Engineering Object-Oriented Semantics Using Graph Transformations

Harmen Kastenberg* Anneke Kleppe†
Arend Rensink

Department of Computer Science, University of Twente
{h.kastenberg,kleppeag,rensink}@cs.utwente.nl

March 24, 2006

*Funded by the Dutch NWO project GROOVE (Grant nr. 612.000.314)
†Funded by the Dutch NWO project GRASLAND (Grant nr. 612.063.408)

Contents

Abstract

In this paper we describe the application of the theory of graph transformations to the practise of language design. We have defined the semantics of a small but realistic object-oriented language (called TAAL) by mapping the language constructs to graphs and their operational semantics to graph transformation rules. In the process we establish a mapping between UML models and graphs.

TAAL was developed for the purpose of this paper, as an extensive case study in engineering object-oriented language semantics using graph transformation. It incorporates the basic aspects of many commonly used object-oriented programming languages: apart from essential imperative programming constructs, it includes inheritance, object creation and method overriding. The language specification is based on a number of meta-models written in UML. Both the static and dynamic semantics are defined using graph rewriting rules.

In the course of the case study, we have built an Eclipse plug-in that automatically transforms arbitrary TAAL programs into graphs, in a graph format readable by another tool. This second tool is called Groove, and it is able to execute graph transformations. By combining both tools we are able to visually simulate the execution of any TAAL program.

1 Introduction

A widely recognized proposal for combating the maintenance and evolution problems faced in software engineering is the 'model driven approach', brought to the worlds attention by the OMGs Model Driven Architecture (MDA) framework [28, 22, 17, 26]. MDA builds upon and greatly extends UML; its cornerstones are meta-modelling and model transformation.

In an MDA model transformation, a source model written in some language (the source language) is translated into a target model written in the same or another language (the target language). Preferably, the intended meaning of the source model remains in the target model. This means that the semantics of the source language must be mapped onto the semantics of the target language. If both semantics are expressed in the same manner, this mapping can be made far more easily.

A commonly used mechanism to define semantics for (object-oriented) languages could facilitate the adaptation of MDA as a software building technique. Our research explores the usefulness of graphs and graph transformations as common mechanism for defining language semantics. In our view using graph transformations for this purpose is an obvious choice. The concept of object as an independent unit that may, or may not have links to other units, already suggests that the state of an object-oriented program could easily be represented as a graph, where objects form the nodes, and the links between the objects from the edges. State changes are naturally represented as graph transformations.

In this report we show the feasibility of this approach by defining the semantics of a small language called TAAL (short for 'The Arend and Anneke Language', and also the Dutch word for language). In Sect. 2 our approach is explained in detail. Sect. 3, Sect. 4 and Sect. 5 explain the concrete and abstract syntax of TAAL, respectively. Sect. 6 explains the construction of a control flow graph from a TAAL program. In Sect. 7 the simulation of a TAAL program is explained. Sect. 8 gives some noteworthy details on the tools that support TAAL. Finally, Sect. 9 gives our conclusions and references to related work.

In this report several UML diagrams are shown. In these diagrams, whenever an association is directed, and the non-navigable side (i.e. the side without arrowhead) is an aggregate, the multiplicity on the aggregate side is considered to be irrelevant. Sometimes multiplicity 0..1 is shown in the diagrams, sometimes not. This was not an explicit choice, but was caused by the inadequacies of the UML diagramming tool used.

2 Approach

In this work we model object-oriented programs as graphs and specify their semantics using graph grammars. The approach we have taken is to define a series of transformations that will transform any TAAL program into a simulation of its execution. The transformations are depicted in Figure 1. The figure also shows how these transformations are currently implemented: on the left hand side this is done on the basis of an Eclipse plug-in [18, 31, 10] whereas on the right hand side transformations are carried out in the Groove tool set [30], which supports the execution and simulation of graph transformations. Implementation details are given in Sect. 8.

2.1 The Transformations

The first two transformations are similar to the first steps in a compiler [3]. The text is parsed and transformed into a Concrete Syntax Tree (CST) or parse tree. Next, a static analysis is performed that transforms the CST into a true graph, i.e. a structure where a node can have more than one incoming edge: the Abstract Syntax Graph (ASG). For instance, type references are resolved and changed into direct references to the node that represents the type. This ASG contains some aspects that are not present in traditional graphs, like nodes that represent sets or ordered lists. Therefore, this form of ASG is called Rich Abstract Syntax Graph (Rich ASG). Details on both transformations can be found in Sect. 3 and Sect. 4.

The Rich ASG is transformed into an XML based format that fits as input to the Groove tool set. The input format for Groove will be called Flat Abstract Syntax Graph (Flat ASG) in the following. The transformation from Rich ASG to Flat ASG is also implemented in the TAAL Eclipse plug-in. Details on this transformation can be found in section 4.3.

In the next step, called flow graph construction, a set of graph production rules is applied to the Flat ASG. These rules add edges to the Flat ASG that represent the flow of control in the program. For instance, to each statement an edge is added that represents the next element to be executed. The result of this transformation is called the Program Graph (PG). A PG thus consists of two interconnected parts: the Flat ASG and the Control Flow Graph (CFG). This step is implemented in Groove. Details can be found in Sect. 6.

Finally, the PG is entered into the Groove Simulator where, using a second graph transformation system called simulation, the program is "executed", resulting in a sequence of Execution Graphs (EGs). During the simulation extra nodes are created that represent instances of the types defined in the program. These nodes are called Value nodes. For each attribute defined in its type, the Value node will have outgoing edges to other nodes representing a place holder for the value of that attribute. The sub-graph containing the Values is called the Value Graph (VG). The VG represents the objects with their instance variables and data values. In compiler terminology, the VG corresponds to the heap. The VG has outgoing edges to the Flat ASG, reflecting type information.

