****

本科毕业设计(论文)

GRADUATION DESIGN(THESIS)

外文文献翻译

本科生院制

2018年3月

# 控制流语义规范与构造

Ruben Smelik, Arend Rensink and Harmen Kastenberg∗

计算机科学系

荷兰特温特大学

摘要：在本文中，我们提出了一种可视化语言，用于指定编程局域网的控制流语义。我们还提供了一个从CFSL到图形生产系统(GPS)的流图构造的转换工具;也就是说，对于语言L，任何CFSL规范都会产生一个GPS，它能从任何L程序(表示为抽象语法图)中构造相应的流图。规范语言足够丰富，可以捕获complex语言结构，包括所有Java语言。

1. 介绍

没有共同的语言来指定编程语言的语义，与语法的EBNF一致（见[7]）。 相反，编程语言的seymosics通常只用自然语言来描述; 一个很好的例子是Java，见[4]。出于以下几个原因，这里有一个问题：如果语义是非平凡的，例如在Java try语句的控制流（带有finally子句）的情况下，很难给出非歧义的自然语言解释; 并且在没有正式规范的情况下，编译器认证，保存正确性转换（如MDA中的参见[9]）或软件验证都没有相应基础。

\*作者受雇于荷兰NWO资助的GROOVE项目（项目编号612.000.314）。

另一方面，正式规范的一个问题是，它们是出了名的难以阅读，因为它们的语言与普通软件工程师的专业知识和经验相去甚远。既然在这种情况下，我们希望编程语言的设计者是规范的作者，而程序员是它的读者之一，那么消除这个障碍是至关重要的。这需要设计一种编程语言的规范语言，这种语言可以让这些类的用户访问。

在本文中，我们限制自己去控制流量语义，这是命令范式中任何编程语言的语义的重要组成部分，并且我们为其规范提供了一种可视化语言，我们简单地称之为控制流规范语言。CFSL的特点是：

* 它在设计语言中的编程结构方面是模块化的; 也就是说，每个语句类型的控制流语义在单独的控制流程规范图（CFSG）中定义; CFSL规范是由一组CFSG构成的。
* 它建立在语言语法的基础上：每个CFSG都是针对有问题的编程语言，从Backus-Naur Form（BNF）语法规则的右手边（或换句话说，抽象语法树的一个片段）开始创建的。
* 可见，每个CFSG都是一个以抽象语法图（ASG）片段为核心的图形，并且有进一步的结构来声明相应的控制流。
* 其自己的语义（即CFSL）是根据从ASG到流程图（FG）的转换来定义的。这些转换是根据图生产系统而定义的。
* CFSL语义定义本身也是通过图形生成系统制定的，该系统将每个CFSG转换为一组图形生成规则，以实现该CFSG的效果。

