(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 101697121 A
(43)申请公布日 2010.04.21

(21)申请号 200910073094.4

(22)申请日 2009.10.26

(71)申请人 哈尔滨工业大学
地址 150001 黑龙江省哈尔滨市南岗区西大直街 92 号

(72)发明人 王甜甜 马培军 苏小红 王宇颖

(74)专利代理机构 哈尔滨市松花江专利商标事务所 23109
代理人 张宏威

(51)Int.Cl.
G06F 9/44(2006.01)
G06F 17/30(2006.01)

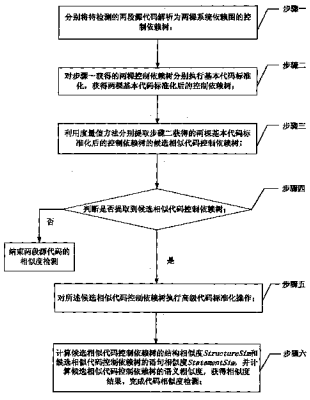
权利要求书 2 页 说明书 14 页 附图 4 页

(54)发明名称

一种基于程序源代码语义分析的代码相似度检测方法

(57)摘要

一种基于程序源代码语义分析的代码相似度检测方法，涉及计算机程序分析技术和计算机软件的重复代码检测方法。它解决了现有的对语法表示不同但语义相似的代码的相似度检测准确度低、计算复杂度高，以及无法实现大规模程序代码相似度检测的问题。它的方法为：分别将待检测的两段源代码解析为两棵系统依赖图的控制依赖树，并分别执行基本代码标准化；利用度量值方法提取两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树；对提取的候选相似代码执行高级代码标准化操作；计算语义相似度，获得相似度结果，完成代码相似度检测；本发明适用于源代码剽窃检测、软件组件库查询、软件缺陷检测以及程序理解等场合。



CN 101697121 A

1. 一种基于程序源代码语义分析的代码相似度检测方法，其特征是：它由以下步骤完成：

步骤一、分别将待检测的两段源代码解析为两棵系统依赖图的控制依赖树；

步骤二、对步骤一获得的两棵控制依赖树分别执行基本代码标准化，获得两棵基本代码标准化后的控制依赖树；

步骤三、利用度量值方法分别提取步骤二获得的两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树；

步骤四、判断是否提取到候选相似代码控制依赖树，如果判断结果为是，则执行步骤五，如果结果为否，则结束两段源代码的相似度检测；

步骤五、对所述候选相似代码控制依赖树执行高级代码标准化操作；

步骤六、计算候选相似代码控制依赖树的结构相似度 StructureSim 和候选相似代码控制依赖树的语句相似度 StatementSim，并根据公式：

$SemanticSim = \lambda_{stru} * StructureSim + \lambda_{stat} * StatementSim$

计算候选相似代码控制依赖树的语义相似度，获得语义相似度结果，完成代码相似度检测；式中：λ_{stru} 为结构相似度的权值、λ_{stat} 为语句相似度的权值。

2. 根据权利要求 1 所述的一种基于程序源代码语义分析的代码相似度检测方法，其特征在于步骤二中所述基本代码标准化的方法为：采用基于系统依赖图的程序标准化方法对步骤一所述控制依赖树进行结构语义等价的基本代码标准化转换；所述基本代码标准化转换包括：复合表达式的标准化转换、表达式的标准化转换和基本控制结构的标准化转换。

3. 根据权利要求 1 所述的一种基于程序源代码语义分析的代码相似度检测方法，其特征在于提取步骤三中利用度量值方法分别提取两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树的方法为：

步骤 A、分别在所述两棵基本代码标准化后的控制依赖树上提取各模块的规模、结构和复杂度特征信息，并分别统计各个模块的度量向量，获得两段代码的度量向量集；所述度量向量集包括：节点个数、运算符个数、选择结构个数、循环结构个数、赋值节点个数、特殊数据类型个数、控制依赖树中最长路径长度、系统函数调用个数、递归调用路径长度和控制依赖边个数；

步骤 B、将步骤 A 获得两段代码的度量向量集，通过公式：

$$sim(v,v') = 1 - \sqrt{\frac{1}{n}\sum_{i=1}^{n}((v_i - v_i')/\max(v_i, v_i'))^2}$$

计算两段代码的度量向量集的相似度 sim(v,v')；式中 i = 1,2,3, ……,10；n = 10；

步骤 C、判断步骤 B 获得的度量向量的相似度 sim(v, v') 是否大于预先设定的度量向量相似度阈值 T_v，如果判断结果为是，则提取此两段代码做为候选相似代码；如果判断结果为否，则结束两段源代码相似度检测。

4. 根据权利要求 1 所述的一种基于程序源代码语义分析的代码相似度检测方法，其特征在于步骤五中的高级代码标准化的方法为：分别对两棵候选相似代码控制依赖树执行标准化转换，所述标准化转换包括：高级控制结构标准化、消除冗余代码、变量重命名、语句重排列和函数调用标准化。

5. 根据权利要求 1 所述的一种基于程序源代码语义分析的代码相似度检测方法，其特征在于步骤六中候选相似代码控制依赖树的结构相似度 StructureSim 是根据公式：

StrcutureSim = StructureMatching(S,T)/|S|

获得的，式中：S 和 T 是两棵候选相似代码的结构树，StructureMatching(S,T) 为 S 与 T 的最大匹配节点对个数，|S| 为候选相似代码的结构树 S 的节点总数。

6. 根据权利要求 1 所述的一种基于程序源代码语义分析的代码相似度检测方法，其特征在于步骤六中候选相似代码控制依赖树的语句相似度 StatementSim 是根据公式：

StatementSim = Σ value(vi)/|S|

获得的，式中：S 为候选相似代码控制依赖树，value(vi) 为 S 中节点的表达式相似度，|S| 为候选相似代码控制依赖树 S 的节点总数。

一种基于程序源代码语义分析的代码相似度检测方法

技术领域

[0001] 本发明涉及计算机程序分析技术和计算机软件的重复代码检测方法，具体涉及一种语义相似的重复代码检测方法。

背景技术

[0002] 重复代码（也称为克隆代码）检测是计算机软件开发和维护活动中一项重要的任务，在源代码剽窃检测、软件组件库查询、软件缺陷检测、程序理解等多个领域中都有广泛的应用。

[0003] 目前已有的重复代码检测方法主要可以划分为：基于文本的方法、结构分析方法、基于度量值的方法、基于相似子图的方法。其中，前两种方法只能检测完全相同或只有很小改动的代码，如标识符重命名或注释改变。而基于度量值的方法虽然计算简单、复杂度较低，但检测的准确度差，且通常不会考虑函数调用的多样化，不能识别模块结构不同但语义相似的代码片段。而由于编程语言的多样性，同样的功能可用多种语法结构实现，同一个算法可能有多种等价的表示方式。这就导致了对于同一个编程任务，源代码的表达方式可以是多种多样的，这种现象称为代码多样化。代码多样化常常出现在现实软件系统中。例如，在开发过程中没有发现某个函数已经实现，而重新实现了一个语法表示不同但语义等价的函数；软件重构的过程中经常进行语义不变的转换，如移动表达式、变量重命名、提取函数或函数内联等；程序剽窃过程中常常以修改源程序加以掩饰，因此重复代码检测方法应该具有处理与识别这些代码多样化的功能，而实现上述处理和识别的功能就需要进行语义分析，对于语义分析，目前还没有完善的解决方案。基于相似子图的重复代码检测方法将源代码表示为程序依赖图 (Program Dependence Graph，PDG)，基于 PDG 的同构性识别相似代码。这种方法考虑程序的语法结构以及数据流信息，但是仍有两个关键问题没有得到良好解决：(1) 对代码多样化的识别能力有限，不能真正实现语义级别的重复代码检测；(2) 较高的计算复杂度。由于数据流分析的复杂度，PDG 的规模有限，并且查找子图同构问题难点，计算复杂度较高。