To represent the execution of an operation or constructor, the execution graph contains so called Frame nodes. Each Frame node has outgoing edges to nodes representing local variables, and an outgoing edge to the next element in the PG to be executed. The sub-graph consisting of Frame nodes and their edges is called the Frame Graph (FG). The frame graph directs the execution of the program. In compiler terminology, the frame graph models the stack during program execution. There can be outgoing edges to the program and value graphs. These edges represent the link with the executable statement in the program from which the frame node has been derived. Details on the simulation step are to be found in Sect. 7.

Together, the VG and FG form an EG. The horizontal line in Fig. 2.1 indicates that graph(s) before the simulation, and during and after the simulation are on a completely different level. The PG is a single graph representing the TAAL program, whereas during simulation the dynamics of the program execution are represented by a series of EGs, each of which represents the system state at a certain point in time. Note that after the simulation has finished the FG has stops to exist, but the VG remains. The final VG represents the effect of the execution of the program.

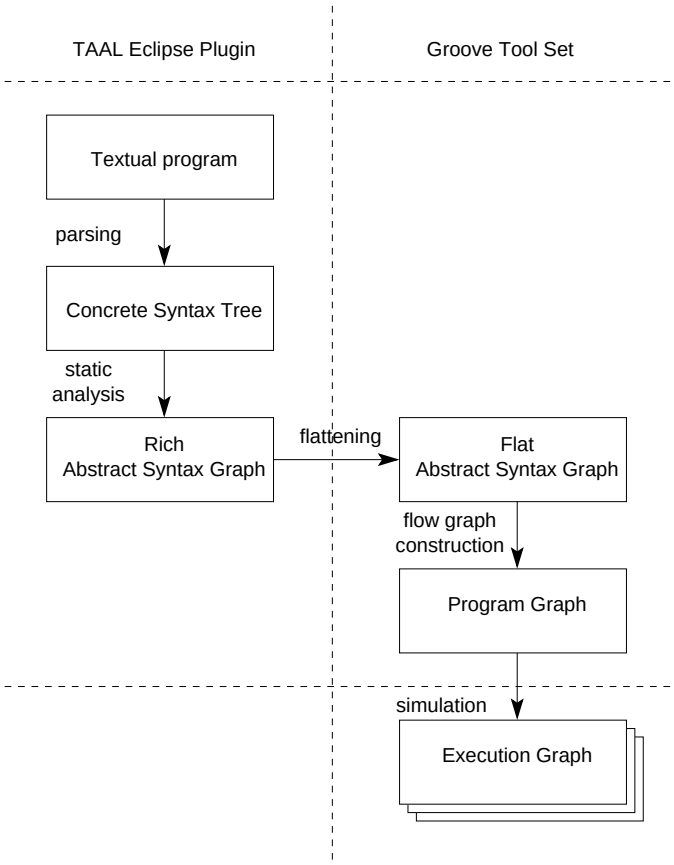


Figure 2.1: Overview of the transformations from program to simulations.

2.2 Definitions

Above we have introduced a large number of concepts that will be used throughout this report. Here we give definitions of each concept.

Definition 2.1 (Concrete Syntax Tree) *A* Concrete Syntax Tree *is a graph representing a TAAL program that contains the results of the parsing phase, also known as parse tree. (See Sect. 3)*

Definition 2.2 (Rich Abstract Syntax Graph) *A* Rich Abstract Syntax Graph *is a graph representing a TAAL program in which each node represents a concept in the TAAL language. (See Sect. 4)*

Definition 2.3 (Flat Abstract Syntax Graph) *A* Flat Abstract Syntax Graph *is a Rich Abstract Syntax Graph in an XML format that can be understood by the Groove tool set. (See Sect. 5)*

Definition 2.4 (Control Flow Graph) *A* Control Flow Graph *is a graph representing the intended flow of control of a TAAL program. (See Sect. 6)*

Definition 2.5 (Program Graph) *A* Program Graph *is an extension of the Rich Abstract Syntax Graph, that also contains the Control Flow Graph for the represented TAAL program. (See Sect. 6)*

Definition 2.6 (Value Graph) *A* Value Graph *is a graph representing a state of an executing (or executed) TAAL program, a model of the program heap. (See Sect. 7)*

Definition 2.7 (Frame Graph) *A* Frame Graph *is a graph representing the state of the executing elements of a TAAL program, a model of the program stack. (See Sect. 7)*

Definition 2.8 (Execution Graph) *An* Execution Graph *is a graph representing an executing (or executed) TAAL program, which consists of three interconnected sub-graphs: the Program Graph, the Value Graph and the Frame Graph. (See Sect. 7)*

2.3 How the Transformations Determine the Semantics

Each of the transformations in Fig. 2.1 determines part of the semantics of the program. The static analysis determines whether the program is statically correct according to certain rules. For instance, each reference to a type must reference a type that is defined somewhere in the program, and each variable must have a type. The static analysis produces the "compile time" errors, if any, in the program.

The flattening determines how a TAAL program can be represented as a traditional graph. For instance, the name of each element in the program is represented as an edge to and from the same node, labelled with the name of the element.

The flow graph construction determines how the control elements in the TAAL program, like while-loops and if-statements, are represented in the program graph. The control language constructs are present in the program from the start. This step determines their exact meaning.

The simulation determines how instances, and how the execution of operations are represented. In other words, the last transformation determines exactly what happens when the program is executed. Note that each execution step in the simulation, i.e. each state change, is given by a graph transformation rule. Note also that the division between the transformations is arbitrary. For instance, the flow graph construction could also have been part of the static analysis.

3 Concrete Syntax Trees

The mini language TAAL incorporates the basic aspects of many commonly used object-oriented programming languages. For instance, the notions of class, attribute, operation, and inheritance are all present. TAAL does not support more complex constructs like packages, or threads, but the language is easily extensible with such notions. The intuition behind the semantics of the language is that a TAAL program has the same meaning as a corresponding Java program.

