文章编号：2096-1472(2021)-08-34-05
DOI:10.19644/j.cnki.issn2096-1472.2021.08.008

基于活动图与顺序图的自动代码生成

文 浩[1,2]，蒋建民[3]，张 仕[1]，洪 中[1]

(1.福建师范大学计算机与网络空间安全学院，福建 福州 350117；
2.中国科学院成都计算机应用研究所，四川 成都 610041；
3.成都信息工程大学软件工程学院，四川 成都 610225)
✉caswh96@foxmail.com; jjm@cuit.edu.cn; shi@fjnu.edu.cn; fjfzhz@fjnu.edu.cn

摘 要：在模型驱动开发过程中，为了减轻开发人员的负担，通常采用自动工具生成代码框架。本文提出了一种基于活动图与顺序图自动生成代码框架的方法，并且基于该方法开发了原型工具。首先，给出了活动图和顺序图的形式化表达；其次，基于活动图和顺序图模型提出了三个实现自动代码生成的算法，该算法是通过活动图来描述对象内部的活动次序，并且利用顺序图体现对象之间的交互；最后，给出了一个原型工具，实现了模型到代码框架的自动转换。该工作可以确保转换的正确性，并且提高软件开发效率。

关键词：模型驱动开发；自动代码生成；形式化方法
中图分类号：TP311.5 **文献标识码**：A

Automatic Code Generation based on Activity and Sequence Diagrams

WEN Hao[1,2], JIANG Jianmin[3], ZHANG Shi[1], HONG Zhong[1]

(1.*College of Computer and Cyber Security*, *Fujian Normal University*, *Fuzhou* 350117, *China*;
2.*Chengdu Institute of Computer Applications*, *Chinese Academy of Sciences*, *Chengdu* 610041, *China*;
3.*College of Software Engineering*, *Chengdu University of Information Technology*, *Chengdu* 610225, *China*)
✉caswh96@foxmail.com; jjm@cuit.edu.cn; shi@fjnu.edu.cn; fjfzhz@fjnu.edu.cn

Abstract: In the process of model-driven development, automatic tools are usually used to generate code frameworks in order to reduce the burden on developers. This paper proposes a method to automatically generate code framework based on activity diagram and sequence diagram, and a prototype tool is developed based on this method. First, this paper gives a formal expression of activity diagrams and sequence diagrams; secondly, three algorithms for realizing automatic code generation are proposed based on the activity diagram and sequence diagram. The algorithm uses activity diagrams to describe the internal activity sequence of the object and uses the sequence diagram to reflect the interaction between the objects; finally, a prototype tool is given to realize the automatic conversion from the model to the code framework. This work can ensure the correctness of the conversion and improve the efficiency of software development.

Keywords: model-driven development; automatic code generation; formal methods

1 引言(Introduction)

在各种开发方法中，模型驱动设计(Model-Driven Design)因其合理与高效已被工业界广泛运用[1-2]。在模型驱动设计框架下，先对系统进行建模，随后通过大量的分析与验证对该模型进行更新与修改，这就使得在设计的早期阶段就可以对错误进行检测和纠正。统一建模语言(UML)就是该开发过程中最常用的一种可视化建模工具，它提供了很多不同类型的图表，分别从不同的视点去建模系统，比如数据、行为、交互、组件架构等机制。

当开发人员用UML完成对系统的建模后，需要进一步将

基金项目：国家自然基金(61772004)；国家自然基金(61361136002)；福建省自然基金(2018J01777)；福建省自然基金(2014J01221)；成都信息工程大学引进人才科研启动项目(KYTZ202009).

抽象模型转换为更具体的模型，并最终转换为可执行代码。但在过去的大部分转换过程中，由于没有完整的自动化方法，导致整个过程的成本较高，并且其中的手工部分因为缺乏形式化方法的支撑，也会造成整个过程容易出错[3]。近些年来，由于自动生成代码领域的火热，关于从UML模型生成对应代码的研究变得越来越多[4−5]。但遗憾的是，目前大部分的研究都是将单一的UML模型与代码之间进行关联，这样的策略在一定程度上会使得生成的代码缺失一定的实施细节。为了解决上述问题，本文选择了UML中的两种模型，一种是活动图(Activity Diagram)，主要用于对业务流程进行建模；另一种是顺序图(Sequence Diagram)，主要用于描述软件的对象或者进程间的交互行为。基于这两种不同图的不同视点，本文提出了一个更加完善的自动代码生成的方法。

