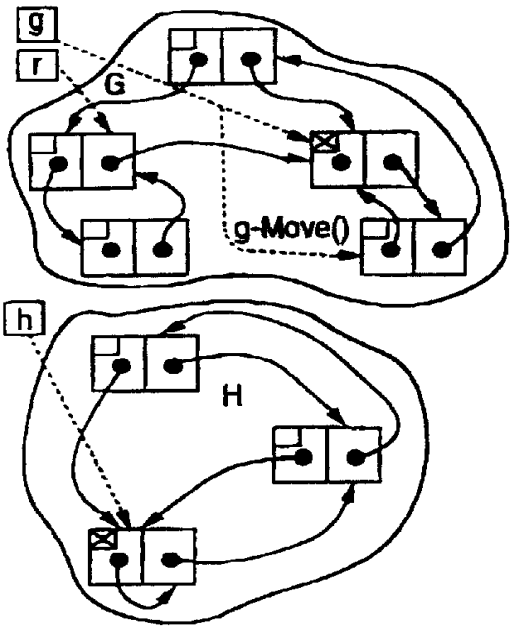

Root
C = (V,M)
V
M
(d)
Root
C2 = (V2,M2)
V2
M3
C1 = C2 ⊕ (V1,M1)
C1 = C2 ⊕ (V3,M3)
V1
M1
V3
M3
V = V1 ⊕ V2
M = M1 ⊕ M3

图 25d




```
Node g, h;
method P(...,Node f) {
  /* 1 */ g = g.Move();
          h = h.Move();
  /* 2 */ h = h.Insert(new Node);
          ⋮
  /* 3 */ x.R(...,f.Move());
          ⋮
  /* 4 */ if (f==g) ?...
  /* 5 */ if (g==h) F...
          ⋮
  /* 6 */ f.Token=False;
          g.Token=True;
  /* 7 */ if (f.Token)? ...
          ⋮
  /* 8 */ f.Token=True;
          h.Token=False;
  /* 9 */ if (f.Token) T ...
}
```

图 26

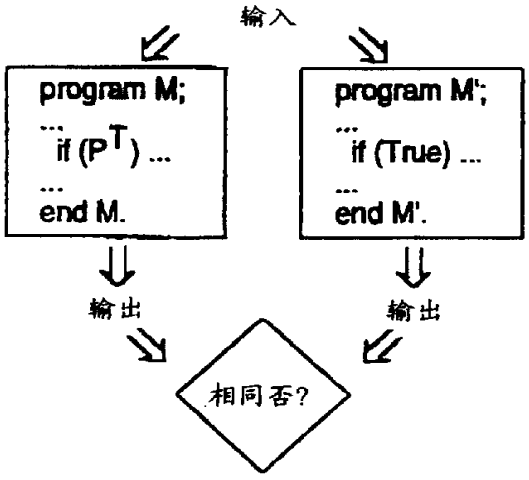


图 30

```
thread S {
  int R;
  while (1) {
    R = random(1,C);
    X = R*R;
    sleep(3);
  }
}
```

```
thread T {
  int R;
  while (1) {
    R = random(1,C);
    X = 7*R*R;
    sleep(2);
    X *= X;
    sleep(5);
  }
}
```

```
int X, Y;
const C = sqrt(maxint)/10;
main () {
  S.run(); T.run();
    ...
  if ((Y - 1) == X) F <= P
    ...
}
```

