西安电子科技大学

硕士学位论文

程序流程图结构分析与识别技术的研究与实现

姓名：陈科

申请学位级别：硕士

专业：计算机软件与理论

指导教师：刘西洋

2011-01

摘要

随着嵌入式领域中模型驱动开发技术的日益成熟和广泛应用，工程的开发效率及自动化程度被高度重视。其中代码自动生成技术将开发人员从繁琐的代码实现中解放出来使其最大程度的关注系统逻辑的设计，从而保证了系统设计的合理性、高效性和鲁棒性。

针对程序流程图模型的代码自动生成技术，其难点在于流程图的结构分析与识别。本文基于结构化程序设计思想的原则，充分考虑流程图中半结构化和非结构化元素的特征，设计并实现了标准程序流程图的结构分析与识别算法。结构化流程图中基本控制结构之间形成了复杂的组合与嵌套关系。结合上下文结构信息，算法运用图论知识准确识别各个循环结构及分支结构，同时在条件判断结点识别算法中判断循环结构的类型。通过遍历循环结构寻找控制流的改变，算法识别出流程图中有 continue、break 和 return 含义的半结构化元素。针对非结构化流程图，本文提出了结构化验证规则，验证算法依此辨别出流程图中有 GOTO 含义的非结构化元素。

本文参与实现的流程图代码自动生成工具经大量测试用例验证，能够正确识别流程图的结构组成和非结构化元素并生成含有半结构化语句的 C 程序代码。

关键字： 程序流程图 结构分析 代码自动生成 半结构化 GOTO 语句

ABSTRACT

As the model-driven development technology in embedded field has become more mature and extensive use, development efficiency and automation degree is highly valued. The automatic code generation technology liberates developers from the tedious code, allowing them to pay the greatest degree of attention to the design of system logic, thus ensuring the system design of rationality, efficiency and robustness.

For automatic code generation technology of program flowchart model, the difficulty lies in the structure analysis and identification of flowchart. Taking full account of the characteristics of semi-structured and unstructured elements in flowchart, the paper designs and implements the analysis and recognition algorithms for standard program flowcharts based on structured program design principles. The basic control structures in flowcharts have complex relationship of combination and nesting. Incorporating structural context, the algorithms accurately identify each loop structure and branch structure by applying graph theory. At the same time, the type of loop structure is determined in the condition node identification algorithm. Searching changes of the control flow by traversing loop structure, the algorithm identifies semi-structured elements with the meaning of continue, break and return in flowchart. For unstructured flowchart, the paper proposes several structural validation rules, which are used to identify unstructured elements with the meaning of "GOTO".

An automatic code generation tool for program flowcharts is developed according to the algorithms proposed by the paper. Experimenting on plenty of test cases, the tool is proved able to correctly identify the structure components of flowcharts and unstructured elements and generate corresponding C codes containing the semi-structured statements.

Keyword: **program flowchart** **structure analysis** **GOTO statement** **automatic code generation** **semi-structured flowchart**

第一章 绪论

1.1 研究背景

本文工作来源于西安电子科技大学软件工程研究所软件测试技术实验室承担的北京某航天自动控制研究所的某研究课题，该课题主要致力于研究航天控制领域的层次划分方法和分层建模技术，并实现对应的组态建模及代码生成仿真工具，软件实现从模型建模、验证、仿真到代码生成的层次化建模过程，并且完成飞行控制软件所需构件的开发和管理。实验室课题组运用 Eclipse 插件开发技术，实现了 Eclipse 平台下状态图、流程图层次化建模的功能，并在图形建模的基础上实现了层次化代码自动生成工作。

模型提供了一个物理系统的抽象，模型可以使得工程师们忽略无关的细节而把注意力放到最重要的部分来思考系统的整体设计。工程中的所有工作内容都是依赖模型来理解复杂的、真实世界的系统。模型被运用于很多方面：预测系统的质量，当系统的某些方面变化时推理特定的属性变化和特征等等。模型也可以作为实现某种物理系统的先驱被开发，或者模型可以根据一个已存在的系统或者开发中的系统进行构建并作为理解现实系统行为的帮助手段。

长久以来，在软件开发过程中，人们总是会发现这个问题的存在：需求工程师、系统分析师、软件工程师、测试人员之间缺少一个共同的交流平台，使得一个项目从需求分析开始无法完整统一的交付到后面的阶段，结果软件工程师根据所获得的信息编写出来的代码，不是用户需求真正表达和需要的。模型驱动开发[1]（Model-Driven Development，MDD）的应用，很好的解决了以上问题，将整个软件开发过程用标准模型来统一表示，消除开发过程中各种参与者之间的差异化理解和隔阂。UML[7]的出现，很好的对模型进行了标准化定义，使模型的表示毫无二意可言。模型驱动开发将分析需求、设计模型与代码同等对待[2]，并将模型和代码更好地对应集成起来，通过模型来大大增加迭代方案的可行性，而不仅仅是直接修改代码。模型驱动开发涉及到不同的技术，这些技术贯穿了整个软件的开发过程，包括模型驱动需求工程、模型驱动设计、模型的代码生成[10]、模型驱动测试以及模型软件演变等。针对模型驱动开发的理念，国际对象管理组织(Object Management Group，OMG)定义了一种软件开发框架MDA[29]（Model Driven Architecture），它是基于UML及其他标准的框架，支持软件设计和模型的可视化以及存储。MDA中最关键的要素是模型，即在MDA框架中模型占据主导地位。利用模型驱动开发使得项目相关人员参与进来，都通过这个共同的模型进行交互，对参与的系统有着共同统一的认识，这其中不仅仅是上面列举的人员，还可以包括

客户、项目经理、主管、第三监理方、评估者等等参与项目的人。总的来说，如果能够定义一个抽象的、独立于任何技术的模型，再通过代码生成器[8,9]生成相关的框架程序或者部分具体技术的业务代码，不但减少了开发人员重复代码的编写，大大提高了开发效率，另外一方面，也使得程序员把精力集中在创造性的开发上，同时减少了因为开发人员的疏忽而造成的系统bug。自动生成的代码具有统一的命名风格和程序风格，从而使代码更易于阅读和维护，更重要的是代码生成集成了软件设计和开发人员的最佳实践模式，使生成的目标代码具有更好的性能和质量保证[18]。

代码生成首先通过归纳总结现有成熟的单元测试工具设计和实现，提炼出统一的符合测试人员操作习惯的单元测试工具主框架。该框架将当前测试建模工程中的测试项映射为层次化结构，并按照工程师目前使用的界面框架语言生成易于理解的主框架代码，便于测试工具开发人员在生成的代码基础上进一步扩展。

1.2 国内外研究现状

上世纪八十年代人们就已经提出了基于模型驱动开发的概念。以色列的David Harel教授曾经在对反应式系统建模的研究当中提出了以状态图(statechart)为主的层次化建模方法[20]。同时期，针对以色列领域工业集团Lavi战斗机领域软件系统的开发，引入了Statechart[3]建模及其支持工具Statemate。实际上，Statechart就起源于1983年David Harel建模的以色列Lavi战斗机领域软件系统。正因为如此，Statechart及其派生如Stateflow一直广泛的应用于领域系统高层功能行为的建模，尤其是模式控制逻辑的图形化建模。

随后，建模技术的发展和代码自动生成[11]技术的出现，使得基于模型驱动的开发方法才真正在软件开发领域应用起来。状态图作为一种精确表示行为的方法，可以更好的协调工程之间的交流[19]，状态图在建模当中的应用非常广泛。上世纪末本世纪初，美国领域管理局NASA喷气动力实验室JPL首次在深空1(Deep Space1)空间计划中应用了基于状态图（Statechart）的建模和代码生成技术，取得了初步的成效。但是由于Statechart的建模和代码生成均借助Matlab Stateflow实现，因而生成的代码需要经过NASA工程师费时费力且容易出错的后处理，才能得到NASA工程师谙熟的、完全符合NASA软件开发规范与硬件平台约束的实际系统代码，封闭的与领域系统成熟建模风格与代码模板脱节的代码生成算法的局限性首次显现。在随后的2002年NASA JPL在Deep Impact计划中应用了类似的Stateflow建模与代码生成技术，封闭代码生成算法的本质缺陷进一步凸显。

与上述NASA基于Simulink/Stateflow的领域系统开发技术类似，法国Esterel公司针对领域系统的模型驱动开发工具SCADE Suite同样以类似Simulink的数据

流框图“确定性数据流”（Deterministic Data Flow）和类似 Stateflow 的 Statechart 变种“安全状态机”（Safe State Machine）建模领域系统的行为（例如领域系统的模式逻辑和控制律），其特色是针对领域系统的高安全性需求对建模和代码生成施加了一套严格的安全性规范。

在总结上述实践经验的基础上，2004 年，NASA JPL 定义了“基于状态的实时系统架构与代码自动生成”开发规范 STAARS（STate-based Architecture and Auto-coding for Real-time Systems），一改之前基于封闭的为生成仿真代码而设计的通用建模工具的思路，强调针对成熟领域软件实践，从中提炼出相应的建模风格与代码模板，并在针对实时嵌入式系统高度优化的开源状态机框架代码 Quantum Framework[15]（QF）的基础上自主开发 Statechart 代码生成器 Auto-Coder，以生成与工程师手工编写的代码一样可读，且符合相关规范的实际系统代码。依据成熟建模风格与代码模板、建构在开源状态机框架之上的 Auto-Coder 初步实现了从仿真代码生成到实际系统代码生成的跨越。

近年来，模型驱动开发已成为软件工程技术的研究热点和发展趋势，也是当前软件自动生成的研究趋势[16,17]。基于MDD的开发平台目前已经在国外开发出来，比如IBM公司的RSA(Rational Software Architect)提供了模型到模型、模型到代码的自动转换，支持模型驱动开发的全过程[27]。Telelogie公司的Tau Developer也是基于MDD开发平台，实现了从设计模型生成可执行软件的功能[28]。目前市场上已经存在很多商品化设计的基于状态图建模的自动化工具，它不仅能自动地从状态图生成代码，而且能够在图形级别进行状态模型的查错和测试[24,25,26]。在航空航天系统中也越来越多地应用了基于Statechart + 数据流框图的模型驱动开发，相关的论文和研究报告在美国领域学会AIAA频繁发表。但是应该看到，Statechart + 数据流框图建模的是更多是领域系统的高层功能行为包括模式逻辑和控制律，这样的功能行为必须要映射到多处理器上多个任务之间的交互，才能生成高效的实际系统代码。这就需要进一步引入对多处理器上多任务及其交互的建模。法国国家科学研究中心系统分析与架构实验室LAAS-CNRS研发的实时建模语言TURTLE及其工具TTool[4]为我们提供了一个有益的借鉴。TURTLE采用实时扩展的活动图建模任务的实时行为以及任务之间的交互，TTool中的代码生成器解析该活动图自动生成对应的RT-LOTOS代码，借助RT-LOTOS验证工具RTL检测TURTLE活动图中的错误。

近期以来，一个重要的发展趋势是模型驱动开发开始应用于复杂测试脚本的开发，活动图是建模测试脚本直观而且自然的模型。对于领域系统尤其领域系统而言，类似 TURTLE 的实时扩展活动图进一步为建模测试脚本中必备的实时性提供了参考和借鉴。可以预见，随着对测试脚本开发效率的进一步需求，基于模型驱动的开发将更广泛地应用于领域系统包括领域系统的测试中。

另一方面，近些年来计算机辅助软件工程(computer aided software

engineering，CASE)有了很大的发展[5]。计算机辅助软件工程是采取系统化工程方法，利用计算机帮助设计人员完成设计任务的理论、方法和技术。它综合了计算机图形学、人机交互技术、工程数据库和设计方法学等多个领域的理论、方法和技术，建立具有辅助设计功能的系统，以帮助设计人员在计算机上完成设计模型的构造、分析、优化和输出等工作。计算机辅助设计可提高设计的自动化程度和质量，缩短设计周期，借助计算机强大的计算能力，完成一些常人难以完成的设计任务。

在针对程序流程框图代码自动生成的可视化编程[15]工具中，多数采取的方法是首先对程序框图之间的各种关系及其含义进行解析，将之分解为顺序执行关系、判断嵌套关系、循环嵌套关系和跳转嵌套关系，在此基础上生成一个自动转换系统。该系统包括用户接口控制模块、图形符号解析模块、代码生成模块、编译执行模块。

很早就有学者对流程图代码生成技术进行了研究。Martin C. Carlisle 等人曾经提出了 RAPTOR[13]，一种基于流程图的程序开发环境。通过跟踪流程图的执行，RAPTOR 程序以可视化的形式创建和运行。Kanis Charntaweekhun 等人也曾提出利用流程图进行可视化编程的思想并设计出了相应的工具[14]。

然而现有的可视化编程辅助工具针对流程图代码自动生成方面对用户和流程图有着各种限制。例如索诺玛加州州立大学的 Tia Watts 等人开发了基于结构化流程图模型的可视化编程工具 THE SFC EDITOR[6]，用于辅助软件开发人员设计和可视化系统流程、算法逻辑。该工具提供了结构化流程图中多种基本控制结构组件如不同类型的循环结构、分支结构等等，开发人员通过这些组件的组合和嵌套表现不同需求的业务逻辑。同时，工具提供了模型代码框架的自动生成功能，一定程度上提高了开发效率和开发质量。工具 SFC 提供了既定的 for、while 和 do while 循环结构单元以及具有单分支或者多分支的分支结构单元等供用户来选择进行流程图的构建。这样使得用户的思维局限在代码设计层次而不是算法逻辑设计层次；另外该工具能够识别的流程图结构是完全结构化的，用户无法随心所欲的进行模型设计以及逻辑设计，需要花费额外的精力来选用不同的结构单元设计符合该工具要求的特定的流程图。因此该工具的功能简单有限、过于死板缺乏灵活性。又如印度 G.H.Raisoni 大学的 Hemlata Dakhore 和 Anjali Mahajan 等人针对结构化流程图转化 C 代码的问题进行了研究，将流程图的图结构转换为树结构，设计了流程图的结构化文档存储方案，并提供了文档的结构语法分析器和代码生成器。其工作基于分支可达集合的计算，以递归的方式对流程图进行结构的分析和转换。利用其开发的工具，程序开发的初学者可以方便的构建结构化流程图进行对应代码的自动生成，而对于灵活的半结构化流程图的结构分析，该工具有待进一步提高。

1.3 论文研究问题

本课题组实现的流程图代码自动生成工具遵循标准流程图规范，允许用户最大程度的注重算法逻辑的设计，灵活的进行流程图的设计。同时该工具能够结合嵌入式领域特有的结构化代码安全规范，识别出流程图中违反编码安全规范的各种元素（例如具有 GOTO 含义的流程连线、缺少 false 分支的条件判断结点、非结构化流程图元素等等）并发出警告，而对于流程图中合法的结构化元素，工具能够正确识别并判断其代码含义（包括具有 continue、break 和 return 代码含义的半结构化流程图元素）。针对流程图中可能出现的各种相互组合及嵌套的循环结构和分支结构，工具能够自动进行结构类型的识别并选用合适的循环类型进行代码生成，无须用户特别指定。课题组设计和实现的流程图代码自动生成工具解放了系统设计工程师的思维，具有极大的灵活性和智能性。

流程图是一个有向图，流程图代码自动生成的过程本质就是将图结构线性化的过程。首先考虑一般的结构化流程图。按照结构化程序设计的观点，任何算法功能都可以通过由程序模块组成的三种基本程序结构的组合：顺序结构、分支结构和循环结构通过组合、嵌套来实现。标准流程图通过各种矩形、菱形结点以及流程线来构建这三种基本结构以及各种复杂的组合、嵌套关系，如图 1.1 所示。顺序结构本身就是线性的，因此线性化的主要工作在于分支结构和循环结构的处理。从图中容易看出，流程图中的分支结构和循环结构都由一个菱形结点引领，如果能够识别出流程图中每个菱形结点表示的基本结构是分支结构还是循环结构，以及每个分支结构的出口和循环结构的类型，那么就可以像编译中的语法分析那样将线性化的流程图转换为代码片段。

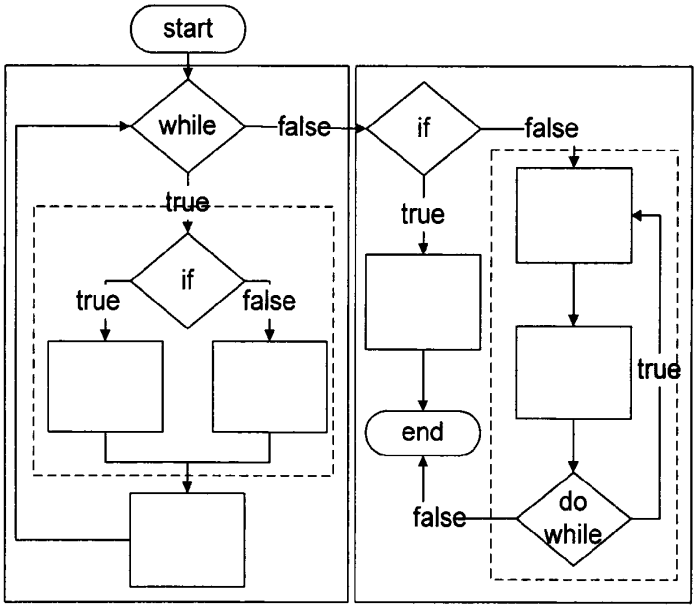


图 1.1 结构化流程图是控制结构之间的组合与嵌套

当结构化流程图中加入了一些灵活性的元素，例如具有 continue、break 和 return 含义的流程线，甚至包含了非结构化元素形成非结构化流程图时，图中的各种基本结构之间严格的组合、嵌套关系被一定程度的打破，如图 1.2 所示。从图中可以看到，具有 continue 与 break 含义的流程图元素使得嵌套结构之间有了多条通路，具有 return 含义的流程图元素使得一个基本结构可以拥有直接通往流程图结束结点的通路，而具有非结构化的 GOTO 含义的流程图元素使得基本结构之间有了任意的非法通路，基本结构之间可能形成相互重叠的情况。为了能够正确识别结构化中灵活的 continue、break、return 和非结构化的 GOTO 元素并做相应的转化或者错误处理，在上述的识别过程中应充分考虑各种流程图元素具有的特征以及对识别带来的各种影响。

