第44卷 第9期
2012年9月

哈 尔 滨 工 业 大 学 学 报
JOURNAL OF HARBIN INSTITUTE OF TECHNOLOGY

Vol. 44 No. 9
Sep. 2012

活动图并发语义代码自动生成算法设计

吴翔虎，曲明成，李建中，王志超

（哈尔滨工业大学 计算机科学与技术学院，150001 哈尔滨）

摘 要： 针对活动图能够比状态图更自然和直观地显示程序的并发行为，为达到图形化描述程序的并发行为并自动生成代码的目标，通过分析活动图的图元语义，以 fork、join、activity、initial、activity final、flow final 等6个图元作为图形建模和代码生成的基础，提出了一套代码自动生成算法. 该算法把活动图拆分成若干独立的活动子图；再把每个活动子图解析成若干进程和信号量；最后对每一个进程和信号量进行代码生成. 实验证明，基于本算法开发的原型系统取得了较满意的效果，同时也证明了所提出的方法和算法的正确性、有效性.

关键词： 代码自动生成；活动图；并发语义

中图分类号： TP311 **文献标志码：** A **文章编号：** 0367－6234(2012)09－0085－06

Design of automatic code generation algorithm based on concurrency semantics of activity diagrams

WU Xiang-hu, QU Ming-cheng, LI Jian-Zhong, WANG Zhi-chao

(School of Computer Science and Technology, Harbin Institute of Technology, 150001 Harbin, China)

Abstract: Compared with state diagram, activity diagram can be used to display the concurrent behavior of program in a more natural and intuitive way. Six primitives of initial, fork, join, flow final, activity final and activity were selected as the basis for graphical modeling and automatic code generation. A XML document format was defined to describe the activity diagram, then the XML document was parsed based on DOM, after that original activity diagram was split into separate activity sub-diagrams; and then each activity diagram was parsed into a number of processes and semaphores and codes. The methods and algorithms proposed were tested by designing and implementing a software system and good results were achieved, it showed that the methods and algorithms were right and effective.

Key words: automatic code generation; activity diagram; concurrency semantic

基于 UML 模型的代码自动生成[1-3]是一种以 UML 模型为起点，可以直接生成多层系统结构，并同时保留原有模型中层次关系的代码自动生成技术[4]. 例如基于状态的代码自动生成工具 I-Logix, Rhapsody 以及基于流程图的代码生成工具都属于该技术范畴[5-7].

现有研究中，在基于活动图或者状态图来生成代码的过程中，状态图被用来作为生成顶层的代码框架，而在刻画程序的执行逻辑时只是单纯的使用流程图. 因为状态图对并发的语言较难转化为代码，所以以状态图生成代码框架的研究成果对于并发的任务或者系统线程的支持并不理想[8]. UML 活动图可以有效的描述系统的执行流程、状态和并发活动，可以作为研究多线程并发的有力手段[9]. 为了能够在建模过程中对并发活动进行较好的支持，本文采用活动图的 fork、join、activity、initial、activity final、flow final 等 6 个图元来描述系统的并发行为. 通过将活动图解析成若干活动子图和同步信号量来实现生成程序代码的目标. 基于本文算法开发的原型系统取得了较满意

收稿日期： 2011－03－14.
基金项目： 国家高技术研究发展计划资助项目(2005AA742013).
作者简介： 吴翔虎(1968—)，男，教授；
李建中(1950—)，男，教授，博士生导师.
通信作者： 曲明成，qumingcheng@126.com.

的效果,同时也证明了所提出的方法和算法的正确性、有效性.

1 设计目标

本文通过对 UML 活动图进行分析,设计实现了一个以 UML 活动图为基础,进行代码自动生成并验证的系统. 系统选取了活动途中的 6 个图元作为基本图元,分别为 initial、flow final、activity final、activity、fork、join.

系统的整体流程如图 1 所示.