Most keywords in TAAL are the same as the keywords in Java (e.g. class, extends, super). The keyword self is used to indicate the current object, instead of the Java keyword this. Operations may have local variables, these are listed after the keyword locals. Listing 1 contains an example of a TAAL program. The BNF definition of the concrete syntax of TAAL can be found in Appendix A.

```
program vase

  { new Vase().changeFlower(new Rose()) }

class Vase
  myFlower: Flower;
  height: Integer := 20;

  changeFlower(newFlower: Flower)
  locals tempVase: Vase := new Vase();
  {
    tempVase.myFlower := self.myFlower;
    while newFlower.length.largerThan( height.plus(15) ) do
      newFlower.cut();
    endwhile
    self.myFlower := newFlower;
  }

  getColor(): String {
    return myFlower.getColor();
  }
endclass

class Flower
  color: String := 'yellow';
  length: Integer := 50;

  getColor(): String {
    return color;
  }

  cut() {
    length := 35;
  }
endclass

class Rose extends Flower
  myColor: String := 'red';

  getColor(): String {
    color := myColor;
    return super.getColor();
  }

  cut() {
```

```
    length := length.minus(5);
  }
endclass

endprogram
```

Listing 1: An example TAAL program.

3.1 Meta-model

The concrete syntax of TAAL is described in a UML metamodel. The mapping of the BNF rules to the meta-classes is simple. The names of the non-terminals in the BNF rules directly correspond to the names of the meta-classes. For instance, the rule for ParsedProgram is:

This rule states that a ParsedProgram has a name, a ParsedExpression, and a number of ParsedTypeDecl's. The metamodel in Fig. 3.1 shows that the meta-class ParsedProgramhas an attribute name, an association with meta-class ParsedExpressionwith multiplicity one, and an association with meta-class ParsedTypeDeclwith multiplicity many, marked 'ordered'. The mapping between the BNF rule and the metamodel is such that name is set as value of the name attribute of the ParsedProgram, the ParsedExpression is set as the startExp, and the ParsedTypeDecl's are set as the types of the ParsedProgram. Note that the ParsedTypeDecl's are added to the types in the order in which they appear in the input. This rule holds for all elements in the metamodel marked 'ordered'.

The metamodel can be divided into three parts: the types, the statements, and the expressions, each of which will be explained the following.

3.1.1 Types

The types in the concrete syntax metamodel can be found in Fig. 3.1.

ParsedOperDecl The class ParsedOperDecl represents a user defined operation. It may have a number of formal parameters, and a number of local variables, each of which are ParsedVarDecl instances. The body of the operation is represented by one of the statement types, the ParsedBlockStat. The return type of the operation is represented by a ParsedTypeRef instance.

ParsedProgram The class ParsedProgram represents a complete program. The fact that a program needs to start somewhere, is represented in TAAL by a single start-expression for every program. (In Java this has been resolved by requiring that the class to be executed must have a main-method.) The only sensible expression to start a program is one that creates an object and calls a method on this object. In the metamodel this is represented by the association called startExp to the meta-class ParsedExpStat. Next to the start expression each program contains a number of user defined types, represented in the metamodel by the association called typesto ParsedTypeDecl.

ParsedTypeDecl The class ParsedTypeDeclrepresents a user defined type. It may have a reference to a supertype, represented in the metamodel by the association called superType to ParsedTypeRef. Furthermore, a ParsedTypeDecl can have operations and attributes, represented by the associations called operations and attributes, respectively.

ParsedTypeRef The class ParsedTypeRef represents a reference to a type. Its only attribute is the name of the type to which it is a reference. During static analysis this reference needs to be resolved.

ParsedVarDecl The class ParsedVarDecl represents a declaration of a variable. This variable may be used as attribute, as formal parameter, or as local variable. Any variable has a type, represented by a ParsedTypeRef, and it may have an expression that represents its initial value.

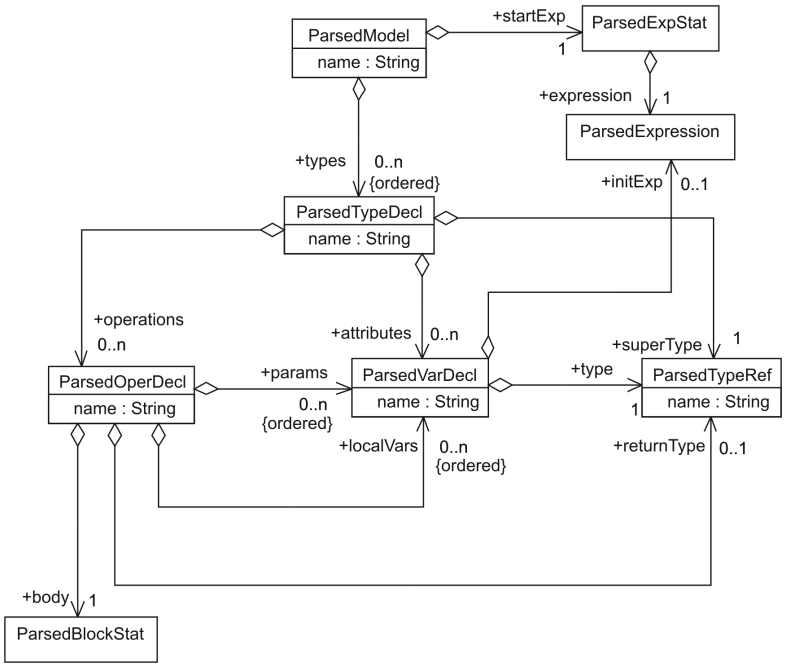


Figure 3.1: The types in the CST meta-model.

3.1.2 Statements

The statements in the concrete syntax meta-model can be found in Fig. 3.2.

ParsedAssignStat The class ParsedAssignStat represents an assignment. It holds references to the expressions that represent respectively the left hand side of the assignment and the right hand side of the assignment.

ParsedBlockStat The class ParsedBlockStat represents an ordered list of statements. The body of an operation implementation, as well as the body of a while-statement, the then-part and the else-part of an if-statement usually are block-statements.

ParsedConditionalStat The class ParsedConditionalStat represents a conditional statement or if-statement. It holds a condition, a reference to an expression, and one or two references to statements representing the then-branch and the else-branch.