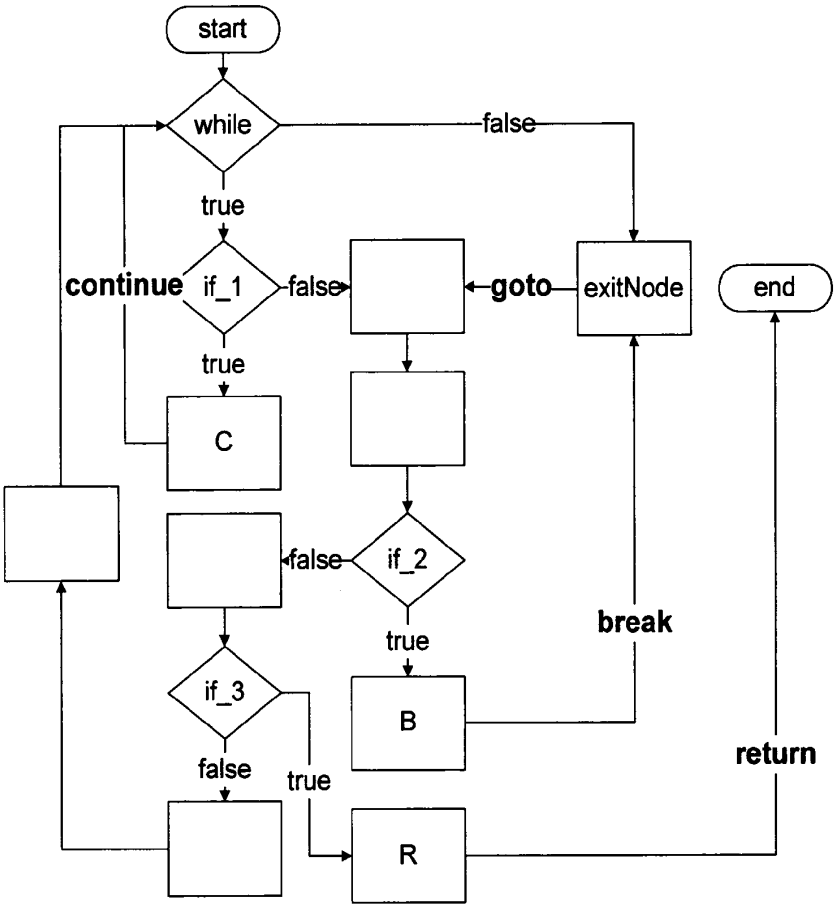


图 1.2 非结构化流程图

本课题组实现的流程图代码自动生成工具系统框架如图 1.3 所示。由系统框架图可知，工具共分为三个模块。用户利用流程图建模模块绘制流程图，流程图解析器将绘制的流程图解析为永久存储的 xml 文件或者工具自定义的流程图数据结构，而代码生成模块负责处理该数据结构，输出流程图对应的代码片段或者指出流程图中不符合代码安全规范的非结构化元素。代码生成模块又分为五个子模块。结构识别模块找出流程图中各个分支/循环结构并识别出其中可能存在的各种

特殊元素；循环结构处理模块将每个循环结构线性化；半结构化元素处理模块将continue、break 和 return 元素转换为结构化流图；分支结构处理模块判断分支结构的出口并将其线性化;线性化代码生成模块遍历线性化的流程图生成代码文本；错误处理模块涉及各个模块，一旦发现流程图有不符合代码安全规范的非法元素即生成错误信息。

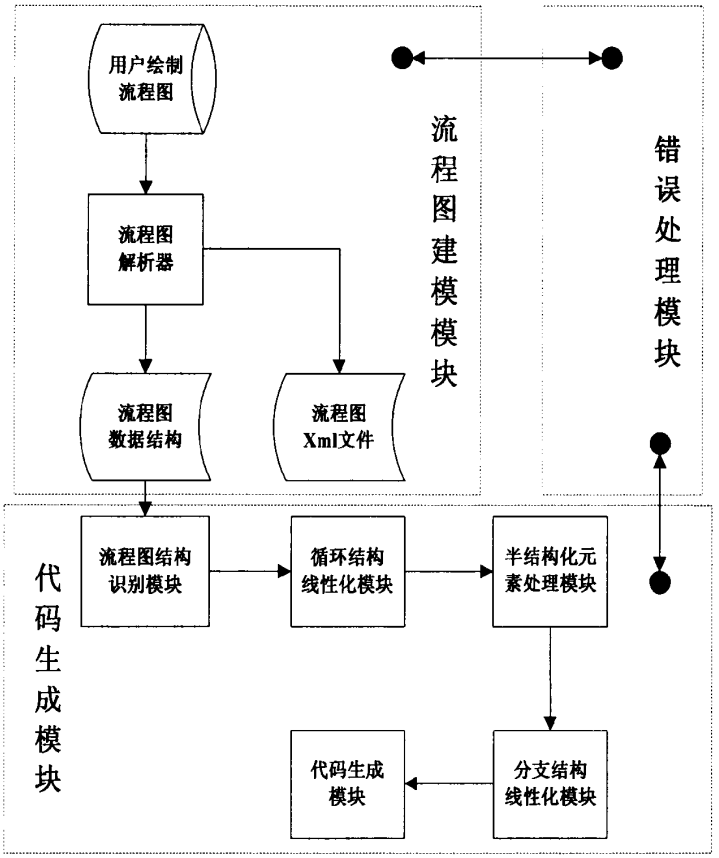


图 1.3 流程图代码自动生成工具系统框架图

1.4 本文研究工作

本文研究工作主要集中在流程图代码生成模块中结构的分析与识别方面，依据图 1.3 的系统框架，本文主要工作如下：

1、运用图论的相关知识识别流程图中的基本分支结构、循环结构；

2、识别基本结构之间的组合、嵌套关系；

3、根据不同循环结构类型的特征及强连通分量上下文结构信息计算循环结构的条件判断结点，同时辨别流程图中所有菱形结点所属的循环类型或分支类型；

4、根据半结构化程序设计的特征，对流程图中可能存在的半结构化元素进行

识别和标记；

5、结合结构化程序设计的准则，识别流程图中可能存在的非结构化元素；

6、从结构化程序设计方面进行流程图合法性的初步检查。

1.5 本文的组织结构

本文共分六章，第一章为绪论，主要讨论了本文的研究背景、所研究的问题和本文所做的工作；第二章介绍了程序流程图结构分析与识别技术所需要的理论基础；第三章给出了结构化、半结构化流程图结构识别的算法框架，重点讨论了辨别基本控制结构、定位循环结构条件结点、判断循环结构类型以及识别半结构化元素的算法；第四章在结构识别的基础上，简单讨论了流程图的线性化处理以及结构化的验证；第五章详细阐述了本文主要研究工作在代码自动生成工具中的实现，给出了部分测试用例的实验结果；第六章总结全文，给出了本文存在的有待改进的问题以及进一步的研究方向。

第二章 流程图代码自动生成相关技术

流程图代码自动生成技术是在对流程图进行结构上的分析和识别基础上，按照一定的编码规范，运用目标语言中定义的语法和基本控制结构，生成能够表达流程图控制流执行路径的代码。

本章阐述了进行流程图代码自动生成时所需要的一些理论基础，包括结构化程序设计、结构化流程图和半结构化流程图元素等。

2.1 结构化程序设计

结构化程序设计方法的产生，是20世纪60年代末70年代初爆发的“软件危机”所引起的。20世纪60年代末70年代初，随着计算机硬件技术的发展和应用范围的不断扩大，人们已经开发了各种大型软件系统，如操作系统、数据库管理系统、航空和旅馆订票系统等。开发这些大型软件系统，编制周期长，工作量大，并且由于传统程序设计方法的局限性，设计出的软件系统可靠性差，往往隐藏着许多错误，软件的开发和维护过程中遇到了一系列严重的问题。一方面，客观上需要研制大量大型软件系统；另一方面，按照原有的方法研制软件周期长、可靠性差、维护困难，已不能适应新形势的要求，这就是“危机”之所在。

面对这种严峻的现实，软件人员不得不重新考虑过去所忽视的程序设计中的一些基本问题。图灵奖获得者Dijkstra曾在ACM通讯上发表了一篇题为《GOTO语句是有害的》文章，指出了GOTO语句的破坏性，首先向传统的程序设计方法提出了挑战，在计算机界引起了一场大辩论，辩论的焦点是是否应该从高级程序设计语言中取消GOTO语句[12]。主张取消的一方强调了GOTO语句的危害性，而主张保留的一方则列举了GOTO语句的各个优点。GOTO语句的危害性为：（1）GOTO语句破坏了程序的静态结构与动态执行之间的一致性；（2）GOTO语句使程序难以阅读，难以查错；（3）GOTO语句影响程序测试，难以进行程序正确性证明；（4）GOTO语句影响程序优化。使用GOTO语句的优点有：（1）在块或过程的非正常出口处往往需要GOTO语句；（2）GOTO语句往往使程序执行的效率较高，而且方便灵活；（3）使用GOTO语句可以方便地从循环（甚至嵌套的循环）中跳出。

关于GOTO语句的讨论一直进行了好多年，Dijkstra认为程序质量与程序中所含GOTO语句的数量成反比，程序的可读性与效率之间存在着反作用[22]。D.E. Kunth对此给出了公证的评述。他的基本观点是：不加限制地使用GOTO语句，特别是使用往回跳的GOTO语句，会使程序结构难于理解，应尽量避免。而在另

外一些情形，为了提高程序的效率，同时又不破坏程序的良好结构，有控制地使用一些GOTO语句是可以的。

争论的结果使大家意识到应该把注意力放到程序结构上，随着计算机硬件技术的不断发展，程序结构比起程序的执行效率显得更为重要。这就导致了结构化程序设计方法（structured programming）[21,30]的产生，这是软件发展的一个重要的里程碑。

结构化程序设计是一种设计程序的技术，它采用自顶而下逐步求精的设计方法和单入口单出口的控制结构。自顶而下逐步求精的方法符合人类解决复杂问题的普遍规律，可以显著提高软件开发工程的成功率和生产率。用先全局后局部、先整体后细节、先抽象后具体的逐步求精过程开发出的程序有清晰的层次结构，容易阅读和理解。不使用GOTO语句仅使用单入口单出口的控制结构，使得程序的静态结构和它的动态执行情况比较一致。因此，结构化程序不仅容易阅读和理解，开发时也比较容易保证程序的正确性，即使出错也比较容易诊断和纠正。程序的逻辑结构清晰，有利于程序的测试和正确性证明。

那么，到底什么是结构化程序设计呢？对于这个问题到现在也没有一个统一的定义，计算机科学家Gries对此进行了解释[23]，主要归纳为如下几点：

- 结构化程序设计是一种进行程序设计的原则和方法，它采用自顶向下逐步求精的方法和单入口单出口的控制结构，按照这种原则和方法设计出的程序具有结构清晰、容易阅读、容易修改、容易验证等特点；
- 结构化程序设计讨论了如何将任何大规模和复杂的流程图转换成一种标准形式，使用程序设计语言中的顺序、分支、循环等有限的控制结构通过组合或嵌套来表示程序的控制逻辑；
- 结构化程序设计是避免使用GOTO语句的一种程序设计。

按照这种思想设计出的程序称为结构化程序，它成为人们编制软件时广泛采用的一种方法。

2.2 结构化程序流程图

2.1.1 程序流程图

程序流程图是有向图，通过将多种不同含义的结点和边进行相连来表示程序中的操作顺序和流程。程序流程图中包括：

- 指明实际处理操作的流程符号，它包括一般的流程处理符号和根据逻辑条件确定要执行路径的条件判断符号；
- 指明控制流的流程线符号；

- 便于读、写程序流程图的特殊符号。

图 2.1 中列出了国标程序流程图中常用的几种元素符号和符号的含义。

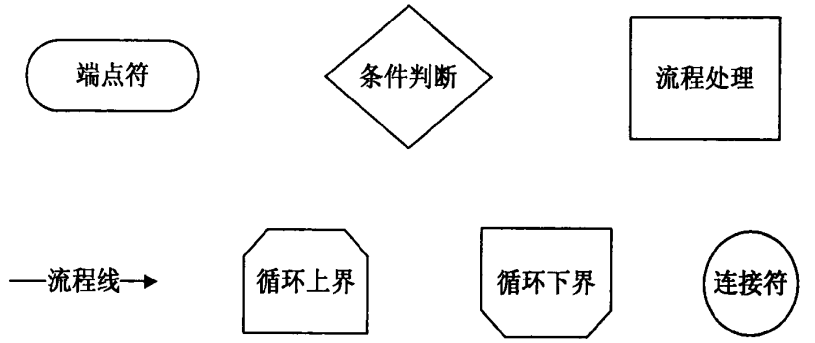


图 2.1 标准程序流程图常用元素符号

端点符：此符号表示转向外部环境或从外部环境转入。一般用于表示程序流程的起始或结束、数据的外部使用以及起源(或终点)。

条件判断：菱形分支结点，表示条件判断或开关类型功能。该符号有两个或两个以上可选择的出口，在对符号中定义的条件进行求值后，有且仅有一个出口被激活。求值结果可在表示出口路径的流程线附近写出。

流程处理：矩形流程结点，表示程序中各种流程处理功能。

流程线：此符号表示程序的控制流，箭头方向为控制流流向。

循环界限：此符号分为循环上界和循环下界两个部分，分别表示循环结构的开始和结束。在该符号的两个部分中要使用同一标识符。初始、增量和终止条件按其操作位置分别出现在上下界内。

连接符：此符号表示转向流程图它处，或自流程图它处转入。它用来作为一条流程线的断点，使该流程线在别处继续下去。对应的连接符应有同一标记。

使用这些符号可以组成顺序、分支和循环的结构，进而按照设计好的处理逻辑构成完整的流程图。

2.1.2 结构化程序流程图

在结构化程序设计思想的指导下，运用程序流程图设计和表述程序的处理逻辑，符合结构化程序设计思想的设计方案是结构化的程序流程图。

顺序的流程结点以及基本的分支和循环控制结构构成结构化程序流程图的过程满足如下几条规则：

- 流程图的构成由最简单的程序流程图开始，初始只含有一个矩形结点；
- 组合规则：流程图中的任意一个矩形结点都能够被两个顺序的矩形结点替代；
- 嵌套规则：流程图中的任意一个矩形结点都能够被任意一个控制结构（顺

序、分支和循环）替代；

- 构成规则：规则 2 和规则 3 可以以任意的次序多次运用从而构成复杂的结构化流程图。

如图 2.2 所示，由以上四条规则构成一幅复杂流程图的过程。

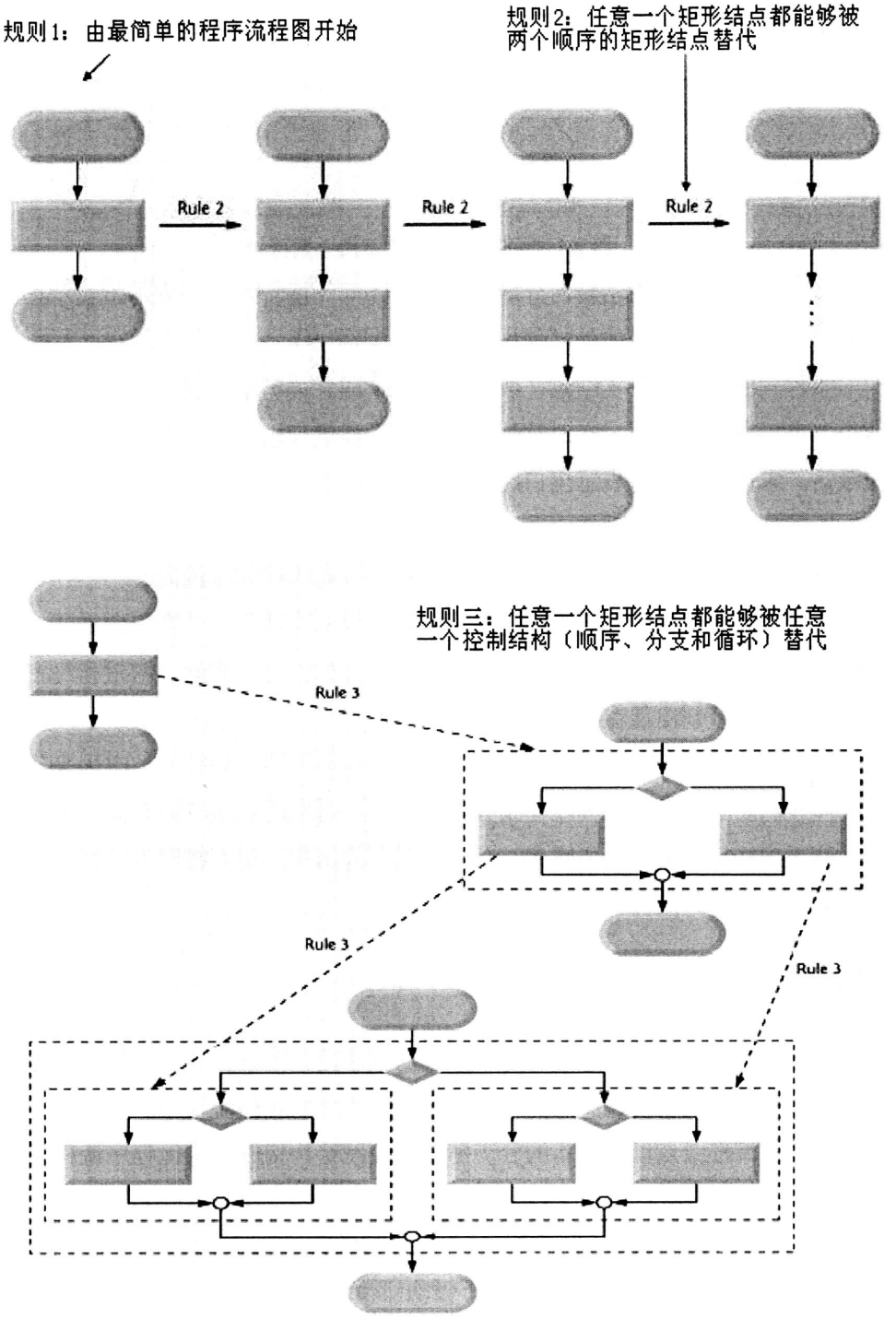


图 2.2 结构化程序流程图的构成规则

由结构化流程图的构成规则可以得到如下几条结论：

1、结构化流程图中只会出现控制结构组合、嵌套的关系，不可能出现交叠的关系。如图 2.3 和图 2.4 所示，流程图中控制结构的构成关系以及非结构化的流程图示例，该非结构化流程图中，多个循环结构出现了交叠的关系。