发明内容

[0004] 本发明是为了解决现在的重复代码检测方法存在的对语法表示不同但语义相似的代码的相似度检测准确度低、计算复杂度高，以及无法实现大规模程序代码相似度检测的问题，从而提出一种基于程序源代码语义分析的代码相似度检测方法。

[0005] 一种基于程序源代码语义分析的代码相似度检测方法，它由以下步骤完成：

[0006] 步骤一、分别将待检测的两段源代码解析为两棵系统依赖图的控制依赖树；

[0007] 步骤二、对步骤一获得的两棵控制依赖树分别执行基本代码标准化，获得两棵基本代码标准化后的控制依赖树；

[0008] 步骤三、利用度量值方法分别提取步骤二获得的两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树；

[0009] 步骤四、判断是否提取到候选相似代码控制依赖树，如果判断结果为是，则执行步骤五，如果结果为否，则结束两段源代码的相似度检测；

[0010] 步骤五、对所述候选相似代码控制依赖树执行高级代码标准化操作；

[0011] 步骤六、计算候选相似代码控制依赖树的结构相似度 StructureSim 和候选相似代码控制依赖树的语句相似度 StatementSim，并根据公式：

[0012] $SemanticSim = \lambda_{stru} * StructureSim + \lambda_{stat} * StatementSim$

[0013] 计算候选相似代码控制依赖树的语义相似度，获得语义相似度结果，完成代码相似度检测；式中：λ_{stru} 为结构相似度的权值、λ_{stat} 为语句相似度的权值。

[0014] 步骤二中所述基本代码标准化的方法为：采用基于系统依赖图的程序标准化方法对步骤一所述控制依赖树进行结构语义等价的基本代码标准化转换；所述基本代码标准化转换包括：复合表达式的标准化转换、表达式的标准化转换和基本控制结构的标准化转换。

[0015] 所述提取步骤三中利用度量值方法分别提取两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树的方法为：

[0016] 步骤 A、分别在所述两棵基本代码标准化后的控制依赖树上提取各模块的规模、结构和复杂度特征信息，并分别统计各个模块的度量向量，获得两段代码的度量向量集；所述度量向量集包括：节点个数、运算符个数、选择结构个数、循环结构个数、赋值节点个数、特殊数据类型个数、控制依赖树中最长路径长度、系统函数调用个数、递归调用路径长度和控制依赖边个数；

[0017] 步骤 B、将步骤 A 获得两段代码的度量向量集，通过公式：

[0018] $$sim(v,v') = 1 - \sqrt{\frac{1}{n}\sum_{i=1}^{n}((v_i - v_i')/\max(v_i, v_i'))^2}$$

[0019] 计算两段代码的度量向量集的相似度 sim(v,v')；式中 i = 1,2,3,……,10；n = 10；

[0020] 步骤 C、判断步骤 B 获得的度量向量的相似度 sim(v, v') 是否大于预先设定的度量向量相似度阈值 T_v，如果判断结果为是，则提取此两段代码做为候选相似代码；如果判断结果为否，则结束两段源代码相似度检测。

[0021] 所述步骤五中的高级代码标准化的方法为：分别对两棵候选相似代码控制依赖树执行标准化转换，所述标准化转换包括：高级控制结构标准化、消除冗余代码、变量重命名、语句重排列和函数调用标准化。

[0022] 所述步骤六中候选相似代码控制依赖树的结构相似度 StructureSim 是根据公式：

[0023] $StrcutureSim = StructureMatching(S,T)/|S|$

[0024] 获得的；

[0025] 所述步骤六中候选相似代码控制依赖树的语句相似度 StatementSim 是根据公式：

[0026] $StatementSim = \sum value(vi)/|S|$

[0027] 获得的，式中：S 为候选相似代码控制依赖树，value(vi) 为 S 中节点的表达式相似度，|S| 为候选相似代码控制依赖树 S 的节点总数。

[0028] 有益效果：本发明的代码相似度检测方法可以识别语义相似的重复代码，可以应用于大规模程序代码的分析。本发明的方法通过将源代码表示为改进的系统依赖图，降低了系统依赖图表示的复杂度；并且根据代码标准化，将系统依赖图进行语义不变的转换，消除了各种代码的多样化，从而实现语义级别的重复代码检测；以及改进基于度量值的方法，在代码标准化的基础上，以模块单位计算度量值，提高度量值相似度计算的准确性；结合以上两种改进方法，通过在计算语义相似度前引入计算复杂度较小的基于度量值的候选相似代码“过滤”机制，缩小搜索空间，并且语义级别的匹配是基于树的匹配，较大程度上降低了计算复杂度，能够处理大规模程序。

附图说明

[0029] 图 1 是本发明的方法的流程示意图；图 2 是本发明的方法所采用的基于系统依赖图的程序标准化模型；图 3 是具体实施方式一中所述嵌套选择结构转换成多分支结构的转换过程示意图；图 4 是具体实施方式一中所述相邻的选择结构的合并过程示意图；图 5 是具体实施方式一中所述用 if-else 结构替换 if-continue 结构的转换过程示意图；图 6 是具体实施方式一中所述消除 if-break 结构过程示意图；图 7 是图 6 的转换结果示意图；图 8 是具体实施方式一中所述用多分支的 if、if-else 结构替换 if-return 结构的转换过程示意图。

具体实施方式

[0030] 具体实施方式一、结合图 1 ～图 8 说明本具体实施方式，一种基于程序源代码语义分析的代码相似度检测方法，它由以下步骤完成：

[0031] 步骤一、分别将待检测的两段源代码解析为两棵系统依赖图的控制依赖树；

[0032] 步骤二、对步骤一获得的两棵控制依赖树分别执行基本代码标准化，获得两棵基本代码标准化后的控制依赖树；

[0033] 步骤三、利用度量值方法分别提取步骤二获得的两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树；

[0034] 步骤四、判断是否提取到候选相似代码控制依赖树，如果判断结果为是，则执行步骤五，如果结果为否，则结束两段源代码的相似度检测；

[0035] 步骤五、对所述候选相似代码控制依赖树执行高级代码标准化操作；

[0036] 步骤六、计算候选相似代码控制依赖树的结构相似度 StructureSim 和候选相似代码控制依赖树的语句相似度 StatementSim，并根据公式：

[0037] $SemanticSim = \lambda_{stru} * StructureSim + \lambda_{stat} * StatementSim$ 计算候选相似代码控制依赖树的语义相似度，获得语义相似度结果，完成代码相似度检测；式中：λ_{stru} 为结构相似度的权值、λ_{stat} 为语句相似度的权值。

[0038] 步骤二中所述基本代码标准化的方法为：采用基于系统依赖图的程序标准化方法对步骤一所述控制依赖树进行结构语义等价的基本代码标准化转换；所述基本代码标准化转换包括：复合表达式的标准化转换、表达式的标准化转换和基本控制结构的标准化转换。

[0039] 所述提取步骤三中利用度量值方法分别提取两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树的方法为：

[0040] 步骤 A、分别在所述两棵基本代码标准化后的控制依赖树上提取各模块的规模、结构和复杂度特征信息，并分别统计各个模块的度量向量，获得两段代码的度量向量集；所述度量向量集包括：节点个数、运算符个数、选择结构个数、循环结构个数、赋值节点个数、特殊数据类型个数、控制依赖树中最长路径长度、系统函数调用个数、递归调用路径长度和控制依赖边个数；