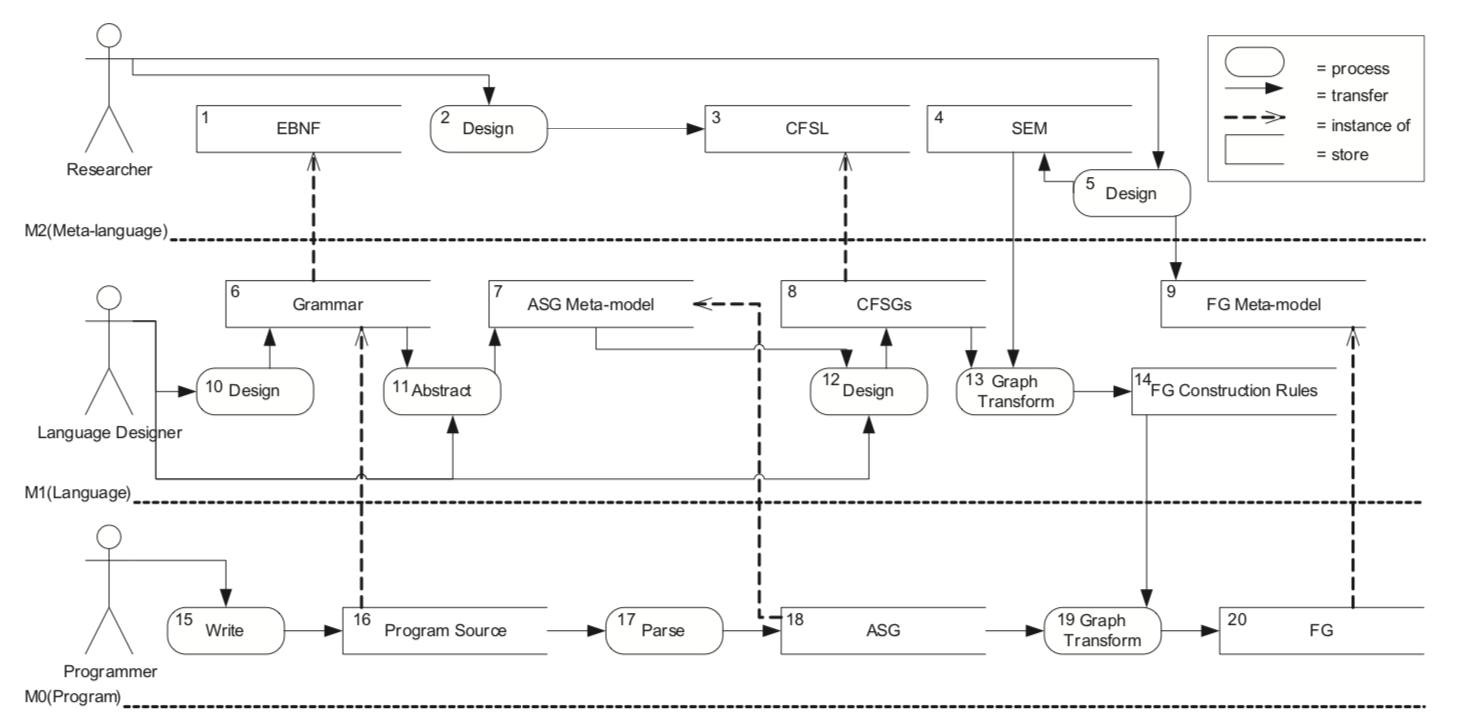
 图1:本研究中涉及的元素的概述以及它们所驻留的层次。

图1显示了我们如何设想在本文中描述的语言在实践中使用的语言，在模型，和参与者它们之间的相互关系。我们辨别出三个层次。 最低级别（M0）是用特定编程语言编写的程序级别（请注意在其他上下文中程序等同于模型）; 中级（M1）是编程语言的级别，最高级（M2）是定义编程语言（部分）的元语言的级别。每个级别都专注于某个角色：分别有定义元语言的研究人员，定义这些元语言的编程语言语法和语义的语言设计人员，以及使用这些特定编程语言编写程序的程序员。

语言设计者在BNF（1）中指定了一种新的编程语言（图1中的6）的语法。这个语法被抽象为ASG的元模型（7）。接下来，他指定了CFSL中语言的控制流语义（3）。通过应用（13）设置的与语言无关的FG元规则（我们命名为SEM）（4），他以一组语言特定的FG构造规则的形式获得了CFSL语义（14）。

程序员用这种新语言写出（15）一些程序（16），符合语法（6）。 该程序被解析（17），产生符合ASG元模型（7）的ASG（18）。 ASG由先前生成的一组FG构造规则（14）转换为符合FG元模型（9）的FG（20）。

我们研究的状况是我们已经详细阐述了必要的概念和转换，但是CFSL的具体图形语法尚未针对可读性进行优化。

在剩下的部分中，我们介绍了方法的要点;对于完整的详细讨论，请参阅本文[14]。第2节描述了CFSL(3)和第3节定义了CFSL语义的转换(14)。第4节简要讨论了CFSL语义(4)的构建。最后，第5节给出了结论并讨论了相关的工作。

1. 控制流程规范语言

在介绍中我们提到了CFSL的几个重要特征; CFSL是一种基于编程语言语法的模块化、可视化、基于图形的规范语言。

在本文中，一个图是由一组有限的节点和一组有限的标记的有向的二进制边组成。节点没有按照定义进行标记，但可以具有指向自身的传出边缘。这些边称为节点的自边。在图形上，节点表示为黑色矩形，边缘表示为黑色箭头。自我边可以表示为节点的标签（在矩形内部图形化地描绘），或者表示为具有相同开始和结束节点的箭头。

每个CFSG符合CFSL元模型，即，当将CFSL元模型表示为图（称为类型图）时，存在从每个CFSG的元素到类型图的元素的唯一映射。这个元模型由编程语言独立的图元（即节点和边）组成，用于表示控制流。出于多种原因，我们选择图形作为我们规范语言的基础。我们更喜欢一个比树结构更具表现力的结构，就像存在于解析的合成树中一样，或者不太明确地存在于（E）BNF语法规则中。此外，通过使用图形，我们可以将图形变换应用于CFSG，我们将在第4节中看到。

2.1. 抽象语法

如上所述，我们在语言语法上使用CFSGs，这是在它(上下文无关的)语法中编码的。为使非终端机具有更强的先进性，我们采用了以非终端机命名的BNF规则;例如,

WhileStatement ::= <WHILE> <LPAR>

condition:Expression

<RPAR> body:Statement

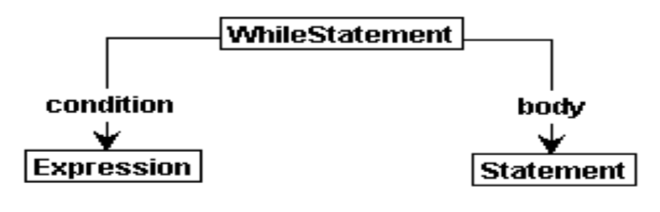
 由此，我们生成了一个抽象的语法图(ASG)，其中省略了纯粹的句法细节。这些struc表示的是图形，而不是树，因为在某些特定的情况下(例如，在Java中，一个标记的break-statement)元素需要被共享。在ASGs中，父元素和子元素之间的树关系被保留，为了一致性，由la- belled child的边。图2显示了与bnf规则对应的ASG。