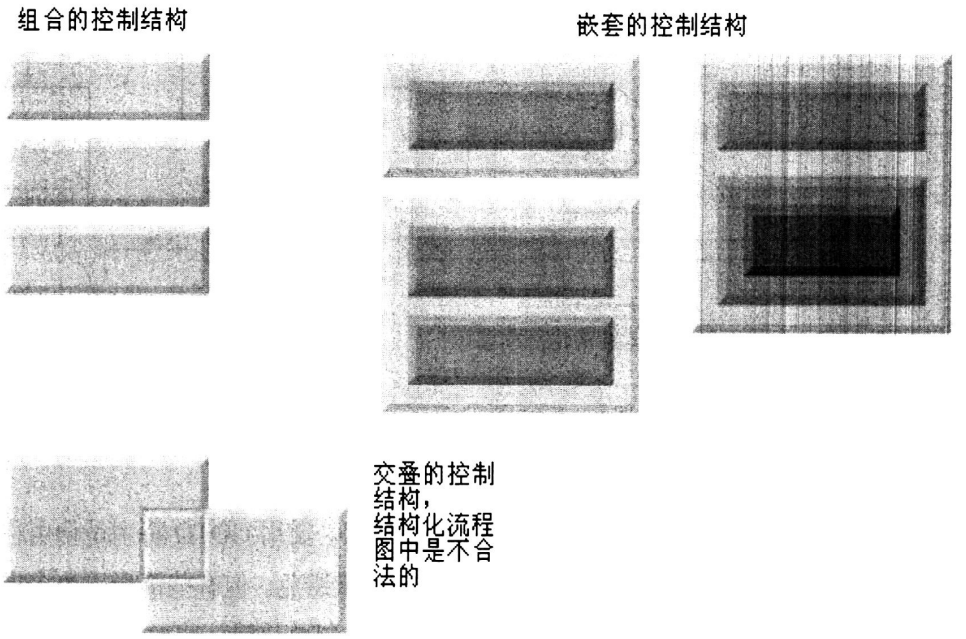


图 2.3 流程图控制结构的构成关系

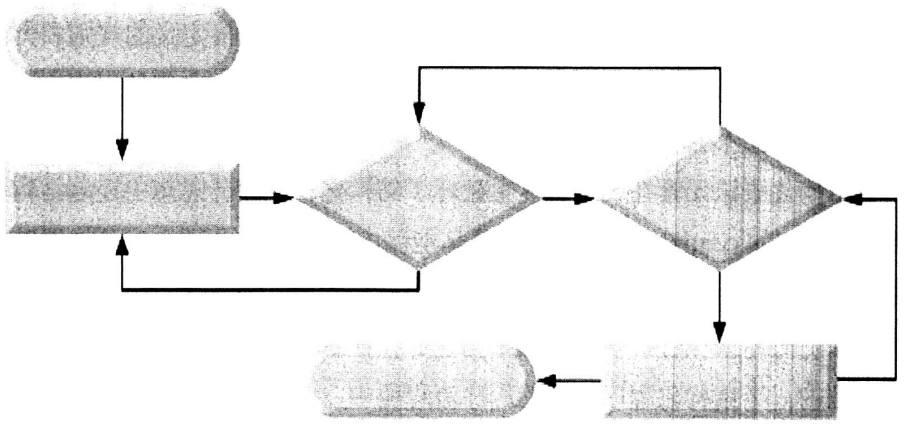


图 2.4 控制结构交叠的非结构化流程图

2、结构化流程图中，控制结构有且仅有一条入口流程线，有且仅有一条出口流程线。如图 2.5 所示，该图中的分支结构存在两个入口，是典型的非结构化流程图。

3、结构化流程图中，对于每一个结点都有一条从开始结点到结束结点的通路经过该结点。如图 2.6 所示，非结构化流程图中存在孤立的结点。

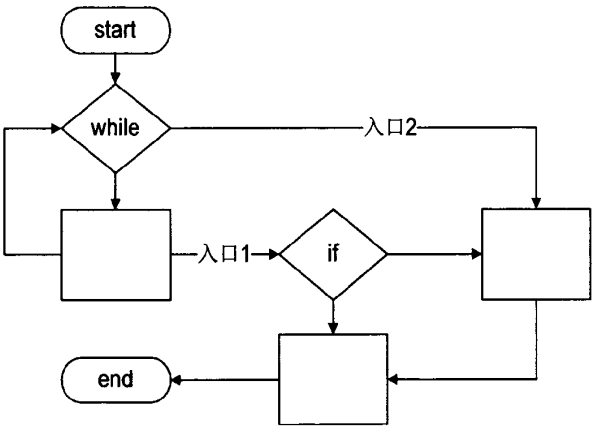
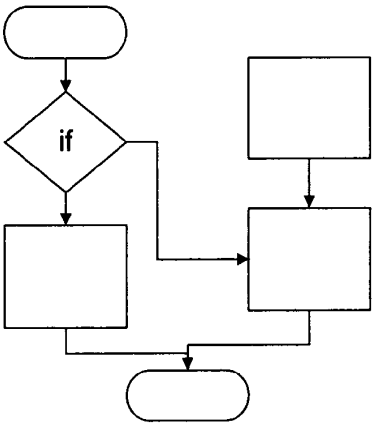


图 2.5 多入口控制结构的非结构化流程图图



2.6 带有孤立结点的非结构化流程图

2.3 半结构化程序流程图

2.1 小节中阐述了 GOTO 语句的危害性和优点。滥用 GOTO 特别是使用往回跳的 GOTO 语句，会使程序结构难于理解，应尽量避免。但有些情形下为了提高程序的效率，同时又不破坏程序的良好结构，有控制地使用一些 GOTO 语句是可以的。基于这些原因，在结构化程序设计思想基础上，大多数结构化程序设计语言加入了少量的扩展语法元素，如 C 语言中的 continue、break 和 return 语句。这些语法元素相当于严格控制下的“GOTO”语句，有着严格定义的规则和特定的使用场景。本文将这些语法元素统称为半结构化语法元素，它们使得程序结构更加灵活简练，又不会带来危害。

半结构化语法元素在一定程度上破坏了结构化程序设计思想中控制结构单入单出等准则。如 continue 语法使得循环体能够中途结束执行，break 语法能够强制退出循环的执行，将其运用在分支结构中造成分支出口不一致；return 语法使得程序能够随时结束执行，使得控制结构能够在任意处以程序结束作为结构出口。

相应的，半结构化程序对应的程序流程图中，某些流程线使得控制流表现出上述半结构化的特征。如图 2.7 所示的半结构化程序流程图，图中标注出了 continue、break 和 return 三种半结构化流程线。值得一提的是，这些半结构化元素只有位于分支结构中才有意义，能够发挥改变控制流的作用，否则其后面的流程结点无法执行，也会使循环结构失去意义。如图 2.8 所示，左图中的 continue 是没有意义的，右图中的 break 和 return 将循环体控制流中断，而实际上该 while 循环结构没有任何作用。

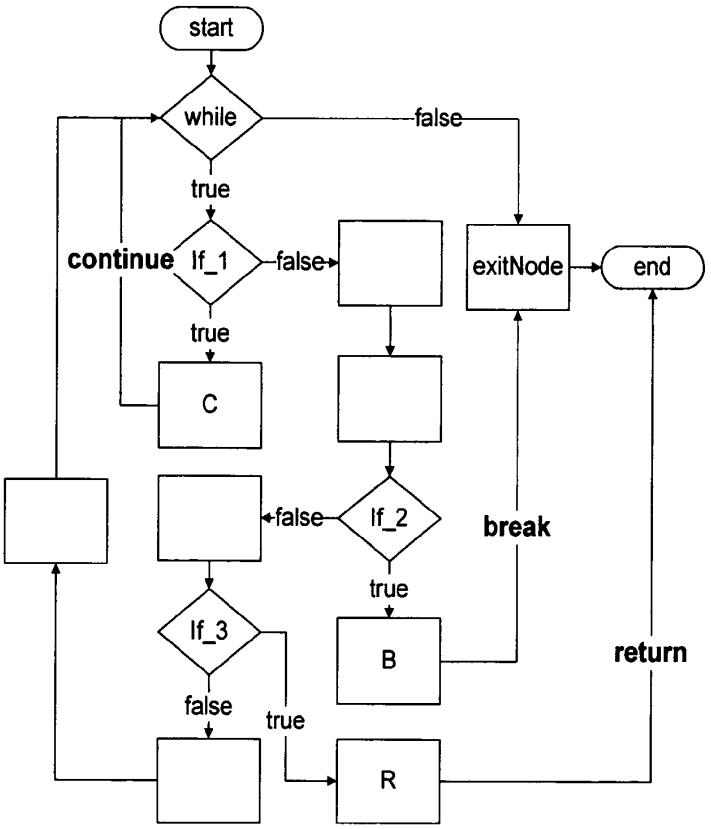


图 2.7 半结构化流程图

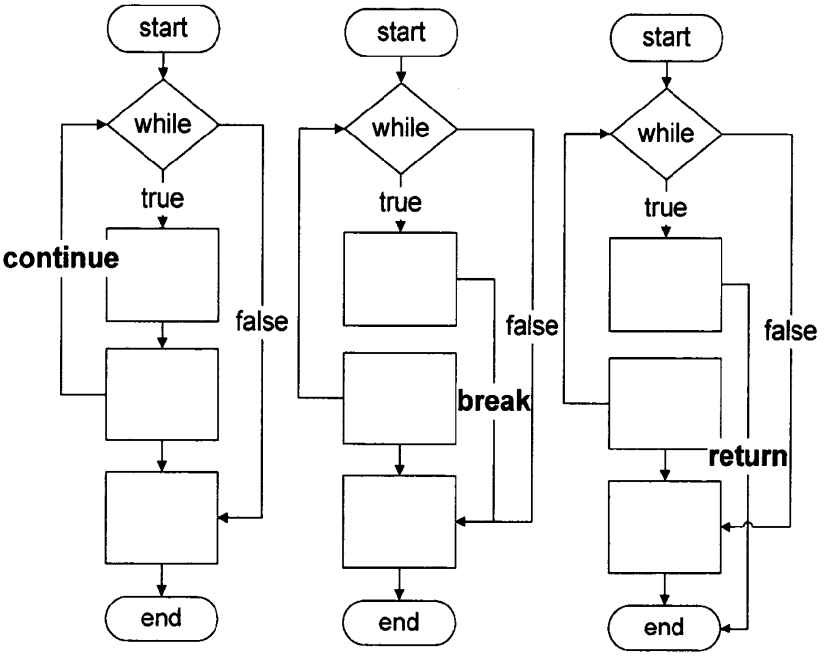


图 2.8 无意义的半结构化元素

2.4 本章小结

本文研究实现的流程图结构分析与识别算法适用于结构化、半结构化流程图，对于非结构化流程图，算法能够辨别其中的非结构化元素。算法基于的理论在于结构化、非结构化程序设计的思想。在结构化程序设计思想的指导下，程序流程图中的控制结构表现出严格的组成原则。

本章详细阐述了 GOTO 语句在程序设计中的重要影响，由此讨论了结构化、半结构化以及非结构化程序设计的不同和优缺点。随着计算机科学的日益发展，半结构化程序设计在工程应用中承担着举足轻重的地位。本文正是基于半结构化程序设计的思想，对其程序流程图进行了结构分析与识别的设计和实现。

第三章 程序流程图的结构分析与识别

本章主要讲述了程序流程图分析中基本控制结构的识别技术与方法。这些基本控制结构包括循环结构和分支结构，它们以复杂的组合、嵌套关系构成完整的流程图。循环结构和分支结构的共同点是都由菱形的判断结点引领，本文通过研究两种控制结构对应的流程图块的特征来进行流程图的结构分析与识别。只有正确的识别流程图中各种基本控制结构的类型才能够正确生成相应代码。由于循环结构较之分支结构具有更加明显的特征，且作用域容易圈定，因此本文从循环结构入手进行流程图的结构分析与识别。

3.1 循环结构的识别

3.1.1 流程图中循环结构的特征

在程序中，循环结构中的循环体语句块执行结束后程序的控制流会回到循环的条件判断语句，如果条件成立则控制流继续从循环体的第一条语句开始执行，否则控制流执行的下一条语句一定不属于循环体。对应程序的流程图，循环结构的特征是循环体中最后一个流程结点的后继结点必定是该循环结构的菱形判断结点，而判断结点的 true 分支后继结点必定是循环体中第一个流程结点，false 分支后继结点一定不是循环体中的任何流程结点。代码中常见的循环结构有 for、while 和 do while 三种，其中 for 循环与 while 循环在流程图上的表现形式一致。不失一般性，本文只识别和区分流程图中的 while 与 do while 循环结构，其不同之处在于菱形判断结点与循环体之间的执行顺序关系。图 3.1 列举了几种典型的循环结构流程图。

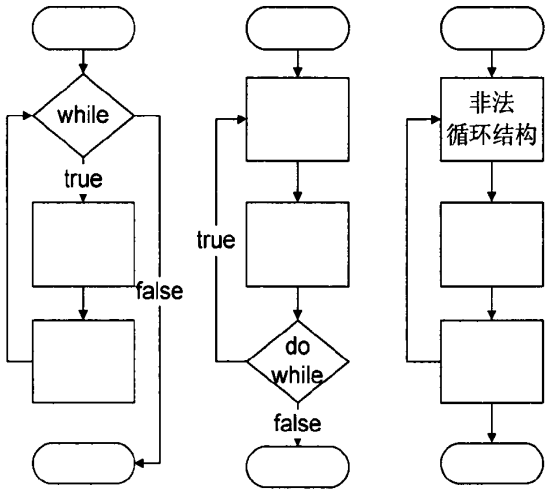


图 3.1 流程图中的循环结构

仔细研究不难发现，循环结构的流程图的典型特征是存在一条流程回线使得整个结构形成了一个环路，并且这条流程回线两端的结点必然包含该循环结构的判断结点，而这两个结点之间的所有通路上的所有结点都属于该循环结构的循环体（若循环结构中包含带有 break、return 含义的流程图元素，那么循环体还应包含其他一些非通路上的结点，这种情况后文详细阐述）。流程图中的顺序结构和分支结构中不可能存在这种环路，因此如果能够找出流程图中的环路结构并且验证该环路结构符合前述各种特征，即可断定该环路是循环结构。在有向图中，环路结构形成了有向图中的强连通子图（强连同分量）。

3.1.2 流程图中的强连通分量

定义 3.1 在有向图 G 中，如果两个顶点 vi，vj 间有一条从 vi 到 vj 的有向路径，同时还有一条从 vj 到 vi 的有向路径，则称两个顶点强连通(strongly connected)。如果有向图 G 的每两个顶点都强连通，称 G 是一个强连通图（SCG, strongly connected graph）。非强连通图有向图的各个极大强连通子图称为强连通分量（SCC, strongly connected components）。

流程图中的流程回线使得环路结构中任意两个流程结点之间存在一条有向路径，因此，在流程图中计算强连通分量并且验证每个强连通分量是否符合上述特征即可判断该 SCC 是合法的循环结构还是非法的结构。

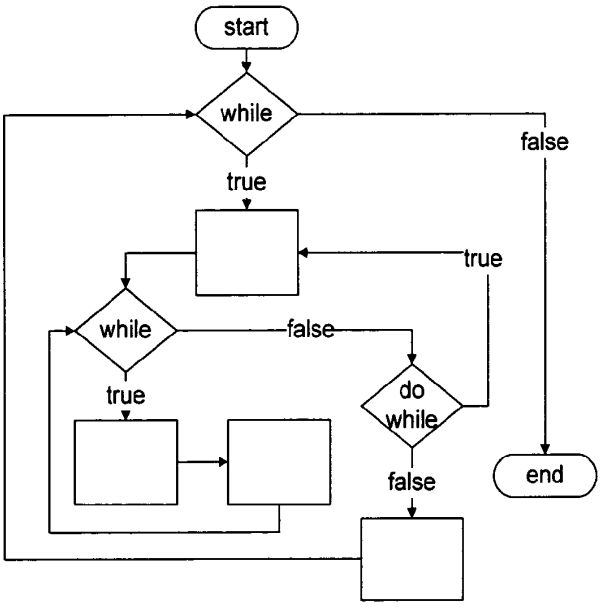


图 3.2 多个循环结构层层嵌套的流程图

有一点值得注意，求出流程图中的所有强连通分量并不代表找到了流程图中所有的循环结构。由于结构化流程图中，基本控制结构之间可能有着复杂的嵌套关系，如果存在多个循环结构层层嵌套的情况，如图 3.2 所示的 while-do

while-while 三层循环嵌套示例，由强连通分量的极大性可知，计算得到的强连通分量只对应最外层的 while 循环结构。针对这种情况，本文设计了一种递归求取循环结构的方法。算法将整个流程图作为初始的 SCC，计算内部的强连通分量，然后消去构成每个 SCC 环路的流程线，继续迭代直到找不到任何结点数大于 1 的强连通分量（这里假设流程图中不存在单一结点的空循环，否则算法的递归结束条件需稍作修改）。具体的计算流程如算法 3.1 所示。

算法 3.1 递归求取流程图中所有的循环结构

算法输入：被分析的流程图 SCC

算法输出：被分析的流程图中所有循环结构的强连通分量集合 totalSCCSet

```
void reverseSearchSCC(SCC)
{
    //在输入的流程图 SCC 中计算强连通分量集合 sccSet
    sccSet = computeSCC(SCC);
    if (sccSet is empty)
        return;                    //若集合 sccSet 为空，返回到上一层递归
    for each scc in sccSet
    {
        if (scc.nodeNum == 1)
            continue;              //若 scc 的结点数等于 1，则不是循环结构
        //将 scc 加入全局的强连通分量集合 totalSCCSet 中
        totalSCCSet.add(scc);
        disableCircle(scc);        //消去 scc 中构成环路的流程回线
        //将消去回线的 scc 作为新的流程图，递归寻找其内部的 scc
        reverseSearchSCC(scc);
    } //for
} //reverseSearchSCC
```

算法结束后，totalSCCSet 中存储了流程图中所有循环结构对应的强连通分量。算法中用于计算强连通分量的算法 computeSCC 和消去流程回线的算法 disableCircle 下文详细阐述。

3.1.3 强连通分量的计算

由强连通分量的定义可知，任何一个强连通分量必定是原有向图的深度优先搜索树的子树。那么只要在深度优先遍历时，把当前搜索树中未处理的结点加入一个堆栈，遍历回溯时确定每个强连通分量的子树的根，然后根据这些根从树的

最底层开始，从栈中取出每一个强连通分量即可。下面分析如何确定强连通分量的根和如何从最底层取强连通分量。

如何确定强连通分量的根，在这里维护两个数组，一个是 DFN[1..n]，一个是 Low[1..n]，其中 DFN[u]表示结点 u 的开始访问时间戳，即遍历时 u 的次序编号，Low[u]为 u 或 u 的子树能够追溯到的最早的栈中结点的次序号。按照定义，Low[u]满足公式(3-1)。

```
Low[u] = min
(
    DFN[u],
    Low[v], [u,v]为树枝边，u 为 v 的父结点                    式(3-1)
    DFN[v], [u,v]为指向栈中结点的后向边（非横叉边）
)
```

这样，在一次深度优先遍历的回溯过程中，如果发现 Low[u]==DFN[u]，那么当前结点 u 就是一个强连通分量的根，为什么呢？因为如果它不是强连通分量的根，那么它一定是属于另一个强连通分量，而且它的根是当前结点的祖宗，那么存在包含当前结点的到其祖宗的回路，可知 Low[u]一定被更改为一个比 DFN[u]更小的值。

如何取强连通分量，如果当前结点为一个强连通分量的根，那么它的强连通分量一定是以该根为根结点的子树。在深度优先遍历的时候维护一个堆栈，每次访问一个新结点，就压入堆栈。当前结点是这个强连通分量中最先被压入堆栈的，那么在当前结点以后压入堆栈的并且仍在堆栈中的结点都属于这个强连通分量。假设一个结点在当前结点压入堆栈以后压入并且还存在，同时它不属于该强连通分量，那么它一定属于另一个强连通分量，但当前结点是它的根的祖宗，那么这个强连通分量应该在此之前已经被取出。算法 3.2 描述了求取强连通分量的步骤。

算法 3.2 求取流程图中的强连通分量

算法输入：被分析的流程图 FC

算法输出：被分析的流程图中强连通分量的集合 SCCSet

```
void computeSCC(FC.startNode)
{
    n = FC.startNode;                  //从流程图的开始结点进行遍历
    DFN[n] = Low[n] = ++Index;         //为结点 n 设定次序编号和 Low 初值
    Stack.push(u);                     //将结点 n 压入栈中
    for each Edge (u, v) in FC         //枚举每一条边，向下一层递归遍历
    {
        if (v is not visted)           //如果结点 v 未被访问过
```

```
        {
            computeSCC(v);                    //递归继续向下找
            Low[u] = min(Low[u], Low[v]);
        }
        else if (v is in Stack)               //如果结点 v 还在栈内
            Low[u] = min(Low[u], DFN[v]);
    } //for
    if (DFN[u] == Low[u])                     //找到一个强连通分量，结点 u 是根
    {   //强连通分量集合中新增一个 SCC 元素
        SCCSet.add(new SCC());
        //栈顶到 u 的所有结点都属于该强连通分量
        do
        {   //栈顶结点 w 退栈，w 是该强连通分量的一个结点
            w = Stack.pop();
            SCCSet.getTail().addNode(w);      //结点 w 加入新增强连通分量中
        }while(u == v);
    } //if
} //computeSCC
```