[0041] 步骤 B、将步骤 A 获得两段代码的度量向量集，通过公式：

[0042] $$sim(v,v') = 1 - \sqrt{\frac{1}{n}\sum_{i=1}^{n}((v_i - v_i')/\max(v_i, v_i'))^2}$$

[0043] 计算两段代码的度量向量集的相似度 sim(v,v')；式中 i = 1,2,3,……,10；n = 10；

[0044] 步骤 C、判断步骤 B 获得的度量向量的相似度 sim(v, v') 是否大于预先设定的度量向量相似度阈值 T_v，如果判断结果为是，则提取此两段代码做为候选相似代码；如果判断结果为否，则结束两段源代码相似度检测。

[0045] 所述步骤五中的高级代码标准化的方法为：分别对两棵候选相似代码控制依赖树执行标准化转换，所述标准化转换包括：高级控制结构标准化、消除冗余代码、变量重命名、语句重排列和函数调用标准化。

[0046] 所述步骤六中候选相似代码控制依赖树的结构相似度 StructureSim 是根据公式：

[0047] StrcutureSim = StructureMatching(S,T)/|S|

[0048] 获得的，式中：S 和 T 是两棵候选相似代码的结构树，StructureMatching(S,T) 为 S 与 T 的最大匹配节点对个数，|S| 为候选相似代码的结构树 S 的节点总数。

[0049] 所述步骤六中候选相似代码控制依赖树的语句相似度 StatementSim 是根据公式：

[0050] StatementS im = Σ value(vi)/|S|

[0051] 获得的，式中：S 为候选相似代码控制依赖树，value(vi) 为 S 中节点的表达式相似度，|S| 为候选相似代码控制依赖树 S 的节点总数。

[0052] 本发明的基本思想是：首先，读取待检测的两段源程序文件进行预处理，创建程序的系统依赖图；然后，执行基本代码标准化，消除对度量值计算有较大影响的代码多样化；接下来，基于度量值的方法提取候选相似代码，快速过滤掉大部分不相似的代码片段，缩小后续语义级别代码检索的空间；下一步，对提取的候选相似代码执行高级代码标准化，消除代码多样化，尽量使等价的语义具有相同的表示方式；最后，通过在结构和语句表达式级别匹配标准化的控制依赖树，计算语义相似度，从而识别语法结构不同而语义相似的代码。

[0053] 下面对本发明的方法做进一步详细说明：

[0054] 在步骤一中，对两段源代码进行词法分析和语法分析以及控制流、数据流分析，构造程序的系统依赖图。

[0055] 这里，词法分析将程序代码转换为可以进行语法分析的 token 流。语法分析将生成程序的抽象语法树表示。

[0056] 然后生成系统依赖图，系统依赖图是程序的图形表示。系统依赖图可以分解为控

制依赖子图和数据依赖子图。控制依赖子图由节点和控制依赖边构成，表示程序的控制流信息；数据依赖子图由节点和数据依赖边构成，表示程序的数据流信息。

[0057] 本发明改进传统的系统依赖图表示形式。将控制依赖子图表示为有序树的形式，本发明称之为控制依赖树。并且只在必要时，即高级代码标准化时，对提取的候选相似代码分析数据依赖子图中的数据流信息，由于候选相似代码只占源程序的一小部分，因此可以有效地降低系统依赖图表示的复杂度，从而使之适合于分析大规模程序。

[0058] 在步骤二中，在系统依赖图的控制依赖子图上执行基本代码标准化。

[0059] 基于度量值的方法提取候选相似代码会因代码多样化而导致度量值计算偏差，从而产生漏检。为了避免这一问题，本发明提出基于系统依赖图转换的代码标准化方法。

[0060] 基本代码标准化根据标准化规则对控制依赖树进行结构语义等价的转换，直到任何规则都不能应用为止。可以消除复合表达式多样化、表达式多样化以及控制结构多样化等会影响度量值相似度计算的准确度的代码多样化的问题。

[0061] 本发明的程序标准化方法采用王甜甜、苏小红、马培军在中国武汉发表的《消除代码多样化的程序标准化》(2008，306 ～ 309，EI 收录号：20091211962490) 和上述三人在中国上海发表的《消除模块程序中代码多样化的程序标准化》(2008，21 ～ 24，EI 收录号：20091211972107，ISTP 收录号：BIX53) 中记录的采用基于系统依赖图 (System Dependence Graph，SDG) 的程序标准化方法。这种方法改进了传统的系统依赖图表示，使其充分表示程序的语法结构与语义。并针对已有的指针分析算法不适合于程序标准化的问题，基于控制依赖树和改进的指向表示方法提出流敏感和上下文敏感的指针分析算法，提高指针分析和数据流分析的准确性，并使得指针分析结果可直接应用于程序标准化转换中。最后将改进的系统依赖图、指针分析算法与程序标准化过程有机结合提出基于系统依赖图的程序标准化模型，根据程序标准化转换规则，对系统依赖图进行语义不变的转换，从而消除代码多样化。

[0062] 常见的代码多样化可以概括为以下几种：

[0063] (1) 代码格式多样化：语义等价的代码可能具有不同的代码格式。例如，不同个数的注释和空行。在解析程序时可以通过删除注释和空行消除这种多样化。

[0064] (2) 复合语句多样化：有些程序元素可以用一条语句表示，也可以用多条语句构成的语句序列表示。

[0065] (3) 表达式多样化：由于各种运算符的优先级和结合性不同，导致语义等价的表达式可以用多种形式表示。

[0066] (4) 控制结构多样化：源代码中通常含有三种控制结构，即顺序、分支和循环。这些控制结构中有时还包含控制转移语句，每种控制结构可以用多种语法表示。

[0067] (5) 冗余代码：程序中可能包含一些无用的语句或使用不同数目的临时变量。

[0068] (6) 函数调用多样化：实现一个算法，函数的模块结构可以是多种多样的，函数的个数以及函数调用语句的位置都可能不同。

[0069] (7) 标识符命名多样化：语义等价的代码中的变量名称可能不同。

[0070] (8) 语句顺序多样化：语义等价的代码中的语句排列顺序可能不同。

[0071] (9) 数据结构多样化：语义等价的代码可能用不同的数据结构表示，如指针与数组表示的多样化等。

[0072] 基于系统依赖图的程序标准化方法对这些代码多样化进行处理，识别包含这些多样化的相似代码。

[0073] 下面结合图 2 具体说明基于系统依赖图的程序标准化规则：

[0074] 图 2 为基于系统依赖图 SDG(System Dependence Graph) 的程序标准化模型，其中 SDG 是程序代码的语义表示。如果待检测的一个源代码 P 与另一个源代码 P' 具有相同的 SDG 表示，则所述两个源代码 P 与 P' 是语义等价的。如果它们具有相似的 SDG 表示，则所述两个源代码 P 与 P' 是语义相似的。然而，如果所述两个源代码 P 与 P' 语义相似，它们的 SDG 不一定相似。

[0075] 语义等价的转换：如果一个转换改变了代码的计算行为却不改变计算结果，则称该转换为语义等价的转换。使用一系列语义等价的转换可将采用相同算法且语义上等价的代码标准化为相同的 SDG。

[0076] 程序标准化：程序标准化是根据一系列转换规则对 SDG 进行语义等价的转换的过程，程序标准化可以消除代码多样化。

[0077] 首先解析源程序，将源程序表示为一种中间表示形式。为了使程序的中间表示方式既便于程序标准化转换，又便于在指针分析过程中利用标准化转换结果简化指针分析，考虑到树结构便于执行指针分析与程序转换，因此本方法提出控制依赖树 CDT(Control Dependence Tree，) 的程序中间表示形式，在表示程序语法结构的同时充分表达程序的语义信息。