图2：Java while语句的A S G

根据现有的一个经验法则，我们可以说，对于编程语言的抽象语法中的每个非终端，我们都必须设计一个CFSG。

2.2. CFSL元模型

控制流信息描述了执行程序的单个原子指令的顺序。在本文中，我们区分了三种类型的控制流：顺序流，有条件流和破坏性流。

我们将通过CFSL元模型（图4）中涉及的元素讨论每种控制流。首先，我们将描述构成元模型基础的一些元素。

标记为AbstractSyntaxElement的节点可以看作是一个通用节点，代表ASG中所有可以在执行期间传送控制权的节点。为特定语言结构指定CFSG时，KeyElement边将用作该AbstractSyntaxEle-节点的自边。标有入口和出口的边分别标识AbstractSyntax-Element的实际执行开始或结束的点。退出节点既可以是相关的AbstractSyntaxElement之一，也可以是新引入的节点（图4中最左边的未标记节点）。

顺序流指的是控制流的类型，其中语句以它们在程序中出现的顺序执行。随后执行的语句都是通过被标记的流的边缘进行的。

当执行顺序基于某个表达式的值时，就存在条件流。对于每个值，表达式可以计算到我们引入一个分支节点(通过分支边缘连接到KeyElement)，它指的是带有条件边缘的原始表达式，以及具有分支边缘的相应值。branchDe- fault-edge表示在没有其他分支应用时所使用的分支。

最后，我们认为如果声明是突然终止的，声明会引入破坏性流程。这将由一个Abort节点来模拟，然后控制器通过abort-edge流向该Abort节点。该Abort节点具有指向导致中断的AbstractSyntaxElement的原因边缘。在某些破坏性流动情况下，尚不清楚继续执行该计划的声明。这是通过引入abortFrom和resumeAbort-边来建模的。有关如何指定的详细信息，请参阅[14]。元模型伴随着一系列不同元素组合的限制。例如，一个CFSG最多可以有一个出口边，而元模型最多可以有两个出口边。附加约束的完整列表可以在[14]中找到。

2.3. 示例CFSG设计

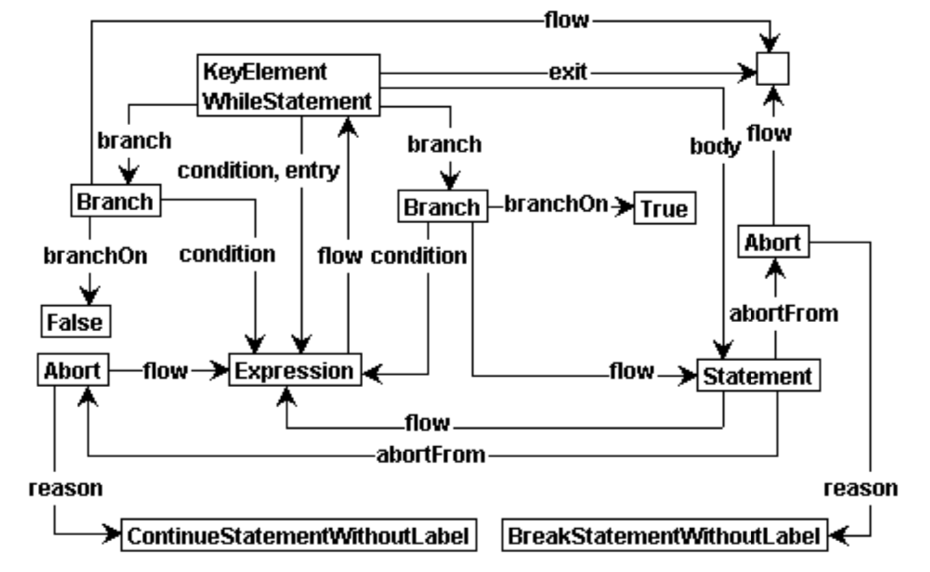
 作为一个例子，我们演示了如何为Java while语句设计CFSG（图3）。

图3：Java while语句的CFSG

我们从图2所示的ASG开始。我们将WhileStatement指定为KeyElement，将其条件表达式作为入口点（WhileStatement的执行从评估条件开始），并引入未标记的节点作为它的出口。我们指定顺序流：从Expression到WhileStatement的流边和从Block（隐含地从它的出口）到表达式（隐式地到它的入口）的流边。第一个流程边缘表示在评估条件之后，控制权转移到WhileStatement，在此处决定是否继续迭代主体。第二个流程边缘表示身体执行后，条件将被重新评估。

条件流由来自WhileStatement节点的两个传出分支边指定：在false上，通过将控制转移到出口节点来完成WhileStatement（通常），如果为true，则为另一次迭代输入正文。