该算法每个结点都被访问了一次，且只进出了一次堆栈，每条边也只被访问了一次，假设流程图中结点数为 N，流程线数为 M，则算法 3.2 在时间复杂度为 O(N+M)的情况下一次性求出流程图中所有的强连通分量。设流程图中循环结构的数目为 P，则算法 3.1 的时间复杂度为 O((P+1)(N+M))。

3.1.4 强连通分量环路的消除

由于结构化流程图中基本控制结构之间有着复杂的嵌套关系，当多个循环结构层层嵌套时，由强连通分量的极大性可知，computeSCC 得到的强连通分量集合只对应几个最外层的循环结构。要计算内部嵌套的强连通分量，必须将外层强连通分量的环路破坏掉。构成环路的流程回线一定与循环结构的根结点有关，而且强连通分量内部与根结点直连的流程线可能存在多条（例如 continue 或者嵌套的内层循环），如图 3.3 所示，内层嵌套的 do while 循环的 true 分支直连根结点。因此究竟应该消除哪条流程线需要在得到循环类型等上下文信息之后才能确定。环路的具体消除方法放到 3.2 节循环结构的类型判定后详细阐述。

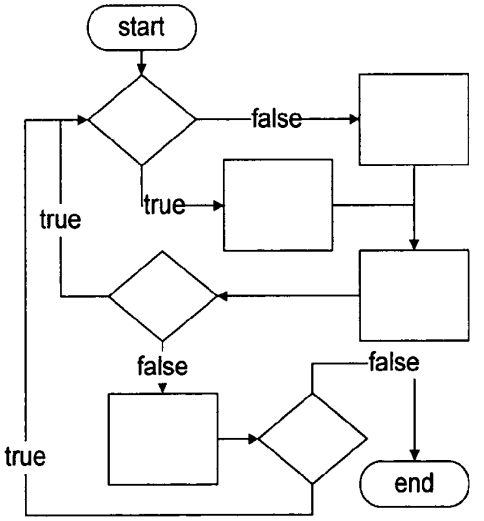


图 3.3 含有多条流程回线的循环结构

3.2 循环结构的类型判定

前文提到流程图中的循环结构存在多种形式，本文针对 while 和 do while 两种不同的形式对流程图中的循环结构进行类型上的判定和区分，其他循环形式总能等价为二者中的一种。由图 3.1 可知，while 与 do while 循环结构的不同在于进行强连通分量计算时 while 结构的根结点是菱形判断结点而 do while 结构的根结点是循环体首结点。

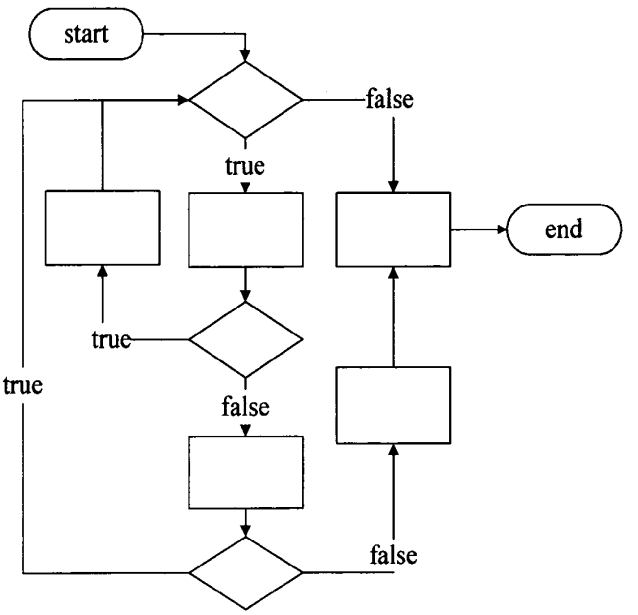


图 3.4 结构不易分辨的流程图

由于循环结构与分支结构之间可能存在复杂的组合、嵌套关系，且二者都包含菱形判断结点，再者循环结构内部可能存在半结构化流程图元素使得控制流错

综复杂，因此无法简单的根据两种循环结构的不同之处来判定循环的类型，还要结合图中其他信息进行判断。如图 3.4 所示，无法简单的判断图中的菱形判断结点是 if 还是 while 抑或是 do while。

因此，当一个复杂的强连通分量中包含多个菱形判断结点时，只有充分结合上下文的信息才能判断哪个结点是引领该循环结构的菱形结点，进而确定循环类型。后文详细阐述了如何寻找引领循环结构的菱形判断结点。

3.2.1 循环结构的条件判断结点

本文通过对简单的循环结构流程图进行观察和研究，总结出循环结构的条件判断结点应该具有的特征，然后按照归纳出来的结论在强连通分量中寻找符合条件的菱形判断结点。若找到唯一确定的结点，则可以立即判断出循环的类型，否则结合其他上下文信息进一步推断哪个结点是真正的条件判断结点。

尽管在程序中，循环结构的条件表达式满足的情况下才执行循环体中的语句。而实际上在程序流程图中，循环结构的条件判断结点的 true 分支并不一定要使得控制流流向循环体。由于本文提出的结构分析算法在复杂情况下会结合分支流向的信息来进行识别，所以这种灵活性无疑给流程图控制结构的分析带来困难。

试想一下，循环结构的强连通分量中，所有的菱形判断结点要么引领一个 if 分支结构，要么引领一个循环结构（对于多分支的判断结点可在预处理时识别并标记为 switch 结构，本文中提到的菱形结点默认指两分支的条件判断结点）。对于那些 true 分支流向强连通分量外部且 false 分支流向内部的判断结点，如果在结构分析之前将其两分支互换并作相应标记，这不会对结构的分析结果产生任何影响。在结构识别完毕后，对标记过的判断结点进行审核，若 if 结构的菱形结点分支被互换，则再次互换恢复原来的控制流向；若循环结构的菱形结点分支被互换，则直接将结点上附加的代码条件取反，以保证最终的代码生成正确无误。

按照上述方法对强连通分量进行预处理，处理后的流程图循环分支流向与代码一致，结构分析和识别时无需额外考虑。

由图 3.1 可知，while 结构的判断结点是所在强连通分量的根结点，并且其 false 分支首结点不属于强连通分量；do while 结构的判断结点的 true 分支首结点是所在强连通分量的根结点，并且其 false 分支首结点不属于强连通分量。因此，循化结构的判断结点应至少同时满足以下两个条件：

1、false 分支的首结点不属于强连通分量；

2、判断结点要么是强连通分量的根结点，要么 true 分支的直接后继是根结点。

按照上述条件在强连通分量中寻找循环结构的判断结点，搜索结果有如下几

种可能性。

1、 找不到合法的条件判断结点。很明显，这种情况下的流程图不是合法的结构化流程图。例如图 3.5 所示，图中强连通分量的两个判断结点都不符合上述条件，根结点 false 分支首结点仍然在强连通分量内部，另外一个结点的 true 分支并未直连根结点。实际上，图中的循环结构与分支结构形成了交叠的关系，不符合结构化的构成准则。

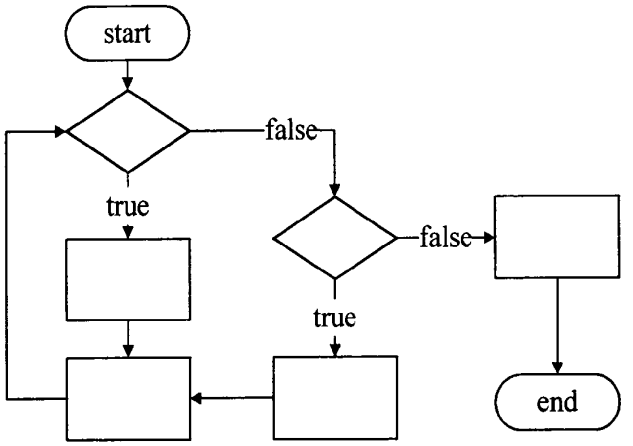


图 3.5 无合法判断结点的非结构化循环结构

2、 找到唯一一个合法的条件判断结点。若该结点是强连通分量的根结点，那么该循环一定是 while 类型。若该结点不是强连通分量的根结点且根结点是普通流程结点，那么该循环一定是 do while 类型，否则无法简单的判断图中的条件判断结点是 if 还是 while 抑或是 do while，该情况与第 3 点中的类似情况一起讨论和处理。

3、 找到两个合法的条件判断结点。若其中一个是强连通分量的根结点，例如图 3.4 所示，此时无法简单的判断图中的判断结点是 if 还是 while 抑或是 do while，从直观上看，其与第 2 点最后的情况都是两个菱形结点相连且下面的菱形结点符合条件，对于这种情况后文将详细说明如何判断该循环结构的类型。若两个都不是根结点，则属于多个非根判断结点的情况，例如图 3.6 所示，A、B 结点都是符合条件的判断结点。该情况与第 4 点中的类似情况一起讨论和处理。

4、 找到多于两个合法的条件判断结点。若其中一个是强连通分量的根结点，例如图 3.7 所示，根结点 ROOT 与结点 A、B 都是符合条件的判断结点，则该循环一定是 ROOT 结点引领的 while 循环类型，后文详细证明；若都不是根结点，类似图 3.6 同样属于多个非根判断结点的情况，后文将详细说明如何判断。

由以上结论，共有三种特殊情况需要结合上下文结构信息进一步分析和证明。

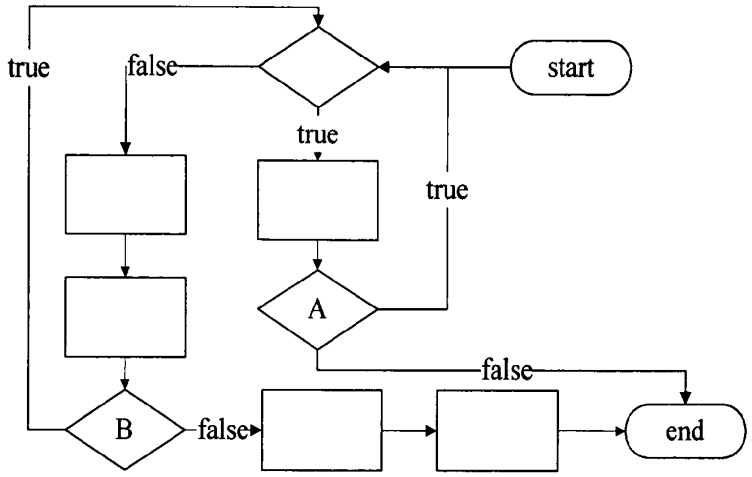


图 3.6 多个非根判断结点的循环结构

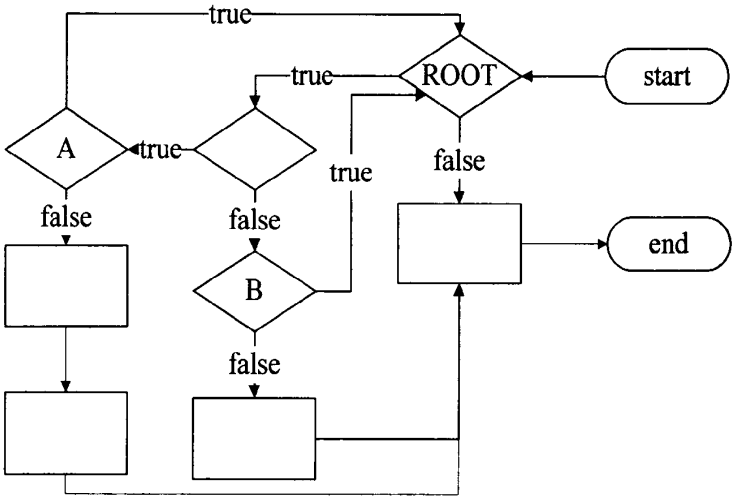


图 3.7 多个判断结点（含根结点）的循环结构

3.2.2 条件判断结点的真假分支

针对第一种特殊情况，当强连通分量的根结点是菱形结点，并且找到的符合条件的判断结点至多两个且仅有一个非根结点时，直观上看强连通分量中构成环路的流程回线两端都是菱形，此时可以结合两个判断结点的真假分支是否在强连通分量的内部来进一步推断循环的类型。为简便起见，将根结点称为 head，将另外一个非根的条件判断结点称为 tail，结合真假分支在强连通分量内外的信息（根据 3.2.1 小节中的处理，不会有 true 外 false 内的情况出现，且 tail 一定满足 true 内 false 外），可以构造如下的枚举真值表 3.1：

表 3.1 条件判断结点的真假分支枚举表

head \ tail	true 内 false 内	true 内 false 外	true 外 false 内	true 外 false 外
true 内 false 内				
true 内 false 外				
true 外 false 内				
true 外 false 外				

由于 tail 需要满足上一节中提到的两个条件，因此它一定是 true 内 false 外，只需进一步讨论 head 结点的不同情况。

1、true 内 false 内：head 结点不可能是 do while 类型。当它是 if 结构时，整个循环结构只能由 tail 引领并且只能是 do while 类型，如图 3.8(a)所示。当 head 是 while 结构时，由 true 内 false 内可知这个 while 循环是嵌套在强连通分量内部的，因此整个循环结构只能由 tail 引领且只能是 do while 类型，如图 3.8(b)所示。

综上，强连通分量为 do while 类型，由 tail 引领；head 类型未知，需下层递归中进行判定。

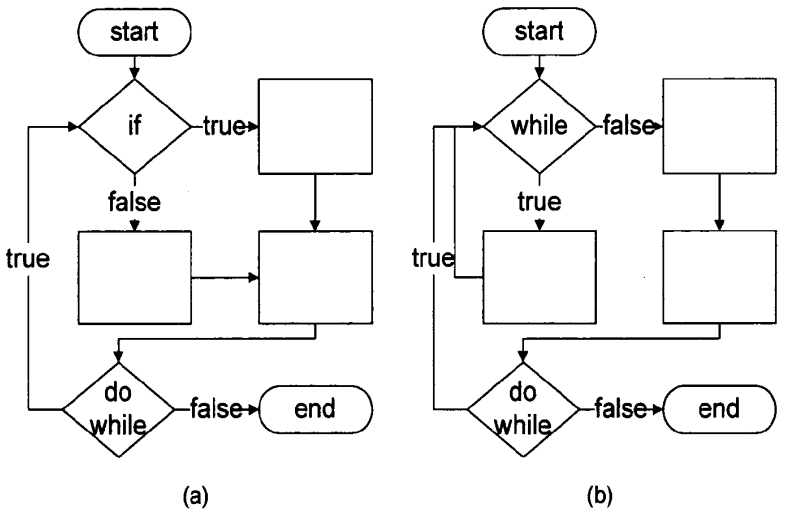


图 3.8 head 是 true 内 false 内的两种情况

2、true 内 false 外：head 结点与 tail 结点二者之一引领强连通分量对应的循环结构，另外一个结点引领的可能是分支结构也可能是循环结构，但一定是嵌套在内部的。由于其也具有 true 内 false 外的属性，对于 if 结构，一定是通过有 break/return 含义的半结构化元素使控制流流出强连通分量，而对于循环结构，内层嵌套的循环出口应属于外层强连通分量，也就是 true 内 false 内的情况，否则二者之间的关系一定是非结构化的。因此另外一个结点一定不会是循环结构，除非这是非结构化流程图。本文结合 head 与 tail 引领的两个结构的出口

结点之间的可达顺序进行结构类型的判定。先给出判定结果，再给出详细的判定说明。

最终判定结果有两种：

结果一：强连通分量为 head 引领的 while 结构，tail 的类型在下层递归中进行分析和识别，应该是 if 类型。

结果二：强连通分量为 tail 引领的 do while 结构，head 的类型在下层递归中进行分析和识别，应该是 if 类型。

判定过程如下：

记 head 的 false 分支首结点为 B 结点，tail 的 false 分支首结点为 A 结点。分支结构流出强连通分量的 false 分支有两种可能，下面分别进行分析。

1) 通过 break 含义的流程线流出循环。在此情况下，A、B 结点之间满足的关系有如下可能：i. A != end && B != end && A != B && A 与 B 之间存在强连通分量外部的通路；ii. A != end && B != end && A == B。

对于情况 i，当存在 A 到 B 的不流经强连通分量内部的路径时，例如图 3.4 所示，说明 A 结点所在分支的控制流通过 break 流程线流向了循环出口 B 结点，循环类型的判定结果为结果一；当存在 B 到 A 的不流经强连通分量内部的路径时，循环类型判定结果为结果二。

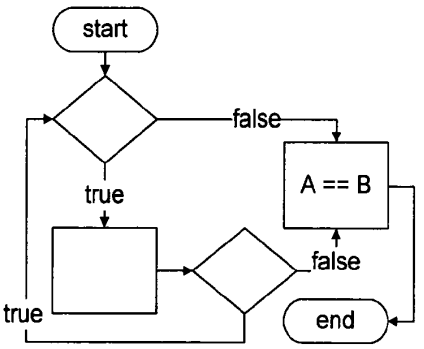
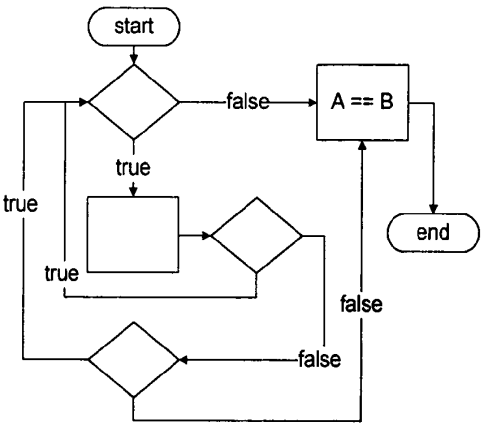


图 3.9 等价的两种循环结构图



3.10 多入边的 while 循环结构

对于情况 ii，还要结合 head 结点的入边情况进行具体讨论。当 head 结点的入边有且仅有一条（即 tail 直连 head 的流程线，如图 3.9 所示）时，两种判定结果是等价的，即强连通分量既可以解释为 while 循环又可以解释为 do while 循环。本文默认采取结果一。当 head 结点的入边有多条时，如图 3.10 所示，若 tail 引领了 do while 循环，当 head 结点是 if 类型时，head 的其他入边只能解释为 do while 结构，那么 if 的 false 分支通过 break 跳出了多层循环不符合结构化特征，当 head 结点是 while 类型时，其 false 分支直接跳出外层 do while 循环也不符合结构化特征；而如果强连通分量是 head 引领的 while 循环，那么其入边尾结点都可解释为

if类型，符合结构化特征。

2) 通过 return 含义的流程线流出循环。在此情况下，A、B 结点之间满足的关系有如下可能：i. A 与 B 结点中至少有一个是 end 结点；ii. A != end && B != end && A 与 B 结点之间不存在强连通分量外部的通路。

