软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于分支标记的数据流模型的代码生成方法*

苏 卓[1,2], 王东艳[3], 杨镒箫[1,2], 张明睿[1,2], 姜 宇[1,2], 孙家广[1,2]

[1](清华大学 软件学院,北京 北京 100084)
[2](教育部信息系统安全重点实验室(清华大学),北京 北京 100084)
[3](北京大学 信息科学技术学院,北京 北京 100871)
通讯作者: 姜宇, E-mail: jiangyu198964@126.com

摘 要: 模型驱动开发以其低错误率,易仿真,易验证的特点在嵌入式软件开发中被广泛应用.近年来,基于模型的嵌入式软件开发方法及相应工具也在逐渐发展和完善.数据流模型是各种建模工具中使用最为频繁的语义模型,然而各种工具对于数据流模型的代码生成能力却参差不齐,特别是对于数据分支组件的支持,当前主流的建模工具都采用各种方式来回避复杂的分支建模及对应的代码生成.但是,分支建模是非常重要的,使用分支组件可以更清晰地表现出数据流的数据传递逻辑. 为了解决复杂分支建模带来的代码生成难题,本文针对具有复杂分支组合的数据流模型提出了一种基于分支调度标记的代码生成方法.在本文提出的算法中,首先通过拓扑排序确定模型的调度顺序,再根据不同分支的影响对组件进行分支标记,之后根据组件的分支标记构造一个基于控制流的代码生成位置表,最后即可根据代码生成位置表进行各种主流语言的代码生成.本文通过构造四个具有复杂分支的数据流模型实例进行代码生成,并在生成代码行数和运行时间等方面与 Simulink 和 Ptolemy 进行对比,进一步说明了我们的代码生成方法在复杂分支组合情况下的通用性以及本文工作的价值和意义.
关键词: 模型驱动开发;数据流;代码生成;分支;嵌入式系统
中图法分类号: TP311

中文引用格式: 苏卓,王东艳,杨镒箫,张明睿,姜宇,孙家广. 基于分支标记的数据流模型的代码生成方法.软件学报. http://www.jos.org.cn/1000-9825/0000.htm
英文引用格式: Su Z, Wang DY, Yang YX, Zhang MR, Jiang Y, Sun JG. Code generation method of data flow model based on branch marking. Ruan Jian Xue Bao/Journal of Software, 2020 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

Code generation method of data flow model based on branch marking

SU Zhuo[1,2], WANG Dong-Yan[3], YANG Yi-Xiao[1,2], ZHANG Ming-Rui[1,2], JIANG Yu[1,2], SUN Jia-Guang[1,2]

[1](School of Software, Tsinghua University, Beijing 100084, China)
[2](Key Laboratory for Information System Security, Ministry of Education, Beijing 100084, China)
[3](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

Abstract: Model-driven development is widely used in embedded software development because it has a low error rate while it is easy to simulate and verify. In recent years, model-based embedded software development methods and their corresponding tools are also gradually developing and improving. Data flow model is the most frequently used semantic model among all kinds of modeling

* 基金项目: 国家自然科学基金(62022046, U1911401, 61802223); 科技部重点研发计划(2019YFB1706200); 华为清华可信研究项目(20192000794)
Foundation item: National Natural Science Foundation of China (62022046, U1911401, 61802223); National Key Research and Development Project (2019YFB1706200); Huawei-Tsinghua Trustworthy Research Project (20192000794)
收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

tools. Nonetheless, the code generation ability of various tools for data flow model is uneven, especially for the branching actors. It is well known that current mainstream modeling tools adopt various ways to avoid complex branch modeling and the generation of its corresponding code. However, branch modeling is very important, and it makes the data transfer logic of the data flow more clearly by using branch actors. In order to solve the problem of code generation caused by complex branch modeling, this paper proposed a code generation method based on branch schedule marking for data flow model aimed at complex branch combinations. As for the algorithm proposed in this paper, firstly, the scheduling order of the model was determined by topological sorting. Secondly, a code generation location table based on control flow was constructed according to the branch marks which marked by the influence of different branches. Finally, code generation of various mainstream languages could be carried out in terms of the code generation location table. By constructing four instances of data flow models with complex branches for code generation and comparing them with Simulink and Ptolemy in terms of lines of code generation and elapsed time, this paper further illustrates the universality of our code generation methods in complex branch combinations and the value and significance of this work.
Key words: model-driven development; data flow; code generation; branch; embedded system

模型驱动开发在嵌入式系统领域中被广泛使用，它具有低代码、不易出错、容易仿真和验证的特点[1-5]。其中数据流模型是使用最为广泛的计算模型。在数据流模型中，组件按照先后顺序执行，读取组件输入端口的数据，并将计算得到的结果通过输出端口和相应的连线向后续组件传递。例如加减法组件，首先在组件的输入端口获得加数和减数等输入数据，将加数相加、减去减数，最后在组件的输出端口输出结果[6]。数据流模型中更高级的功能还包括复合组件和状态机等，复合组件是将一些基础组件进行封装，作为一个更大的计算单元使用，一般地，复合组件是可以嵌套的，而状态机的应用使得状态转换相关的控制逻辑的建模更加方便。数据流中的这些组件通常被组合起来使用，从而描述整个模型系统。

数据流模型及相关的支持工具在工业界和学术界越来越得到人们的关注，例如 Simulink、Ptolemy-II 等[7-14]。但是，这些工具强大的建模能力却给模型验证和代码生成带来了非常大的挑战，所以很多建模工具要么对模型进行约束限制，要么不支持某种功能的代码生成，比如 Ptolemy-II 就不支持数据流模型中分支组件的代码生成。代码生成器有限的代码生成能力会很大程度上限制工具的使用场景，因为在工业届很多开发人员不仅仅希望工具能够进行建模和仿真，还希望能够支持代码生成以减少他们的编码工作[15-17]。

我们分别调研了工业界和学术界使用较多的两种建模工具：Simulink 和 Ptolemy-II。经调查，我们发现它们都弱化了对于数据分支组件的支持。在 Simulink 中，用户可以通过 If 子系统或者 Switch Case 子系统来实现分支表达，这种表达方式会将不同分支中的组件分别包装在不同的子系统组件中，这里的子系统也属于一种复合组件。这样一来，一方面不便于观察和修改模型的逻辑和结构；另一方面当我们需要嵌套分支的时候，就需要用嵌套多层的子系统去实现，这对于建模来说是相当繁琐的，根据我们的了解，企业中的开发人员大多选择直接在模型中嵌入代码去实现分支逻辑的计算。在 Ptolemy-II 中，建模和仿真是支持分支组件的，但正如前面提到的，它不支持数据流模型中分支组件的代码生成，另外，对于 Ptolemy-II 中代码生成能力较强的针对离散事件模型的代码生成器，也直接按照事件传递机制去生成代码，也就是说，生成的代码是一整套离散事件模拟器[18]。然而，通过这种方式生成的代码几乎无法在实际生产中使用，因为它太过庞大，并且运行效率非常低。

综上所述，不能很好地支持分支组件及其代码生成是影响这两种建模工具使用的比较重要的因素之一。可想而知，分支组件对于数据流模型的建模来说至关重要，它能很好地将模型中的分支逻辑表达出来，更重要的是分支组件的代码生成可以支撑起整个建模工具的完备性，这不但要求代码生成器能处理简单的分支表达，还要求它能够处理任意复杂的分支组件的组合，因为任何分支路径合的表达在建模层面都是有意义的。

对分支组件的控制流分析是非常复杂的，对带有分支组件的数据流模型进行控制流转换主要会面临三个挑战：(1)分支组件所带来的两个或多个控制流分支什么时候可以合并。当然，面对这个问题，最坏的方法是将分支之后的所有组件在每个分支中都生成代码，这种方法显然会生成大量的冗余代码，特别是当存