ParsedExpStat The class ParsedExpStat represents a statement that consists of a single expression.

ParsedReturnStat The class ParsedReturnStat represents a return statement. It holds a reference to the expression that represents the value to be returned, called value.

ParsedStatement The class ParsedStatement represents a statement in the program. It is an abstract class, which is indicated in the diagram by the fact that its name is in italics.

ParsedWhileStat The class ParsedWhileStat represent a while loop. It holds a condition and a body. The condition is a reference to an expression, the body is a reference to a statement.

3.1.3 Expressions

The expressions in the concrete syntax meta-model can be found in Fig. 3.3.

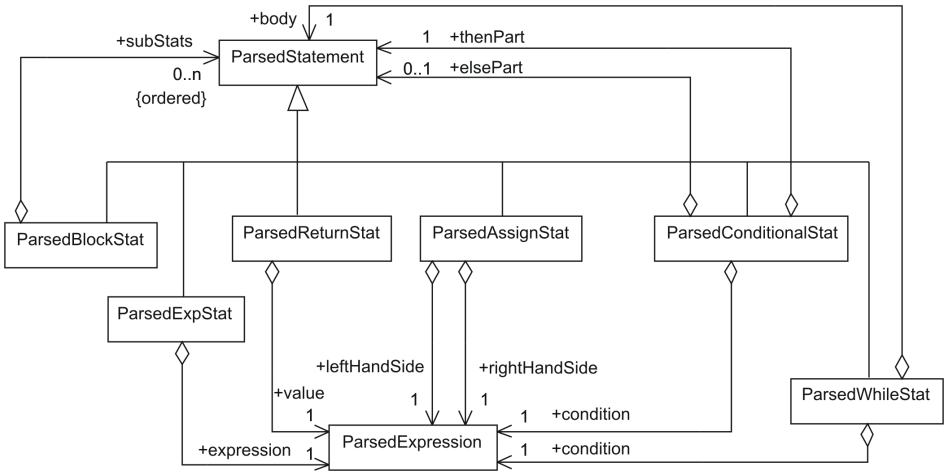


Figure 3.2: The statements in the CST meta-model.

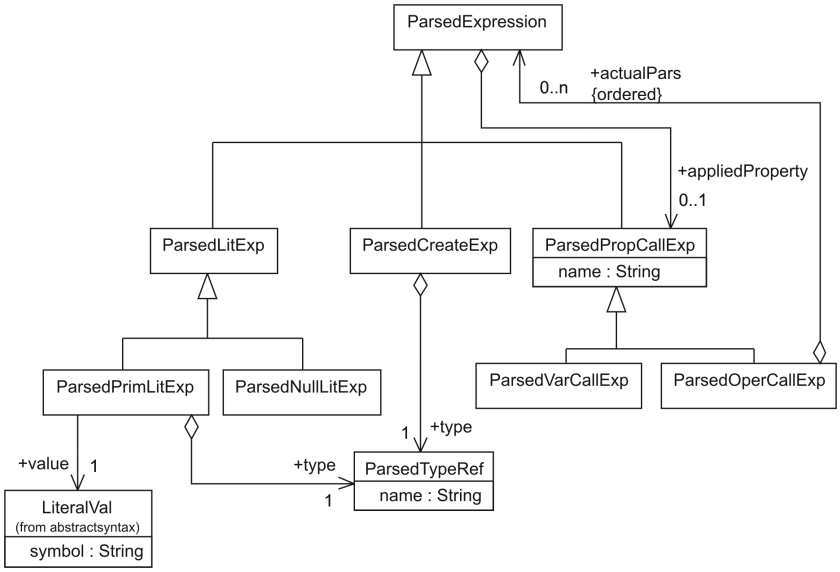


Figure 3.3: The expressions in the CST meta-model.

ParsedCreateExp The class ParsedCreateExp represents an expression whose value is a newly created object.

ParsedExpression The class ParsedExpression represents any expression in the program. It is an abstract class, which is indicated in the diagram by the fact that its name is in italics.

ParsedLitExp The class ParsedLitExp represents a literal expression. It is an abstract class.

ParsedNullLitExp The class ParsedNullLitExp represents the 'null literal expression.

ParsedOperCallExp The class ParsedOperCallExp represents a call to an operation. It holds a list of expressions that are the actual parameters to this call.

ParsedPrimLitExp The class ParsedPrimLitExp represents any literal expression except 'null. If the expression is a string, the link called type will hold a reference to the standard library type 'String. If the expression is numeric, the link called type will hold a reference to the standard library type

‘Real. It holds a reference to a LiteralVal instance that holds the actual value. Note that LiteralVal is a meta-class that is part of the ASG meta-model (see Sect. 4).

ParsedPropCallExp The class ParsedPropCallExp represents either an operation call or a reference to a variable. Instances may be part of a compound expression. According to the BNF rules an expression may consist on the one hand of an expression, and on the other hand of a dot and a variable name or operation call. The part after the dot is represented by an instance of this class, or rather, because this class is abstract, by instance of one of its subclasses. The name attribute holds the name of the called variable or operation.

ParsedVarCallExp The class ParsedVarCallExp represents a call to a variable.

3.1.4 Auxiliary Elements

In the metamodel for the CST two extra meta-classes are present.

LiteralVal The class LiteralVal is part of the value graph meta-model. It represents a literal value in the program, like ‘10, or ‘myString. It is used in the CST meta-model because the exact literal value must be used throughout the series of transformations up and until it is part of the Execution Graph.

ParsedElement The class ParsedElement represents any element that is recognized during parsing. ParsedElement has three attributes (line, column, and filename) that hold information on where the element is found in the input text. Each element of the CST meta-model inherits from the class ParsedElement. It is an abstract class.

4 Rich Abstract Syntax Graphs

As explained earlier, the abstract syntax for TAAL comes in two flavours: the rich and the flat abstract syntax. In this section we will explain the first of these two.