对于情况 i，当 head 结点的入边有且仅有一条时，例如将图 3.9 中的结点 A/B 设为 end 结点，两种判定结果是等价的，本文默认采取结果一。当 head 结点的入边有多条时，若所有入边的尾结点都是判断结点时，例如将图 3.10 中的结点 A/B 设为 end 结点，两种判定结果同样是等价的，可以解释为 while 循环内嵌套多个 if 结构也可以解释为多层 do while 循环嵌套只有 head 结点为 if 类型；否则若入边尾结点存在矩形结点，结论只能是结果一，例如图 3.11 所示，该循环只能是 head 引领的 while 循环。

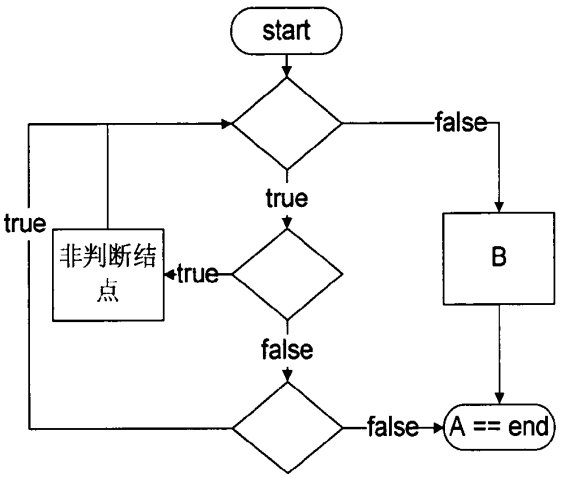


图 3.11 多入边的 while 循环结构

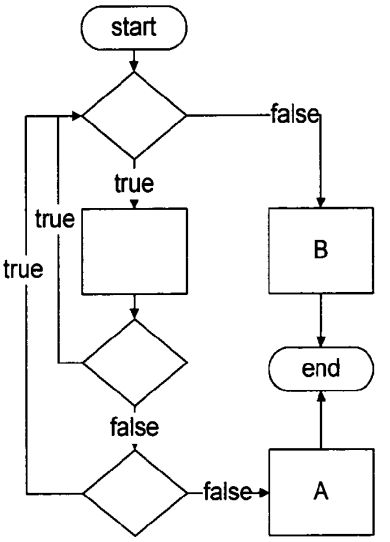


图 3.12 多入边等价的两种循环结构

对于情况 ii，与情况 i 的判定方式完全一致。例如图 3.12 所示，head 结点的入边有多条，且都是判断结点，那么该循环可以解释为 while 内嵌套多个 if 结构也可以解释为两个嵌套的 do while 循环，而 head 结点是最内层的 if 结构，其 false 分支通过 return 流程线跳出多层循环。

3) true 外 false 外：head 结点不可能是 do while 类型，也不可能是 while 类型。因此整个循环结构只能是由 tail 引领的 do while 类型。而 head 的类型需在下层递归中进行判定，且结果应该是 if 结构。

3.2.3 多个非根判断结点的区分

当强连通分量中存在多个符合条件的非根的判断结点且根结点不符合条件时，该循环一定是 do while 类型，原因如下。

首先根结点必须是菱形判断结点且一定是嵌套在内层的 while 循环，否则多

个条件判断结点对根结点的入边找不到结构化的存在理由。以图 3.6 为例进行说明，结点 A、B 是符合条件的非根判断结点。A、B 之中只有一个是真正引领循环的结点，不失一般性，假设 B 是引领 do while 循环的结点，A 结点等其他判断结点的 true 分支能够直连根结点只能解释为它们位于根结点引领的 while 循环中，其他情况都是非结构化的解释。

根结点的 while 循环嵌套在 do while 循环内部，除了引领 do while 循环的结点，其他判断结点都在 while 循环体内部，因此要找到引领 do while 循环的判断结点只需绕过 while 循环，从其 false 分支首结点开始遍历强连通分量并滤过根结点及其 true 分支，遍历结束后访问到的判断结点一定是真正引领 do while 循环的结点。若没有访问到或者访问到多个则说明流程图一定不是结构化的。

在图 3.6 中，从根结点的 false 分支开始遍历，恰好访问到唯一的一个判断结点 B，因此 B 结点是引领 do while 循环的条件判断结点。

3.2.4 多个条件判断结点中 while 的确定

当强连通分量的根结点以及两个或两个以上的非根结点都是符合条件的循环判断结点，那么这个循环一定是 while 类型。

以图 3.7 为例反证这个结论。图中的强连通分量有三个菱形判断结点 ROOT、A 和 B，它们都满足 true 内 false 外的条件。其中 ROOT 为根结点，A、B 为循环体结点且 true 分支直连 ROOT 结点。如果这个循环不是 ROOT 结点引领的 while 类型，那么一定是 A 或者 B 引领的 do while 类型。不失一般性，假设 A 是 do while 类型(B 是 do while 时同理)。此时 ROOT 不可能是 do while。如果 ROOT 是 while，则由 ROOT 的 true 内 false 外的特征可知，这个嵌套在 do while 循环内部的 while 循环的出口结点在父亲 do while 的强连通分量外部，相当于循环结束后直接跳出两层循环，不符合结构化的规范。因此 ROOT 一定是 if。此时 B 不可能是 while，因为 true 分支直连 ROOT 无法解释且 false 分支跳出两层循环；B 不可能是 do while，同样因为 false 分支跳出两层循环；B 也不可能是 if，因为 true 分支直连 ROOT 无法解释只可能是 GOTO。

由以上推断可知，若循环是 do while 类型则一定是非结构化流程图。若循环是 while 类型，A、B 等条件判断结点可以都解释为 if-continue-else-break 的半结构化形式。证毕。

3.2.5 循环结构判定算法

由3.1和3.2小节的分析可以归纳出循环结构类型判定算法的步骤为首先计算强连通分量中引领循环的菱形判断结点，然后根据上下文结构信息进行判断。算

法详细设计如下。

算法 3.3 计算强连通分量中引领循环的菱形判断结点

算法输入：被分析的强连通分量 scc

算法输出：scc 中引领循环的条件判断结点 leadingNode

```
Node findLoopLeadingNode(scc)
{
    boolean headSatisfy = false;
    Node temNode = null;
    int tailNum = 0;

    for each Edge e in scc.rootNode.outList
    {   //scc 的根结点是 true 内 false 外则说明 root 满足基本条件，加入链表中
        if (!scc contains e)
        {
            headSatisfy = true;
            scc.leadingNodeList.add(scc.rootNode);
            break;
        }
    }
    for each Edge e in scc.rootNode.inList
    {   //遍历根结点的所有入边的尾结点，如果 false 分支首结点在 scc 外，则满足基本条件，加入链表中
        temNode = e.tailNode;
        if (!scc contains temNode)
            continue; //不在 scc 中的结点直接滤掉
        for each Edge ee in temNode.outList
        {
            if (!scc contains ee.headNode)
            {   //false 分支首结点在 scc 外
                tailNum++;
                scc.leadingNodeList.add(temNode);
            }
        }
    }
```

```
if (headSatisfy)
{   //根结点符合条件
    if(tailNum == 0)
    {   //根结点为符合条件的判断结点
        return rootNode;
    }
    else if(tailNum == 1)
    {   //非根结点 tail 为符合条件的判断结点
        Node temNode = scc.leadingNodeList.get(1);
        scc.leadingNodeList.clear();
        scc.leadingNodeList.add(temNode);
        return temNode;
    }
    else
    {   //符合基本条件的非根结点大于 1，取根结点
        scc.leadingNodeList.clear();
        scc.leadingNodeList.add(scc.rootNode);
        return scc.rootNode;
    }
} //if headSatisfy
else
{
    if (tailNum == 0)
    {   //没有符合条件的判断结点
        scc.leadingNodeList.clear();
        return null;
    }
    else if(tailNum == 1)
    {   //恰好只有一个非根结点符合条件
        return scc.leadingNodeList.get(0);
    }
    else
    {   //多个符合条件的非根结点中选一个，根结点应为菱形结点
        int validLeadingNum=0;
        Node tem = null, tem2 = null, validLeadingNode = null;
```

```
queue q;
scc.rootNode.setVisitFlag(true);
tem = scc.rootNode.outList.get(1).headNode();
tem.setVisitFlag(true);
if (scc.leadingNodeList contains tem)
{
    validLeadingNum++;
    validLeadingNode = tem;
}
q.add(tem);
while (!q.isEmpty())
{   //从根结点的 false 分支首结点 BFS，寻找最外层的非根结点
    tem = q.dequeue();
    for each Edge e in tem.outList
    {
        tem2 = e.headNode;
        if (tem2.getVisitFlag() || !scc contains tem2)
        {   //已访问过或不在 scc 中，不遍历
            continue;
        }
        else if(scc.leadingNodeList contains tem2)
        {   //访问到了一个符合要求的非根结点
            validLeadingNum++;
            validLeadingNode = tem2;
            tem2.setVisitFlag(true);
            q.enqueue(tem2);
        }
        else
        {
            tem2.setVisitFlag(true);
            q.enqueue(tem2);
        }
    } //for
}//while
```

```
            if(validLeadingNum == 1)
            {   //成功找到最外层的非根结点
                scc.leadingNodeList.clear();
                scc.leadingNodeList.add(validLeadingNode);
                return validLeadingNode;
            }
            else
                return null;
        } //if tailNum > 1
    } //if !headSatisfy
} // findLoopLeadingNode
```

算法 3.4 判断循环结构的类型

算法输入：被分析的强连通分量 scc

算法输出：scc 对应的循环结构的类型 scc.style

```
boolean recognizeLoopStyle(scc)
{
    Node headNode = scc.rootNode;
    Node tailNode = scc.leadingNodeList.get(0);
    if (headNode == tailNode)
    {   //head 结点 true 内 false 外，是符合条件的判断结点，循环类型是 while
        if (scc contains headNode.outList.get(0).headNode)
        {
            headNode.setTag(WHILE);
            scc.setLoopStyle(WHILE);
            scc.setConditionNode(headNode);
            return true;
        }
        else
            return false;
    }
    //head != tail
    if (headNode.getStyle()==REC && tailNode.getStyle()==RHOM)
    {   //headNode 是矩形结点，tailNode 是菱形结点，循环是 do while 类型
        tailNode.setTag(DOWHILE);
        scc.setLoopStyle(DOWHILE);
```

```
        scc.setConditionNode(tailNode);
        return true;
    }

    if (!(headNode.getStyle() == RHOM && tailNode.getStyle() == RHOM))
        return false;  //head 和 tail 结点都是菱形才可能合法
    boolean flag1, flag2, flag3, flag4, flag5;
    Node A = null, B = null;
    flag1 = isTInFOut(headNode, scc);  //只有 true 内 false 外 flag1 才为 true
    flag2 = isTInFOut(tailNode, scc);
    if (flag1 && flag2)
    {   //head 和 tail 结点都是 true 内 false 外
        B = findNodeOutSCC(headNode, scc);  //head 结点 false 分支首结点为 B
        A = findNodeOutSCC(tailNode, scc);  //tail 结点 false 分支首结点为 A
        flag3 = (A == B);  //A 与 B 结点相同则 flag3 为 true
        //A 与 B 中至少有一个为流程图结束结点则 flag4 为 true
        flag4 = (A.getStyle() == END || B.getStyle() == END);
        //A 与 B 都不是流程图结束结点
        //且 A 与 B 之间不存在不流经 scc 内部结点的路径，则 flag5 为 true
        flag5 = (A.getStyle() != END && B.getStyle() != END \\
                && !hasPath(A, B, scc) && !hasPath(B, A, scc));
        if (flag3 || flag4 || flag5)
        {   //循环结构是 head 引领的 while 类型
            headNode.setTag(WHILE);
            scc.setLoopStyle(WHILE);
            scc.setConditionNode(headNode);
            return true;
        }
        else if (!flag3 && hasPath(A, B, scc))
        {   //循环结构是 head 引领的 while 类型
            headNode.setTag(WHILE);
            scc.setLoopStyle(WHILE);
            scc.setConditionNode(headNode);
            return true;
        }
```

```
        else if (!flag3 && hasPath(B, A, scc))
        {   //循环结构是 tail 引领的 do while 类型
            tailNode.setTag(DOWHILE);
            scc.setLoopStyle(DOWHILE);
            scc.setConditionNode(tailNode);
            return true;
        }
        else if (flag1 && !flag2)
        {   //head 结点 true 内 false 外，tail 结点不是 true 内 false 外
            //只能是 head 引领的 while 类型
            headNode.setTag(WHILE);
            scc.setLoopStyle(WHILE);
            scc.setConditionNode(headNode);
            return true;
        }
        else if(!flag1 && flag2) //head !TinFout, tail TinFout
        {   //head 结点不是 true 内 false 外，tail 结点 true 内 false 外
            //只能是 tail 引领的 do while 类型
            tailNode.setTag(DOWHILE);
            scc.setLoopStyle(DOWHILE);
            scc.setConditionNode(tailNode);
            return true;
        }
        else
            return false;  //其他情况都是非结构化流程图，无法识别循环的类型
        } //if (flag3 || flag4 || flag5)
    } //if (flag1 && flag2)
    return false;  //其他情况都是非结构化流程图，无法识别循环的类型
} //recognizeLoopStyle
```

3.2.6 强连通分量的环路消除算法

由 3.1.4 小节可知，要计算嵌套在循环内部的强连通分量，必须将外层强连通分量的环路破坏掉。

当循环结构是 while 类型时，构成强连通分量环路的流程回线是 tail 结点直连

循环判断结点即根结点的那条边。除此之外，循环体内部可能存在其他多条根结点的入边，在半结构化流程图中，这些边都是具有 continue 含义的半结构化元素。因此，while 循环根结点的入边应全部消除，否则半结构化的 continue 会被误认为构成了内层的循环而造成识别错误。

当循环结构是 do while 类型时，构成强连通分量环路的流程回线是循环判断结点即 tail 结点直连根结点的那条边。除此之外，循环体内部可能存在其他多条根结点的入边。但由于外层循环是 do while 类型，这些其他入边不可能是半结构化元素，只可能是内部构成循环的环路甚至非结构化的 GOTO 语句，当然这些需要下层递归进行分析和识别。因此，do while 循环只需将 tail 直连根结点的边消除即可。

需要消除的流程线只对内层强连通分量的计算产生影响，其本身是构成外层循环不可缺少的，因此这里所谓的消除只是设置某个标志使得计算强连通分量时忽略这些流程线，并不是真的删除它们。

算法 3.5 消除强连通分量的环路

算法输入：待处理的强连通分量 scc

算法输出：构成 scc 环路的流程线被设置禁用标志

```
boolean disableCircle(scc)
{
    if (scc.getLoopStyle() == WHILE)
    {    //while 类型，禁用 root 的所有强连通分量内的入边
        for each Edge e in scc.rootNode.inList
        {
            if (scc contains e.tailNode)
                e.setEnableFlag(false);
        } //for
        return true;
    } //if
    else if (scc.getLoopStyle() == DOWHILE)
    {    //do while 类型，禁用 tail 到 root 的流程线
        Node tailNode = scc.getConditionNode();
        Edge trueEdge = tailNode.outList.get(0);   //默认 true 分支的下标为 0
        if (trueEdge.headNode == scc.rootNode)
        {
            trueEdge.setEnableFlag(false);
            return true;
```

```
        }
        else
            return false;
    }
    else
        return false;
} //disableCircle
```

3.3 分支结构的识别

当流程图中某个强连通分量及其内部嵌套的所有强连通分量对应的循环结构类型确定后，该强连通分量中其他未知类型的条件判断结点一定是引领分支结构的 if 类型，当然，引领 switch 的多分支判断结点的出度大于 2，依据此特征无需分析可直接识别其结构类型。

3.4 半结构化流程图元素的识别

在识别循环结构类型的过程中，内部嵌套的 if 结构中有可能存在半结构化元素。半结构化元素的定义详见 2.3 小节。这些半结构化元素使得循环体中的控制流出现了中途结束、跳出循环甚至直接转向结束的特殊情况。依据这些特征，可以同时识别出流程图中相应的带有 continue、break 和 return 含义的半结构化元素。这些控制流被更改的分支路径上的结点原本应该属于分支所在的循环结构，因此在识别这些半结构化元素的同时，这些结点也加入到循环结构对应的强连通分量中，对其进行扩展。

当一个循环结构的强连通分量中嵌套了分支结构，且分支结构中存在半结构化的流程线时，循环结构单入单出的结构化特性被改变。如图 2.7 所示，分支中具有 continue 含义的流程线使得循环体控制流直接转向条件判断结点；具有 break 含义的流程线使得循环体控制流流出循环结构流向循环的出口结点；具有 return 含义的流程线使得循环体控制流直接流向流程图的结束结点。

3.4.1 continue 流程线的识别

在引领循环结构的条件判断结点的入度流程线中，如果流程线的尾结点属于强连通分量，那么该流程线具有 continue 含义，如图 2.7 中标记 continue 的流程线将 while 循环中 if_1 结构的 true 分支 C 结点之后的控制流转向了循环开始的判断结点。识别算法设计如下。

算法 3.6 识别循环结构中 continue 含义的流程线

算法输入：待识别的强连通分量 scc

算法输出：完成流程线的 continue 标记

```
void recognizeC(scc)
{
    Node conditionNode = scc.getConditionNode;
    for each Edge e in conditionNode.inList
    {   //scc 中判断结点的入边不是 break，则为 continue
        //同时是 continue 和 break 的情况，外层的 continue 不可以覆盖内层的 break
        if (scc contains e.tailNode && e.getTag() != BREAK)
            e.setTag(CONTINUE);
    }
}
```

3.4.2 break 和 return 流程线的识别

在循环结构内部嵌套的 if 分支中，带有 break 或者 return 含义的流程线使得整个分支从强连通分量中脱离出来。因此，对这样的分支进行深度优先遍历，直到访问到流程图的结束结点或者该循环结构的出口结点（即循环的条件判断结点的 false 分支首结点），遍历中连向结束结点或出口结点的流程线分别被识别为 return 或者 break，如图 2.7 中标记 break 的流程线将 if_2 结构的 true 分支 B 结点之后的控制流转向了循环出口结点 exitNode，标记 return 的流程线将 if_3 结构的 true 分支 R 结点之后的控制流转向了流程图结束结点 end。识别算法设计如下。

算法 3.7 识别循环结构中 return 和 break 含义的流程线

算法输入：待识别的强连通分量 scc，循环出口结点 exitNode，流程图结束结点 endNode

算法输出：完成流程线的 return 和 break 标记，将本应属于 scc 的扩展结点加入集合 scc.extendNodeSet

```
void recognizeBR(scc)
{
    for each Node node in scc
    {
        if (node.getStyle() != IF)
            continue;
        for each Edge e in node.outList
```

```
        {
            if (!scc contains e.headNode)
            {   //对 scc 中所有 if 结构不在 scc 中的分支进行遍历识别
                //DFS 从该分支首结点到 exitNode/endNode，将指向 exitNode 或
endNode 的边记录到 breakSet 和 returnSet 中
                if (e.headNode == endNode)
                    returnSet.add(e);
                else if (e.headNode == exitNode)
                    breakSet.add(e);
                else   //对该分支进行 DFS，遍历中寻找 return/break
                    calcBRDFS(e.headNode, exitNode, endNode, \\
                                breakSet, returnSet,  scc);
            } //if
        } //for
    } //for
    //注意不能把内层判断出来的标记给覆盖掉，比如外层判断的 break 覆盖内层
判断的 return 就错了。优先识别为 return，原因是若 if 出去的分支中嵌套有循环
会误识别为 break。
    for each Edge e in breakSet
    {
        if (e.getTag() == NORMAL)
            e.setTag(BREAK);
    }
    for (each Edge e in returnSet)
        e.setTag(RETURN);
} //recognizeBR

void calcBRDFS(fnode, exitNode, endNode, breakSet, returnSet, scc)
{
    if (fnode != exitNode && fnode != endNode)
        //将由于 if 的半结构化语句脱离 scc 的结点加入本该属于的 scc 中
        scc.extendNodeSet.add(fnode);
    if (fnode != exitNode)
    {
        for each Edge e in fnode.outList
```

```
        {
            if (e.headNode == endNode)
                returnSet.add(e);
            else if (e.headNode == exitNode)
                breakSet.add(e);
        } //for
    } //if
    else
        return;  //遇到边界结点，终止当前路径的遍历进行回溯遍历
    for each Edge e in fnode.outList
        //继续遍历结点的孩子结点
        calcBRDFS(e.headNode, exitNode, endNode, breakSet, returnSet, scc);
    return;
} //calcBRDFS
```