我们依次给出了活动图和顺序图模型的形式化定义，并基于两个模型之间的关联，将模型元素与代码语句进行了对应，从而实现了代码生成。本文给出的三个算法，则对应到如何实现代码生成的自动化或半自动化。

2　活动图和顺序图(Activity diagram and sequence diagram)

活动图被广泛运用于建模工作流或模拟业务流程，它在本质上是一种流程图，着重表现从一个活动到另一个活动的控制流，而顺序图主要是用于显示对象之间交互的图。相比于使用单一UML模型，将活动图与顺序图之间进行关联后，可以结合不同的视点以及两种图不同的特性，使生成的代码框架更加完整，细节更多，并更实用。在本节中，首先给出了活动图和顺序图的形式化定义，随后提出了一个关联函数，用于描述活动图与顺序图之间的关系。

活动图是一种运用节点和边的组合，可视化描述活动执行过程的UML图，其中的节点可以分为活动节点、对象节点和控制节点。活动节点是活动图中最主要的元素之一，它用来表示一个活动；对象节点是用来帮助定义活动中对象流的抽象活动节点；而控制节点则是一种可以协调其他节点之间流的特殊的活动节点。下面给出了一个活动图抽象语法的定义。

定义1：一个活动图是一个九元组$AD=<A,AO,Dn,Mn,Fn,Jn,R,Ia,Fa>$，其中：

(1)$A=AO\cup Dn\cup Mn\cup Fn\cup Jn\cup Ia\cup Fa$；

(2)AO，活动节点和对象节点的集合；

(3)Dn，选择节点的集合；

(4)Mn，合并节点的集合；

(5)Fn，分叉节点的集合；

(6)Jn，汇合节点的集合；

(7)$R\subseteq A\times A$，活动和节点间关系的集合；

(8)Ia，初始节点的集合；

(9)Fa，终止节点的集合。

对活动图的形式化定义来自我们目前的工作[6−8]。为了简化处理，在本文中不区分对象节点和活动节点，即将所有对象节点都视作活动节点。同样为了方便描述，我们给出了前置集和后置集的定义，即对于任意节点$x\in A$，它的前置集和后置集可以分别表示为${}^{\cdot}x=\{y\in A|(y,x)\in R\}$和$x^{\cdot}=\{y\in A|(x,y)\in R\}$。

例1：图1(a)是一个活动图，可以表示为$AD=<A,AO,Dn,Mn,Fn,Jn,R,Ia,Fa>$，其中$A=\{i,a,b,p,c,d,e,g,h,j,k,dn1,dn2,mn1,mn2,fn,jn,f\}$，$AO=\{a,b,p,c,d,e,g,h,j,k\}$，$Dn=\{dn1,dn2\}$，$Mn=\{mn1,mn2\}$，$Fn=\{fn\}$，$Jn=\{jn\}$，$R=\{(i,a),(a,mn1),(mn1,b),(b,dn1),(dn1,p),(p,mn1),(dn1,c),(c,dn2),(dn2,d),(d,mn2),(dn2,e),(e,fn),(fn,g),(fn,h),(g,jn),(h,jn),(jn,j),(j,mn2),(mn2,k),(k,f)\}$，$Ia=\{i\}$，$Fa=\{f\}$。并且可以很明显得知，对于A中任意一个元素，比如$b$，可以得到${}^{\cdot}b=\{mn1\}$和$b^{\cdot}=\{dn1\}$。

顺序图可视化地描述了对象之间按照时间顺序进行信息交互的过程，它将交互展示为一个二维图表，垂直维度上是一条被称为生命线的虚线，该虚线绘制在每个对象的下面，用于体现时间与顺序；而水平维度上主要体现在对象之间的信息交互，即消息的传递。下面给出了一个顺序图抽象语法的定义。

定义2：一个顺序图是一个三元组$SD=<S,O,M>$，其中：

(1)S，信号的集合；

(2)O，对象的集合；

(3)$M\subseteq S\times O\times O$，消息的集合。

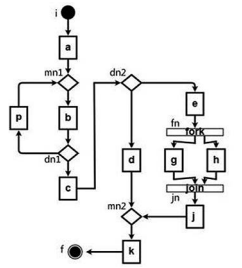
在本文中，因为只关注对象之间的交互内容，所以在给出顺序图的形式化定义时，选择了一种比较简易的表达方式，该方式能更清晰地描述消息的传递过程。