4.1 Meta-model

Fig. 4.1 to Fig. 4.3 contain the UML meta-model for the Rich Abstract Syntax Graph. Like the Concrete Syntax Tree meta-model, this meta-model can be split into parts: the types, the statements, the expressions. Additional is the part that represents the build-in types and operations: the standard library. In the following, each element in the meta model, together with its context, will be explained.

4.1.1 Types

The different types that are supported by TAAL, together with their relationships, are shown in Fig. 4.1.

NullType Class NullType represents the type of a special value: the *null value*. This type has only one instance and conforms to any other type. This class is a *singleton class* [19].

ObjectType Class ObjectType is a specialization of class Type. It enables the programmer to define new types in the program. Class ObjectType has a superType-relationship with itself modelling inheritance-relationships between instances of this class. Class ObjectType has a relationship called attributes with class VarDecl representing the instance variables.

OperDecl Operations that are declared for objects are captured by the class OperDecl. It explicitly refers to the Type-class it belongs to by means of a relationship called owner. Every operation may require multiple parameters in order to be executed. This is modelled by the parameters-relationship with the class VarDecl. An operation may also have local declared variables. Therefore, the localVars-relationship has been introduced. In order to track the referred operation implementation at run-time it is needed to assign a signature to every operation. This will be explained in more detail in Sect. 6(TBC).

OperImpl The meta-model explicitly distinguishes between abstract and concrete operations. Abstract operations, i.e. operations that do not have an implementation, are instances of the OperDecl class. Concrete operations are instances of the OperImpl class. The OperImpl class therefore has a relationship with Statement called body. See Section 4.2 from more information on the difference between the OperDecl and OperImpl classes.

PrimitiveType TAAL includes a number of elementary data-types that are represented by the class PrimitiveType (e.g. String, Integer and Real). (See also Section 4.3.)

Program The class Program represents the whole program. In a program multiple data-structures can be declared and referred to. The data-structures being declared are the types of the program represented by the types-relationship with the class Type. When a program has multiple types, these types are ordered reflecting the order in which they appear in the code. As mentioned before, the fact that a program needs to start somewhere, requires every program to have a single start-expression. Therefore, the class Program has a relationship with class ExpStat (which is explained in section 4.1.2).

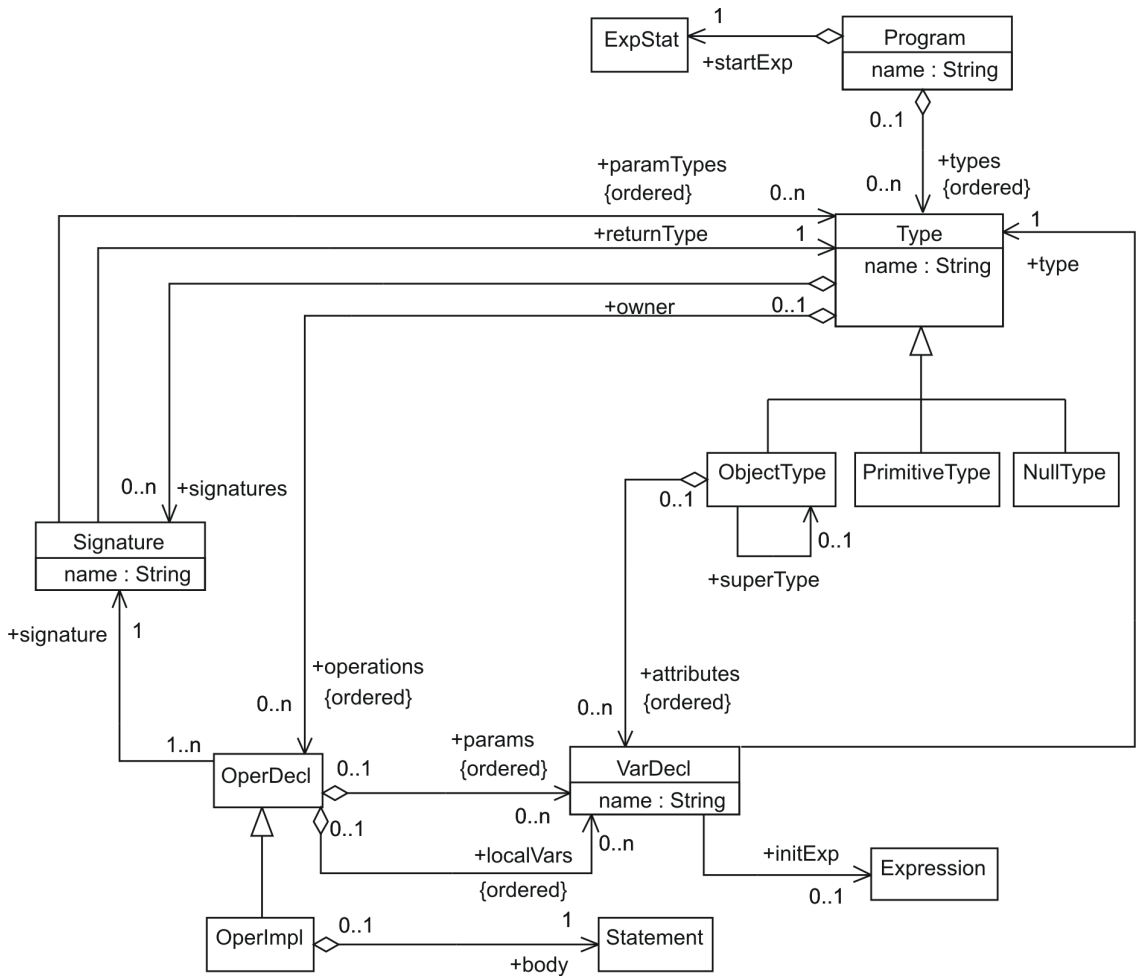


Figure 4.1: The types in the ASG meta-model.

Signature The class Signature stores the elements that uniquely identify the operations, i.e. the types of the parameters (this is an ordered 'list), the return type, and the name of the operation. In Sect. 7, it will be made clear that method signatures play a crucial role in the dynamic method lookup process.