图 27

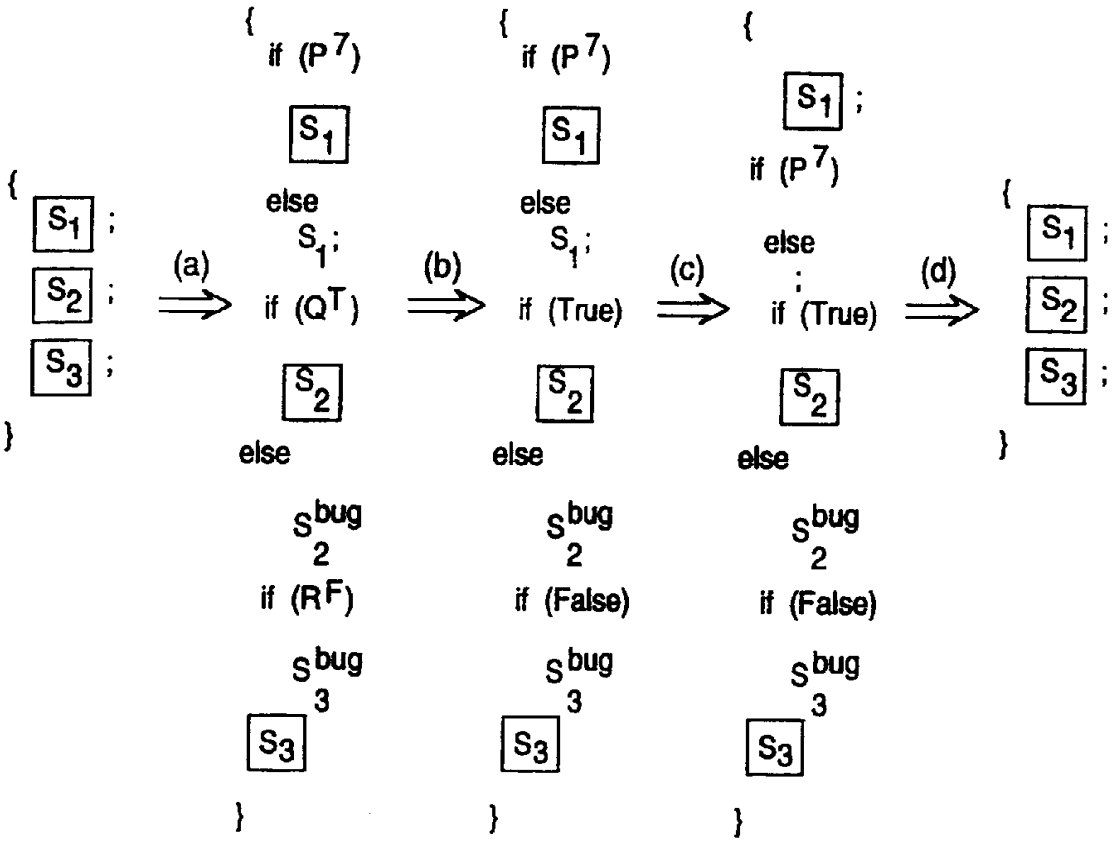


图 28

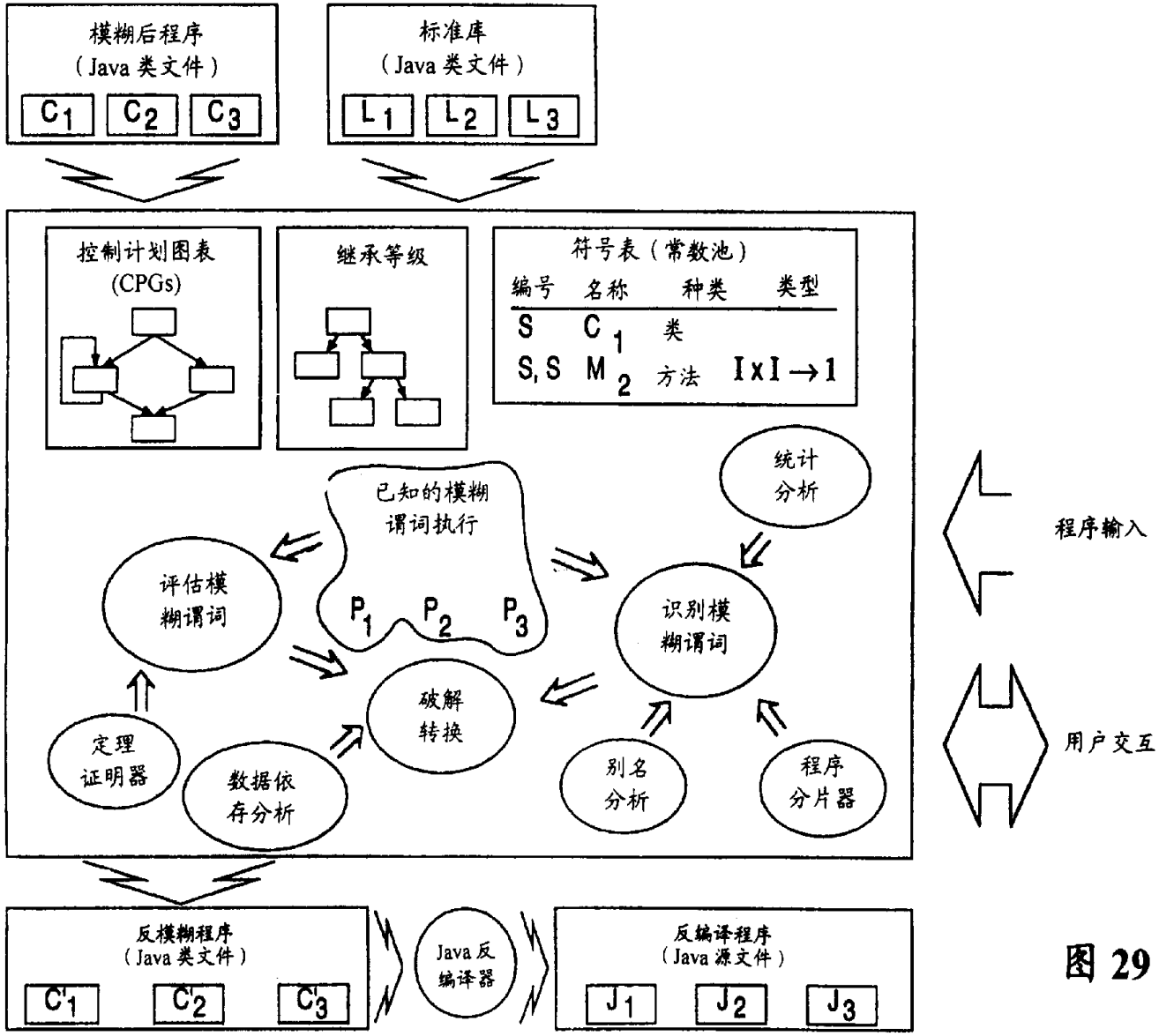

模糊后程序
（Java 类文件）
C1 C2 C3
标准库
（Java 类文件）
L1 L2 L3
控制计划图表
(CPGs)
继承等级
符号表（常数池）
编号 名称 种类 类型
S C1 类
S,S M2 方法 I x I → I
统计
分析
已知的模糊
谓词执行
P1 P2 P3
评估模
糊谓词
识别模
糊谓词
破解
转换
定理
证明器
数据依
存分析
别名
分析
程序
分片器
程序输入
用户交互
反模糊程序
（Java 类文件）
C'1 C'2 C'3
Java 反
编译器
反编译程序
（Java 源文件）
J1 J2 J3

图 29

模糊			质量			量度	节
目标	操作	转换	有效性	弹性	开销		
布局		打乱标识符	中	单向	免费		5.5
		改变格式化	低	单向	免费		5.5
		删除注释	高	单向	免费		5.5
控制	计算	插入死的或无关的数据	由模糊谓词的性质和构架插入时的嵌套深度决定			μ1, μ2, μ3	6.2.1
		扩展循环条件				μ1, μ2, μ3	6.2.2
		可推导变为不可推导				μ1, μ2, μ3	6.2.3
		增加冗余运算符				μ1	6.2.6
		删除编程习语	中	强	中	μ1	6.2.4
		列表解释	高	强	昂贵	μ1	6.2.5
	聚集	内联的方法	中	单向	免费	μ1	6.3.1
		外联的语句	中	强	免费	μ1	6.3.1
		交叉方法	由模糊谓词的性质决定			μ1, μ2, μ3	6.3.2
		复制方法				μ1, μ7	6.3.3
		块循环	低	弱	免费	μ1, μ2	6.3.4
		展开循环	低	弱	便宜	μ1	6.3.4
		循环空隙	低	弱	免费	μ1, μ2	6.3.4
	排序	对语句重排序	低	单向	免费		6.4
		对循环重排序	低	单向	免费		6.4
		对表达式重排序	低	单向	免费		6.4

图31a-1