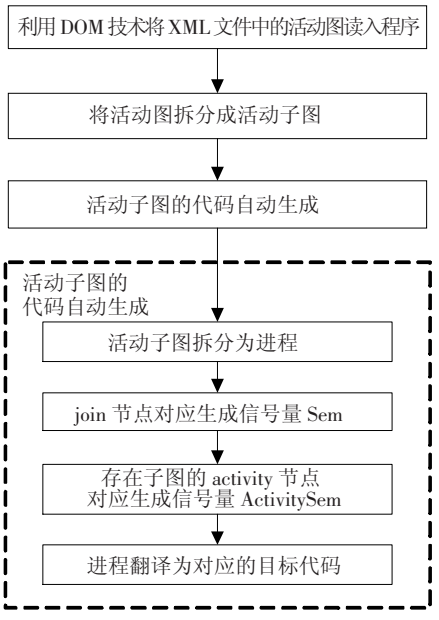

图 1 验证系统处理流程

本系统中活动图用 XML 可扩展标记语言来描述,活动图可以通过 DOM 技术解析 XML 文件从而以树状结构载入系统. 再进行根遍历 XML 树,将活动图可以划分为若干个独立的子活动图. 之后利用相应的算法将每一个子活动图转化为若干进程,同时添加对这些进程进行并发控制的信号量,并将生成的进程和信号量翻译成相应代码. 本文选择 java 语言作为目标代码,由于算法的通用性,不难获得在其他平台上运行的其他语言形式的目标代码.

2 活动图的 XML 描述

UML 活动图可以通过 XML 可扩展标记语言来进行准确的描述,并且可以利用 DOM 技术把 XML 文件解析成一个树状的数据结构,解析得到树状结构通常形式为:以 root 节点作为根节点,根节点的下一层为所以活动图的节点和边,其中所有边都在节点之后. 另外用递归的形势表示 activity 节点的子图(若包含).

对于图 2 中的示例活动图,本文根据上述所制定的规则,绘制出图 3 所示的对应的 XML 树状结构.

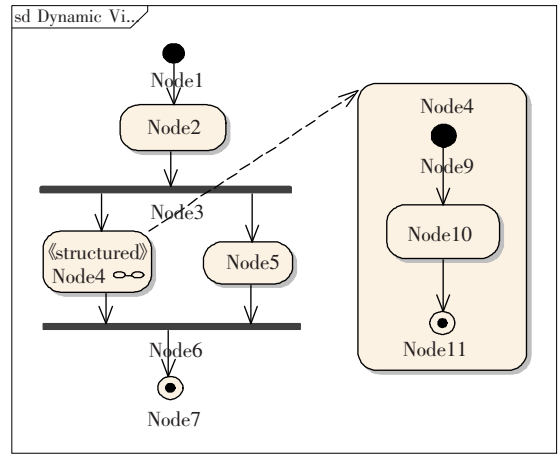

图 2 一个活动图实例

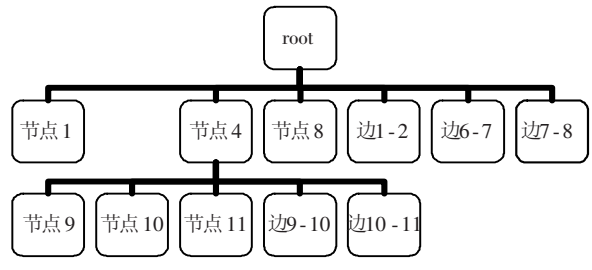

图 3 示例活动图所对应的树状结构

活动图的节点和边相应的数据结构定义如下:

节点:struct node {int id; String kind;}

边:struct edge {int source; int target;}

节点的数据结构 node 中的属性 id 记录原活动图中节点的图元序号,属性 kind 记录节点类型;边的数据结构 edge 中属性 source 记录边的源节点序号,属性 target 记录边的目标节点序号.