Type Typing is a very important concept in object-oriented languages. The purpose of class Type is to model typing at a generic level. The class Type has three subclasses, namely ObjectType, PrimitiveType and NullType, classifying the kind of type being referred to at a certain point in the program. The Type class has a relationship with class OperDecl called operations representing a (possibly empty) set of operations. It is abstract, which is indicated in the diagram by the fact that its name is in italics.

VarDecl Class VarDecl represents the declaration of a variable, which can be either an instance variable, a local method variable, or a formal method parameter. The type of the variable is captured by the type-relationship with class Type. Furthermore, this class has a relationship with class ObjectType called owner representing that every instance variable belongs to at most one object. The relationship initExp with class Expression represents the initialization of the instance variable at declaration. Note that this relationship is not optional (see Section 4.2).

4.1.2 Statements

TAAL supports the use of a variety of statements. The different kinds of statements and their relationships at meta-level will be explained in this section. Fig. 4.2 gives an overview of the different supported statements.

AssignStat The class AssignStat represents the statements in which a particular value will be assigned to some variable. The value that will be assigned is represented by the relationship with class Expression called source. The variable that will be assigned is referred to by the relationship with class VarDecl called assignedVar.

BlockStat The class BlockStat has a relationship with its own superclass Statement called subStats. This relationship represents the embedding of all kinds of statements within a single block.

ConditionalStat The class ConditionalStat represents the well-known if-then or if-then-else construction in programming languages. It consists of two required, and one optional element. The first required element is the condition which determines where to continue the program. This is modelled by the condition-relationship with class Expression. The second required element is a statement representing the then-part which will be executed when the conditional-expression evaluates to true. This is modelled by the thenPart-relationship with class Statement. There is another relationship with this class called elsePart, which may or may not exist for a particular conditional statement.

ExpStat The class ExpStat, represents an expression used as a statement, i.e. in such a way that its value is discarded. Typical examples are object creation and method invocation.

ReturnStat The class ReturnStat has a single relationship with class Expression called value, representing that a return-statement should return a value of an expression of a specific type.

Statement The class Statement is the super class of all specific statements that can occur in a TAAL-program. It has six subclasses, namely ConditionalStat, ExpStat, ReturnStat, BlockStat, WhileStat and AssignStat. It is abstract, which is indicated in the diagram by the fact that its name is in italics.

WhileStat The class WhileStat represents a while-statement. It holds a reference to an instance of class Expression by the condition-relation that figures as the condition, as well as a reference to an instance of Statement that figures as the body to be executed in case the condition returns true.

4.1.3 Expressions

The different expressions that are supported by TAAL, together with their relationships, are shown in Fig. 4.3.

CreateExp The class CreateExp represents a new object expression. It has a link with the ObjectType of which the newly created object should be an instance.

Expression The class Expression represent an expression in a TAAL program. It is abstract, which is indicated in the diagram by the fact that its name is in italics.

LiteralExp Instances of the class LiteralExp represent literal expressions. It is abstract, which is indicated in the diagram by the fact that its name is in italics.

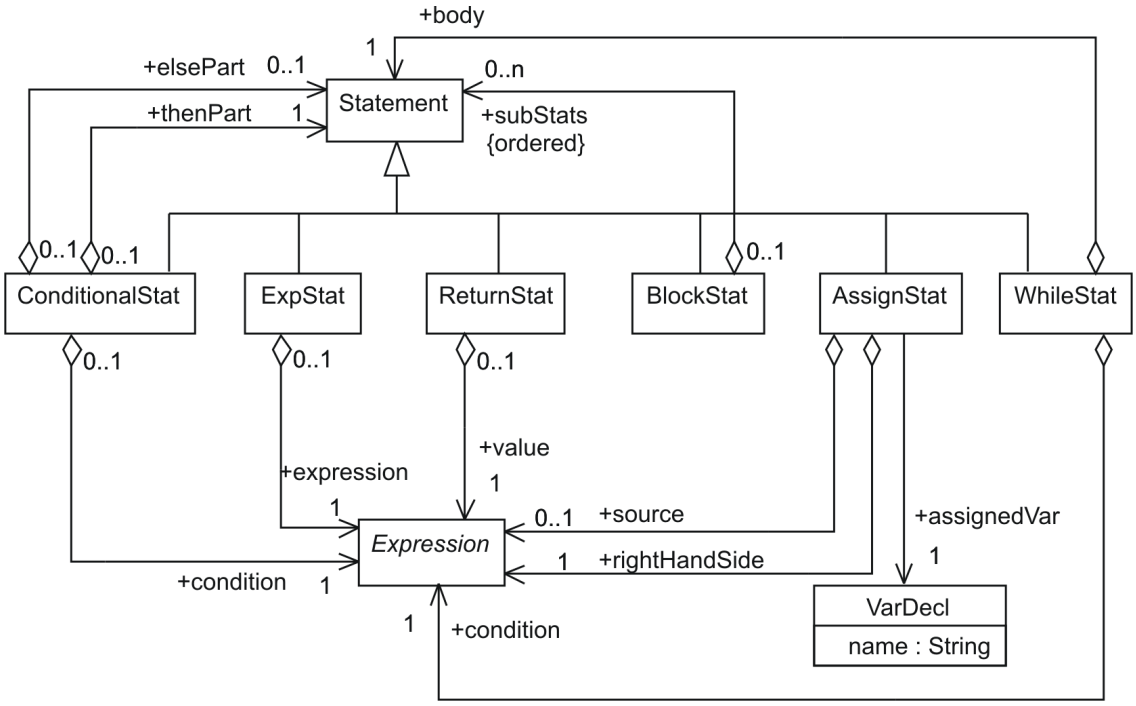


Figure 4.2: The statements in the ASG meta-model.