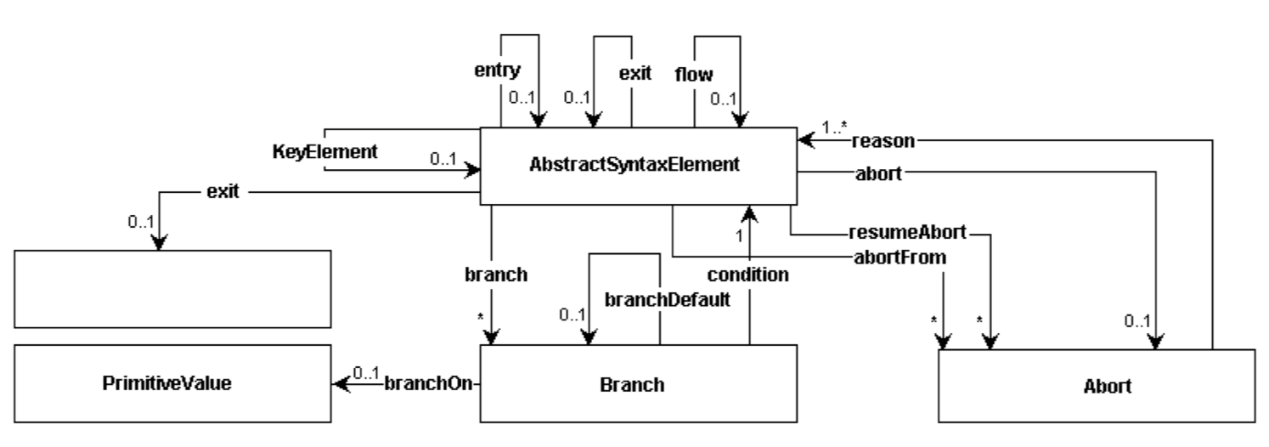
当程序到达例如中断或继续语句时，WhileStatement的主体的执行可能会中断。这由具有源自该机构的传入abortFrom-边缘的两个Abort-节点指定。 每个Abort节点都有一个外出原因边缘，通过该边缘可以跟踪导致破坏性流的AbstractSyntaxElement。图3指定当达到break-statement时，控制流向WhileState的出口。相反，当执行continue语句时，WhileStatement的条件将被重新评估。这里不讨论中断和继续的标签变体。

图4：CFSL元模型

3. CFSL semantics

CFSL的语义是通过从抽象语法图到相应流程图的映射来定义的。在本节中，我们通过一个图形生成系统将该映射形式化，该系统将任何抽象语法图形转换为流程图形。

3.1.流程图

流程图（FG）是可视化程序中控制流程的广为人知的方式（例如[3]）。在流程图中，语句显示为节点，箭头或图边显示控制在各个语句之间如何传输。 有许多可能的建模级别，从详细到抽象：例如，可以从分支中的实际值中抽象出来，并且仅表示存在多个结果的事实。我们使用一个非常详细的模型，主要由图4中已经显示的元素组成，其中控制流程的所有方面都忠实地表示出来，以便实际上可以基于FG表示对程序进行精确模拟。

为了从ASG构建FGs，我们使用图形转换，从本质上丰富了现有的结构; 换句话说，对于我们来说，流程图是用控制流信息进行分解的抽象语法图。

3.2.图形转换

图转换是一种系统的，基于规则的转换技术。它具有坚实的研究基础[12]，并在计算机科学的许多领域得到应用。

图生成系统（GPS）是一组图生产规则，每个规则都可以将源图转换为称为目标图的新图。该规则指定它应用的条件以及它对源图所做的更改。从技术上讲，图形生成规则1由两部分重叠的图形组成，左侧L和右侧R以及一组负向应用条件N，这些条件也是（连接）图与L部分重叠。为了应用该规则，将左侧L与源图G（的一部分）相匹配，之后将G中的L的图像替换为R的副本;但匹配只有在它不能扩展到N中的任何图表时才有效。换句话说，否定应用条件中的结构在源图中是被禁止的。在我们对本文中使用的规则（从GROOVE工具中获取）的视觉呈现中，我们将所有这些元素组合到一个图中，是由四种元素组成：

* 读者：存在于L和R中的元素。它们必须出现在L的源图中才能匹配并保存在目标图中;
* 擦除器：元素存在于L中，但不存在于R中。它们在源图中相匹配但未保留在目标图中，即它们被删除。
* 创建者：元素在L中不存在，但在R中存在。它们被引入目标图形。
* 禁运：L中不存在的元素，但存在于N中的负面应用条件之一中。

为了从视觉上区分这四种类型，每个元素都有独特的颜色和形式，如图5所示：读者是黑色的，橡皮擦是蓝色虚线（黑色和白色演示文稿中为深灰色）创建者为绿色（浅灰色为黑色和白色演示文稿）和禁运是红色虚线（黑色和白色演示文稿中的深灰色）。



(a) 读者 (b) 擦除器 (c) 创作者 (d) 禁运

图5:图形生成规则元素。

应用GPS（而不是单一规则）的效果是通过应用单个规则而得到的，只要有适用的规则; 当不能应用更多规则时，转换会终止并返回结果图。这个过程中的两个重要方面是：

* 在转换过程中的任何时候，几条规则可能适用于中间图，并且它们的应用不会立即产生相同的结果。然而，为CFSL生成的GPSs被设计为融合的，这意味着规则的应用顺序不会影响最终的图形。（我们并没有正式证明合并;但是，我们认为可以证明这一点，因为我们的规则基本上只是增加了结构，并没有将其删除，并且规则没有使用禁运。）
* 转换过程可能无法终止。再次，我们将CFSL语义设置为终止; 在这种情况下，由于我们基本上实施了对非周期ASG的两遍遍历这一事实。（同样，我们还没有正式证明这一点。）

为了执行图形转换，我们使用GROOVE。

3.3. 流程图构造方法

流程图构建规则通过转换ASG来构建FG。在这个转换过程中，控制流信息元素被引入ASG。为了一致性，我们在FG中使用与CFSG中指定控制流程相同的元素（参见图4）。我们的FG建筑方法包括我们所做的一系列设计选择：