[0078] 然后对 CDT 进行基本标准化转换，以消除部分代码多样化，生成标准化的 CDT，从而简化随后的指针分析。

[0079] 接下来，遍历标准化的 CDT，计算各个节点的指针别名。程序标准化转换需要确保程序的语义不变，对指针分析的准确性要求较高，因此这种标准的指针分析算法采用了流敏感和上下文敏感的分析方法。

[0080] 然后，就可以利用指向信息集合执行数据流分析，在 CDT 的基础上创建数据流图，进一步执行高级标准化转换，消除代码多样化。最后生成的标准化 SDG（由 CDT 和数据流图构成），可直接应用于程序分析中。

[0081] 基于系统依赖图的程序标准化模型将系统依赖图创建、指针分析与程序标准化过程有机结合在一起，既利用程序标准化简化指针分析，又将指针分析结果直接应用于程序标准化转换中，提高程序标准化转换的代码多样化消除率，从而提高程序分析的灵活性。

[0082] 本标准将标准化分为两类，其中在执行转换时不需要数据流与指针别名信息的标准化称为基本标准化。基本代码标准化根据一系列标准化规则对 CDT 进行结构语义等价的转换，直到任何规则都不能应用为止。

[0083] 所述基本代码标准化包括：

[0084] 1、拆分复合语句

[0085] 复合语句被拆分为等价的语句序列，消除复合语句多样化：

[0086] • 将表达式转换为只引用变量而不定义变量的形式，从而消除副作用。例如：

[0087] while((i = j/k) < 1){...} → i = j/k ;while(i < 1){...i = j/k ;}

[0088] • 拆分变量声明语句。例如：

[0089] int i, j ;→ int i ;int j ;

[0090] • 拆分变量声明与初始化语句。例如：

[0091] int i = 0 ;→ int i ;i = 0 ;

[0092] • 拆分输入 / 输出语句。例如：

[0093] scanf(“ % d% d”,&i,&j) → scanf(“ % d”,&i) ;scanf(“ % d”,&j) ;

[0094] • 拆分连续的赋值语句。例如：

[0095] i = j = k ;→ j = k ;i = j ;

[0096] • 如果数组下标是复合表达式，则用临时变量替换该复合表达式。例如：

[0097] r = a[b[i]] ;→ t = b[i] ;n = a[t] ;

[0098] • 如果复合表达式中含有函数调用语句，则用临时变量替换该函数调用语句。例如：

[0099] if(x+sqrt(y) > 0){...} → t = sqrt(y) ;if(x+t > 0){...}

[0100] •如果函数调用语句中的实参是复合表达式，则用临时变量替换该复合表达式。例如：

[0101] i = fun(j+k) → t = j+k ;i = fun(t) ;

[0102] •如果 return 语句返回的是复合表达式，则用临时变量替换该复合表达式。例如：

[0103] return i+j → t = i+j ;return t ;

[0104] 2、表达式标准化

[0105] 算术表达式标准化利用算术运算符的分配律、交换律、恒等式等属性，建立转换规则，用以消除表达式多样化。这些规则被反复调用，对表达式语法树进行转换，直到任何规则都不适用于表达式。布尔表达式标准化与算术表达式标准化规则类似，只是它的转换规则是根据布尔运算符的属性定义的，并且应用于布尔表达式。如下为几个典型的表达式标准化规则，其中 A1，A2，A3 为常量、变量或子表达式。

[0106] R1 :A1+(A2+A3) → (A1+A2)+A3

[0107] R2 :(A1||A2)&&A3 → (A1&&A3)||(A2&&A3)

[0108] R3 :A2 < A1 → A1 > A2

[0109] 宏替换是一种特殊的表达式标准化，用宏定义的常量或表达式替换宏定义符号。例如，#define PI 3.1415area = PI*r*r ;→ area = 3.1415*r*r ;

[0110] 3、基本的控制结构标准化

[0111] 对选择结构和循环结构进行转换，消除一些控制结构多样化。

[0112] • 统一选择结构的表示形式，将各种选择语句表示为 selection 结构。

[0113] • 统一循环结构的表示形式，将各种循环语句表示为 iteration 结构。

[0114] •合并相邻的选择结构。如果两个相邻的选择结构满足下列条件，则将它们合并为一个选择结构：

[0115] (1) 它们在 CDT 中是兄弟节点；

[0116] (2) 只有一个选择分支；

[0117] (3) 条件表达式是互斥的。例如：

[0118] if(exp) if(exp)

[0119] A A

[0120]

⇨

[0121] if(! exp) else

[0122] B B

[0123] • 删除永远得不到执行的选择分支。例如：

[0124] if(1)A else B → A

[0125] if(0)A else B → B

[0126] if(e)if(e)A else B → if(e)A

[0127] • 删除逻辑表达式恒为假的循环单元。例如：

[0128] do A while(0) ;→ A

[0129] while(0)A ;→空语句，从 CDT 中删除该循环结构。

[0130] •通过将条件表达式进行与操作，将嵌套选择结构转换成多分支结构。转换过程如图 3 所示。

[0131] • 合并相邻的选择结构。如果多个相邻的 selection 结构满足以下条件：

[0132] (1) 具有相同个数的选择分支；

[0133] (2) 相应分支的条件表达式相同。

[0134] 则如图 4 所示，将它们合并为一个选择结构。

[0135] 在步骤三中，执行基于度量值的候选相似代码提取。

[0136] 如果两段代码实现相同的算法，则它们通常具有相似的规模、结构、I/O 模式等程序特征，本发明改进基于度量值的方法，提出基于度量值的候选相似代码提取方法。在语义级别比较之前，利用效率很高的基于度量值的候选相似代码提取方法，快速过滤掉大部分不相似的代码片段，缩小语义级别代码检索的空间。由于基于度量值的方法计算简单，复杂度低，因此本发明将其用于预处理阶段，快速地筛选出可能的相似代码，然后再对可能的相似代码进行更精确的基于系统依赖图的语义分析，这样，不仅有助于提高算法的效率，还能确保得到准确的检测结果。

[0137] 在两个待检测的源代码 P 与 P' 中提取候选相似代码的算法主要包括两部分：首先，计算 P 与 P' 的各个模块的度量向量，然后，计算度量向量相似度，并根据设定的度量向量相似度阈值 Tv，提取候选相似代码。

[0138] 假设模块 $f \subset P$ 与模块 $f' \subset P'$ 的度量向量分别为 $v(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10})$，$v'(v_1', v_2', v_3', v_4', v_5', v_6', v_7', v_8', v_9', v_{10}')$，则 v 与 v' 的相似度计算方法如下。

[0139] $$sim(v,v') = 1 - \sqrt{\frac{1}{n}\sum_{i=1}^{n}((v_i - v_i')/\max(v_i, v_i'))^2} \qquad (1)$$

[0140] 如果 $sim(v, v') > T_v$，则将函数 f 和 f' 所直接调用或间接调用的控制依赖树子树构成的集合，以及函数 f 和 f' 所直接调用或间接调用的控制依赖树子树构成的集合，加入到候选相似代码列表中。

[0141] 在步骤五中，执行高级代码标准化。

[0142] 将需要进行数据流分析的代码标准化称为高级代码标准化。代码标准化的目的是消除代码多样化，使等价的语义有相同的表示方式，从而提高代码语义相似度计算的准确性。标准化过程中进行语义不变的转换是由一系列替换规则限制的。为确保应用这些规则