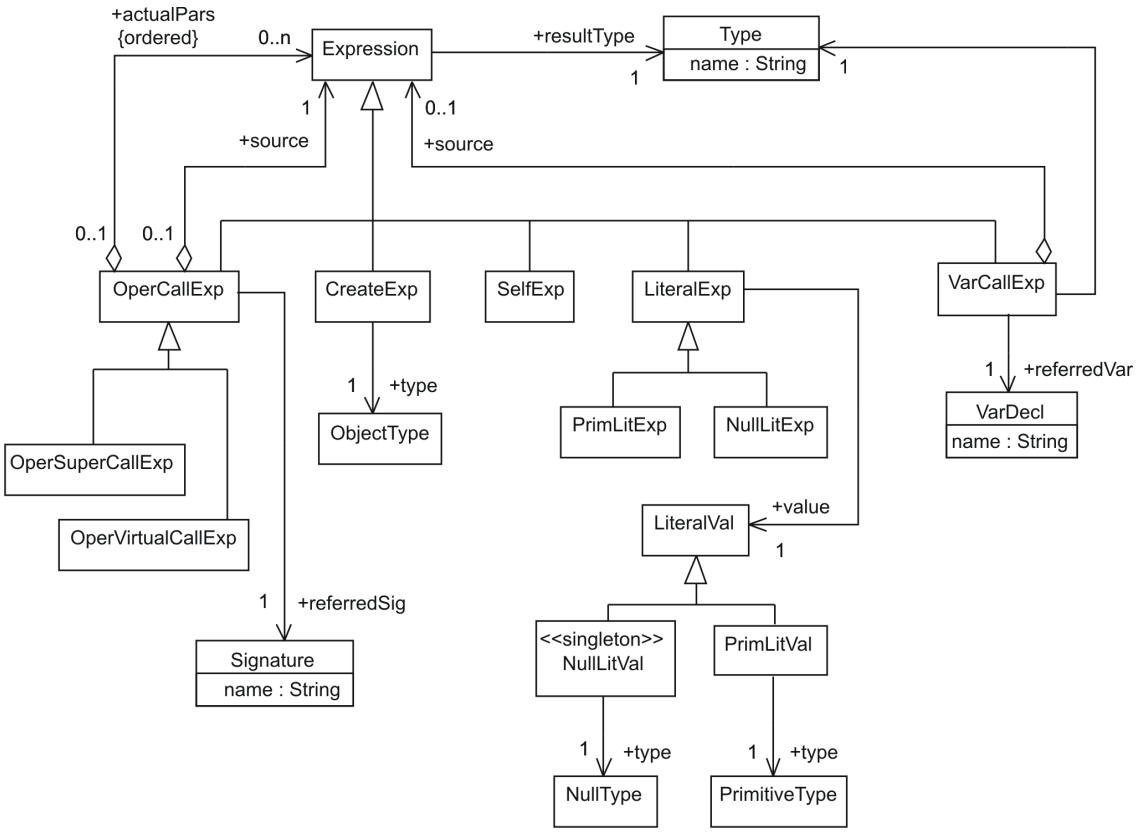


Figure 4.3: The expressions in the ASG meta-model.

NullLitExp The class NullLitExp represents the literal expression 'null'. This literal expression has as type the NullType instance. NullLitExp is a *singleton* class.

OperCallExp The class OperCallExp represents a reference to an operation of a type. Each OperCallExp instance may have actual parameters. This is represented by the relationship between OperCallExp and Expression called actualPars. An OperCallExp instance also holds a relation to the Signature of the operation of which it is an invocation. Note that an OperCallExp does not reference an OperDecl or OperImpl, because of the dynamic method lookup that supports inheritance in TAAL. The class OperCallExp is abstract, which is indicated in the diagram by the fact that its name is in italics.

OperSuperCallExp The class OperSuperCallExp represents the explicit calling of an operation declared in one of the super classes of the current class. It is created when the input contains operation calls like "super.xxx()".

OperVirtualCallExp The class OperVirtualCallExp represents a reference to an operation. It represents the 'normal' operation call, in contrast with the explicit calling of an operation of the supertype.

PrimLitExp PrimLitExp is the meta-class for all literal expressions that have as type a PrimitiveType.

SelfExp The class SelfExp stands for the expression that consists of the keyword self.

VarCallExp The class VarCallExp represents a reference to a variable. Each VarCallExp instance holds a relationship with the attribute, local variable, or formal parameter of an operation. In the metamodel this is indicated by the association with class VarDecl called referredVar.

4.1.4 Auxiliary Elements

In the metamodel for the ASG one extra meta-class is present, which does not appear in Fig. 4.1 - 4.3, but is shown in Fig. 4.4.

AbstractSyntaxElement The class AbstractSyntaxElement represents any element in the ASG. It is the abstract superclass of all classes in the ASG metamodel. AbstractSyntaxElement has three attributes (line, column, and filename) that hold information on where the element is found in the input text. This information is used for error messages.

4.2 Differences between CST and ASG Meta-models

There are notable differences between the meta-models of the concrete syntax tree and the rich abstract syntax graph. These differences are explained in the following list.

- There are no counterparts for the NullType and PrimitiveType in the CST meta-model. The reason is that these types may not be defined by a user of TAAL. All instances of both classes are part of the standard library, which will be explained in Section 4.3.
- The ASG meta-model does not contain a counterpart for the class ParsedTypeRef. Instances of this class are resolved and replaced by references to, i.e. associations with, the type itself.

- The association from VarDecl to Expression called initExp is not optional, whereas the corresponding relationship in the CST is optional. If there is no initExp to a ParsedVarDecl, then the static analysis will create a VarDecl with the NullLitExp instance as initial expression. If the type of the declared variable is one of the predefined types Real and Integer, its initial

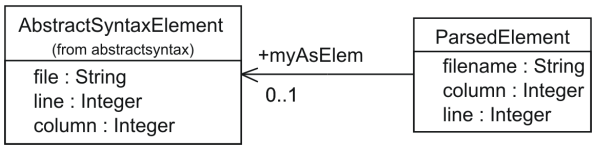


Figure 4.4: The link between CST and ASG meta-models.

value will be set to 0. In the case the variable is of type Boolean, which is also a predefined type, its initial value is 'false.