1. 对于每种类型的抽象语法元素，我们设计一个（首选）或几个流程图构建规则，以引入必要的控制流元素。
2. 流程图构建过程自顶向下运行，从正在构建的流程图的根节点开始，到结束于原始语句的级别。
3. 对于图中的每个流元素，我们创建它的入口和出口（关于控制流）。最初，我们统一为这些入口和出口提供辅助流量连接器; 然后这些在建设过程中合并。
4. 我们使用自下而上的解决方案流程解决颠覆性流程问题。

我们在下面讨论这些选择。

Ad 2.在我们的案例中，流程图的构建是一个顶级的，

这意味着我们从ASG节点开始，ASG节点被定义为正在建设中的FG的根，并且继续沿着ASG子节点。更具体地说，我们首先将根节点标记为符合FG构建的条件，使用特殊的自我边缘标记构建，并自上而下传递这些标记。 FG的构建规则是在构建边缘上匹配。

Ad 3.所有带有控制流语义的ASG元素都被认为是FlowElement，并且有一个入口和出口。特定流量要素的控制流程必须考虑到其子女在ASG中的执行情况。然而，由于是自上而下的建设，当父母流动元素正在建设中时，其子女的控制流程尚未确定。这引入了一个问题：未知子流元素的执行开始于哪里（即它的输入未知）。我们的解决方案是为每个流量元素的入口和出口边缘提供控制流连接器节点（FlowConnector）作为（初始）目标。父节点流量元素可以将他们的流量边缘连接到这些连接器节点，并且子节点可以引入他们的内部控制流量，从其入口和出口流量连接器开始和结束。我们的策略是统一引入入口和出口流量连接器，然后通过将节点与其他流量元素合并，去除多余的流量连接器。

图6显示了Java while语句的流程图构造规则示例。在此规则中，我们看到使用构建标记，合并FlowConnector以及引入顺序（流）和条件分支（Branch）控制流程。

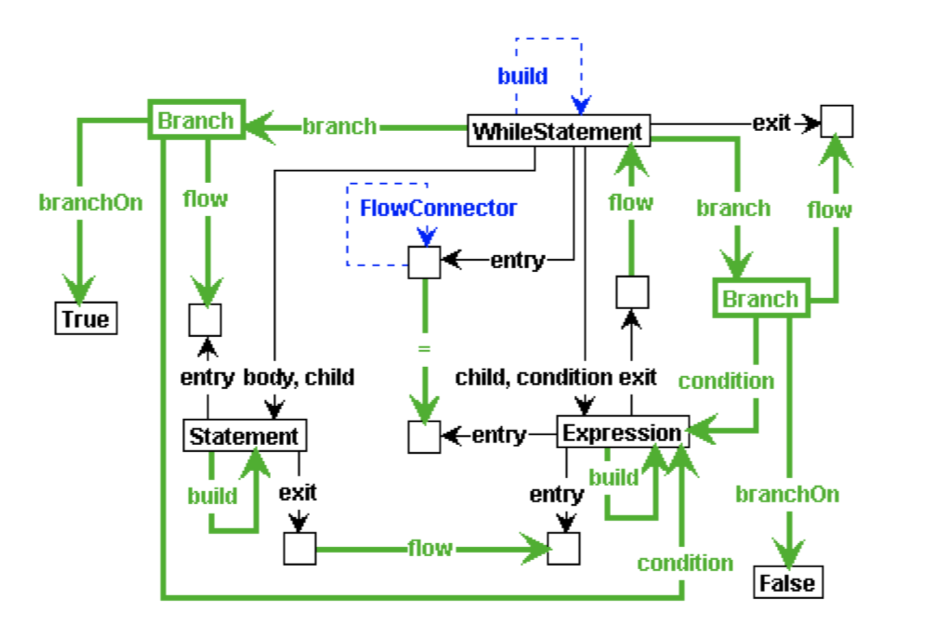
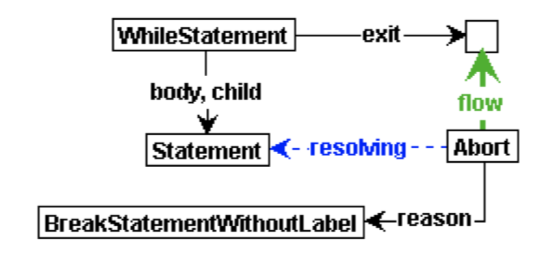


图6:Java while-statement(由SEM生成)的FG构造规则

Ad 4.破坏性流动的构建涉及更多。这里的主要问题是规则必须检查突发完成状态的FG上下文，以确定所产生（发布）流的目标语句。哪些类型的流程元素可以作为目标取决于突发完成语句的类型。我们指的是找到正确的目标流程元素并将突破性流程引入此陈述作为突发完成解决方案的过程。我们使用自下而上的解决方法。首先，当一个突发完成语句被标记为构造时，相应的流程图构造规则将一个中止边缘引入中止节点，并带有一个解析标记，指示应该解析一个情景。接下来，解析标记在语法树中向上传播，直到到达解析流元素。最后，与突发完成声明和重新解决目标相匹配的另一个构造规则消除了标记并完成了破坏性流程。图7显示了如何解决while语句正文中的break-statement（与图3相比）。