图 2 所示活动图的 XML 文件为:

```
<root>
<node id="1" type="initial"></node>
<node id="2" type="activity"></node>
<node id="3" type="fork"></node>
<node id="4" type="activity">
    <node id="9" type="initial"></node>
    <node id="10" type="activity"></node>
    <node id="11" type="final"></node>
    <edge from="9" to="10"></edge>
    <edge from="10" to="11"></edge>
</node>
<node id="5" type="activity"></node>
<node id="6" type="join"></node>
<node id="7" type="activity"></node>
<node id="8" type="final"></node>
<edge from="1" to="2"></edge>
<edge from="2" to="3"></edge>
<edge from="3" to="4"></edge>
<edge from="3" to="5"></edge>
<edge from="4" to="6"></edge>
<edge from="5" to="6"></edge>
<edge from="6" to="7"></edge>
<edge from="7" to="8"></edge>
</root>
```

3　活动图拆分成活动子图

利用树的中序遍历算法遍历一遍整个活动图，获得遍历的结果为一系列独立的活动子图和带有嵌套子树的 activity 节点. 在这里，本文只需要解决活动子图的本层树状结构而不需要考虑它的 activity 节点是否还嵌套了子树.

上述的算法，将带有嵌套问题的活动图问题转化为处理若干独立活动子图的过程.

中序遍历一遍活动图，解决活动图中的嵌套问题，生成若干不带嵌套的独立活动子图的算法的伪代码如下：

```
输入:root 节点(活动图的根节点)
输出:生成独立的活动子图根节点数组 Root[ ]
{
    建立和初始化 taskQueue 作为任务队列;
    建立和初始化 Root[ ],用来存储生成结果的所有活动子图的根节点;
    将活动图的根节点 root 放入 taskQueue;
    root 存入 Root[ ];
    while(! taskQueue. empty( ))
    {
      tmpRoot = taskQueue. front( );
      对以 tmpRoot 为根的树进行一遍中序遍历
      {
        if(该节点是带有嵌套子树的 activity 节点)
        {
          taskQueue. push(该节点)
          该节点存入 Root[ ];
        }
      }
      taskQueue. pop( );
  }
  return Root[ ];
}
```

通过上述算法，将图 2 所示的活动图拆分成图 4 所示的分别以原图根节点 root 和节点 node 作为根节点的两个活动子图.

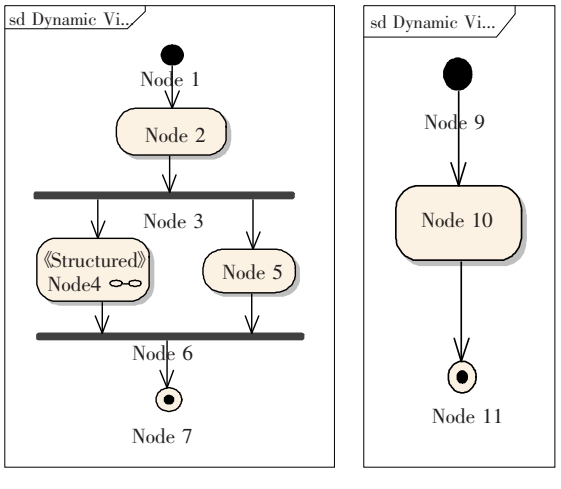

图 4　图 2 活动图拆分为活动子图

4　活动子图的代码自动生成

为了要实现通过活动子图的代码自动生成，首先按照相应规则将独立的活动子图拆分为若干的进程. 本文中进程的拆分规则定义为：开始节点一定是 fork、join、initial 节点，结束节点一定是 fork、join、final 节点.

在进行拆分后，本文通过运用建立两种信号量 Sem 和 Activity 来解决并发进程的时序控制问题.

信号量 Sem，设置在 join 节点上. 初始值设为在进入 join 节点的进程数目，每当有进程进入 join 节点时进行减一操作，只有当 Sem = 0 时才会创建后续的进程. 信号量 Sem 保证了所有的并发进程都执行完后才会继续执行.

信号量 ActivitySem，设置存在嵌套子图的 activity 图元上. 初始值设为 1，在 activity 图元的子图进程结束时，将其置为 0. 同时在外层 activity 图元之后监测 ActivitySem 的值，为 0 时继续往下执行. 信号量 ActivitySem 保证了嵌套在 activity 中的子图能够在 activity 后续任务之前执行，保证语义的正确性.