前后，代码具有相同的语义，因此，还需在控制依赖树的基础上进行数据流分析，保持数据流依赖和控制依赖关系不变，这样才能保持它们的语义等价。本发明将需要进行数据流分析的代码标准化称为高级代码标准化。高级代码标准化可以消除控制结构多样化、函数调用多样化、冗余代码、变量名多样化、语句顺序多样化等会影响语义相似度计算的准确度的代码多样化。高级代码标准化规则保持流依赖和控制依赖不变，从而保持了它们的语义联系不变。高级控制结构标准化用于消除一些控制结构多样化。包括：

[0143] • 合并相邻的具有相同控制条件表达式的多个循环单元。例如：

[0144] for(i = b ;i <= f ;i++)　　for(i = b ;i <= f ;i++)

[0145] A ;⇨ {

[0146] for(j = b ;j <= f ;j++)　　A ;

[0147] B ;　　j = i ;

[0148]　　B ;

[0149]　　}

[0150] • 提取循环体中值不变的语句。例如：

[0151] for(i = 1 ;i < 10 ;i++)　　m = n/q ;

[0152] {⇨ t = m+1

[0153] m = n/q ;　　for(i = 1 ;i < 10 ;i++)

[0154] a[i] = m+1 ;　　{

[0155] }　　a[i] = t ;

[0156]　　}

[0157] •消除 if-continue 结构。用 if-else 结构替换 if-continue 结构，转换过程如图 5 所示。

[0158] •消除 if-break 结构。通过增加控制循环条件，消除 break 语句。如果图 6 中，表达式 e2 与 B1 和 B2 都不存在数据依赖关系，则转换成图 7 左侧部分的形式，否则引入一个中间变量 var，转换成图 7 右侧部分的形式。

[0159] • 消除 if-return 结构。用多分支的 if、if-else 结构替换 if-return 结构，转换过程如图 8 所示。

[0160] • 转换循环语句中某些特殊的条件表达式，例如：

[0161] (a)for(i = 0 ;i < n+1 ;i++) → for(i = 0 ;i <= n ;i++)

[0162] (b)for(i = 0 ;i <= 999 ;i++) → for(i = 0 ;i < 1000 ;i++)

[0163] (c)for(i = 0 ;i <= n-1 ;i++) → for(i = 0 ;i < n ;i++)

[0164] (d)for(i = 1 ;i < n ;i++)A → for(i = 0 ;i <= n ;i++)A，其中 i 不在 A 中出现。

[0165] 2、函数调用标准化

[0166] 函数调用标准化用于消除函数调用多样化，把不含递归调用的模块进行函数内联，将其转换成一个没有函数调用语句的语义等价的程序。

[0167] 函数内联用被调函数的函数体的拷贝替换函数调用语句，内联后的程序与内联前的程序是语义等价的。在理论上函数内联展开可能导致程序规模的指数级增长，但是在实际应用中这种问题很少出现。本发明对子程序进行内联操作，子程序的规模通常是有限的，因此不会出现这种问题。

[0168] 目前人们已经提出了多种函数内联算法和策略，但大多是用于优化编译器中的，不适合直接应用于程序分析和程序匹配。为此，本发明提出基于函数调用树与 PDG 的函数内联算法。为各个子程序建立函数调用树，因此可以通过后根序遍历函数调用树，确定当前要进行内联操作的主调函数和被调函数。

[0169] 函数内联等价于复制被调函数的 PDG，并将其内联到主调函数的 PDG 中。PDG 的内联操作有如下几步：

[0170] (1) 将被调函数的 PDG 中声明的变量重命名。

[0171] (2) 处理参数传递并修改节点间的数据依赖关系。

[0172] (3) 处理返回值传递并修改节点间的数据依赖关系。

[0173] (4) 修改节点间的控制依赖关系。

[0174] (5) 删除主调函数中的函数调用节点、被调函数中的 entry 节点以及和它们相关联的边。

[0175] 经过内联操作后，每个不含递归调用的模块都转换为一个不含函数调用语句与子函数的 PDG，从而消除了函数调用多样化。

[0176] 递归函数不进行内联操作。针对同一个编程任务编写的递归程序，大都具有类似的模块结构，只是函数名字、参数名字和顺序可能有所区别，这种多样化可通过变量重命名和重排语句顺序进行消除。

[0177] 3、消除冗余代码

[0178] • 常量传播。例如：

[0179] i = 0 ;j = i ;k = j+1 ;→ i = 0 ;j = 0 ;k = 1 ;

[0180] • 消除恒等表达式。例如：删除语句 i = i。

[0181] • 消除公共子表达式。例如：

[0182]
```
if(i > 1)                    t = n-1 ;
```

[0183]
```
 j = i/(n-1) ;               if(i > 1)
```

[0184] ⇨

[0185]
```
else                          j = i/t ;
```

[0186]
```
  k = i*(n-1) ;              else
```

[0187]
```                           k = i*t ;
```

[0188] • 消除死代码。如果一个变量没有被任何语句引用，则删除该变量的声明语句；如果赋值语句的左值没有被任何语句引用，则删除该赋值语句。例如：

[0189]
```
main()                      main()
```

[0190]
```
{                           {
```

[0191] ⇨

[0192]
```
 int i,j ;                  int i ;
```

[0193]
```
 i = 1 ;                     i = 1 ;
```

[0194]
```
 j = 1 ;                     printf(“%d”,i) ;
```

[0195]
```
 printf(“%d”,i) ;      }
```

[0196] }

[0197] 4、指针必然别名替换

[0198] 由于互为别名的两个命名对象指向同一个存储区域，所以对同一个存储地址的引用方式可能因别名的存在而产生多样化。例如程序：

```
[0199] int add(int x,inty){          //s1
[0200]     returnx+y;                //s2
[0201] }                             //s3
[0202] main(){                       //s4
[0203]   int sum = 0;                //s5
[0204]   int i = 1;                  //s6
[0205]   int*p = &sum;               //s7
[0206]   int*q = &i;                 //s8
[0207]   int(*f)(int,int) = add;     //s9
[0208]   while(*q < 11){             //s10
[0209]    *p = (*f)(*p,*q);          //s11
[0210]       *q = *q+1;              //s12
[0211]    }                          //s13
[0212]    printf("%d/n",*p);     //s14
[0213]    printf("%d\n",*q);     //s15
[0214] }
```

[0215] 程序中 *q 与 i 互为别名，因此既可以通过 *q 也可以通过 i 访问相应的内存地址，赋值语句 *q = *q+1 ;可以等价地表示为 i = i+1 ;，这就是由指针别名引入的代码多样化。

[0216] 别名信息可以分类为两种：必然别名 (Must alias) 和可能别名 (May alias)。必然别名是指在程序的所有执行中都出现的别名关系；可能别名是在程序的某些执行中出现的别名关系。两个变量如果互为必然别名，则可以相互替换。而如果它们互为可能别名，则不可进行替换，因为它们不总是等价的。指针变量的必然别名替换的思想是用必然别名替换指针变量，以减少程序中由于别名对导致的代码多样化。

[0217] 指针必然别名替换的算法，如程序：

[0218] 算法：AliasesReplacement

[0219] 输入：具有指向集合的 CDT

[0220] 输出：替换了具有必然别名的指针变量和表达式的 CDT

[0221] Begin

[0222] foreach CDT 中的节点 N

[0223] if N 的表达式引用了指针变量 pvar then

[0224] if 存在 <*pvar, t, D> ∈ IN_N || (pvar, t, D) ∈ IN_N

[0225] && 不存在$(*pvar,t',D) \notin GEN_N$ && 不存在$(*pvar,t',P) \notin GEN_N$ then

[0226] 用 t 替换节点 N 的表达式中的 *pvar 或 pvar

[0227] endif

[0228] endif

[0229] endfor

[0230] End

[0231] 检查 SDG 的每个节点，如果其对应的语句中引用了指针变量 pvar，但该指针变量在此处未因赋值而产生新的别名，就分析该节点的入口处的别名信息集合。如果入口处的别名信息集合中含有唯一的形如 <*pvar，t，D> 或者 <pvar，t，D> 的别名对，而节点对应的语句中又引用变量 pvar 却没有产生新的别名，则用 t 替换节点的语句中所含的 *pvar 或者 pvar。