3.4.3 半结构化流程图元素的结构转化

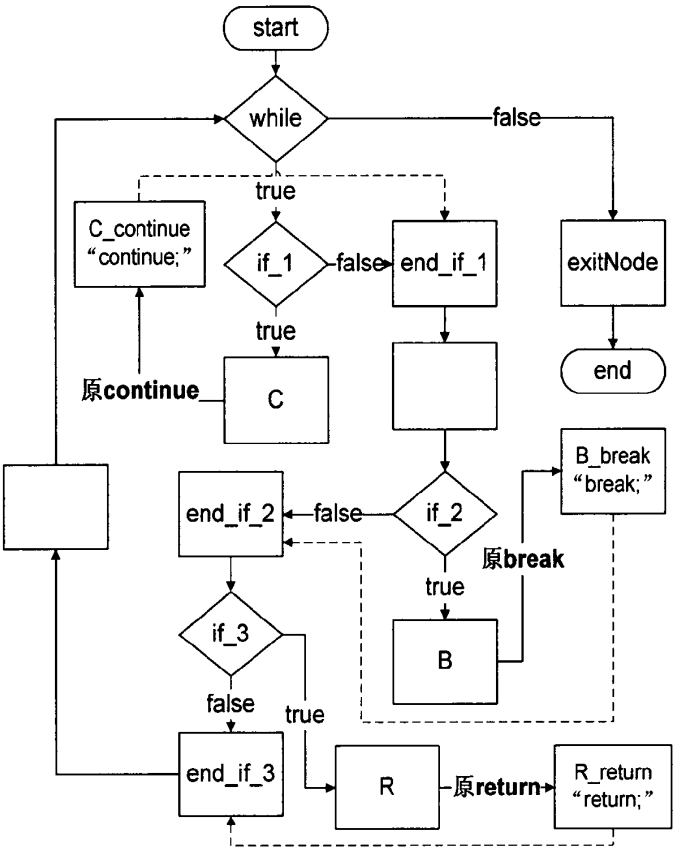


图 3.13 半结构化元素的结构转化

流程图中的半结构化元素对控制流造成的影响对代码自动生成带来了困难。针对半结构化元素，为了方便生成相应的 continue、break 和 return 代码，设法将半结构化流程图转化为结构化流程图，同时将半结构化元素的信息记录在转化后的结构化流程图中。尽管转化后的控制流发生了改变，但半结构化的语句代码能够正确生成。

结构化的分支结构中，所有分支的控制流必将在某结点处汇合。含有半结构化元素的分支的控制流被引向了他处，因此只需在这些半结构化流程线中加入携带代码信息的普通流程结点，将半结构化的控制流连向这些结点，并忽略这些元素对控制流带来的影响，将结点的后继重新设置为结构化控制流本该流经的汇合点。图 2.7 中非结构化的 continue、break 和 return 流程线经过如此处理之后，半结构化流程图转换成结构化的图 3.13，其中新增的结点 C_continue、B_break 和 R_return 中携带了原有的半结构化元素的代码信息。然而此时由于分支结构尚未处理，无法得知控制流本该流经的汇合点（标识为 end_if_*），所以控制流的填充需要在后面处理分支结构的阶段中继续进行，填充的流程线如图中虚线所示。

3.5 本章小结

本章详细分析并阐述了程序流程图的结构分析中基本控制结构的识别技术与方法。这些基本控制结构包括循环结构和分支结构，它们以复杂的组合、嵌套关系构成完整的流程图。同时针对流程图中存在的半结构化、非结构化元素也进行了识别算法的讨论。

针对循环结构，通过递归式的强连通分量计算来确定循环结构的作用域；通过寻找循环结构的条件结点判断循环类型，同时定位分支结构的条件结点。在寻找条件结点的过程中，结构之间复杂的嵌套关系和存在的半结构化元素给结点的定位带来了相当大的困难。本章详细讨论了各种可能性，将多数情况进行归纳总结。

针对半结构化元素，本章列举了 continue/break/return 流程线具有的特征，将它们的识别嵌入到了流程图的结构分析中。为了方便后续处理，应对这些半结构化元素进行预处理。本章同时给出了将其结构化处理的方法建议。

本章详细讨论了以上问题，并详细设计了相应的处理算法。

第四章 流程图的结构化验证及线性化处理

本章阐述了在流程图的结构分析与识别过程中如何辨别非结构化流程图，提出了结构化流程图的验证规则。同时阐述了在流程图结构合法的情况下，如何将分支、循环等基本控制结构线性化处理，将流程图转化为线性结构。

4.1 流程图的结构化验证

结构化流程图的分析工作不仅要能够识别出正确的结构化流程图中各种基本控制结构及其组合、嵌套关系，还要能够鉴别非结构化流程图，精确的指出非结构化流程图中违反结构化各项规范的结点和流程线。

4.1.1 基本验证

根据 2.2 小节中的描述，程序流程图定义了程序中所有可能的控制流流向。流程图中的每一个流程结点都应该在某条从开始结点到结束结点的控制流路径上。结构化程序流程图应首先满足如下 N 条简单规范：

1、 流程图中有且仅有一个开始结点，其入度为 0，出度为 1；有且仅有一个结束结点，入度>=1，出度为 0。

2、 流程图中所有矩形流程结点的入度>=1，出度为 1。

3、 流程图中所有菱形判断结点的入度>=1，出度为 2，两条出度流程线上的 true 和 false 标记必须完整。

4、 流程图中所有流程线都要有合法的头结点和尾结点。

5、 从流程图的开始结点对全图进行深度优先遍历，遍历过程中访问到的结点集合必须是流程图中所有结点的集合，同样访问到的流程线集合必须是所有流程线的集合。

4.1.2 控制结构之间的关系验证

根据 2.2 及 2.3 小节中的描述，半结构化流程图中的分支和循环结构之间的关系只有组合和嵌套两种，不可能出现交叠的情况。每个基本控制结构的入口都是单一的，由于半结构化元素的拓展，基本结构的出口可能有多处，但其流向都有既定的限制。

依据上述规范，对流程图中识别得到的各个循环结构的强连通分量作如下验证：

1、 识别半结构化元素的过程中，若某些流程线不符合半结构化的限制条件，应发现这些非结构化的地方。本文基于结点的后继可达集合提出了验证方法。如图 4.1 所示，图中的流程线 E 使得控制流流向了循环结构出口结点的后继可达集合，最终流向 end 结点，乍看下该分支是通过 return 流程线跳出循环的，而实际上它并不是半结构化的 return。从循环结构出口结点 exitNode 进行深度优先遍历，遍历过程中忽略本强连通分量中的结点，记访问到的后继可达集合为 Set1，则 Set1 表示循环执行完毕后即将执行的其他所有结点；对于循环结构中的分支结构，若某分支不属于该强连通分量，则从该分支首结点进行深度优先遍历，遍历过程中忽略本强连通分量中的结点，若遇到 exitNode 则回溯，记访问到的后继可达集合为 Set2，则 Set2 表示该分支执行的所有结点。若分支中含有半结构化元素，则 Set2 中的结点要么是属于该强连通分量的扩展结点，要么是 break 引向的 exitNode，要么是 return 引向的 endNode。根据式(4-1)计算集合 Set，若 Set 非空则分支的后继执行结点中出现了循环结束的后继执行结点，这显然是非结构化的现象。图 4.1 中，Set1 = {exitNode, D, B, C, endNode}，Set2 = {A, B, C, endNode}，Set = {B, C}非空，可见指向 B 结点的流程线 E 是非结构化的，只能解释为 GOTO。

$$Set = Set1 \cap Set2 - \{exitNode, endNode\} \qquad 式(4\text{-}1)$$

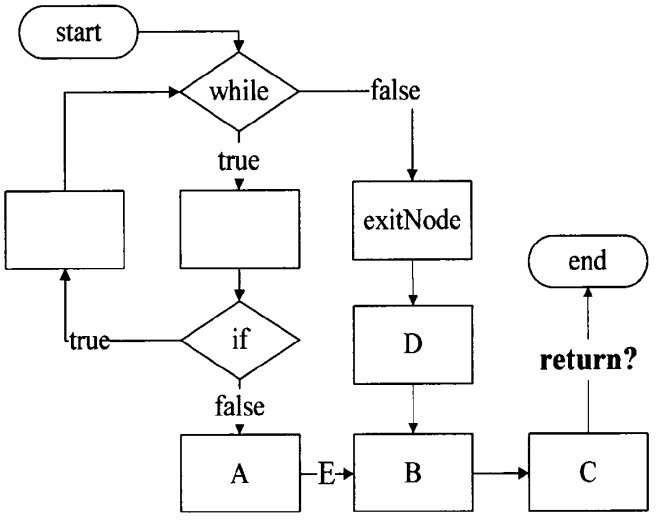


图 4.1 非结构化的“return”流程线

2、 检查强连通分量中除根结点的其他所有结点（包括识别半结构化元素过程中添加的扩展结点），这些结点的所有前驱结点都应该属于强连通分量，从而保证了循环结构入口单一。

3、 检查强连通分量中引领循环的判断结点的 false 分支首结点，该结点

属于父强连通分量。否则循环结构之间的嵌套关系被破坏，出现了控制流跳出多层循环的非结构化情况。

4.2 流程图的线性化

流程图中各个基本控制结构的分析和识别工作完成之后，为了方便代码自动生成，需要进一步识别各个结构的边界从而进行线性化处理。

4.2.1 循环结构的线性化

循环结构识别结束后即可进行线性化处理。本文算法框架中借鉴了标准流程图中循环界限的标识，将构成循环结构强连通分量中的环路去掉，并在循环的入口结点及出口结点处加入了循环上下界结点对整个循环的范围进行限定。处理后的循环结构可以按照线性的方式从循环上界开始遍历至循环下界，当然，对于内部嵌套的循环结构也递归式的做了线性化处理。

4.2.2 分支结构的线性化

循环结构的线性化操作完成后，流程图中仅剩分支结构以及未明确后继结点的半结构化信息结点。为了能够将分支结构转化为对应的代码，还需要明确分支结构的出口，即 true 分支和 false 分支的汇合点。分支结构的汇合点满足如下条件：汇合点的入度 = 作为分支结构汇点的数量 + 1。本文依据此条件分别在 true 分支和 false 分支中寻找合法的汇合点。由于流程图中同样存在分支结构层层嵌套的关系，因此寻找汇合点的算法也是以递归的方式处理的。当分支中存在未确定后继的半结构化信息结点时，汇合点的入度不满足上述表达式。在这种情况下，算法依据 if 结构的 true 分支中含有 continue/break/return 语句时 else 分支相当于不存在的特性，将这些半结构化信息结点的后继直接设置为 false 分支的首结点，同时将此首结点作为分支结构的汇合点。

4.3 本章小结

本章阐述了结构化流程图的分析算法设计中，流程图应满足的结构化验证规则。为了便于代码生成，本章同时给出了如何将分支、循环等基本控制结构线性化处理，将流程图转化为线性结构的大致方法。

第五章 流程图结构分析与识别的实现

本文的流程图结构分析与识别模块采用eclipse下插件开发的方式以java语言实现。系统建模模块将流程图存储为 XML 格式，本模块解析并建立相应流程图的数据对象，所有的分析与识别算法都围绕该对象展开。

5.1 相关数据结构设计

5.1.1 流程图

流程图是有向图，常用的存储方式有邻接矩阵、邻接表、十字链表等形式。在流程图的代码自动生成过程中，有对流程图进行结点与边的添加删除操作，因此不适于使用邻接矩阵或邻接表来存储该有向图。本文采用结点与边之间的链接（引用）方式来存储流程图，同时以 hash 表存储结点 ID 与结点引用之间的映射，以便快速的存取结点。

```
public class FlowChart
{   //流程图
    //结点ID与引用的映射集合
    private HashMap<Integer, FlowChartNode> nodeMap;
    private ArrayList<FlowChartConnecter> connecterList; //流程线数组
    private FlowChartNode startNode;                     //开始结点
    private FlowChartNode endNode;                       //结束结点
    //构造函数、数据成员的读写操作略
}
public class FlowChartNode
{   //流程图结点
    private String codeString;                           //结点对应的代码
    private int ID;                                      //唯一标识，从1开始
    private ArrayList<FlowChartConnecter> inList;        //入边流程线引用集合
    private ArrayList<FlowChartConnecter> outList;       //出边流程线引用集合
    private ENUMNODETYPE style;      //结点类型：RHOM，REC，START，END
    private ENUMTAG tag;  //结点标记：IF，WHILE，DOWHILE，\\
                          UPPER，LOWER，CONTINUE，BREAK，RETURN
    private boolean visitFlag;                           //遍历访问标志
```

```
    private int SCCLevel;                    //所在的强连通分量的层次
    private int DFSDistance;                 //DFS遍历顺序
    private int SCCDFN;                      //计算SCC的遍历序号
    private int SCCvisit;                    //计算SCC的访问标志
    private int SCCLow;                      //计算SCC的时间戳
    //构造函数、数据成员的读写操作略
}
public class FlowChartConnecter
{   //流程线
    int connecterID;                         //流程线唯一标识
    FlowChartNode headNode;                  //箭头指向的结点
    FlowChartNode tailNode;                  //流程线尾部的结点
    ENUMTAG tag;     //流程线标记：CONTINUE，BREAK，RETURN，NORMAL
    boolean enableFlag;                      //流程线启用标志
    //构造函数、数据成员的读写操作略
}
//流程图中特殊标志分类定义如下
public enum ENUMNODETYPE
{   //结点类型
    RHOM,               //菱形
    REC,                //矩形
    START,              //开始结点
    END                 //结束结点
    };
public enum ENUMTAG
{   //标记
    IF,
    WHILE,
    DOWHILE,
    UNKNOWN,        //未知
    UPPER,          //循环上界
    LOWER,          //循环下界
    CONTINUE,
    BREAK,
    RETURN,
```

```
    NORMAL,          //初始化为NORMAL
    EXTENDED,            //scc的扩展结点
};
```

5.1.2 循环结构强连通图

```
public class FlowChartSCC
{
    private int ID;                                    //强连通图的唯一标识
    private int level;                                 //强连通图的层次
    private Set<FlowChartNode> nodeSet;                //强连通图的结点集合
    private Set<FlowChartNode> extendNodeSet;          //强连通图的扩展结点集合
    private FlowChartNode headNode;                    //强连通图的根结点
    private ArrayList<FlowChartNode> tailNodeList;     //tail结点
    private ENUMTAG loopStyle;                         //循环类型
    private int upperNodeID;                           //循环上界结点ID
    private int lowerNodeID;                           //循环下界结点ID
    private int conditionNodeID;                       //引领循环的菱形结点ID
    //构造函数、数据成员的读写操作略
}
```

5.2 流程图结构分析与识别的实现

流程图的结构分析与识别算法从循环结构对应的强连通分量入手，以递归的形式找出流程图中各个循环结构及其组合、嵌套关系，然后根据强连通分量分支、出口等上下文结构信息判断循环结构的类型，同时确定 if 分支结构的菱形判断结点。

5.2.1 分析与识别的递归框架

```
public boolean flowChartStructureAnalysis(FlowChartSCC scc)
{   //找到最外层的强连通分量
    ArrayList<FlowChartSCC> tmpScgList = computeSCC(scc.getHeadNode(), scc);
    Iterator<FlowChartSCC> iter = tmpScgList.iterator();
    while(iter.hasNext())
    {
```

```
FlowChartSCC tmpScg = (FlowChartSCC)iter.next();
Set<FlowChartNode> tmpScgNodeSet = tmpScg.getNodeSet();
if (tmpScgNodeSet.size() == 1)
{   //如果强连通分量只有一个结点
    for (FlowChartNode node : tmpScgNodeSet)
    {   //如果是菱形，标识结点
        boolean bResult = recognizeOneNode(node);
        if (!bResult){
            return false;
        }
    }
    continue;
}
if (!judgeInitialScgLegal(tmpScg))
{   //初步检查scc的合法性，看是否有入边进入到scc的多个结点
    return false;
}
//多个结点构成的强连通分量对应一个循环结构，将其加入sccList中
sccList.add(tmpScg);
//在强连通分量中寻找引领循环的判断结点
boolean bResult = findLoopLeadingNode(tmpScg);
if (!bResult){
    return false;
}
//把所有指向scc外的变为F分支
boolean changeResult = changFOut(tmpScg);
if(!changeResult){
    return false;
}
//识别循环结构的类型
int tem = recognizeLoopStyle(tmpScg);
if (tem != 0){
    return false;
}
//将被调换TF分支的if结点复原
```

```
        boolean changeBackResult = changBackIfTF(tmpScg);
        if (!changeBackResult){
            return false;
        }
        //禁用强连通分量的环路，以便寻找嵌套的循环结构
        disableCircle(tmpScg);
        //递归调用，在强连通分量内部继续分析和识别
        bResult = flowChartStructureAnalysis(tmpScg);
        if (!bResult){
            return false;
        }
        //验证循环结构的出口结点属于父scc
        bResult = judgeSccLevelLegal(tmpScg, scc);
        if (!bResult){
            return false;
        }
        //在强连通分量中识别半结构化流程图元素
        bResult = recognizeCBR(tmpScg);
        if (!bResult){
            return false;
        }
        //验证强连通分量中除根结点的其他所有结点（包括扩展结点）
        //这些结点的所有前驱结点都应该属于强连通分量
        bResult = judgeSccEntryLegal(tmpScg, scc);
        if (!bResult){
            return false;
        }
    }

/*globalSCC中的scc都处理完后，对于不属于任何scc的if结构中的return边要进行标记。这里暂时将end结点所有不是break的入边标记为return，这样整个程序正常结束后仍然生成return代码*/
    if (scc.getLevel() == 0)
    {
        for (FlowChartConnecter c : flowChart.getEndNode().getInList())
```

```
        {
            if (!c.getTag().equals(ENUMTAG.BREAK))
                c.setTag(ENUMTAG.RETURN);
        }
    }
    return true;
}
```

5.2.2 非结构化错误处理

本文结构化流程图的分析工作能够识别出正确的结构化流程图中各种基本控制结构及其组合、嵌套关系，同时能够辨别非结构化流程图，精确的指出非结构化流程图中违反结构化各项规范的结点和流程线。