例2：图1(b)是一个顺序图，可以表示为$SD_1=<S_1,O_1,M_1>$，其中$S_1=\{c1,c2,c3,c4,c5\}$，$O_1=\{AD,x,y\}$，$M_1=\{(c1,AD,x),(c2,x,y),(c3,y,x),(c4,x,AD),(c5,AD,y)\}$。

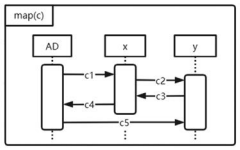
为了在上述活动图与顺序图之间建立联系，本文利用一个关联函数，将每个顺序图视为活动图中的某一个活动节点的精化，即活动图中的每一个活动都可以通过顺序图展开，精化后的活动图相比于原来的活动图具有更多的细节。

设Φ为活动节点的集合，Σ为所有顺序图的集合；称函数$map:\Phi\to\Sigma$为“关联函数”。也就是说，对于所有的$x\in AO$，都可以通过关联函数map得到与之对应的顺序图$map(x)=<S,O,M>$，即通过一次关联函数，得到的顺序图$map(x)$是对活动节点x所属的活动图AD的一次细节填充。

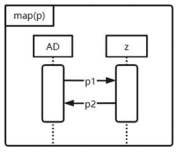
例3：图1(a)表示的是一个活动图$AD=<A,AO,Dn,Mn,Fn,Jn,R,Ia,Fa>$，图1(b)、图1(c)分别表示的是两个顺序图$SD_1$和$SD_2$。当$c$，$p\in AO$，分别存在$map(c)=SD_1$，$map(p)=SD_2$，即可以将$SD_1$和$SD_2$视作活动节点$c$和$p$所属的活动图$AD$的细节填充，其余的活动节点保持不变。



(a)AD



(b)SD_1



(c)SD_2

图1 一个活动图和两个顺序图的实例

Fig.1 Examples of an activity diagram and two sequence diagrams

3 代码生成规则(The rules for code generation)

前面介绍了关联函数，建立了活动图与顺序图之间的联系，基于这种联系，此部分提出了具体的代码生成规则。首先，给出了一种代码生成函数，使活动图和顺序图中的节点与程序语句间形成对应关系；接着设计了三个算法，用于实现基于活动图和顺序图的代码自动生成。

3.1 代码生成函数

把UML图转换成代码框架的核心问题，就是如何使得图形中的核心元素——节点，与代码语句一一对应。为了在活动图和顺序图的节点与Java语句之间建立联系，下面给出了一个代码生成函数的定义。

设Ψ是活动节点和消息的集合，Λ是构成Java语言的语句集合；称函数$g:\Psi\rightarrow\Lambda$为“代码生成函数”。

当代码生成函数的输入分别为活动节点和消息时，可以得到：

(1) $\forall x\in AO$，$x^{\cdot}\subseteq Dn$：$g(x)$=boolean x()；

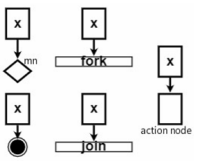
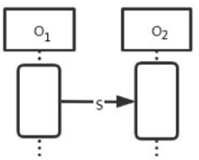
(2) $\forall x\in AO$，$x^{\cdot}\subseteq A\backslash Dn$：$g(x)$=void x()；

(3) $\forall x\in M$，$x=(s,o_1,o_2)$：$g(x)=o_2.s()$。

表1给出了从活动节点和消息到Java代码的详细对应关系。当代码生成函数的输入是活动图的活动节点时，可以根据该活动节点的后件进行分类：一种是当活动节点的后件为选择节点时，因为选择节点需要一个布尔值的输入，所以将该活动节点对应的方法的返回值设置成布尔型。另一种是当活动节点的后件为非选择节点时，因为除了选择节点，其他的节点并不需要其前件的输出作为该节点的输入，所以将该活动节点对应的方法的返回值设置为空。当代码生成函数的输入是顺序图的消息时，可以通过识别该消息的发送对象与接收对象生成对应的方法，即该消息的接收对象执行该消息的对应方法。

表1 活动节点和消息的转换规则

Tab.1 Transformation rules for action nodes and messages

节点类型	图形节点	对应代码
		boolean x()
$x\in AO$		void x()
$x\in M$		o_2.s()