[0232] 对程序：

```
[0233] int add(int x,inty){            //s1
[0234]     return x+y;                 //s2
[0235] }                               //s3
[0236] main(){                         //s4
[0237]    int sum = 0;                 //s5
[0238]    int i = 1;                   //s6
[0239]    int*p = &sum;                //s7
[0240]    int*q = &i;                  //s8
[0241]    int(*f)(int,int) = add;      //s9
[0242]    while(*q < 11){              //s10
[0243]      *p = (*f)(*p,*q);          //s11
[0244]      *q = *q+1;                 //s12
[0245]    }                            //s13
[0246]    printf("% d\n",*p);          //s14
[0247]    printf("% d\n",*q);          //s15
[0248] }
```

[0249] 中的示例程序进行别名替换，由指针分析算法分析得到 s7，s8，s9 分别产生别名对 <*p，sum，D>，<*q，i，D>，<*f，add，D>，因此 s10，s11，s12，s13，s14 与 s15 的 IN 集合为 {<*p，sum，D>，<*q，i，D>，<*f，add，D>}。然后执行必然别名替换，s10，s11 与 s14 中的 *p 被 sum 替换。s11，s12 与 s15 中的 *q 被 i 替换。s11 中的 *f 被 add 替换。最后 s7，s8 与 s9 变为未被引用的语句而被删除。最后得到语义等价的程序：

```
[0250] int add(intx,int y){
[0251]    return x+y;
[0252] }
[0253] main(){
[0254]    int sum = 0;
[0255]    int i = 1;
[0256]     while(i < 11){
[0257]       sum = add(sum,i);
[0258]       i = i+1;
[0259]    }
```

[0260] printf(“ % d\n”, sum) ;

[0261] printf(“ % d\n”, i) ;

[0262] }

[0263] 5、变量重命名

[0264] 如果两段代码语义等价，则对应的变量的类型和出现频率通常是相同的。因此可根据变量的类型以及出现的频率给变量重命名。

[0265] 首先，将相同类型的变量按照出现的频率排序。然后，将变量重命名为“##_#”的形式，其中，第一个“#”表示变量的类型，如，“i”表示整型，“c”表示字符型，“f’ 表示浮点型，“F”表示 FILE ；第二个“#”表示特殊的类别，如，“A”表示数组，“P”表示指针，“S”表示结构体 ；第三个“#”表示该变量出现频率的排序。例如，iA 01 表示一个数组变量，类型是 int 类型，出现的频率在所有 int 类型变量中排列第一。循环控制变量是特殊命名的，第一个字符是‘x’。递归程序的函数名的形式为 recSub_Func_i，其中，i 为该函数被第一次调用的顺序。非递归函数名的形式为“Sub_Func_i”其中，i 为该函数在模块中定义的顺序。最后，用新的变量名替换 SDG 中旧的变量名。通过变量重命名可以识别变量名字不同的但语义相似的代码片段。

[0266] 6、重排语句顺序

[0267] 重排语句顺序的算法如程序 ：

[0268] 算法 :StatementsReordering

[0269] 输入 :模块的各个 PDG

[0270] 输出 :语句重排序后的 PDGs

[0271] Begin

[0272] 后根序遍历 CDT ；

[0273] foreach CDT 中的节点 N

[0274] if N 有两个或两个以上的控制依赖子节点 then

[0275] 假设 S1 与 S2 是 N 的子节点

[0276] if S1 是 S2 的左兄弟并且 S1 与 S2 间不存在数据依赖关系 then

[0277] if symbol(S2) 的字符排序小于 symbol(S1) then

[0278] 交换 S1 与 S2 在 CDT 中的位置

[0279] endif

[0280] endif

[0281] endif

[0282] endfor

[0283] End

[0284] 所示，可以消除语句排列顺序的多样化。如果两条语句间不存在数据依赖，则它们可以互换位置，这样的语句被重新排序。

[0285] 将 SDG 中的每个节点表示为形为“X:Y”的符号，其中“X”是该节点的类型 (例如，“#declare”，“assignment”，“selection”，“iteration”，“entry”等)，“Y”是该节点的语句或表达式，例如，“assignment :x_01 ＝ 1”。对于 PDG 中两个节点 S1 和 S2，如果 S1 与 S2 具有相同的控制依赖父节点，S1 与 S2 不存在数据依赖关系，并且 S2 的符号比 S1 符号按字

符排序小，则在 CDT 中将 S2 排列在 S1 的前面。经过语句重排序后各个节点在有序树 CDT 中的位置就唯一确定了。

[0286] 在步骤六中，计算语义相似度。

[0287] 在标准化的控制依赖树的基础上进行结构匹配和语句匹配两个级别上的匹配。按以下如公式计算语义相似度：

[0288] $SemanticSim = \lambda_{stru} * StructureSim + \lambda_{stat} * StatementSim$

[0289] 其中 StructureSim、StatementSim 分别为结构相似度和语句相似度，λ_{stru}、λ_{stat} 分别为这两个相似度的权值，满足 $\lambda_{stat} > \lambda_{stru} > 0$，$\lambda_{stru} + \lambda_{stat} = 1$。不但考虑程序结构，而且考虑具体的语句表达式，因此可以准确度量程序的语义相似度。结构匹配和语句匹配方法如下：

[0290] (1) 结构匹配：

[0291] 本发明给出以下两个定义作为结构匹配的基础：

[0292] 定义（结构树）：在控制依赖树基础上提取程序的循环、选择、顺序结构，生成只包含节点类型及其之间的控制依赖边的树型结构，称为结构树。

[0293] 定义（匹配节点对）：两棵结构树 T1、T2 的节点对 (v1,v2) 是匹配节点对，当且仅当：

[0294] 1) v1 与 v2 具有相同的符号，即相同的类型。

[0295] 2) v1、v2 的父节点是匹配节点对。

[0296] 3) 假设 w1 与 w2 是匹配节点对，v1 与 w1 是兄弟，v2 与 w2 是兄弟，那么若 v1 在 w1 前出现，则 v2 在 w2 前出现。

[0297] 4) v1 至多能和 T2 中的一个节点匹配，v2 至多能和 T1 中的一个节点匹配。

[0298] 结构树可充分表示程序的控制结构，因此候选相似代码的结构匹配相似度可由它们的结构树的相似度来度量。求结构树 S 与 T 的最大匹配等价于求 S 与 T 的最大匹配节点对集合。最后按照下式计算结构相似度 StrcutureSim。其中，StructureMatching(S,T) 为 S 与 T 的最大匹配节点对个数，|S| 为结构树 S 的节点总数。

[0299] $StrcutureSim = StructureMatching(S,T)/|S|$

[0300] (2) 语句匹配：

[0301] 结构匹配过程中保留了类型相匹配的节点的信息，因此程序的语句匹配可以在程序结构匹配基础上进一步检查语句、表达式的匹配情况。表达式的匹配通过遍历两个表达式的抽象语法树求得最大匹配，计算相似度，并将各语句的表达式相似度保存在源候选相似代码控制依赖树 S 的各个节点中。最后，通过遍历 S 获得各个节点的表达式相似度 value，按照公式：