图7：由于断言（由SEM生成）而中止while语句的FG构造规则

在某些情况下，在突然完成解决后，它会重新引入（恢复）。 特别是，这中现象在试验陈述的最后部分发生，我们将在下面看到。

**3.4.** Java**流程图构建示例**

我们在清单1所示的Java代码片段中说明了FG构建过程。清单中有一个while语句，其中包含try-statement与finally-part。 try语句的主体包含一个break语句。 我们将看到，在执行break语句时，首先将控制转移到finally语句，并在执行后立即执行突发完成，从而终止包含while语句的执行。

while (true) try {

... break; ...

} finally { ...

}

清单1：示例Java代码片段

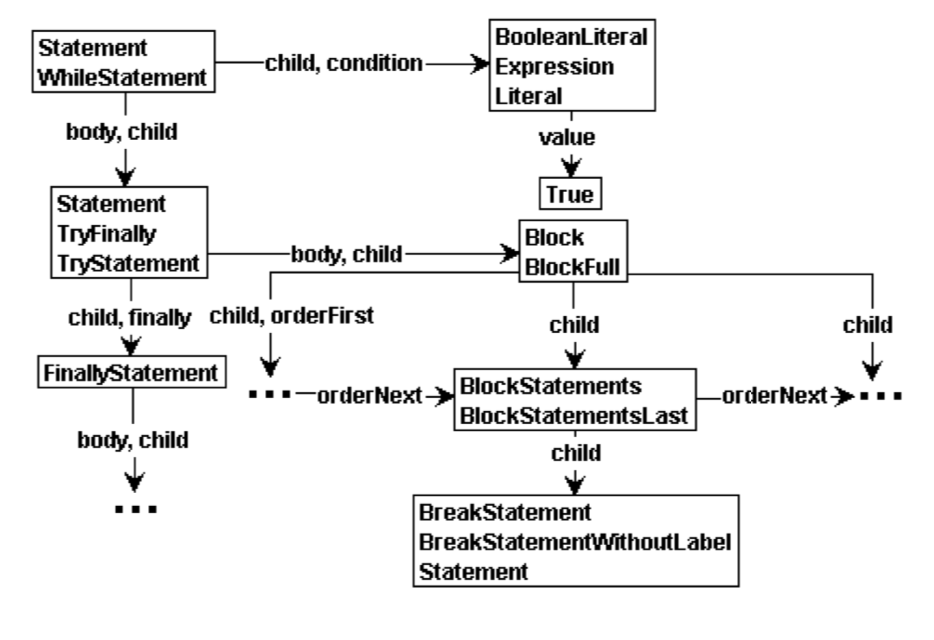
 与此代码段对应的ASG如图8所示。图中的点对应于列表中的点，即省略的上下文。 当我们将Java FG构建规则应用于此ASG时，将执行构建过程。 首先，创建入口和出口FlowConnector; 然后自上而下的流程结构开始，在while语句开始。到达break-statement后，突然完成会自下而上传播。它被解析为导致finally语句，然后重新结束while语句。结果FG如图9所示。