在分支嵌套的时候，生成的代码量将会呈指数增长。所以，为每个分支确定一个合适的分支合并位置是关键。(2)分支组件嵌套该如何进行有效的控制流分析。每个组件都会受到其前面的全部分支组件的分支的影响，那么当该分支后面存在分支(甚至在该分支后面存在更多分支)时，如何针对多层的分支信息进行分支化简及合并是一个复杂的问题。(3)分支组件存在分支交叉时该如何生成控制流代码。在模型表达层面，两个分支组件各自的一个分支都汇聚到了同一个组件上，这种表达意味着汇聚到同一个组件上的两个分支，只要至少一个分支可以触发，该组件就可以执行，所以那该件的数据源可能会有三种情况(数据分别来自两个分支和数据共同来自两个分支)。多个分支组件之间的分支交叉可能会带来非常多的分支执行路径的可能，这也是自由使用分支组件所带来的代码生成的最大的难题。

本文针对数据流分支的代码生成问题进行了深入的研究，提出了一种针对数据流模型中任意复杂的分支组件组合的代码生成算法，该算法主体是按照数据流的拓扑排序从前向后依次进行的，首先根据分支组件的分支信息为其后继组件添加控制流标签，由于某些组件会处于不同的分支下，所以之后要将组件所有的分支情况进行化简合并。所有组件的分支信息都计算完之后，构造一个组件代码生成位置表，通过各个组件的分支信息，向这个列表中插入分支和组件数据源信息。由于存在分支，有的组件可能会被插入到多个位置。最终得到的列表就是完整的基于控制流的组件执行顺序，按这个生成位置表对各个组件分别生成代码即可得到完整的模型的代码。

1 研究思路

针对复杂的数据流模型，首先要对模型进行调度分析，根据模型中的分支组件为所有组件计算分支标记，在分支标记过程中对组件的分支信息进行化简处理。其次，根据每个组件的简化分支信息构造组件代码生成位置表。最后，根据组件代码生成位置表就可以生成完整的复合模型调度顺序的代码。针对数据流模型中分支组件的代码生成方法的主体框架如图 1 所示：

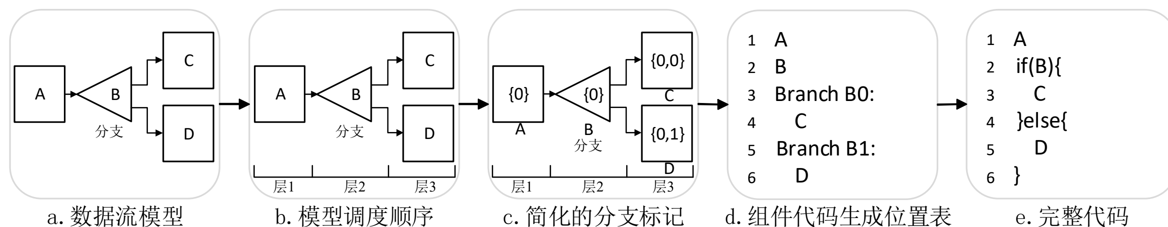


Fig.1 Framework of code generation method for branch actors in data flow model

图 1 针对数据流模型中分支组件的代码生成方法框架

第一步，对数据流模型进行调度分析。数据流模型的调度顺序是按照拓扑排序确定的，首先找到模型中没有数据输入的组件，将它们设置为第一层，之后把这些组件在模型中去除，再次寻找没有数据输入的组件，将这些新找到的组件设置为第二层，以此类推。对于模型中的数据延时组件，我们将其视为首尾断开的两个组件，用这样的方式来打破模型的环路。

第二步，根据调度顺序，逐层确定分支标记。从模型的第一层开始，对组件添加分支标记，这些分支标记是会根据组件的连接关系向后传递的。默认模型最开始就有一个分支，我们将处于第一个分支上的组件标记为{0}。之后如果遇到分支组件，就在分支组件后面进行叠加标记，同样的，分支组件的第一个分支用{0}标记，第二个分支用{1}标记，如图 1(c)所示。由于分支信息是从前向后叠加的，所以在对模型中任何一个组件进行分支标记前，首先对该组件所有的前驱组件的分支进行合并化简处理，这样就可以尽早地完成分支的合并，以避免代码生成的“分支爆炸”的现象。这里的“分支爆炸”指的是由于分支组件的组合使用（包括多个分支组件的不同分支的交汇以及分支组件的嵌套）会造成后续组件可能的执行情况数会成指

数增长。如果每个组件都继承前驱组件带来的所有分支情况，并且按组件所有的分支情况生成的带有各种分支条件的代码将会非常冗长且低效。具体的分支标记和化简方法会在第 2 节中详细描述。

第三步，根据分支标记生成基于控制流的组件代码生成位置表。建立一个空的组件代码生成位置表，将最开始的被标记{0}分支的组件，按调度顺序依次插入到生成位置表中，遇到分支组件则在生成位置表中插入分支数量个 Branch 标记，后续组件若满足这个 Branch 标记的分支状态就插入到这个 Branch 标记后面，如图 1(d)所示。在遇到分支合并的组件时，这些 Branch 标记就会被删除，后面再有新的分支则会创建新的 Branch 标记。更加详细的代码生成位置表的构造方法会在第 3 节中说明。

最后，直接根据基于控制流的组件代码生成位置表生成目标编程语言的代码，这里的生成位置表相当于伪代码的作用，可以直接翻译为各种主流的编程语言。

2 模型调度顺序的计算

模型调度顺序的计算是生成代码的第一步。我们将模型视为一个有向无环图，通过对该图进行拓扑排序确定各个组件的执行顺序。这里需要注意的是对于带有环的数据流模型，我们要将环路中寄存器类型的组件拆分为存和取两部分，通过这种方式打破环路。用于代码生成的数据流模型，如果存在数据环路，那么其环路中必定有延时、队列、数据池等寄存组件，否则一旦执行到环路，模型将进入死循环。接下来要对模型进行拓扑排序，找到模型中没有任何输入数据的组件，这些组件作为排序的第一层。然后消除第一层的组件和与它们连接的所有数据流连线，继续寻找没有任何输入数据的组件，把新找到的这些组件作为排序的第二层。后面的各层以此类推。由于已经事先将数据环打破，所以该排序可以在有限步骤内完成。我们仅对模型进行分层，而没有必要对层内的组件执行顺序进行计算，因为同层的组件的执行不会相互影响。

为了更直观更清晰地介绍模型调度顺序的计算和后续算法，下面采用一个带有分支嵌套的模型进行介绍，模型如图 2 所示。图 2 中，用矩形表示一般组件，对于一般组件我们可以将其简化地理解为根据给定的输入数据计算得到输出数据，输出数据再根据组件间的连线向后传递；用三角形表示分支组件，为了描述起来更加简单，这里仅采用二分支的组件进行演示，如图中的 B、F、G 这三个组件就是二分支组件，这些组件的左边有一个输入端口，右边有两个输出端口，这些组件会根据特定条件将输入端口的数据选择性地输出到第一个输出端口或第二个输出端口的其中一个，而另一个输出端口则不会向后输出数据，也不会触发后面的组件执行。整个模型的调度顺序是从左向右进行的，按拓扑排序进行分层会将整个模型分为层 1 到层 8 共 8 层。这个模型包含了除分支交叉(不同的分支组件之间的分支发生合并)的多种复杂分支结构，例如，在图 2 中，有 B 和 F 构成的分支嵌套的情况、B 和 E 构成的分支提前结束的情况、H 和 I 和 M 构成的不同数据源的分支合并的情况、M 和 J 和 K 和 N 构成的跨多个分支的分支交叉和合并的情况。