最后，翻译每个拆分出来的进程，获得其对应的目标代码.

4.1　活动子图拆分为进程

为了避免活动子图拆分成若干进程后出现进程名称冲突的情况，规定下面的进程命名规则为：

拆分后得到的进程 Thread 名字为 Thread$i_1_i_2_\cdots_i_k$，其中 $i_1\_i_2\_\cdots\_i_k$ 为该进程在原活动图中先后经历的节点.

按照上述的进程命名规则，可以避免进程名字重复，做到根据节点唯一缺点进程名.

进程类 Thread 中通过用链表 idList 存储进程在活动图中先后经历的节点序列来实现存储进程信息.

```
class Thread
{
  public LinkedList < id > idList;
}
```

此外，为了判断节点和边是否已经被纳入到进程中，建立 Node 类和 Edge 类，通过 tag 属性来标识是否已经被纳入某进程.

```
class Node {int id; String type; Boolean tag;}
class Edge {int from; int to; Boolean tag;}
```

实现将活动子图拆分成为若干进程的算法 GetThread( )的伪代码如下所示：

```
输入:root 节点,活动子图的根节点 t
输出:进程数组 ArrayList < Thread > ,拆分的结果
GetThread( )
{
    建立并初始化 ArrayList < Node > 和 ArrayList < Edge > ;
    遍历该以 root 为根节点的树
      Node temp = 当前遍历到的树的节点;
      temp. tag = false;
      if( temp 是活动子图的节点 )
        ArrayList < Node > . add( temp) ;
      if( temp 是活动子图的边)
        ArrayList < Edge > . add( temp) ;
    遍历该以 root 为根节点的树
    {
      Node temp = 当前遍历到的节点;
      if ( temp. tag = = false)
      {
        idList = new ArrayList < int > ;
        idList. add( temp. id) ;
        while( temp 不是 join、fork 或 initial)
        {
            int nextid = 0;
            boolean flag = false;
          for each edge in ArrayList < Edge >
          {
              if( temp. id = = edge. to)
              {
                nextid = edge. from;
                flag = true;
                 break;
              }
          }
          If( flag) idList. add( nextid) ;
    }
    while( temp 不是 join、fork 或 final)
    {
        int nextid = 0;
        boolean flag = false;
      for each edge in < Edge >
      {
          if( temp. id = = edge. from)
          {
              nextid = edge. to;
              flag = true;
              break;
          }
      }
      If( flag) idList. add( nextid) ;
    }
    将 idList 中的所有节点对应在 ArrayList < Node > 中的元素
的 tag 值置 true;
  }
 }
}
```

经过上面的 GetThread( )算法,将图 2 示例的活动图中以 root 为根节点的活动子图拆分成图 5 中所示的 Thread1_2_3 和 Thread3_5_6、Thread3_4_6 以及 Thread6_7_8 这 4 个进程.

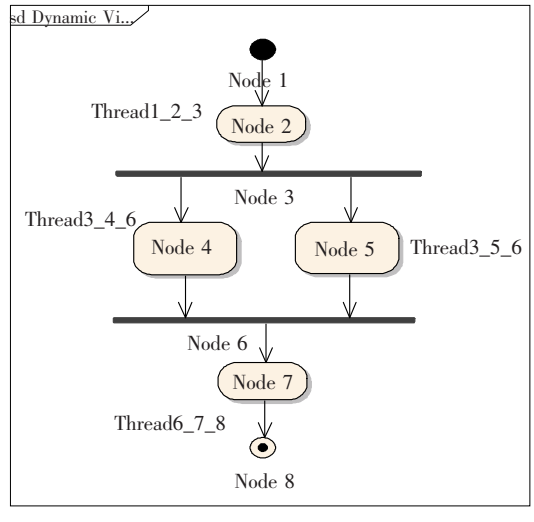

图 5 活动子图拆分为进程

4.2 join 节点对应生成信号量 Sem

按照上面所设计的规则,对每一个 join 节点生成一个信号量 Sem,命名为 Sem + 节点编号,同时计算进入 join 节点的进程数目,并赋作信号量的初始值.