图8：清单1的ASG。

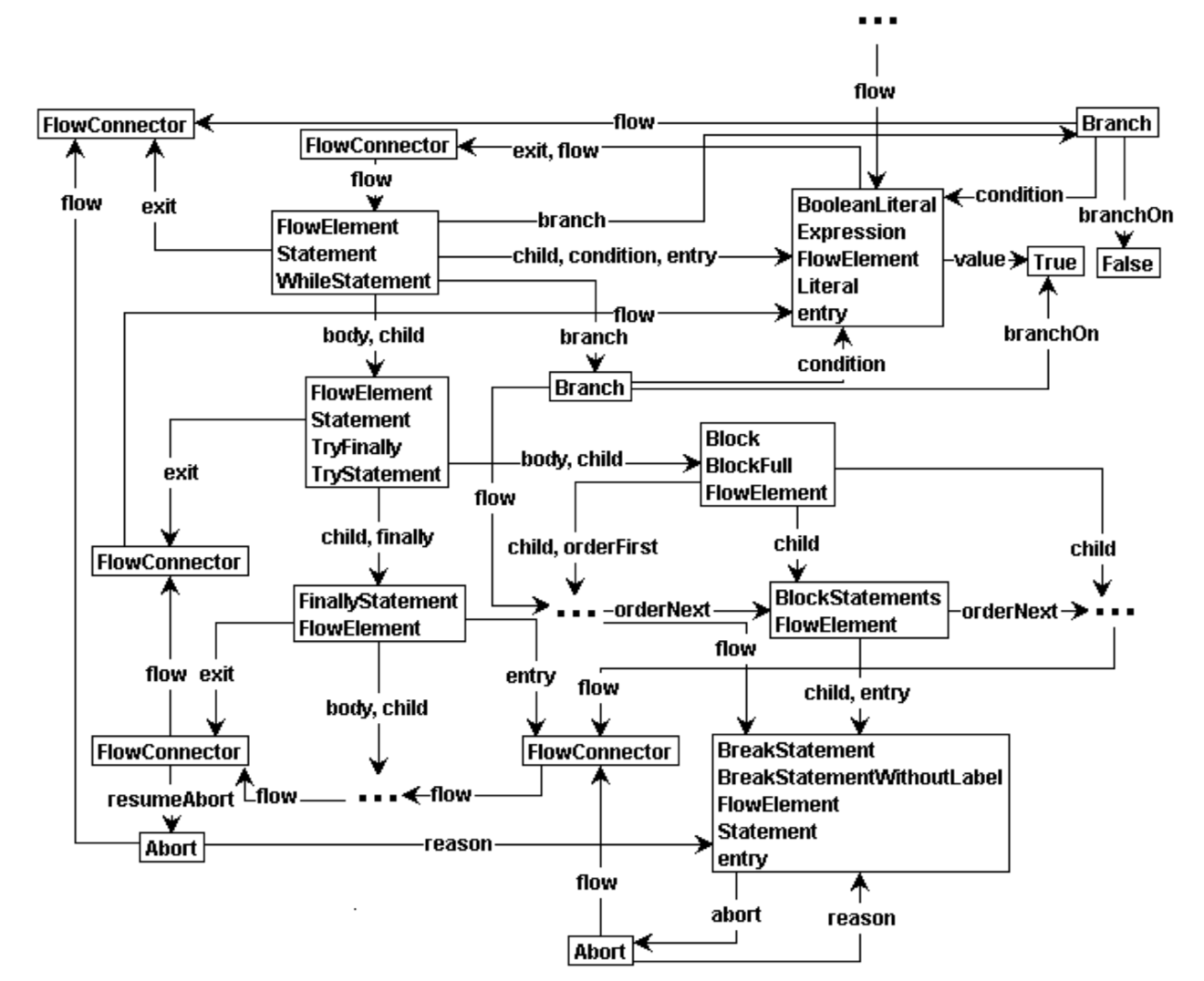


图9：清单1的构造FG

4. 构建CFSL语义

我们现在一方面拥有CFSG，为语句类型指定了控制流语义;另一方面，FG构建规则实际上为ASG生成流程图。后者实际上是作为前者的语义。为了定义语义，我们再次使用图转换; 也就是说，我们在这里（简要地）描述一个GPS，它可以将任何CFSL规范（即CFSG集合）转换为相应的一组FG规则。 我们使用名称SEM来指代这个GPS。

SEM中的规则可以被认为是元规则，因为它们是创建图形生产规则的图形生成规则。显然，SEM是独立的编程语言：它可以应用于任意CFSL规范，SEM规则仅与CFSL元模型（图4）中的元素相匹配，仅引入FG元素（包括F G构建所需的辅助元素）。

与前一部分讨论的FG建设案例不同，SEM并不总是将CFSG映射为单一规则：相反，单个CFSG可能会为FG建设带来多重规则。我们已经看到了一个例子：图3的CFSG产生图6和图7等规则。换句话说，SEM不融合：相反，每个规则应用序列都被考虑，而转换过程的结果是不能进一步转换的目标图的集合。

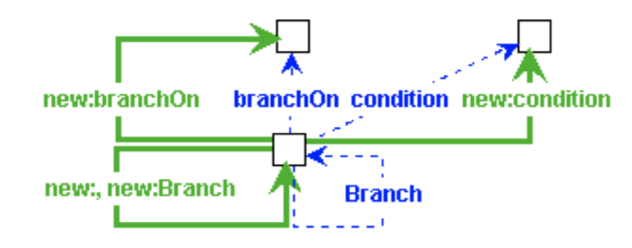
例如，SEM中的一个规则如图10所示。该规则代表构建条件流的FG构建规则的一个步骤，如第2节所述。请注意使用前缀“new”在创作者中：这反映了目标规则中节点和边的角色指示。

图10：SEM元规则示例。

1. 结论

我们认为CFSL在实现我们的目标方面是成功的，我们在介绍中提出：它是一种用模块化和可视化方式指定控制流语义的语言，作为语法定义的扩展。我们通过提供完整的Java控制流语义规范来验证语言，包括异常处理和其他类型的突发终止; 见[14]。 在这样做的时候，我们也证明了图形变换在这种情况下的有用性，作为定义CFSL语义以及构建实际流程图的工具。

也就是说，在我们可以声称拥有我们真正期望的语言设计者可以使用的语言之前，有很多要改进的地方：

* 我们应该证明任何由CFSG构建的GPS的合流都可以实现目标。
* 我们应该可以从一个任意的EBNF语法开始，而不是BNF。
* 正如引言中所提到的，CFSL的具体合成还有很多不足之处：图3中的规范图虽然没有多余的信息，但并不是一个可读的格式。通过在CFSL中引入一个属性类型的符号，可以获得很多; 例如，分支的边缘和原因边缘可以变成属性。另一个很好的可能性是使用hyperedges而不是Branch和Abort-nodes。例如，图11将表示与图3中相同的CFSG，但使用更丰富的（临时）可视语法。
* 作为视觉语言的替代方法，引入文本符号可能是一个好主意，以便与基础BNF语法的关联更加明显。