在结构分析和识别的过程中，算法在任意阶段都有可能找到流程图中存在的非结构化现象。例如在进行强连通分量的结构化验证时发现多入口、结构交叠等非结构化情况；在寻找循环结构的引领判断结点时找不到或找到多个符合条件的结点；在判断循环结构类型的过程中发现找不出结构化的解释或者外层和内层的判断出现不一致的情况；在识别循环结构中的半结构化元素时发现某些流程线有着非结构化的 GOTO 语义等等。

在整个层次化代码生成框架中一旦用户绘制的流程图中出现了非结构化元素，建模工具应及时指出这些地方，并向用户给出图中的错误和警告信息。本文借鉴面向对象中异常处理的思想，定义了一个全局的错误处理类，在算法的任意阶段，一旦发现流程图无法进行结构化解析，则将相关的错误和警告信息写入错误处理类对象中，强制算法中止并返回错误标志。上层建模模块可以提取错误信息并在界面上加亮提示用户。错误处理类和处理机制设计如下。

```
public class ErrorMessage
{
    private boolean noError;                                    //错误标志
    private Set<FlowChartNode> errorNodeSet;    //错误提示结点集合
    private Set<FlowChartConnecter> errorConnecterSet;   //错误提示流程线集合
    private String errorMessage;                                //错误提示信息
    public void setErrorNodeSet(Set<FlowChartNode> flowChartNodeSet) {
        this.errorNodeSet = flowChartNodeSet;
    }
    public void setErrorConnecterSet(
```

```
            Set<FlowChartConnecter> flowChartConnecterSet) {
        this.errorConnecterSet = flowChartConnecterSet;
    }
    public void setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
    }
    //构造函数、数据成员的读写操作略
} //ErrorMessage
public ErrorMessage getLastError()
{
    return errorMessage;
}
```

当算法执行过程中发现了非结构化情况，算法调用 ErrorMessage 类的几个 set 函数将可能的非结构化结点、流程线等元素记录在 errorMessage 对象中。算法给上层建模模块返回预定的错误标志。建模模块接收到错误标志调用 getLastError() 函数即可从 errorMessage 对象中提取分析算法记录下来的错误信息，最后向用户给出提示。

5.2.3 算法复杂度分析

设流程图中结点总数为 N，流程线总数为 M，循环结构的总数为 P。

本文实现的数据流结构分析和识别算法每一层迭代处理一个循环结构，整个流程图作为最外层的一个强连通图，因此算法共处理了(P+1)个循环结构。

从空间上看，需要辅助空间的操作大多在于强连通分量的计算、流程图的遍历以及算法的递归迭代上。详细开销分析如下。

计算强连通分量 computeSCC 的空间开销为(3N)，用于结点辅助标记和遍历栈；计算引领循环的判断结点 findLoopLeadingNode 的空间开销为(N)，用于 BFS 的遍历队列；判断循环结构的类型 recognizeLoopStyle 的空间开销为(N)，用于计算结点间路径的 DFS 遍历栈；禁用强连通分量环路 disableCircle 的空间开销为 1；识别循环结构中半结构化元素 recognizeCBR 的空间开销为(2N+M)，用于 Set1 和 Set2 集合计算的 DFS 遍历栈以及 break/return 流程线集合的存储。

综上所述，算法辅助空间开销为(P+1)(7N+M)，空间复杂度为 O((P+1)(N+M))。

从时间上看，开销大多在于算法对流程图的遍历和递归迭代上。computeSCC 的时间复杂度为 O(N+M)，findLoopLeadingNode 的时间复杂度为 O(N+M)，recognizeLoopStyle 的时间复杂度为 O(N+M)，disableCircle 的时间复杂度为 O(1)，

recognizeCBR 的时间复杂度为 O(N+M)。

综上所述，算法的时间复杂度为 O((P+1)(N+M))。

5.3 流程图的代码自动生成实验

5.3.1 实验选取

基于流程图的结构分析与识别技术，课题组实现了流程图代码自动生成模块。为了验证该模块的正确性、鲁棒性以及完备性，同时测试模块的代码生成效率，本文针对不同方面筛选了 150 个典型的流程图，详细设计了测试用例库如表 5.1 所示。

表 5.1 流程图代码自动生成模块测试用例

用例类别		用例描述	用例数
结构化/半结构化流程图用例	循环结构	简单的 while/do while 循环，T、F 分支反向的循环	4
		head/tail 结点均为菱形结点的各种复杂情况	16
		多循环组合、嵌套	12
	分支结构	简单的 if 分支	4
		多分支组合、嵌套，包含共用汇点，无半结构化元素	4
		多分支组合、嵌套，包含共用汇点，有半结构化元素	16
	综合	循环与分支混合，三层以上嵌套，无半结构化元素	4
		循环与分支混合，三层以上嵌套，有半结构化元素	16
		其他任意设计的测试用例	20
非结构化流程图用例	违反结构化原则	流程结点、流程连线不完整	4
		循环中不存在合法判断结点	4
		多入口循环结构，包含扩展结点入口	6
		循环没有出边或循环的出边不正确	6
		循环的出口非结构化	6
		分支结构汇点不存在、不一致	10
		分支结构的 T、F 分支有共用	4
	违反半结构化	break 到外层循环	6
		continue 到外层循环	6
		break/return 连向循环出口结点的后继可达集合	6

5.3.2 实验结果

针对近 100 个正确的结构化、半结构化测试用例，以及超过 50 个错误的非结构化测试用例，流程图代码自动生成模块正确分析和识别出了每个流程图的结构，同时生成了语义正确的代码；另一方面辨别出了非结构化流程图并依照错误处理机制指出了造成非结构化特性的流程结点和流程线。

图 5.1 至 5.6 是几个典型的半结构化流程图测试用例，经模块识别后生成的相应代码如表 5.3 和表 5.4 所示。图 5.7 至 5.10 是几个典型的非结构化流程图测试用例，模块检测出的错误元素已经被可视化工具加亮标红显示，具体错误见图注。

实验平台为 HP Pro 2080 MT，CPU 为 Intel Core2 E7500 3Ghz，内存 2G，运行环境是 Eclipse Application。采取执行 100 次取平均值的方式测试得到上述合法用例从解析流程图 xml 到识别结构、代码自动生成的时间开销如表 5.2 所示，结合 5.2.3 小节算法复杂度的分析可知课题组实现的流程图代码生成模块效率较高。

表 5.2 部分合法测试用例时间开销

用例 ID	结点数	流程线数	循环结构数	分支结构数	时间开销(ms)
图 5.1	17	20	3	3	6.124
图 5.2	15	19	1	4	4.75
图 5.3	18	22	1	4	7.406
图 5.4	14	19	1	4	4.626
图 5.5	19	23	0	5	8.562
图 5.6	15	21	3	4	5.718

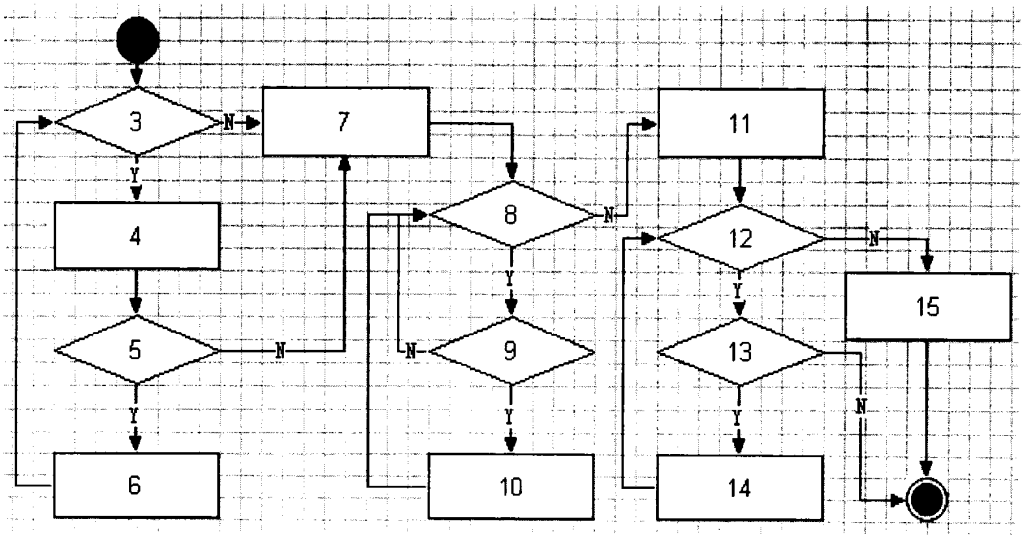


图 5.1 有半结构化元素的循环结构组合用例

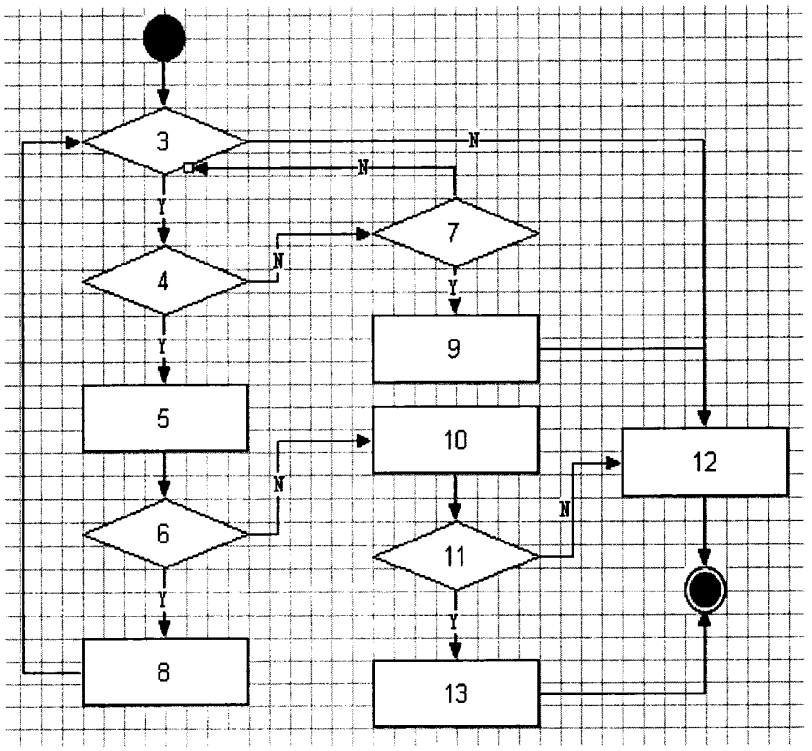


图 5.2 while 循环结构中有半结构化元素的分支结构嵌套用例

表 5.3 代码自动生成实验结果

图 5.1 代码	图 5.2 代码	图 5.3 代码

void function()
{
while(3)
{
4;
if(5)
{
6;
continue;
}
break;
}
7;
while(8)
{
if(9)
{
10;
continue;
}
}
11;
	void function()
	{
	while(3)
	{
	if(4)
	{
	5;
	if(6)
	{
	8;
	continue;
	}
	10;
	if(11)
	{
	13;
	return;
	}
	break;
	}
	if(7)
	{
		void function()
		{
		do
		{
		3;
		if(4)
		{
		5;
		if(6)
		{
		10;
		continue;
		}
		12;
		if(13)
		{
		14;
		break;
		}
		15;
		return;
		}

```
    while(12)
    {
        if(13)
        {
            14;
            continue;
        }
        return;
    }
    15;
}
```

```
            9;
            break;
        }
    }
    12;
}
```

```
        if(7)
        {
            8;
            break;
        }
        9;
        return;
    } while (11);
    16;
}
```

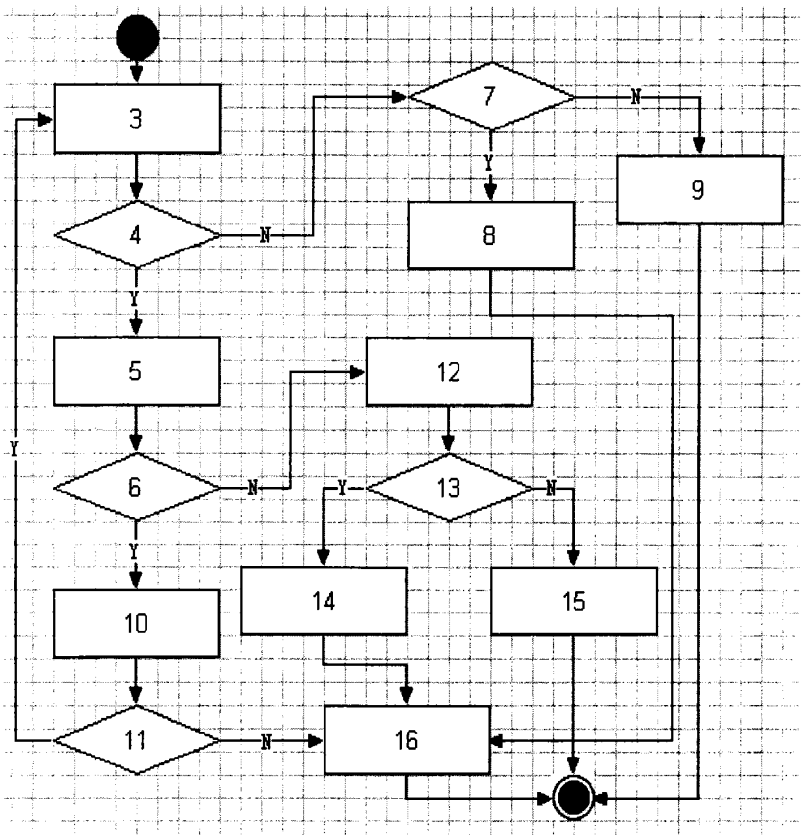


图 5.3 do while 循环结构中有半结构化元素的分支结构嵌套用例

表 5.4 代码自动生成实验结果

图 5.4 代码	图 5.5 代码	图 5.6 代码

void function(){
if(3){
while(4){
if(5){
6;
continue;
}else{
}
	void function(){
	if(3){
	if(4){
	5;
	if(8){
	10;
	return;
	}
		void function(){
		do{
		3;
		if(9)
		{
		if(!(10))
		{
		continue;

```
            if(7){
                continue;
            }else{
            }
            8;
            break;
        }
    }else{
        do{
            9;
            if(10){
                continue;
            }else{
            }
            break;
        }while(11);
    }
    12;
    return;
}
```

```
            11;
        }
        else{
            6;
            if(9){
                12;
            }
            else{
                13;
            }
        }
        14;
    }
    else{
        7;
    }
    if(15){
        16;
        return;
    }
    17;
}
```

```
            }
            if(!(11))
            {
                break;
            }
            if(!(13))
            {
                return;
            }
        }
        do{
            4;
            do{
                5;
            }while(6);
        }while(7);
    }while(8);
    12;
}
```