3.2 算法设计

本文的自动代码生成策略是，通过依次访问活动图中的活动节点以及各类节点之间的执行顺序来生成代码框架中的类定义与主函数内部的执行逻辑，再通过识别顺序图对象之间的通信来完善方法的调用。基于关联函数map和代码生成函数g，本节提出了算法1$Gcd(AD,set_1)$、算法2$Sc(AD)$和算法3$Gel(AD,set_2,lp,AD_code)$，用于实现从活动图自动生成代码。

算法1$Gcd(AD,set_1)$用于生成代码框架中的类定义，如图2所示。该算法以活动图AD和顺序图的集合为输入，输出该活动图AD对应生成的代码AD_code。下面给出算法的具体解释。

第1 行是根据输入的活动图的名字创建相应的Java包。第2—13 行则是遍历该活动图中的所有活动节点。其中，第3—6 行是识别当前活动节点的对象，如果在当前的代码框架中没有与该对象名相同的类，则添加对应的类定义；第7 行是向类中添加该活动节点对应的方法；第8—12 行是通过映射函数map寻找到set_1中用于精化活动图节点的对应顺序图，并且识别对应顺序

图中所有的消息，生成对应的方法调用。第14—15 行是向代码框架中添加Main类，并且向其中添加main()方法，最后是输出生成的Java代码框架。

Algorithm 1 生成类定义Gcd

Input:
一个活动图$\mathcal{AD}=\langle A, AO, Dn, Mn, Fn, Jn, R, Ia, Fa\rangle$和一个集合$set_1=\{map(x)|x\in AO\}$

Output:
代码框架$\mathcal{AD}_code$

```
1: 创建一个Java的包,以活动图的名字命名,记作AD_code;
2: for each x ∈ AO do
3:    识别该活动对应的对象,记作x_obj;
4:    if AD_code中没有x_obj类 then
5:       向AD_code里添加x_obj类;
6:    end if
7:    向x_obj类中添加g(x);
8:    if map(x) ∈ set1 then
9:       for each m ∈ {(a,b,c) ∈ M|b = x_obj} do
10:         向g(x)中添加c.a();
11:      end for
12:   end if
13: end for
14: 向AD_code中添加Main类,并且在Main类中添加main()方法;
15: 输出AD_code;
```

图2 生成类定义的算法1

Fig.2 Algorithm 1 for generating class definitions

算法1执行完成之后，活动图对应的代码框架中已经有了类定义的部分，但一个完整的代码框架除了类定义及方法声明，其主函数中还应该有具体的执行逻辑。为了准确描述活动图中顺序、选择、循环、并发这四种不同的情况，下面分别给出了算法2$Sc(AD)$和算法3$Gel(AD, set_2, lp, AD_code)$，用于生成main()方法内具体的程序执行逻辑。

算法2$Sc(AD)$是为了找出活动图中的环，识别可能会循环执行的活动节点，如图3所示。该算法以活动图AD为输入，输出AD所有环中的活动节点的集合lp。以图1(a)中的活动图AD为例，算法2的输出为$lp=Sc(AD)=\{b, q\}$。下面给出该算法的具体解释。

Algorithm 2 找出活动图中的环Sc

Input:
一个活动图$\mathcal{AD}=\langle A, AO, Dn, Mn, Fn, Jn, R, Ia, Fa\rangle$

Output:
存放环中的活动节点的集合lp

```
1: 给出一个集合lp,令lp = ∅;
2: 给出一个集合s,令s = ∅; //存放遍历的节点
3: Function DFS(x, y)
4: for each z ∈ x° do
5:    if z == y then
6:       lp = lp ∪ s;
7:       s = ∅;
8:       return lp;
9:    else if z ∈ Fa then
10:      s = ∅;
11:      break;
12:   else
13:      s = s ∪ {z};
14:      x = z;
15:      DFS(x, y);
16:   end if
17: end for
18: end Function
19: for each x ∈ Dn do
20:    y = [°x]; //该语句表示取前置集中的节点赋给y
21:    lp = lp ∪ DFS(x, y);
22: end for
23: lp = lp ∩ AO;
24: 输出lp;
```