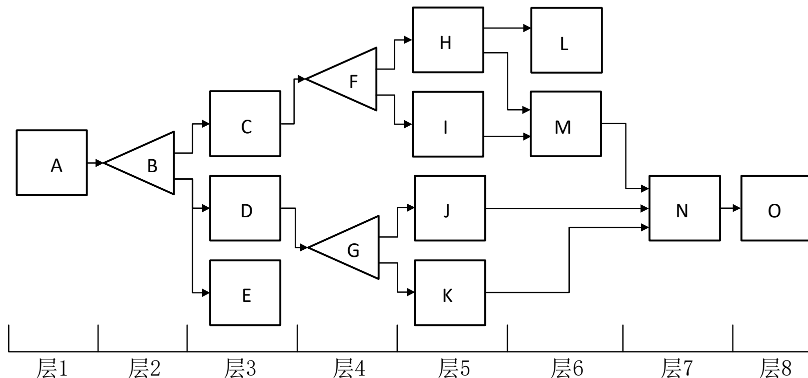


Fig.2 Schematic diagram of a model with complex branches structure

图 2 具有复杂分支结构的模型示意图

3 分支标记的计算与简化

在介绍分支标记算法前，先明确几个数据表示的概念：分支(*Branch*)，表示分支组件的单个分支，它由分支组件和分支号(该分支组件的第几个分支)这两个数据唯一表示，记为 Id_{Actor}，*Id* 表示分支号，*Actor* 表示分支组件；分支路径(*BranchPath*)，表示由多层分支组件带来的不同的分支情况，记为{$Branch_1$, $Branch_2$, $Branch_3$,…}；分支路径表，也称为一个组件的分支标记，表示某个组件所有可能的分支路径及分支路径上的数据源的集合，记为{$BranchPath_1$:$Actor_1$, $BranchPath_2$:$Actor_2$, $BranchPath_3$:$Actor_3$,…}。

对于一个已经确定了各个调度层的模型，首先要对模型中的组件进行分支标记。分支标记方法如算法 1 所示。算法 1 的整体逻辑是从前向后逐层遍历整个模型，对每层中的每个组件单独计算分支信息。算法 1 的第 3 行，获取当前组件的所有前驱组件。若当前组件没有任何前驱组件，则直接标记当前组件的分支为带有空数据的分支路径，如算法 1 第 4-7 行所示；若存在前驱组件，则构造一个分支路径表，分支路径表是一个 map 结构，key 存储分支路径，value 存储由 key 分支路径所带来的数据源组件。接下来，第 9-23 行遍历当前组件的所有前驱组件，获取它们的分支路径信息。其中，若前驱组件不是分支组件，则直接继承该前驱组件的分支路径信息，并把该前驱组件作为当前组件的当前分支路径的数据源，如第 18-22 行所示。若前驱组件是分支组件，则要在该前驱组件的分支路径的基础上连接该前驱组件的分支信息作为新的分支路径，如第 11-17 行所示。在获得了当前组件的所有分支路径之后，对这个组件的所有继承来的分支路径做简化处理，再赋值给当前组件的分支信息，如 24-25 行所示。

算法 1. 基于数据流模型的分支标记算法.

输入: 确定调度顺序的数据流模型 *ModelLayers*.

输出: 模型中所有组件的分支标记 *BranchInfos*

```
1:  for each layer in ModelLayers
2:    for each actor in layer
3:      predecessors = getPredecessors(actor)
4:      if predecessors == ∅ then
5:        BranchInfos[actor] = getBranchPathWithEmptyDataSrc()
6:        continue
7:      end if
8:      branchPathsWithDataSrc = ∅
9:      for each pred in predecessors
10:       branchPathsOfPred = BranchInfos[pred]
11:       if isBranchActor(pred) then
12:         for each branchPath in branchPathsOfPred
13:           for each branch in getActorBranches(pred)
14:             newbranchPath = connectBranchToBranchPath(branchPath, branch)
15:             addDataSrcToBranchPaths(branchPathsWithDataSrc[newbranchPath], pred)
16:           end for
17:         end for
18:       else
19:         for each branchPath in branchPathsOfPred
20:           addDataSrcToBranchPaths(branchPathsWithDataSrc[branchPath], pred)
21:         end for
22:       end if
```

```
23:         end for
24:         branchPathsWithDataSrc = simplifyBranchPath(branchPathsWithDataSrc)
25:         BranchInfos[actor] = branchPathsWithDataSrc
26:     end for
27: end for
28: return BranchInfos
```

分支简化是非常重要的一步，如果没有分支简化操作或者分支简化没有在遍历各个组件的过程中进行，后面将会面临前面提到的代码生成的“分支爆炸”问题。算法 2 详细描述了对于单个组件的分支简化算法，对应到算法 1 中的第 24 行的 *simplifyBranchPath* 函数。该算法的整体思路是遍历当前组件的所有分支路径的组合，找到可以合并的分支路径，并使用合并后的分支路径将其替换，比如{0,0B,0F}和{0,0B,1F}这两个分支路径可以合并为{0,0B}。为了避免无意义的分支路径组合的遍历，首先对所有分支路径按照分支路径的长度从小到大排序，比如{0,0B,0F}、{0,1B}、{0,0B,1F}排序后变为{0,1B}、{0,0B,0F}、{0,0B,1F}，如算法 2 的第 1 行所示。然后从后向前进行匹配寻找可能合并的分支路径，这里的匹配条件为：1)具有相同的分支数量；2)具有完全相同的数据源；3)除分支路径内最后一个分支外，前面的分支完全相同，如{0,0B,0F}和{0,0B,1F}；4)分支路径内最后一个分支的分支组件相同。分支路径匹配的伪代码见算法 2 的第 2-23 行，其中第 9、10 和 20 行用来存储匹配到的所有分支路径。第 24-31 行尝试将匹配到的分支路径进行合并，可以合并意味着所有匹配到的分支路径各自的最后一个分支是互补的，比如{0,0B,0F}和{0,0B,1F}中的 0F 和 1F 在二分支的情况下就是互补的，多分支也是同理。如果可以合并则将合并后的分支路径插入当前分支路径的集合中继续参与匹配，因为有些情况的分支路径集合可以进行多次合并，比如，{0,1B}、{0,0B,0F}、{0,0B,1F}最终可以合并为{0}。之后继续尝试新的匹配，直到遍历完所有分支路径的组合。

算法 2. 对于单个组件的分支信息简化算法.

输入: 组件的所有分支信息组 *BranchPathsWithDataSrc*.

输出: 简化的组件所有分支信息组 *SimpleBranchPathsWithDataSrc*.

```
1:  SimpleBranchPathsWithDataSrc = sortByBranchPathLength(BranchPathsWithDataSrc)
2:  curPos = len(SimpleBranchPathsWithDataSrc) - 1
3:  while curPos>0 do
4:      branchPath = SimpleBranchPathsWithDataSrc[curPos].key
5:      dataSrcs = SimpleBranchPathsWithDataSrc[curPos].value
6:      subBranchPath = branchPath[0:-1]
7:      lastBranch = branchPath[-1]
8:      findPos = curPos – 1
9:      branchPathsWithSameDataSrc = ∅
10:     branchPathsWithSameDataSrc.append(branchPath)
11:     while findPos >0 do
12:         branchPathFind = SimpleBranchPathsWithDataSrc[findPos].key
13:         dataSrcsFind = SimpleBranchPathsWithDataSrc[findPos].value
14:         subBranchPathFind = branchPath[0:-1]
15:         lastBranchFind = branchPath[-1]
16:         if len(branchPath)= len(branchPathFind) and
17:                 dataSrcsFind == dataSrcs and
18:                 subBranchPath == subBranchPathFind and
19:                 getBranchActor(lastBranch) == getBranchActor(lastBranchFind) then
```

```
20:             branchPathsWithSameDataSrc.append(branchPathFind)
21:         end if
22:         findPos = findPos - 1
23:     end while
24:     if len(branchPathsWithSameDataSrc) > 1 then
25:         if canBranchPathsMerge(branchPathsWithSameDataSrc) then
26:             SimpleBranchPathsWithDataSrc.insert({subBranchPath, dataSrcs})
27:             SimpleBranchPathsWithDataSrc.remove(branchPathsWithSameDataSrc)
28:             curPos = curPos - len(branchPathsWithSameDataSrc) + 1
29:             continue
30:         end if
31:     end if
32:     curPos = curPos - 1
33: end while
34: return SimpleBranchPathsWithDataSrc
```