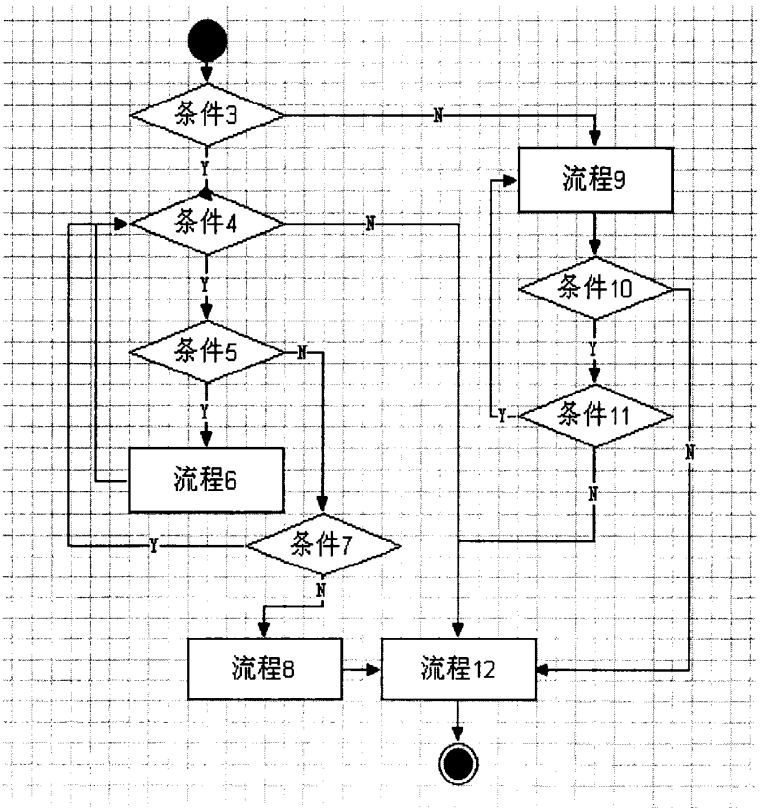


图 5.4 分支结构与循环结构相互嵌套用例

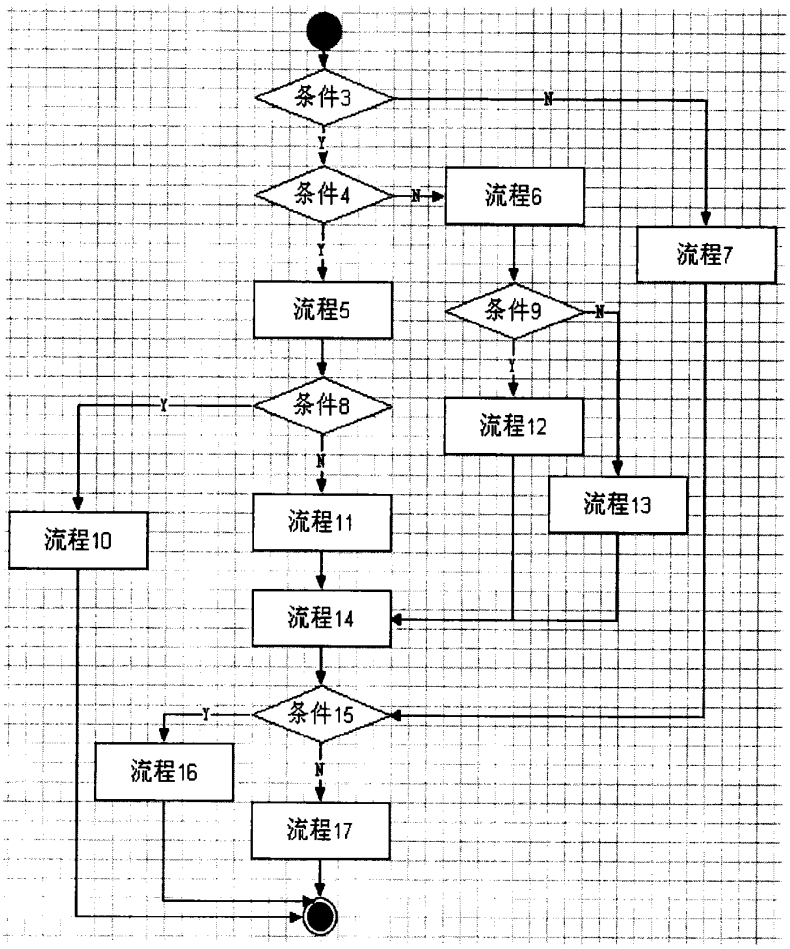


图 5.5 多层分支结构嵌套及组合用例

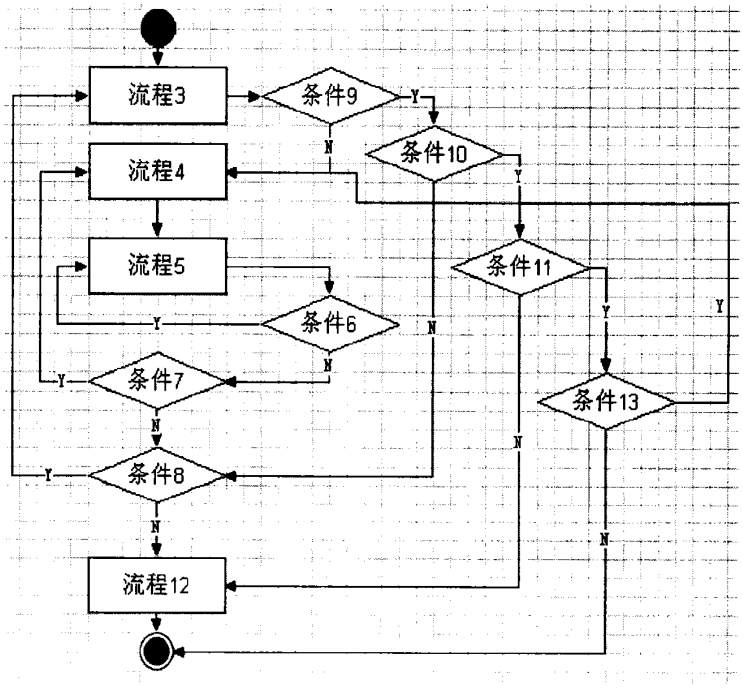


图 5.6 有半结构化元素的三层分支结构及循环结构嵌套用例

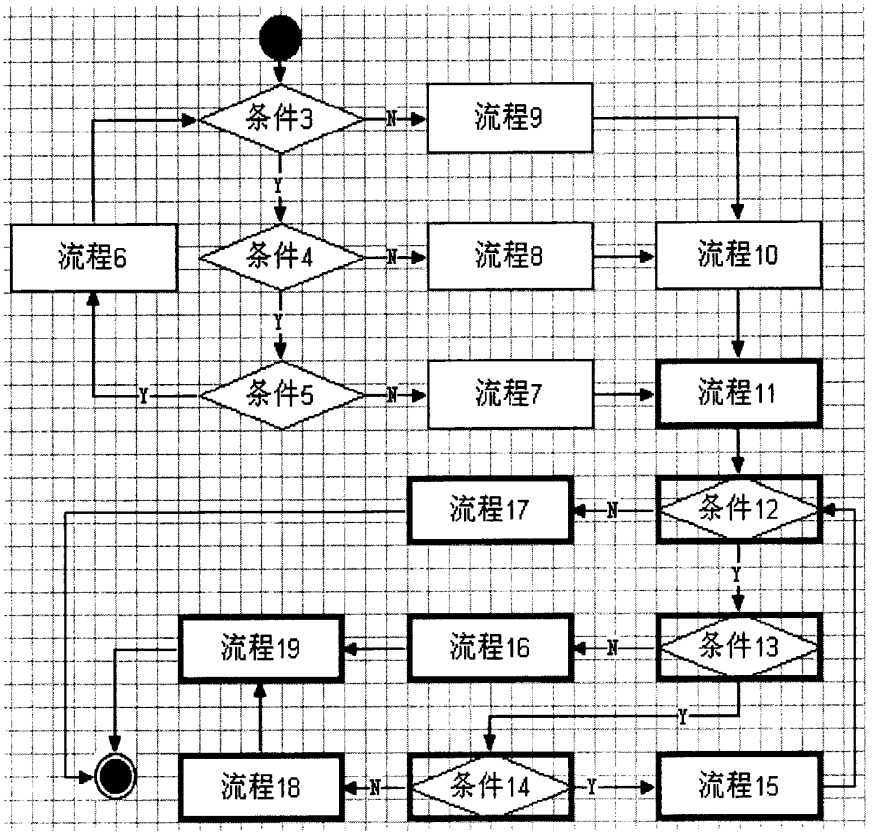


图 5.7 循环结构多出口不一致的非结构化用例

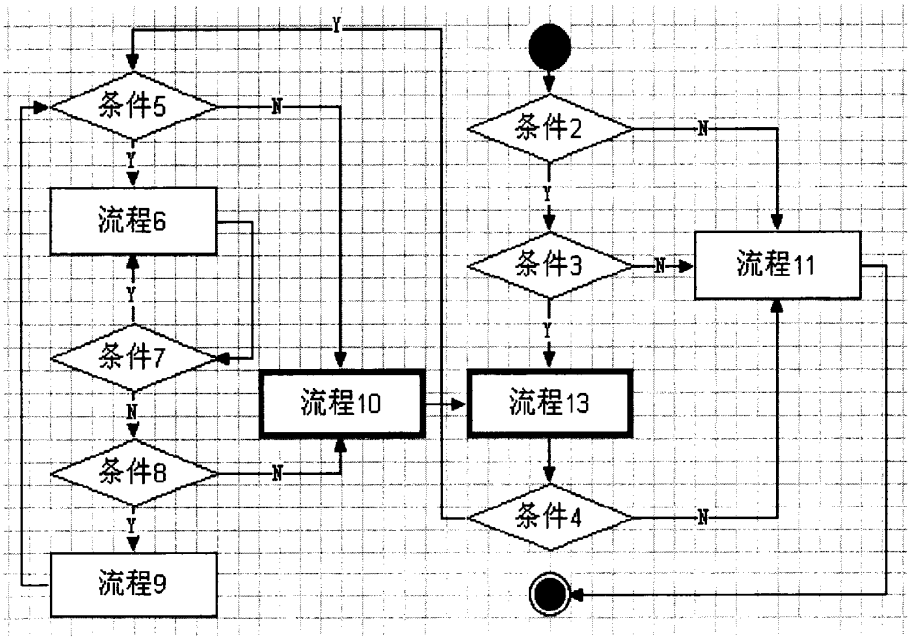


图 5.8 循环结构的出口交叠至上层分支中的非结构化用例

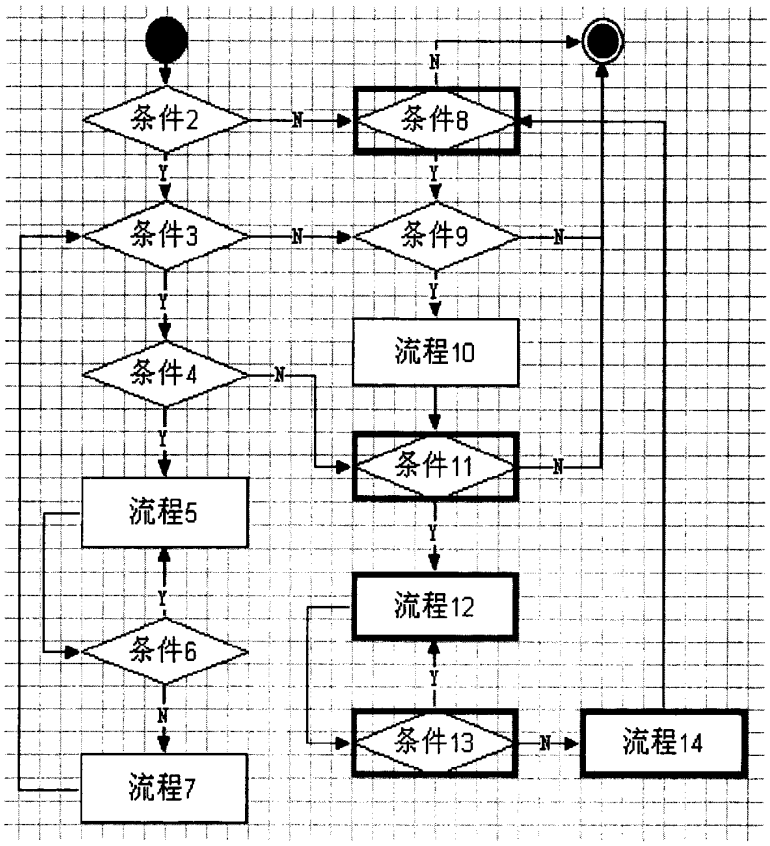


图 5.9 跳出循环的分支与循环的后继可达集合交叠的非结构化用例

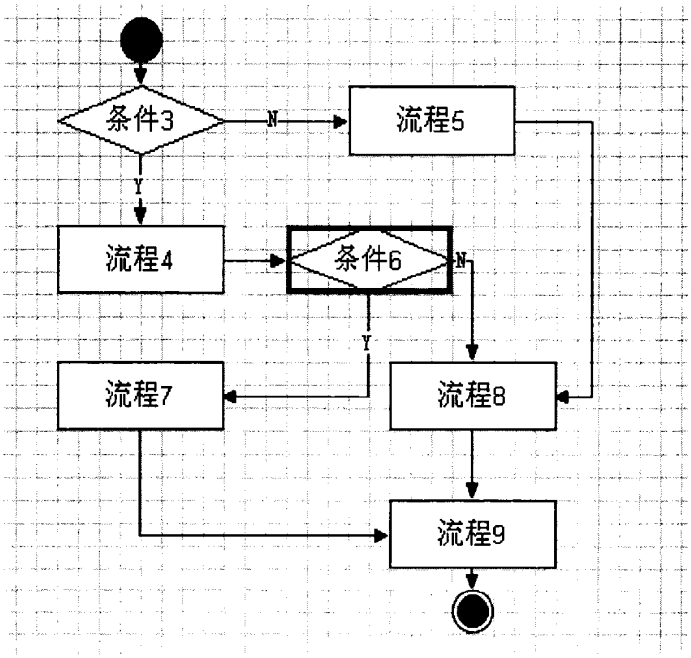


图 5.10 嵌套分支结构无合法汇点的非结构化用例

5.4 本章小结

本章是流程图结构分析与识别的实现和测试部分。

实现方面给出了基本数据结构的设计。针对流程图的结构识别，给出了递归算法框架的实现逻辑。另外，设计了适用于整个工具的错误处理机制。最后给出了整体算法的复杂度分析。

测试部分对实验所运行的测试环境进行了说明。实验采用了有效的测试用例对流程图代码自动生成工具的正确性、容错性和性能进行了一系列评估。实验表明工具实现了既定的功能，识别出了结构化流程图的结构组成，正确生成了相应代码。对于非结构化流程图，工具能够识别其中存在的非结构化元素并加亮给出警告信息。

第六章 结论与展望

6.1 本文工作总结

针对模型驱动开发中层次化建模技术的系统框架，本文主要研究了程序流程图模块的结构分析和识别技术。在充分了解结构化流程图、半结构化流程图以及非结构化流程图结构组成及主要特征的基础上，结合图论相关知识，本文提出了流程图结构分析与识别算法。算法创新点在于对半结构化元素的支持。算法主要分如下几个部分：

1、递归求取流程图中的强连通分量，确定每个循环结构的作用域及组合嵌套关系。

2、根据不同循环结构类型的特征及强连通分量上下文结构信息计算循环结构的条件判断结点，同时辨别流程图中所有菱形结点所属的循环类型或分支类型。

3、根据半结构化程序设计的特殊性，对流程图中带有 continue/break/return 含义的半结构化元素进行识别和标记。

4、从结构化、半结构化程序设计方面进行流程图合法性的检查，在分析和识别过程中定位流程图中存在的非结构化元素。

6.2 工作展望

由于时间与技术原因，本文中有关流程图结构分析与识别的技术研究工作中还存在许多可以改进的地方，具体而言，主要有以下几方面：

1、由于代码中的switch-case结构总能等价的转化为嵌套的if-else结构。因此本文在识别break/return这些半结构化元素时并未考虑switch结构中的情况。

2、非结构化流程图中存在着不受任何限制的带有GOTO含义的流程线，这些流程线破坏了结构化原则，使得流程图在结构的组合嵌套关系、结构的入口出口规则、结构的作用域规则等方面出现了异常情况。在定位这些非结构化元素时，算法的识别准确性还有待进一步提高。

3、循环结构的类型判定算法中，由于判定过程比较复杂，判定逻辑没有得到完全的证明，亟需以更大量、更复杂的测试用例进行实验，从而保证整个算法的完备性。

致谢

硕士研究生的学习即将结束，这段时间是我最受益和最难忘的。在此谨向所有关心、帮助和支持过我的老师、同学、朋友和亲人致以最衷心和最真挚的谢意！

首先衷心感谢我的导师刘西洋教授，本研究及论文是在他的亲切关怀和悉心指导下完成的。刘老师渊博的学识，精益求精的态度，丰富的科研经验让我在学业上跨上了一个新的台阶，是刘老师不断的鼓励让我树立了不怕困难，努力坚持的信心。在生活上，刘老师也给了我无微不至的关怀和帮助。在今后的工作中，我将谨记刘老师的谆谆教导，加倍努力，不辜负刘老师的殷切期望。

感谢王黎明师兄，杜文师兄，李春香师姐，年华师兄，在我研究生的科研和学习过程中他们给了我极大的帮助。感谢秦英师姐，穆浩英师姐，吴桂花师姐，感谢许琳，褚艳利，宋爱龙，周栋武，陈瑞滨，张帆，任中杰，邵帅，许胜之，王帼钦，周明媛，严琳以及 1311 实验室其他所有同学，是他们营造了一个充满学术气氛的环境，给了我很多的关心和帮助，感谢他们中的每一个人！

感谢我所有 606 和 607 的弟兄们，今后不管到哪里，60607 这个独特的数字始终会给我力量。多余的话语就不必了，真心希望弟兄们事业有成！

最后，我要将心底最诚恳的谢意致以我的家人。感谢我的妈妈、爸爸、舅舅姥爷等亲人，二十多年来他们对我的爱是无法用言语表达的。尤其是我亲爱的妈妈，她含辛茹苦的把我培养成人，付出一切使我从小得到了最好的教育并为我铺好了成长的道路，感谢她在生活上对我无微不至的关心和照顾，我一定会努力成才，不辜负她的心血和期望，我也会将这份爱传递下去。特别感谢许琳，感谢她在学业上一直以来给我的鼓励和支持，感谢她在生活上对我细心的关心和照顾，感谢她在我遇到困难挫折时陪在我身边开导我鼓励我，使我能够一直保持乐观和自信，我将时刻提醒自己，不忘记承诺，我将为我们共同的目标奋斗一生。

再一次对我的导师刘西洋教授和所有关心过、帮助过和支持过我的老师、同学、朋友、亲人致以崇高的感谢和敬意！

参考文献

[1] A.W.Brown, J.Conallen, D.Tropeano. "Models, Modeling, and Model Driven Development" in Model-Driven Software Development. S.Beydeda, M.Book, and V Cmlhn, Editors, Springer, Berlin. 2005:1-17.

[2] Thomas Stahl, Markus Völter. Model-Driven Software Development: Technology, Engineering, Management. 2006.

[3] David Harel, Statecharts in the Making: A Personal Account. Communications of the ACM Vol. 52 No. 3, 2009: 67-75.

[4] Ludovic Apvrille, Jean-Pierre, TURTLE: A real time UML profile supported by a Formal Validation Toolkit, IEEE TSE, 30:7, July, 2004.

[5] Shari Lawrence Pfleeger, Software Engineering Theory and Practice, 2001.

[6] Tia Watts. The SFC editor: a graphical tool for algorithm development. Journal of Computing Sciences in Colleges, 4(1):73–85, 2004.

[7] James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual(UML 参考手册(影印版)). 北京:科学出版社, 2004.

[8] Veikko Seppanen. Obstacles in Software Process Teclmology Evolution: Analysis of the Code Generation Case. Euromicro Conference, 2000. Proceedings of the 26th, 2000, 2(5-7):292-299.

[9] Michael W.Whalen, Mats P.E.Heimdahl. On the Requirements of High-Integrity Code Generation.High-Assurance Systems Engineering, 1999. Proceedings 4th IEEE International Symposium On. 1999, (17-19):217-224.

[10] http://www.dzone.com/links/r/executable_models_vs_codegeneration_vs_model_inte.html.

[11] Matt Stephens. Automated Code Generation. http://www.softwarereality.com/programming/code_generation.jsp, 2002.

[12] 朱三元编著. 程序设计方法学[M] . 上海: 上海交通大学出版社, 1983.

[13] Martin C.Carlisle, Terry A. Wilson, Jeffrey W.Humphries, Steven M. Hadfield, RAPTOR: Introducing Programming to Non-Majors with Flowcharts, Journal of Computing Sciences in Colleges, Volume 19 Issue 4, April 2004.

[14] Kanis Charntaweekhun and Somkiat Wangsiripitak. Visual Programming using Flowchart, Communications and Information Technologies, 2006. ISCIT '06.

[15] Geoffrey G. Roy, Joel Kelso, Craig Standing, "Towards a Visual Programming Environment for Software Development, "Proceedings on Software Engineering: Education & Practice, 1998, pp. 381-388.

[16] Klans-Dieter Schewe, Bernhard Thalheim, Conceptual Modelling of Web Information Systems, Data & Knowledge Engineering. 2005, 54(2): 147-188.

[17] H.Tai, K.Mitsui, T.Nerome, M.Abe, K.Ono, M.Hori. Model-Driven Development of Large-Scale Web Applications. IBM Journal of Research and Development. 2004, 48(5-6): 797-809.

[18] 尹彦均. Web 应用代码自动生成平台中代码生成系统的研究与实现. 北京工业大学硕士学位论文. 2007:7-16.

[19] Ed Benowitz, Ken Clark, Garth Watney Auto-coding UML Statecharts for Flight Software, 2006.

[20] David Harel and Michal Politi. Modeling Reactive Systems with Statecharts: The STATEMATE Approch, McGraw-Hill, 1998.

[21] Jay Nelson, Structured Programming Using Processes, proceedings of the 3rd ACM SIGPLAN Erlang Workshop, 2004.

[22] 陈世鸿. 面向对象软件设计方法的研究[J]. 武汉大学学报(自然科学版), 1995, 41(6): 315～320.

[23] 江明德编著. 面向对象程序设计[M]. 北京: 电子工业出版社, 1997.

[24] Douglass, Bruce Powell. Real-Time UML, Second Edition: Developing Efficient Objects for Embedded Systems. Addison Wesley, 1999.

[25] Douglass, Bruce Powell. Doing hard time, developing real-time systems with UML, objects, frameworks, and patterns. Addison Wesley, 1999.

[26] Douglass, Bruce Powell. UML Statecharts. Embedded Systems Programming, 1999, January: 22-42.

[27] A.W.Brown, S.1yengar, S.Johnston. A Rational Approach to Model-Driven Development. IBM Systems Journal. 2006, 45(3).

[28] 模型驱动开发方法的应用研究. 计算机工程, 32(13): 63-65.

[29] OMG. MDA. http://www.omg.org/mda/.

[30] Dijkstra, E.W., Structured Programming. Software Engineering Techniques, NATO Science Committee, pp. 84-87, August 1970.

作者在读期间的研究成果

参与科研：

1. C/C++程序信息流的动态分析，来源于国家“十一五”某预研项目.

2. 流程图的自动代码生成技术，来源于北京某航天自动控制研究所课题.

发表论文：

[1] 王黎明，陈科，许琳，张中宝，刘坚，陈平。软件维护中面向错误诊断的别名分析方法，西安电子科技大学学报.

[2] Xiyang Liu, Ailong Song, Zili Shao, Chunxiang Li, Ke Chen, Wei Wang. Cluster based architecture synthesis minimizing the resources under time constraint, IEEE ICASSP, 2010.