为每个 join 节点生成 Sem 信号量的算法 SemCompiler( )的伪代码如下:

```
输入:root 节点,活动子图的根节点;join 节点;进程数组 Thread;
输出:生成目标文件,Sem + join 节点 id
SemCompiler( )
{
    int count = 0;
    For each thread in ArrayList < Thread >
    {
    if ( thread 最后一个节点 Node. type = = join) count + + ;
    initial Sem = count;
    //获取 join 节点的后续进程
    Thread nextThread;
    for each thread in ArrayList < Thread >
    {
      if ( thread 第 1 个节点 Node. type = = join 节点)
      {
        nextThread = thread;
        break;
      }
    }
    创建文件 sem + id. java;
    按照规定格式写入初始值和后续进程等内容。
}
```

对图 2 示例的活动图中运行 SemCompiler( )

算法后，对 join 节点 6 生成命名为 Sem6 的信号量，该信号量对应的 Sem6. java 文件如下：

```
public class Sem6
{
    private static int semaphore = 2;
    public static synchronized void release( )
    {
        --semaphore;
        if (semaphore = = 0)
        {
            Thread6_7_8 thread6_7_8 = new Thread6_7_8( );
            thread6_7_8. start( );
        }
    }
}
```

4.3　存在子图的 activity 节点对应生成信号量 ActivitySem

相应的，本文同样按照上述设计的规则，对每一个嵌套子图的 activity 节点也生成一个信号量，同样的命名为 Activity + 节点编号，初始值设定为 1.

图 2 中的示例活动图，activity 节点 4 带有嵌套的子图，实现对它进行并发控制的信号量命名为 ActivitySem4，生成的 ActivitySem4. java 文件如下：

```
public class ActivitySem4( )
{
    private static int semaphore = 1;
    public static synchronized void release( )
    {
        --semaphore;
    }
    public static synchronized int test( )
    {
        return semaphore;
    }
}
```

4.4　进程翻译为对应的目标代码

将进程翻译成对应的目标代码的步骤为：

1）利用进程的节点序列 idList 来获得进程的名字，根据进程的名字创建相应文件.

2）依次分析节点序列中的每个节点的内容. 按照节点的不同类型，翻译成该节点对应到目标文件中的代码. 注意进程的第 1 个节点（initial、fork、join 中的某个）不进行翻译，所以从节点序列中的第 2 个节点进行分析翻译.

3）为了使生成结果能够符合 Java 语言的结构特征，将对每个 activity 节点都进行 2 次分析翻译的过程. 其中第 1 次分析翻译过程只生成 activity + id( )，第 2 次才对函数体内的具体内容进行分析翻译.

对进程转化生成为 java 代码的算法 ThreadCompiler( ) 的伪代码如下：

```
输入：进程的节点序列链表 idList
输出：创建生成相应目标文件
ThreadCompiler( )
{
    根据命名规则创建文件。
    List < Node > trlist = idList - > next;
    While(trlist ! = NULL)
    {
        Node tempnode = trlist - > node;
        switch (tempnode. type)
        {
            case: fork
                写入并发进程的启动代码;
                break;
            case: join
                对应 Sem--;
                break;
            case: activity
                idList. add(tempnode);
                if(有嵌套子图) 循环监测 ActivitySem;
                break;
            case: final
                写入 return;
            break;
        }
        trlist = trlist-next;
    }
    trlist = idList - > next;
    While(trlist ! = NULL)
    {
        Node tempnode = trlist - > node;
        if (tempnode. type = = activity)
        {
            写入 activity 中的具体内容;
            if(有嵌套子图)
            {
                写入子图中以 initial 为起始的进程的启动代码;
            }
        }
        trlist = trlist-next;
    }
}
```

5　验　证

本文测试的环境为：PC 实验平台，32 bitWindowsXP 操作系统，2 GB 内存，2.80 GHz 双核 CPU，并采用 Eclipse3. 4. 1 的开发平台.