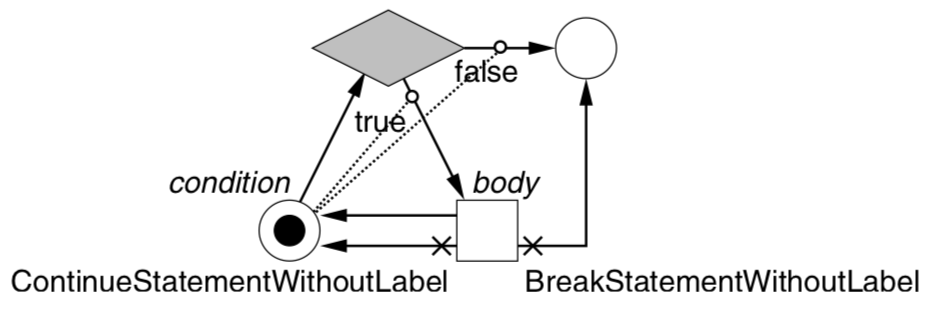
声明

图11：尽管替代具体语法中的控制流规范

5.1. 相关工作

这里报道的工作最初来自[8]，其中我们为（定制）面向对象语言提出了一个完整的基于图转换的语义。流程图构造是该语义的一部分;在那篇论文中，我们为FG建造设计了一个专用的GPS。使用CFSL我们可以更快速，更紧凑和更直观地指定GPS。

另一个研究流程图生成的环境是三图文法（TGGs）[13]。特别适用于模型转换（例如[5]），TGG特别适用于不仅仅是模型的推理，例如（在我们的情况下）ASG和FG，还涉及它们的关系。事实上，三图文法可以被认为是FG构建过程的替代机制。目前尚不清楚是否可以构建与CFSL相似的模拟以便从TGG的能力中受益，事实上这还没有成为研究的主题。

与我们的工作密切相关的是蒙太奇项目，例如[1]中所描述的。这为帮助语言设计者指定编程语言的语法和（静态和动态）语义提供了一个框架。作者与我们分享了许多设计原则;它们也从一组规范中提出了一个完整的编程语言规范，并且在我们的例子中，它们的规范基于语言语法。蒙太奇包含装饰抽象语法树的局部有限状态机器（这些可以在以后连接起来形成流程图）。一个重要的区别在于，在基于文本的操作规则中定义了监视器中的动态语义。这可能导致复杂的非视觉动作规则，例如，对于破坏性流动来说，而在CFSL中这是更直观的结构。

参考文献

[1] M.Anlauff,P.W.Kutter,andA.Pierantonio.Enhancedcon- trol flow graphs in montages. In Proceedings of the Third International Andrei Ershov Memorial Conference on Per- spectives of System Informatics (PSI’99), volume 1755 of LNCS, pages 40–53. Springer, 2000.

[2] H. Ehrig, R. Heckel, M. Korff, M. Lo ̈we, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation, Part II: Single pushout approach and com- parison with double pushout approach. In Rozenberg [12], pages 247–312.

[3] N. E. Fenton and S. L. Pfleeger. Software Metrics: A Rigor- ous & Practical Approach. International Thomsen Publish- ing Inc., 2nd edition, 1997.

[4] J.Gosling,B.Joy,G.Steele,andG.Bracha.JavaLanguage Specification. The Java Series. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2005.

[5] L. Grunske, L. Geiger, and M. Lawley. A graphical specifi- cation of model transformations with triple graph grammars. In A. Hartman and D. Kreische, editors, Model Driven Acri- tecture — Foundations and Applications (ECMDA), volume 3748 of LNCS, pages 284–298. Springer, 2005.

[6] A.Habel,R.Heckel,andG.Taentzer.Graphgrammarswith negative application conditions. Fundamenta Informaticae, 26(3-4):287–313, 1996.

[7] International Organization for Standardization. ISO/IEC 14977: Extended BNF. International Organisation for Stan- dardization, 1996.

[8] H. Kastenberg, A. Kleppe, and A. Rensink. En- gineering object-oriented semantics using graph transformations. Technical report, University of Twente, 2005. Pre-final version available at http://www.cs.utwente.nl/∼rensink/papers/taal-draft.pdf.

[9] Object Management Group. Model Driven Architecture, 2006. http://www.omg.org/mda/.

[10] A. Rensink. The GROOVE Simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Bo ̈hlen, editors, Proceedings of the Second International Workshop on the Applications of Graph Transformations with Industrial Rel- evance (AGTIVE’03), volume 3062 of LNCS, pages 479– 485. Springer, 2004.

[11] A. Rensink and H. Kastenberg. Graphs for object-oriented verification (GROOVE), 2006. http://groove.sf.net/.

[12] G. Rozenberg, editor. Handbook of Graph Grammars and Computing by Graph Transformation, volume I: Founda- tions. World Scientific, 1997.

[13] A. Schu ̈rr. Specification of graph translators with triple graph grammars. In Proceedings of the 20th Interna- tional Workshop on Graph-Theoretic Concepts in Computer Science (WG’94), volume 903 of LNCS, pages 151–163. Springer, 1994.

[14] R. M. Smelik. Specification and construction of control flow semantics. Master’s thesis, University of Twente, 2006. http://www.cs.utwente.nl/∼rensink/papers/Smelik.pdf.