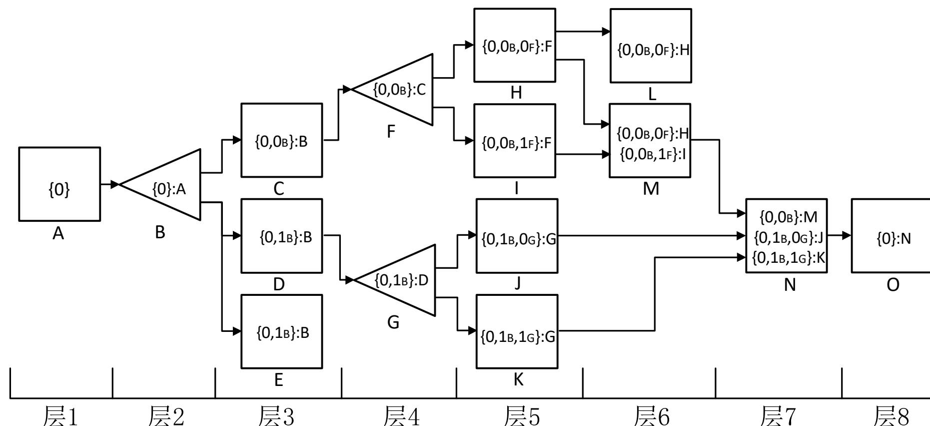


Fig.3 The result of the branch marking of the model in Figure 2.

图 3 图 2 中模型的分支标记结果

下面以图 2 中的模型为例，对整个分支标记算法进行讲解。首先对第 1 层进行标记，由于我们默认模型最开始就处于一个单独的分支中，所以我们对 A 组件标记{{0}}，其中 0 代表第一个分支，且不带有数据源。分支标记逐层进行，接下来对第 2 层中的 B 组件进行标记，因为分支会对其后面所有组件有影响，所以分支标记要向后续组件传递，所以 B 组件也要标记{{0}}，但此时 B 接收来自 A 的输出数据，所以记录在该分支下接受 A 的数据，记为{{0}:A}。接下来标记第三层，由于 B 组件产生了两个分支，所以 C 组件要受到 B 组件第一个分支的影响，同时 C 组件也仍然在最开始的 A 组件所在的分支的控制之下，所以 C 组件要标记为$\{\{0,0_B\}:B\}$，表示其在最初的{0}分支的控制下又在 B 的第一个分支控制下，并且接收来自 B 组件的数据。同理，D 和 E 组件标记为$\{\{0,1_B\}:B\}$。第 4 层的 F 和 G 组件会继续嵌套分支，第 5 层的所有组件相比第 3 层组件的分支标记会多出来由 F 和 G 组件所带来的分支，例如 H 组件的分支标记为$\{\{0,0_B,0_F\}:F\}$，I 组件的分支标记为$\{\{0,0_B,1_F\}:F\}$。第 6 层的 M 组件的分支标记和前面的组件有些不同，M 组件汇聚了来自 H 和 I 组件的两个分支，虽然他们的两个分支合并到了一起，但是 M 组件仍要区分这两个分支，因为在这两个分支下，一个传递的是 H 组件的数据，另一个传递的是 I 组件的数据。显然 M 组件的

代码不能写在 F 分支代码的控制范围之外，要分别在 F 分支代码的两个控制区域中写入带有不同数据源的 M 组件的代码。所以，M 组件的分支标记有两个分支路径和$\{0,1_B,1_G\}$，而其中$\{0,0_B,0_F\}$和$\{0,0_B,1_F\}$这两个分支路径都带来的是M组件的输出数据，所以可以将0_F和1_F进行合并。最终N组件的分支标记为$\{\{0,0_B\}$:M, $\{0,1_B,0_G\}$:J, $\{0,1_B,1_G\}$:K$\}$。最后 O 组件汇聚所有分支，根据算法 2，$\{0,1_B,0_G\}$和$\{0,1_B,1_G\}$可以合并为$\{0,1_B\}$，而$\{0,1_B\}$和$\{0,0_B\}$再次合并为$\{0\}$，并且 O 组件只接收来自 N 组件的数据，所以 O 组件的分支标记为$\{\{0\}$:N$\}$。针对图 2 中的分支标记结果可直接参见图 3。

4 代码生成位置表的构造

在上一节中，我们已经对模型中所有的组件进行了分支标记，这些分支标记就是这些组件的控制流信息，有了这些信息，最直接的生成代码的方法就是逐层逐组件地生成组件代码，如果组件受到分支影响，就在组件的执行代码前加上条件约束。显然，这样做会生成大量的冗余的条件判断语句，为了尽可能地减少冗余的条件判断代码，生成控制逻辑更加清晰的代码，我们构造一个代码生成位置表，通过向分支控制的语句块插入伪代码的方式来确定完整的基于控制流的组件执行顺序。一个完整的代码生成位置表可能会包含以下几种元素，如表 1 所示。

Table 1 Elements contained in the code generation location table

表 **1** 代码生成位置表中包含的元素

元素	参数	对应的代码	描述
ActorExe	Actor,DataSrc	Execution code of Actor	以 DataSrc 为参数，生成对应的 Actor 的执行代码
Branch	BranchPaths	If/Else if/Else/Switch	表示分支的开始，依据 BranchPaths 生成分支判断代码
Inserter	BranchPaths	Null	表示后续满足 BranchPaths 分支条件的组件的插入位置

代码生成位置表的构造方法如算法 3 所述。该算法的主体流程是首先构造一个空的代码生成位置表 *CGLT*，如第 1-2 行所示。然后逐层逐组件地根据组件地分支信息向代码生成位置表中添加元素。这里很重要的一点是每一层在执行的时候会使用上一层计算得到的 *CGLT*，也就是说，组件寻找 *Inserter* 和插入 *Inserter* 以及 *Branch* 不会影响到当前层的其它组件，在每一层计算完之后，由算法第 23 行的 *CGLT.update*() 来整理在这一层中插入的各种元素，让它们能够在下一层的计算中生效。第 5-21 行为针对每个组件的处理代码，首先获得组件的分支路径集合，如第 5-6 行所示。之后根据该组件的分支路径来确定如何修改代码生成位置表。若该组件的所有分支路径不存在冲突，也就是这些分支路径的条件不会同时成立，就对这些分支路径分别处理，如第 8-16 行所示。对于每条分支路径若在代码生成位置表中可以直接找到带有该分支路径的 *Inserter*，就直接在该 *Inserter* 位置插入组件的执行代码；如果没有，则去掉分支路径的最后一个分支作为 *subBranchPath* 去寻找 *Inserter*，如果去掉一个分支后仍然找不到，就再去掉最后一个分支作为新的 *subBranchPath*，直到找到存在的 *Inserter*。这个迭代是有限次数的，因为至少最后会找到带有$\{\{0\}\}$的 *Inserter*。在迭代过程中，如果找到了满足条件的 *Inserter*，首先移除代码生成位置表中所有该 *Inserter* 所在的分支之内的所有 *Inserter*。之后在该 *Inserter* 位置创建多层的分支结构，并在每个分支结构中添加对应的 *Inserter*。例如，迭代寻找前的分支路径为$\{0, 0_B, 0_F\}$，最终找到了带有$\{\{0\}\}$的 *Inserter*，那么就在这个 *Inserter* 的位置依次插入$\{0, 0_B\}$, $\{0, 0_B, 0_F\}$, $\{0, 0_B, 1_F\}$, $\{0, 1_B\}$, $\{0, 1_B, 0_F\}$, $\{0, 1_B,1_F\}$这些分支和对应的 *Inserter*。然后将当前组件的执行代码插入到$\{0, 0_B,0_F\}$对应的 *Inserter* 位置。针对组件的分支路径存在冲突的情况，我们直接在带有$\{\{0\}\}$的 *Inserter* 位置插入这些分支路径的全部组合的分支及对应的 *Inserter*，如第 18-20 行所示。这种情况适用于复杂的分支交叉的情况。比如某个组件的分支路径集合为$\{\{0, 1_X\}, \{0, 0_Y\}\}$(其中 X 组件的 1 分支和 Y 组件的 0 分支可能会同时执行)，则创建$\{\{0, 1_X\}$ && $\{0, 0_Y\}\}$、$\{\{0, 1_X\}\}$、$\{\{0, 0_Y\}\}$这三种分支及对应的 *Inserter*，其中“&&”表示逻辑与，意为两个分支都成立。这个例子最终会生成如代码片段 1 所示代码。在该算法的最后，移除那些分支内容为空的 *Branch* 元素，并移除所有的 *Inserter*，因为它们并