选择 Eclipse 作为开发平台，因为 Eclipse 是一个开发源代码，基于 JAVA 的开发平台. Eclipse 平台不仅包括了 IDE，即 JAVA 的集成开发环境，还提供了相关的调试工具，便于开发和调试 Java 程序.

运用本文的算法，图 5 所示的拆分出的 4 个

进程翻译生成的目标代码如下：

```
//Thread1_2_3.java
public class Thread1_2_3 extends Thread
{
    public void run()
    {
        activity2();
        Thread3_4_6 thread3_4_6 = new Thread3_4_6();
        thread3_4_6.start();
        Thread3_5_6 thread3_5_6 = new Thread3_5_6();
        thread3_5_6.start();
    }
    public void activity2()
    {
        System.out.println("activity2 run");
    }
}
//Thread3_4_6.java
public class Thread3_4_6 extends Thread
{
    public void run()
    {
      activity4();
while(ActivitySem4.test()! = 0)
            {
            }
            Sem6.release();
    }
    public void activity4()
    {
      System.out.println("activity4 run");
      Thread9_10_11 thread9_10_11 = new Thread9_10_11();
      thread9_10_11.start();
    }
}
//Thread3_5_6.java
public class Thread3_5_6 extends Thread
{
    public void run()
    {
      activity5();
      Sem6.release();
    }
    public void activity5()
    {
      System.out.println("activity5 run");
    }
}
//Thread6_7_8.java
public class Thread6_7_8 extends Thread
{
    public void run()
    {
      activity7();
      return;
    }
    public void activity7()
    {
      System.out.println("activity7 run");
    }
}
```

另外目标代码也包括 Sem6.java 和 ActivitySem4.java 两个文件就构成了完整可以运行的并发程序.

6 结 论

1)本文通过采用 UML 活动图的 6 个基础语义来对程序的并发行为进行建模,并采用特定的算法将模型转化为目标代码.

2)通过原型系统的开发,充分证明了采用 6 个基础图元来描述程序并发行为的基本能力,以及由此生成程序代码的可行性.

3)弥补了状态图对程序并发行为描述能力的不足,为基于活动图、状态图、流程来构建并发系统框架和行为逻辑提供了有益的参考.

参考文献：

[1] 张天, 张岩, 于笑丰. 基于 MDA 的设计模式建模与模型转换[J]. 软件学报, 2008,19(9): 2203 - 2217.

[2] 吕瑞峰, 王刚, 闫晓先, 等. 基于模型驱动框架的计算无关层过程建模[J]. 计算机集成制造系统, 2008,14(5): 1 - 8.

[3] LIANG Yizhi, WANG Yanzhang, LIU Yunfei. The formal semantics of an UML activity diagram[J]. Journal of Shanghai University (English Edition), 2004,8(3): 322 - 327.

[4] MEDVIDOVIC N, ROSENBLUM D S, ROBBINS J E, *et al*. Modeling software architectures in the unified language[J]. ACM Transactions on Software Engineering and Methodology, 2002, 11(1): 2 - 57.

[5] DAKHORE H, MAHAJAN A. Generation of C-code using XML parser[OL]. http://www.rimtengg.com/iscet/proceedings/pdfs/advcomp/149.pdf.

[6] CARLISLE M C, WILSON T A, HUMPHRIES J W, *et al*. Raptor: introducing progra mming to non-majors with flowcharts[J]. Journal of Computing Sciences in Colleges, 2004, 19(4): 52 - 60.

[7] KANIS C, SOMKIAT W. Visual programming using flowchart[C]//ISCIT 06. International Symposium on Communications and Information Technologies. Washington, DC: IEEE Xplore, 2006: 1062 - 1065.

[8] SAMEK M. Quantum programming for embedded systems:toward a hassle-free multithreading[J]. C/C++ Users Journal, 2003, 3(1): 1 - 10.

[9] 柳翔. 嵌入式与实时系统开发[M]. 北京: 机械工业出版社, 2006:207 - 208.

(编辑 张 红)