[0302] $StatementSim = \Sigma\ value(vi)/|S|$

[0303] 计算语句相似度 StrtementS im；其中，value(vi) 为节点 vi 的求得的表达式相似度，|S| 为控制依赖树 S 的节点总数。

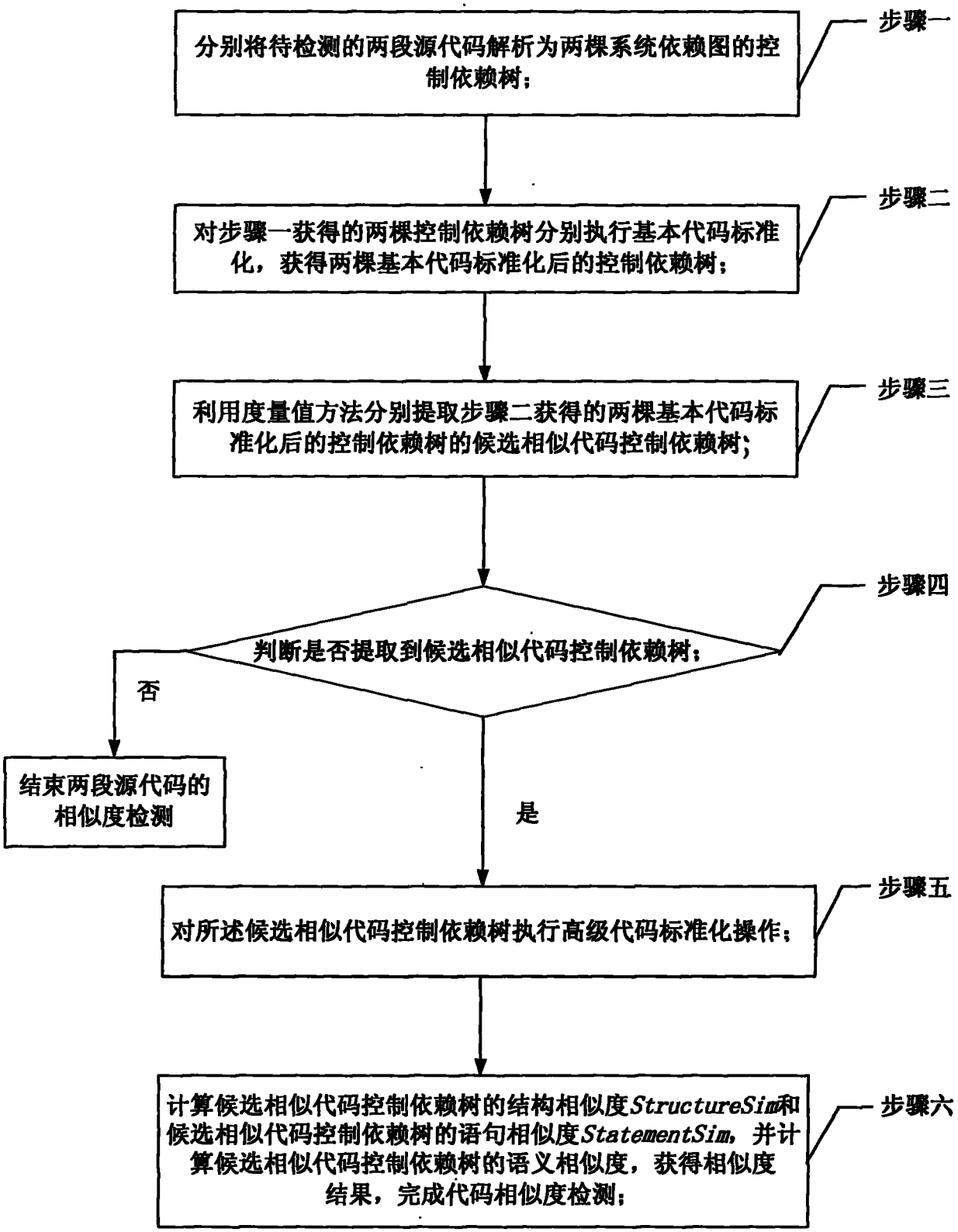

分别将待检测的两段源代码解析为两棵系统依赖图的控制依赖树；
步骤一
对步骤一获得的两棵控制依赖树分别执行基本代码标准化，获得两棵基本代码标准化后的控制依赖树；
步骤二
利用度量值方法分别提取步骤二获得的两棵基本代码标准化后的控制依赖树的候选相似代码控制依赖树；
步骤三
判断是否提取到候选相似代码控制依赖树；
步骤四
否
结束两段源代码的相似度检测
是
对所述候选相似代码控制依赖树执行高级代码标准化操作；
步骤五
计算候选相似代码控制依赖树的结构相似度StructureSim和候选相似代码控制依赖树的语句相似度StatementSim，并计算候选相似代码控制依赖树的语义相似度，获得相似度结果，完成代码相似度检测；
步骤六

图 1

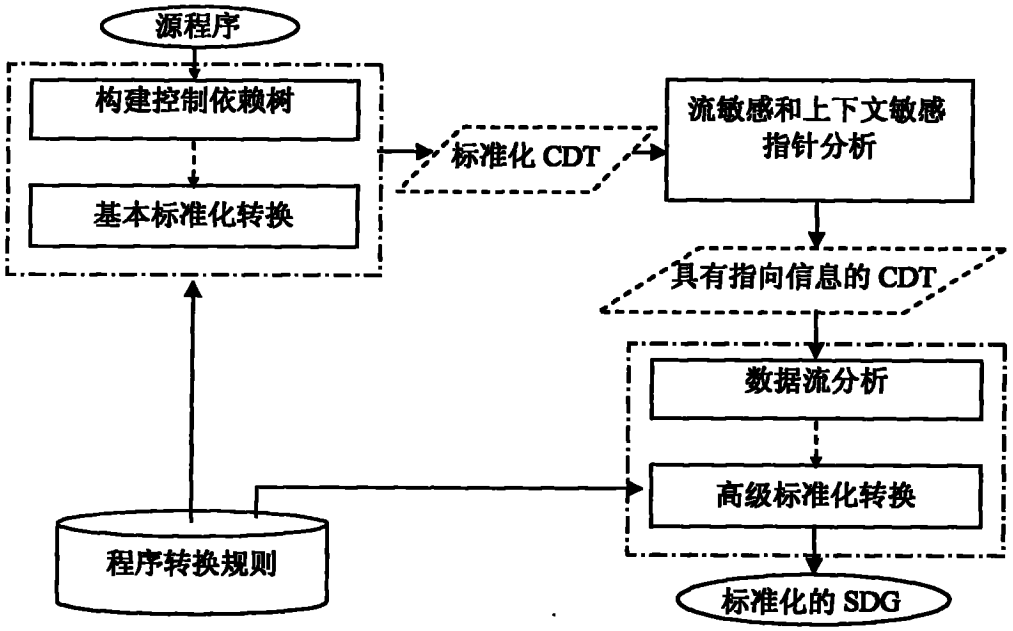

源程序
构建控制依赖树
基本标准化转换
标准化 CDT
流敏感和上下文敏感指针分析
具有指向信息的 CDT
数据流分析
高级标准化转换
程序转换规则
标准化的 SDG