不会生成实际的代码。

代码片段 1. 多分支路径组件对应生成的代码示意

```
1.  if (1X && 0Y){          //Branch
2.      Actor(1X, 0Y)       //ActorExe
3.  }
4.  else if (1X){           //Branch
5.      Actor(1X)           //ActorExe
6.  }
7.  else if (0Y){           //Branch
8.      Actor(0Y)           //ActorExe
9.  }
```

算法 3. 代码生成位置表的构造算法.

输入: 确定调度顺序的数据流模型 *ModelLayers*, 模型中所有组件的分支标记 *BranchInfos*.

输出: 代码生成位置表 *CGLT*(*code generation location table*)

```
1:  CGLT = ∅
2:  CGLT.insertInserter(0, {{0}})
3:  for each layer in ModelLayers
4:      for each actor in layer
5:          branchPathsWithDataSrc = BranchInfos[actor]
6:          branchPaths = branchPathsWithDataSrc.values
7:          if not isBranchPathsConflict(branchPaths) then
8:              for each branchPath in branchPaths
9:                  subBranchPath = branchPath
10:                 while not CGLT.findInserter(subBranchPath) do
11:                     subBranchPath = subBranchPath[0:-1]
12:                 end while
13:                 CGLT.removeInserterContain(subBranchPath)
14:                 CGLT.insertGapBranchAndInserter(branchPath, subBranchPath)
15:                 CGLT.insertActorExe(branchPath, actor, branchPathsWithDataSrc[branchPaths])
16:             end for
17:         else
18:             CGLT.removeInserterContain({0})
19:             CGLT.insertCombinedBranchPathsAndInserter({0}, branchPaths)
20:             CGLT.insertActorExe(branchPath, actor, branchPathsWithDataSrc[branchPaths])
21:         end if
22:     end for
23:     CGLT.update()
24: end for
25: CGLT.removeEmptyBranch()
26: CGLT. removeAllInSerter()
27: return CGLT
```

接下来，我们仍然以图 2 中的模型为例，根据图 3 中计算好的分支标记来构建一个完整的 *CGLT*。由

于受到篇幅和排版的限制，我们没有办法将构造 *CGLT* 的每一步都罗列出来。代码片段 2 中展示了最终生成的 *CGLT*，其中用删除线划掉的代表最终被删掉的元素，在注释中注明了该元素是在什么时候创建和删除的。对于图 2 中的模型，一开始我们构建一个空的 *CGLT*，并插入带有{{0}}分支路径的 *Inserter*，用于指定后续元素插入的位置，对应到代码片段 2 的第 31 行。之后逐层遍历模型，首先是 A 组件，A 组件具有{{0}}分支标记，所以直接将 A 组件的执行语句插入到带有{{0}}的 *Inserter* 前面，对应到代码片段 2 的第 1 行。第 2 层的 B 组件也是同理，只不过 B 组件的执行参数为 A 组件的输出，对应到代码片段 2 的第 2 行。虽然 B 组件是分支组件，但是它对应的分支不在这里创建，而是在后面的组件遍历中按需要创建的。接下来，遍历到第 3 层的 C 组件，这时，我们发现 C 组件的分支标记{{0,0B}}在 *CGLT* 中查不到对应的 *Inserter*，而且 C 组件的分支标记中也只有一个分支路径{0,0B}，所以，这里按照算法 3 的 9-15 行进行处理。去掉分支路径上的最后一个分支，继续查找子分支路径{0}，找到带有{{0}}的 *Inserter*，在该 *Inserter* 前插入 B 组件的所有分支，并将 C 组件的执行语句插入到新的带有{{0,0B}}的 *Inserter* 位置，对应到代码片段 2 的第 3、4、16、17 和 28 行。D 和 E 组件同理。接下来第 4 层的 F 和 G 组件，分别插入到带有{{0, 0B}}和{{0, 1B}}分支标记的 *Inserter* 位置，对应到代码片段 2 的第 5 和 20 行。第 5 层的 H、I、J、K 组件的处理方式同 C 组件。第 6 层的 L 和 M 组件可以直接根据它们的分支标记确定插入位置。第 7 层的 N 组件和 M 组件类似，只不过在处理{0, 0B}分支的时候要把{0, 0B, 0F}和{0, 0B, 1F}对应的 *Inserter* 删除，对应到代码片段 2 的第 10、14 行。第 8 层的 O 组件的处理方式同 M。最后，把 *CGLT* 中的所有 *Inserter* 删除。根据代码片段 2 中的代码生成位置表，可以直接翻译为各种主流的编程语言，将 *ActorExe* 元素转化为组件对应的执行语句，*Branch* 元素转化为 if、else if、switch 等流程控制语句即可。