图3 找出环的算法2

Fig.3 Algorithm 2 for finding circle

第1—2 行分别定义一个集合lp和s，并且令lp和s为空。第3—18 行给出了一个深度优先的递归算法，判断DFS函数中第一个输入的后置节点是否与该输入的前置节点，即第二个输入之间存在连通关系，如果存在，则将存放路径上的所有节点的集合s赋值给集合lp。第19—22 行是遍历活动图中的所有选择节点，寻找活动图中的所有环，并且依次将环中的节点存放到集合lp中。第23—24 行是取集合lp中的所有活动节点，最后输出生成的集合lp。

基于算法1和算法2，算法3$Gel(AD, set_2, lp, AD_code)$可以生成一个完整的代码框架，如图4所示。该算法以活动图AD保存初始节点后件的集合set_2，算法2输出的集合lp和算法1生成的代码框架为输入，而输出则是对应的更新后的代码框架。下面给出该算法的具体解释。

Algorithm 3 生成执行逻辑Gcl

Input:
一个活动图$\mathcal{AD}=\langle A, AO, Dn, Mn, Fn, Jn, R, Ia, Fa\rangle$,一个集合$set_2=\{map(x)|x\in AO\}$,一个集合$lp=Sc(\mathcal{AD})$和$\mathcal{AD}_code$

Output:
更新后的代码框架$\mathcal{AD}_code$

```
1: 给出一个用于存放已执行过的选择节点的集合al,令al = ∅;
2: for each x ∈ set2 do
3:    if x ∈ AO then
4:       if 现在正在执行的线程为'main' then
5:          向main()中添加x_obj.x()语句;
6:       else
7:          寻找到与现在正在执行的线程名相同的线程类，向该类的run()方法
            中添加x_obj.x()语句;
8:       end if
9:       令set2 = x°,返回第2步;
10:   else if x ∈ Dn并且x° ∈ lp then
11:      if 现在正在执行的线程为'main' then
12:         向main()中添加while()语句;
13:      else
14:         寻找到与现在正在执行的线程名相同的线程类，向该类的run()方法
            中添加while()语句;
15:      end if
16:      令al = al ∪ {x};
17:      令set2 = x°,返回第2步;
18:   else if x ∈ Dn ∩ al then
19:      令set2 = x°,返回第2步;
20:   else if x ∈ Dn并且x° ∉ lp then
21:      if 现在正在执行的线程为'main' then
22:         向main()中添加if()语句;
23:      else
24:         寻找到与现在正在执行的线程名相同的线程类，向该类的run()方法
            中添加if()语句;
25:      end if
26:      令set2 = x°,返回第2步;
27:   else if x ∈ Mn then
28:      令set2 = x°,返回第2步;
29:   else if x ∈ Fn then
30:      while |x°| − 1 ⩾ 1 do
31:         创建子线程，以t_{|x°|−1}命名;
32:         |x°| − −;
33:      end while
34:      将创建的子进程和'main'乱序地放入栈中;
35:      令set2 = x°,返回第2步;
36:   else if x ∈ Jn then
37:      while 栈内不为空 do
38:         栈顶的线程出栈;
39:         if 现在执行的线程不为'main' then
40:            向main()中添加现在执行的线程名.join();
41:         end if
42:         跳出当前for循环体;
43:      end while
44:      令set2 = x°,返回第2步;
45:   else if x ∈ Fa then
46:      跳出当前for循环体;
47:   end if
48: end for
49: 输出AD_code;
```

图4 生成执行逻辑的算法3

Fig.4 Algorithm 3 for generating execution logic

第1 行是定义一个集合*al*，用于存放已经存放的选择节点。第2—48 行是使用深度优先的策略，从初始节点的后置节点开始，依次访问活动图中的每一个节点。其中，在第3—9 行里，当前节点为活动节点时，判断当前执行的是否为主线程，如果是主线程，则在main()方法内调用该节点对应的方法，否则在run()方法内进行相同的方法调用(run()方法属于线程类)；在第10—17 行中，当前节点为选择节点并且该节点的后置节点属于集合*lp*时，生成对应的while语句，并且将该选择节点存放到集合*al*中；在第18—19 行中，当前节点为选择节点并且该节点属于集合*al*时，跳转至该选择节点的后置节点；在第20—26 行中，当前节点为选择节点并且该节点的后置节点不属于集合*lp*时，生成对应的if语句；在第27—28 行里，当前节点为合并节点时，跳转至该合并节点的后置节点；在第29—35 行中，当前节点为分叉节点时，根据当前节点的后置节点数减一的数量创建子线程，并且将子线程和主线程乱序地放入栈中；在第36—44 行里，当前节点为汇合节点时，令栈内位于顶部的线程出栈，如果此时的执行线程不为主线程，则在main()方法内调用对应线程的join()方法(主线程等待该线程执行完成后，再继续执行)，然后再继续让栈内的下一个线程出栈，执行上述操作，直至栈内为空；在第45—48 行里，当前节点为终止节点时，跳出所有循环。第49 行是输出最后更新完的代码框架。