模糊			质量			量度	节
目标	操作	转换	有效性	弹性	开销		
数据	存储&编码	改变编码	由编码函数的复杂度决定			μ1	7.1.1
		提升对象级别	低	强	免费		7.1.2
		改变变量生存期	低	强	免费	μ4	7.1.2
		分离变量	取决于原始变量分为多少变量			μ1	7.1.3
		将静态数据转化为过程数据	由生成函数的复杂度决定			μ1, μ2	7.1.4
	聚集	合并级别变量	低	弱	免费	μ1	7.2.1
		因子类	中	↑	免费	μ1, $\mu 7^{b,c,e}$	7.2.3
		插入假类	中	↑	免费	μ1, $\mu 7^{b,c}$	7.2.3
		重分解类	中	↑	免费	μ1, $\mu 7^{b,c,e}$	7.2.3
		分解数组	↑	弱	免费	μ1, μ2, μ6	7.2.2
		合并数组	↑	弱	免费	μ1, μ3	7.2.2
		折叠数组	↑	弱	便宜	μ1, μ2, μ3, μ6	7.2.2
		展平数组	↑	弱	免费		7.2.2
	排序	对方法和实例变量重排序	低	单向	免费		7.3
		对数组重排序	低	弱	免费		7.3

图31a-2

模糊			质量			量度	节
目标	操作	转换	有效性	弹性	开销		
	目标的	HoseMocha	低	极弱	免费	μ1	9
	内在的	增加有别名的正式值以防止分片	中	强	免费	μ1, μ5	9.4
预防性		增加变量依存以防止分片	由模糊谓词的性质决定			μ1	9.4
		增加假的数据依存	中	弱	便宜	μ1	9.1.1
		使用带有副效果的模糊谓词	中	弱	免费	μ1	9.5
		产生使用较难定理的模糊谓词	↑	↑	↑	μ1	9.5

图31b

模糊构架	性质		节
	弹性	开销	
依靠对库函数的调用创建	极弱	由库函数的开销决定	6.1.1
依靠局部(基本块内的)信息创建	极弱	免费... 便宜	6.1.1
依靠全局(基本块间的)信息创建	弱	免费... 便宜	6.1.1
依靠程序间和别名信息创建	完全	便宜... 昂贵	8.1
依靠进程交互和调度创建	完全	便宜... 昂贵	8.2

图32