代码片段 2. 由图 2 中模型生成的代码生成位置表

1: *ActorExe*: *A*() //迭代第 1 层 A 组件时创建
2: *ActorExe*: *B*(*A*) //迭代第 2 层 B 组件时创建
3: *Branch* ({{0, 0B}}) //迭代第 3 层 C 组件时创建
4: *ActorExe*: *C*(*B*) //迭代第 3 层 C 组件时创建
5: *ActorExe*: *F*(*C*) //迭代第 4 层 F 组件时创建
6: *Branch* ({{0, 0B, 0F}}) //迭代第 5 层 H 组件时创建
7: *ActorExe*: *H*(*F*) //迭代第 5 层 H 组件时创建
8: *ActorExe*: *L*(*H*) //迭代第 6 层 L 组件时创建
9: *ActorExe*: *M*(*H*) //迭代第 6 层 M 组件时创建
10: ~~*Inserter*({{0, 0B, 0F}})~~ //迭代第 5 层 H 组件时创建，迭代第 7 层 N 组件时删除
11: *Branch* ({{0, 0B, 1F}}) //迭代第 5 层 H 组件时创建
12: *ActorExe*: *I*(*F*) //迭代第 5 层 I 组件时创建
13: *ActorExe*: *M*(*I*) //迭代第 6 层 M 组件时创建
14: ~~*Inserter*({{0, 0B, 1F}})~~ //迭代第 5 层 H 组件时创建，迭代第 7 层 N 组件时删除
15: *ActorExe*: *N*(*M*) //迭代第 7 层 N 组件时创建
16: ~~*Inserter*({{0, 0B}})~~ //迭代第 3 层 C 组件时创建，迭代第 8 层 O 组件时删除
17: *Branch* ({{0, 1B}}) //迭代第 3 层 C 组件时创建
18: *ActorExe*: *D*(*B*) //迭代第 3 层 D 组件时创建
19: *ActorExe*: *E*(*B*) //迭代第 3 层 E 组件时创建
20: *ActorExe*: *G*(*D*) //迭代第 4 层 G 组件时创建
21: *Branch* ({{0, 1B, 0G}}) //迭代第 5 层 J 组件时创建
22: *ActorExe*: *J*(*G*) //迭代第 5 层 J 组件时创建
23: *ActorExe*: *N*(*J*) //迭代第 7 层 N 组件时创建

```
24:         ~~Inserter({{0, 1B, 0G}})~~      //迭代第 5 层 J 组件时创建，迭代第 8 层 O 组件时删除
25:     Branch ({{0, 1B, 1G}})               //迭代第 5 层 J 组件时创建
26:         ActorExe: K(G)                   //迭代第 5 层 K 组件时创建
27:         ActorExe: N(K)                   //迭代第 7 层 N 组件时创建
28:         ~~Inserter({{0, 1B, 1G}})~~      //迭代第 5 层 J 组件时创建，迭代第 8 层 O 组件时删除
29:     ~~Inserter({{0, 1B}})~~              //迭代第 3 层 C 组件时创建，迭代第 8 层 O 组件时删除
30: ActorExe: O(N)                           //迭代第 8 层 O 组件时创建
31: ~~Inserter({{0}})~~                      //算法开始时创建，算法结束时删除
```

5 案例实验及对比

由于 Ptolemy-II 构建的带有复杂分支的数据流模型看起来更直观些，我们使用 Ptolemy-II 构建三种不同分支情况的模型实例，然后将这些实例按照本文提出的算法最终转化为 C 语言代码。实验所构建的 Ptolemy-II 模型如表 2 中的第一列所示，对应生成的代码如第二列所示。这三个模型都使用离散事件指示器(DE Director)和离散时钟(DiscreteClock)来驱动，模型的仿真时间为 50 个单位时间，离散时钟的采样时间为 1 个单位时间，所以整个模型会反复执行 51 次。模型中的 BooleanSwitch 组件为二分支的分支组件，它会根据它最下面的端口数据的逻辑值选择将左边输入数据传递到右边的哪个端口。Ramp 组件可以产生等差数列，这里我们设置的等差数列首项为-25，公差为 1，该组件每次输出的结构会从-25 开始依次加 1。Expression 为表达式组件，可以把输入数据代入表达式求解，再将结果传递到输出端口。Scale 组件可以将输入数值缩放后输出。SequencePlotter 示波器组件可以将每次输入的数据以折线的形式绘制出来。Counter 组件的输出从 1 开始，之后每次执行输出会增加 1。Display 组件可以将输入数据以文本的形式输出出来。表 2 中第二列显示的代码只是完整代码的一部分，这里只把组件的调度及计算逻辑相关的代码呈现了出来，其它的比如主函数封装、局部变量的声明等和调度无关的代码在这里不加以展示。另外，我们以标准输出“printf”作为各种输出组件对应的生成代码，并且，在由 *CGLT* 转化为代码的过程中，以 BooleanSwitch 组件为数据源的组件直接跨过 BooleanSwitch 组件向前寻找数据源，通过这种方式可以生成更精简的代码。

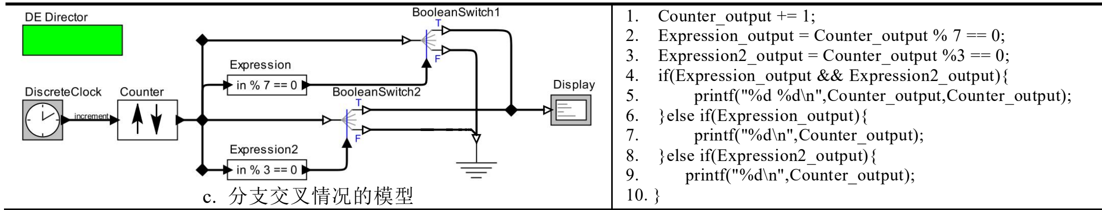
Table 2 Model examples for three different branch situations

表 **2** 三种不同分支情况的模型实例

模型	对应的 C 代码

DE Director
DiscreteClock Ramp Expression in > 0 BooleanSwitch Scale 0.1 SequencePlotter
a. 分支合并情况的模型
	1. Ramp_output += 1;
	2. Expression_output = Ramp_output > 0;
	3. if(Expression_output){
	4. printf("%f\n",Ramp_output);
	5. }else{
	6. Scale_out = Ramp_output*0.1;
	7. printf("%f\n",Scale_out);
	8. }

DE Director
DiscreteClock Ramp Expression in > 0 Expression2 in > -10 BooleanSwitch1 BooleanSwitch2 Expression3 in*5+1 Expression4 in*-4+1 Expression5 in*5+91 SequencePlotter
b. 分支嵌套情况的模型
	1. Ramp_output += 1;
	2. Expression_output = Ramp_output > 0;
	3. Expression2_output = Ramp_output > -10;
	4. if(Expression_output){
	5. Expression3_output = Ramp_output*5+1;
	6. printf("%f\n",Expression3_output);
	7. }else{
	8. if(Expression2_output){
	9. Expression4_output = Ramp_output*-4+1;
	10. printf("%f\n",Expression4_output);
	11. }else{
	12. Expression5_output = Ramp_output*5+91;
	13. printf("%f\n",Expression5_output);
	14. }
	15. }




```
1.  Counter_output += 1;
2.  Expression_output = Counter_output % 7 == 0;
3.  Expression2_output = Counter_output %3 == 0;
4.  if(Expression_output && Expression2_output){
5.       printf("%d %d\n",Counter_output,Counter_output);
6.  }else if(Expression_output){
7.       printf("%d\n",Counter_output);
8.  }else if(Expression2_output){
9.       printf("%d\n",Counter_output);
10. }
```

表 2 中的 a 模型显示一个在深度学习领域中常用的激活函数 LeakyRelu 函数的图像。当 Ramp 组件的输出大于 0 时，SequencePlotter 显示 Ramp 组件本来的输出值；当 Ramp 组件的输出小于等于 0 时，SequencePlotter 显示 Ramp 组件输出值乘以 0.1。模型中的 BooleanSwitch 组件生成了 if-else 语句(第 3 行和第 5 行)；SequencePlotter 组件合并了 BooleanSwitch 组件的两个分支，由于 SequencePlotter 组件在两个分支下的数据源不同，所以在 if 和 else 控制的语句块下各生成了一行代码(第 4 行和第 7 行)；Scale 组件仅在 BooleanSwitch 的 False 分支中，所以只在 else 控制的语句块下生成代码(第 6 行)。

表 2 中的 b 模型显示一个分段函数，由两个 BooleanSwitch 组件控制三个函数段，其中 BooleanSwitch2 组件嵌套在 BooleanSwitch1 下面。当然分段函数的模型也可以直接通过多分支组件比如 Switch 组件构造，这里仅为了演示分支嵌套情况的代码生成。该模型中的两个 BooleanSwitch 组件会生成嵌套的 if-else 语句(第 4、7、8 和 11 行)；分段函数的三个部分分别生成在这三个 if-else 控制的语句块中；最后 SequencePlotter 组件也由于数据源的不一致会生成在三个 if-else 控制的语句块中。