图 2

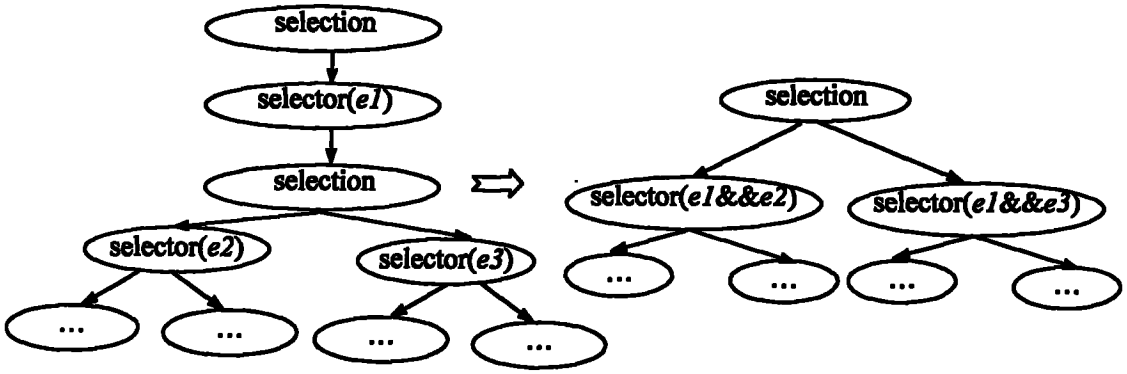

selection
selector(e1)
selection
selector(e2)
selector(e3)
selection
selector(e1&&e2)
selector(e1&&e3)

图 3

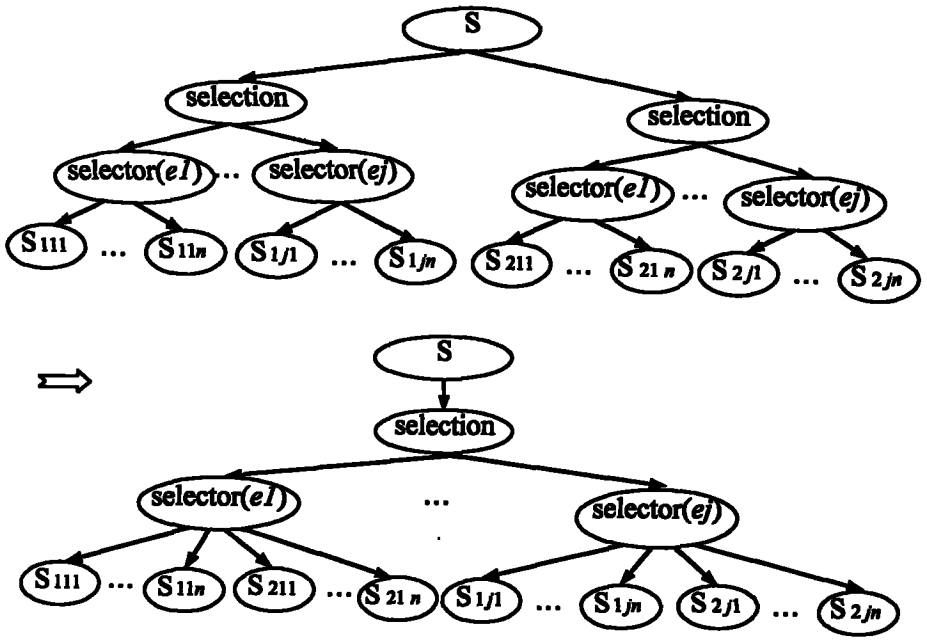

S
selection
selection
selector(e1)
selector(ej)
selector(e1)
selector(ej)
S111
S11n
S1j1
S1jn
S211
S21n
S2j1
S2jn
S
selection
selector(e1)
selector(ej)
S111
S11n
S211
S21n
S1j1
S1jn
S2j1
S2jn

图 4

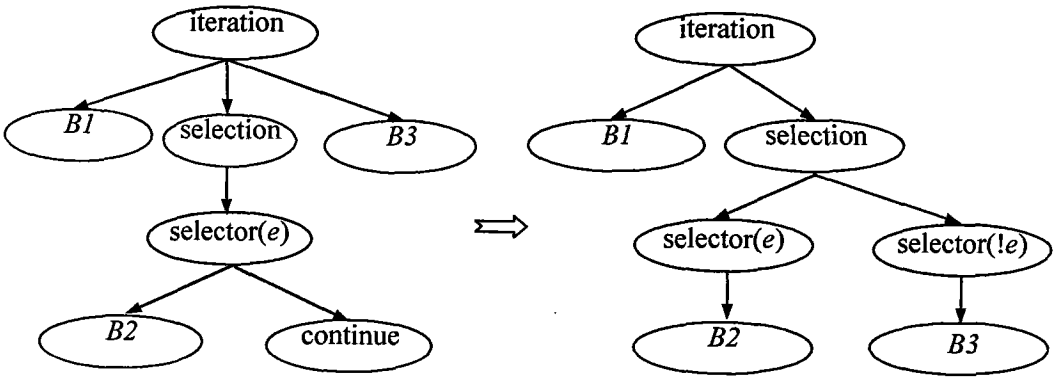

iteration
B1
selection
B3
selector(e)
B2
continue
iteration
B1
selection
selector(e)
selector(!e)
B2
B3

图 5

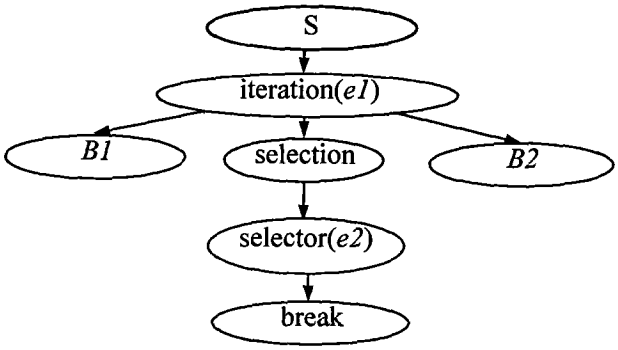

S
iteration(e1)
B1
selection
B2
selector(e2)
break

图 6

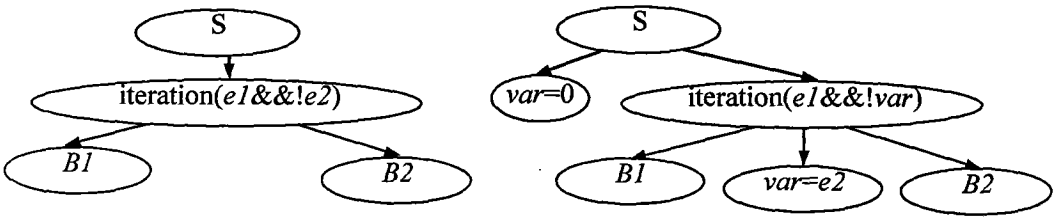

S
iteration(e1&&!e2)
B1
B2
S
var=0
iteration(e1&&!var)
B1
var=e2
B2

图 7

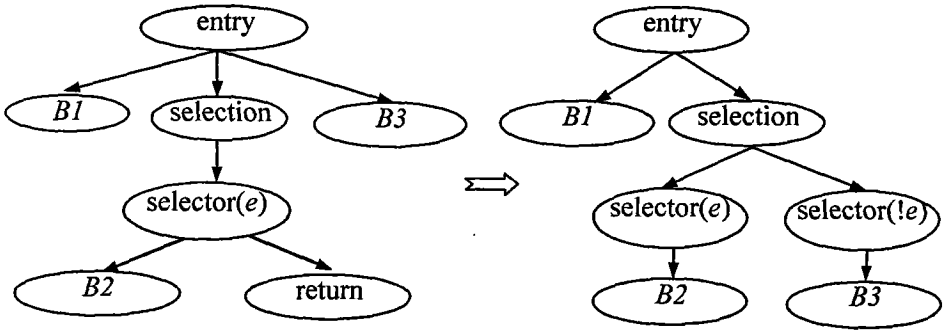

entry
B1
selection
B3
selector(e)
B2
return
entry
B1
selection
selector(e)
selector(!e)
B2
B3

图 8