综上，算法1主要是根据*AO*和set_1中的元素，生成相应的代码语句，其时间复杂度为$O(a\times n)$，其中*a*表示*AO*中的节点个数，*n*表示set_1中与活动节点对应的顺序图的个数。算法2是判断活动图中是否有环，并且将环中的活动节点取出，其时间复杂度为$O(d\times r)$，其中*d*表示*Dn*中的节点个数，*r*表示活动图中边的数量，即*R*中的元素个数。算法3是通过遍历活动图中的所有节点，根据不同的情况生成对应的执行逻辑，其时间复杂度为$O(a\times r)$，其中*a*表示*AO*中的节点个数，*r*表示*R*中的元素个数。在软件开发的设计阶段，因为UML图是根据需求进行绘制的，所以节点数量及图的数量都是有限的，开发人员有充分的时间通过这三个算法生成对应的代码框架。

例4：图5中的代码框架就是通过算法1、算法2和算法3生成的，即$AD_code=Gel(AD,set_2,lp,Gcd(AD,set_1))$，其中*AD*为图1(a)中的活动图，$set_1=\{map(c),map(p)\}$，即分别对应图1(b)、图1(c)中的顺序图$SD_1$和$SD_2$，$set_2=\{a\}$则是对应活动图*AD*中初始节点的后置集合，$lp=Sc(AD)=\{b,q\}$表示*AD*中可能会循环发生的活动节点的集合。

```
1  //由算法1生成
2  public class AD { //类定义
3          void a();
4          boolean b();
5          boolean c() {
6              x.c1();
7              y.c5();
8          };
9          void p() {
10             z.p1();
11         };
12         void d();
13         void e();
14         void g();
15         void h();
16         void j();
17         void k();
18 }
19 public Main {
20 //变量声明
21 }
22 //由算法3生成
23 Main::main(String arg[]) {
24     AD.a();
25     para_1=AD.b();
26     while(para_1) {
27         AD.p();
28         para_1=AD.b();
29     }
30     para_2=AD.c();
31     if(para_2) {
32         AD.d();
33     } else {
34         AD.e();//创建线程t1
35         AD.g();//执行main线程
36         t1.join();//AD.h()在线程t1中
37         AD.j();
38     }
39     AD.k();
40     exit(0);
41 }
```

图5 由UML图生成的代码框架

Fig.5 A code framework generated by a UML diagram

4 原型工具(Prototype tool)

基于开源库GoJS(https://gojs.net/latest/index.html)，我们开发了原型工具codeGeneration(下载链接：https://pan.baidu.com/s/1Z1PzWioR2hRcdmFuwxZPMQ，提取码：wgoc)用于实现UML模型到Java代码框架的自动转换。图6是一个从活动图转换成对应代码框架的实例。

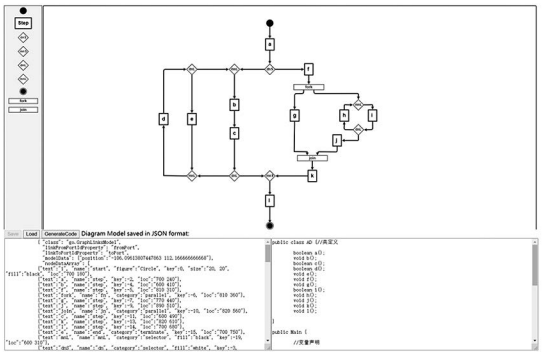


图6 原型工具的转换界面

Fig.6 The transformation interface of prototype tool

5 结论(Conclusion)

基于模型的自动代码生成与一致性验证是实现自动化，以及提高软件可靠性过程中必不可少的一个环节。本文基于UML模型中的活动图和顺序图，给出了一套完整的Java代码自动生

(下转第30页)