表 2 中的 c 模型输出可以被 3 或 7 整除的数，其中能同时被 3 和 7 整数的数会被输出两遍。在该模型中 BooleanSwitch1 和 BooleanSwitch2 组件的 True 和 False 分支分别发生交叉，这会导致 *CGLT* 中 Display 组件和接地组件(表示丢弃数据)的分支标记中会有两个分支路径，且这两个分支路径可能会同时满足。因此，这两个分支总共会有 4 种分支执行情况，但是由于接地组件并不会生成代码，所以这里只生成三个 if-else 语句块(第 4、6 和 8 行)。特别地，在 BooleanSwitch1 和 BooleanSwitch2 组件分支都为 True 的情况下，Display 组件会同时接收来自两个 BooleanSwitch 组件的数据，也就是生成的 printf 代码会输出两个值。

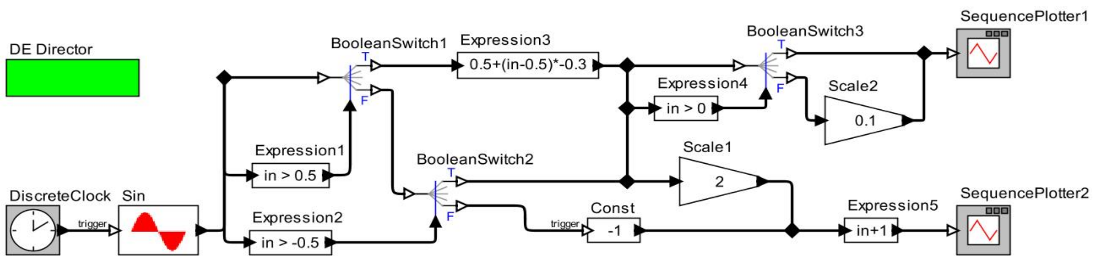
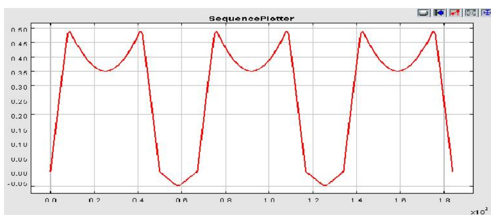
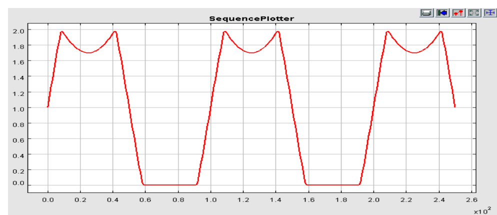


Fig.4 A model with more complex branching.

图 4 具有更复杂分支情况的模型



a. 图 4 模型中 SequencePlotter1 输出的图像



b. 图 4 模型中 SequencePlotter2 输出的图像

Fig.5 The function graph of the model output in Figure 4.

图 5 图 4 中的模型输出的函数图像

另外我们对一个更复杂的数据流模型进行代码生成实验，见图 4，该模型不仅仅具有分支合并、分支嵌套、分支交叉等基本情形，还包含分支嵌套情况下的分支交叉，如图 4 中的 BooleanSwitch1 和 BooleanSwitch2 组件的 True 分支交叉于 BooleanSwitch3、Expression4 和 Scale1 这三个组件前。在数据流语义下，该模型是合法的。该模型会输出两个函数图像，分别如图 5(a)和 5(b)所示。模型中的 Sin 组件会以一定采样间隔输出正弦曲线上的值。由 BooleanSwitch1 和 BooleanSwitch2 组件将正弦曲线分为三部分计算，最后 SequencePlotter1 组件会输出计算后的其中两部分数据，SequencePlotter2 组件会输出计算后的所有三部分数据。根据本文提出的基于分支调度标记的数据流模型的代码生成方法，图 4 中的模型生成的代码如代码片段 3 所示，可以看出，整体的代码逻辑为嵌套的两个 if-else 语句(第 4、15、16 和 26 行)，这两个 if-else 将控制逻辑分成了和图 4 中模型完全一致的三部分。模型中的 Expression4 和 BooleanSwitch3 组件及它们所有后继组件和 Scale1 组件都同时位于 BooleanSwitch1 的 True 分支和 BooleanSwitch2 的 True 分支下，所以它们会同时在两个 if-else 控制的语句块下出现(第 7、9-14 行和 18、20-25 行)。最后 SequencePlotter2 组件会在 if-else 的控制逻辑之后生成代码，并且由于该模型有两个输出组件，所以在生成的 printf 代码中会标识出来是由哪个输出组件输出的数据(第 10、13、21、24 和 31 行)。

代码片段 3. 由图 4 中模型生成的代码

```
1:   Sin_output = sin((Sin_t++)*0.01);
2:   Expression1_output = Sin_output > 0.5;
3:   Expression2_output = Sin_output > -0.5;
4:   if(Expression1_output){
5:       Expression3_output = 0.5 + (Sin_output-0.5) * -3;
6:       Expression4_output = Expression3_output > 0;
7:       Scale1_output = Expression3_output * 2;
8:       Expression5_output = Scale1_output + 1;
9:       if(Expression4_output){
10:          printf("SequencePlotter1: %f\n", Expression3_output);
11:      }else{
12:          Scale2_output = Expression3_output * 0.1;
13:          printf("SequencePlotter1: %f\n", Scale2_output);
14:      }
15:  }else{
16:      if(Expression2_output){
17:          Expression4_output = Sin_output > 0;
18:          Scale1_output = Sin_output * 2;
19:          Expression5_output = Scale1_output + 1;
20:          if(Expression4_output){
21:              printf("SequencePlotter1: %f\n", Sin_output);
22:          }else{
23:              Scale2_output = Sin_output * 0.1;
24:              printf("SequencePlotter1: %f\n", Scale2_output);
25:          }
26:      }else{
27:          Const_output = -1;
28:          Expression5_output = Const_output + 1;
```

```
29:         }
30:   }
31:   printf("SequencePlotter2: %f\n", Expression5_output);
```

最后我们基于上述带有各种分支情况的 4 个模型对 Simulink、PtolemyII 以及我们的代码生成器进行对比，对比指标包括代码行数、代码文件数和代码运行时间。由于 Simulink 不支持数据流的分支表达，所以我们将 4 个模型翻译成等价的 Simulink 模型之后再对其进行代码生成。这样，在 Simulink 中模型通过组件的表达情况也可以作为重要的参考依据，以表明数据流模型中分支组件的重要性。我们使用相同的实验环境对不同代码生成器生成的代码进行编译，之后为了更好地消除随机性误差，我们在相同的环境下运行 10000 次编译后的程序得出它们的万次运行时间(ms)。实验结果如表 3 所示：

Table 3 Comparison between Simulink, PtolemyII, and ours on code generatation

表 **3** Simulink、PtolemyII 和本文工作在代码生成上的比较

模型	组件个数	模型深度	代码生成器	代码行数	代码文件数	运行时间(ms)
分支合并情况的模型(表 2.a)	Simulink: 14 PtolemyII: 6	Simulink: 8 PtolemyII: 6	Simulink	512	9	984
			PtolemyII	11556	52	1718
			本文工作	**18**	**1**	**812**
分支嵌套情况的模型(表 2.b)	Simulink: 32 PtolemyII: 10	Simulink: 13 PtolemyII: 7	Simulink	543	9	937
			PtolemyII	12175	60	1811
			本文工作	**28**	**1**	**843**
分支交叉情况的模型(表 2.c)	Simulink: 32 PtolemyII: 8	Simulink: 12 PtolemyII: 5	Simulink	725	15	280
			PtolemyII	11864	56	1531
			本文工作	**20**	**1**	**108**
复杂分支情况组合的模型(图 4)	Simulink: 58 PtolemyII: 15	Simulink: 21 PtolemyII: 9	Simulink	541	9	1286
			PtolemyII	13940	87	运行时错误
			本文工作	**49**	**1**	**1078**
平均	Simulink: 34 **PtolemyII: 9**	Simulink: 13 **PtolemyII: 6**	Simulink	580	10	871
			PtolemyII	12383	63	1686
			本文工作	**28**	**1**	**710**

本文通过对比 PtolemyII 和 Simulink 构建相同的模型所需的组件个数和模型深度来说明使用分支组件表达模型的效率和复杂度，这里的模型深度表示模型对应的有向无环图的最长路径长度。从表 3 中可以看出，使用 PtolemyII 建模相比 Simulink 可以节省约 73.5%的组件个数，模型深度可以减少约 53.8%，使用数据流分支组件进行建模可以简化建模难度、提升建模效率。

另外，更重要的是，本文提出的解决分支组件代码生成的工作相比 PtolemyII 和 Simulink 各自的代码生成器都有明显的提升：在生成的代码行数上，相比 Simulink 减少约 95.1%，相比 PtolemyII 减少约 99.7%；在生成代码的运行时间上，相比 Simulink 提速约 18.4%，相比 PtolemyII 提速约 57.8%。Simulink 生成的代码会包含很多仿真环境的定义、数据及数据类型的定义，而实际上大部分的定义都是没有被使用的。PtolemyII 会按照事件传递执行的方式生成代码，为模型中的每个组件都生成一个单独的代码文件，并且引入了复杂的数据传递的实现代码，显然这种代码很难被用于嵌入式系统。

6 总结与展望

本文针对带有复杂数据分支结构的数据流模型提出了基于分支调度标记的代码生成方法，该方法首先通过拓扑排序对模型进行调度分层，然后以分支组件为依据对模型中的各个组件进行分支标记，之后根据分支标记生成基于控制流的组件代码生成位置表，最后可直接根据代码生成位置表转化为目标语言代码。

本文所提出的方法解决了数据流模型中由分支组件带来的建模或者代码生成的约束问题，并且可以生成和数据流模型一致的控制逻辑的代码，极大地解放了分支组件在数据流模型中的表达能力。本文中所描述的分支标记算法能够有效地避免由于分支嵌套和交叉所带来的代码生成的“分支爆炸”问题，该算法在分支标记过程中对组件的分支标记进行化简，使组件所拥有的分支标记数量一直保持最少的水平。同时，本文提出了基于分支标记的插入式代码生成方法，通过这样插入式的方法构造的代码生成位置表保证了它的控制逻辑和模型逻辑的一致性，也保证了代码中组件的执行顺序不会发生错乱。此外，通过案例实验并对比PtolemyII 和 Simulink 各自的代码生成器进一步说明了本文工作的价值意义。

本文的算法可以在构建完代码生成位置表之后做进一步的代码级的优化，当生成多个分支结构的时候，如果这些分支结构属于并列的 if 和 else if 关系，并且这些分支结构内部的代码是完全一致的，那么就可以将这些并列的 if 和 else if 合并为一个分支语句块，分支条件用或运算连接。另外，当多个分支嵌套的时候，并且这些分支嵌套只在最内层有组件的执行代码，那么就可以在不影响其它语句执行逻辑的情况下将多层分支化为一个分支，分支条件用与运算连接。在构建完代码生成位置表之后可以进行很多代码级的优化，这样能够在不影响模型语义的情况下尽可能地缩减生成的代码的长度。

References:

[1] Christopher H, Edward AL, Liu J, Liu XJ, Steve N, Xiong YH, Zheng HY. Heterogeneous concurrent modeling and design in java. Vol.1: California, California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2008. 1-34

[2] Joseph TB, Soonhoi H, Edward AL. Ptolemy: A framework for simulating and prototyping heterogeneous systems, 1992.

[3] Jiang Y, Zhang H, Li Z, Deng YD, Song XY Gu M, Sun JG. Design and Optimization of Multiclocked Embedded Systems Using Formal Techniques. IEEE Transactions on Industrial Electronics, 2015.62(2):1270–1278.

[4] Edward AL, Xiong YH. A behavioral type system and its application in Ptolemy II. Formal Aspects of Computing, 2004.16(3):210–237.

[5] Yang ZB, Pi L, Hu K, Gu ZH, Ma DF. AADL: An architecture design and analysis language for complex embedded real-time systems. Journal of Software, 2010,21(5):899−915. http://www.jos.org.cn/1000-9825/3700.htm

[6] Duane AA. A computation model with data flow sequencing [Ph.D. Thesis]. Stanford, CA: Stanford University, 1969.

[7] Jiang Y, Zhang HH, Zhang HF, Liu H, Song XY, Gu M Sun JG. Design of mixed synchronous/asynchronous systems with multiple clocks. IEEE Transactions on Parallel and Distributed Systems, 2015(26):2220-2232.

[8] Philip B, Sanjeev K, Edward AL, Liu XJ, Zhao Y. Modeling of sensor nets in Ptolemy II. In: Proceedings of the 3rd international symposium on Information processing in sensor networks. New York: Association for Computing Machinery. 2004:359–368.

[9] Christopher B, Edward AL, Stavros T. Exploring models of computation with Ptolemy II. In: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. New York: Association for Computing Machinery, 2010:331–332.

[10] Johan E, Jörn WJ, Edward AL, Liu J, Liu XJ, Jozsef L, Stephen N, Sonia S, Xiong YH. Taming heterogeneity-the Ptolemy approach. Proceedings of the IEEE, 2003(1):127–144.

[11] Jiang Y, Zhang HH, Zhang HF, Zhao XY, Liu H, Sun CN, Song XY, Gu M, and Sun JG. Tsmart-galsblock: A toolkit for modeling, validation, and synthesis of multi-clocked embedded systems. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York: Association for Computing Machinery. 2014:711–714.

[12] Paul LG, Jean-Pierre T, and Jean-Christophe LL. Polychrony for system design. Journal of Circuits, Systems, and Computers, 2003(3):261–303.

[13] Jiang Y, Liu H, Song HB, Kong H, Wang R, Guan Y, Lui S. Safety-Assured Model-Driven Design of the Multifunction Vehicle Bus Controller. IEEE Transactions on Intelligent Transportation Systems, 2017(19):3320-3333.

[14] Jiang Y, Song HB, Yang YX, Liu H, Gu M, Guan Y, Sun JG, Lui S. Dependable model-driven development of cps: From stateflow simulation to verified implementation. ACM Transactions on Cyber-Physical Systems, 2018(3): Article 12

[15] Alejandro C, Juan BP, Mercedes R. MEdit4CEP-Gam: A model-driven approach for user-friendly gamification design, monitoringand code generation in CEP-based systems. Information and Software Technology, 2018(95):238–264.

[16] Gabriel S, Jose AG, Ricardo T. Code generation using model driven architecture: A systematic mapping study. Journal of Computer Languages, 2020(56):100935.

[17] Du Y, Guo DH, Chen X, Ren L. Model-Driven Visualization Generation System. Journal of Software, 2016,27(5):1199-1211. http://www.jos.org.cn/1000-9825/4959.html

[18] Zhou G, Man-Kit L, Edward AL. A code generation frame-work for actor-oriented models with partial evaluation. In International Confer-ence on Embedded Software and Systems. Springer, 2007(29):193–206.

附中文参考文献:

[5] 杨志斌,皮 磊,胡 凯,顾宗华,马殿富.复杂嵌入式实时系统体系结构设计与分析语言:AADL.软件学报,2010,21(5):899-915. http://www.jos.org.cn/1000-9825/3700.htm

[17] 杜一,郭旦怀,陈昕,任磊,戴国忠.一种模型驱动的可视化生成系统.软件学报,2016,27(5):1199-1211. http://www.jos.org.cn/1000-9825/4959.html