- The return type of an operation is treated similarly. If the return type is not present in the CST then the return type in the ASG will be set to the NullType.
- In the CST there can only be ParsedOperDecl's present. In the ASG the ParsedOperDecl may be represented by either an OperImpl or an OperDecl instance. If the ParsedOperDecl has a ParsedBlockStat without any sub-statements, then an OperDecl instance is created, else an OperImpl instance is created. Note that this means that the user cannot overwrite an operation with an empty one. The BNF rules would need to be extended in order to differentiate between an operation without a body, and an operation with an empty body.
- Signature's are completely missing from the CST. These instances are created during the static analysis based on the information in the ParsedOperDecl's.
- The left hand side of a ParsedAssignStat is a ParsedExpression. During static analysis this expression is analyzed. If it represents a variable, then in the ASG it is replaced by link to a source expression, representing the source of the variable, and a link, called assignedVar, to the variable itself.
- An Expression in the ASG always has a Type associated. This resultType is determined during static analysis based on the different subclasses of Expression. For instance, the resultType of a CreateExpis always equal to its type, the resultType of an OperCallExp is equal to the return type of the referred Signature, and the resultType of a PrimLitExp depends on its value.
- The class ParsedPropCallExp has no ASG counterpart. It is used to simplify parsing only.
- Instances of the class ParsedOperCallExp are resolved to be instances of either subclass of OperCallExp. The class OperCallExp thus remains abstract.

4.3 The Standard Library and Type Conformance

Most languages have a number of predefined types, and TAAL is no exception. TAALs predefined types are gathered into a set called the standard library. The standard library is implemented as a single class that holds four instances of PrimitiveType, called String, Boolean, Integer, and Real. A very small number of operations are available on these types. For the String type the operation equals is defined. The operations plus and minus are defined for both Integer and Real. There is no infix notation for calling these operations. For instance, the expression 10.minus(3) is the syntax for calling the minus operation.

In every typed language, type conformance is an issue. In TAAL, type conformance is determined by a single (meta)operation in the class Type. It implements a number of very simple rules. First, all types conform to the NullType, and NullType conforms to all other types. Second, if both types are equal they conform to each other. Third, a subclass conforms to any of its superclasses. Fourth, the type Integer from the standard library conforms to the type Real, also from the standard library.

4.4 Static Analysis

During the static analysis, a Rich Abstract Syntax Graph is created from the Concrete Syntax Tree. This is done by traversing the concrete syntax tree a number of times, i.e. it is a multi-pass analysis. Each traversal is implemented in a separate class that implements a visitor over the Concrete Syntax Tree. Many of the passes could be combined if there was a need for efficiency.

Pass 1 For each element in the Concrete Syntax Tree that has a direct abstract syntax counterpart, this abstract syntax counterpart is created. The ASG element is temporarily stored in the Concrete Syntax Tree element. Therefore the class ParsedElement has an association with class AbstractSyntaxElement, called myAsElem, as shown in Figure 9. Parsed elements that do not have a direct ASG counterpart, like ParsedTypeRef and ParsedPropCallExp, are simply skipped. To be able to give meaningful error messages the ASG element gets the information on the line, column, and file from its parsed element. If the parsed element has a name, the name of its ASG element will be set to the same value.

ParsedTypeDecl instances are always transformed into ObjectType instances. Users may not define PrimitiveType instances, these are hold by a dedicated class called StandardLib. If a ParsedPrimLitExp has as value the string self, a SelfExp instance is created instead of a PrimLitExp instance. Each ParsedOperDecl is checked whether it contains a body. If the body is present the ASG element will be of type OperImpl, otherwise it will be of type OperDecl. For both a Signature instance is created and stored in the signature role of the OperDecl meta-class. The name of the Signature instance is set. In this pass the signature is not yet added to the owner of the operation. The value of each NullLitExp will be set to a LiteralVal instance with symbol null. There is just one such LiteralVal instance in the complete system.

Pass 2 In the next pass the simpler associations in the ASG are set. All values are set to the ASG element of the corresponding association in the CST. For instance, the value of myAsElem.startExp in a ParsedProgram will be set to startExp.myAsElem. The values of the following associations are set:

- BlockStat.subStats
- ConditionalStat.elsePart
- ConditionalStat.thenPart
- OperDecl.locals
- OperDecl.params
- OperImpl.body
- PrimLitExp.value
- Program.startExp
- Program.types
- Type.attributes
- Type.operations
- WhileStat.body

Pass 3 In pass 3 type references are analyzed, and the standard library is initialized. Each reference to a ParsedTypeReference instance is replaced by a reference to the correct type. First the types in the program are searched, if the reference cannot be found, the standard library is searched. The resultType of OperCallExp and VarCallExp instances is not set in this pass. This pass sets the values of the following associations:

- CreateExp.resultType (equal to CreateExp.type)
- CreateExp.type
- CreateExp.type
- NullLitExp.resultType
- ObjectType.conformsTo (for user defined types this equals ObjectType.super)
- ObjectType.super
- OperDecl.signature.returnType
- PrimLitExp.resultType (the difference between Real and Integer is analyzed here)
- VarDecl.type

Pass 4 The signature of each operation is completed in pass 4. If an operation has a signature that equals a signature that already exists in the program, the operation will refer to the existing signature. Thus all operations with the same signature will refer to the same signature object, i.e. node in the flat abstract syntax graph. Signatures are considered to be equal when their names are equals and their list of parameter types are equal. Both the elements, and the order of the elements in the list of parameter types must be equal. Note that the VarDecl instances that constitute the parameters obtain their type in pass 3. Therefore the parameter types in the Signature instances can only be set after pass 3. The value of the following association is set in this pass:

- OperDecl.signature.paramTypes

The value of the following association may be changed, if the found signature is equal to an existing signature:

- OperDecl.signature

Pass 5 Pass 5 analyses to which Signature an OperCallExp instance refers, and to which VarDecl a VarCallExp instance refers. In order to find the first, the enclosing TypeDecl is searched for signatures, if no matching signature is found its supertypes are searched recursively. Before searching for a matching signature, the actual parameters are tested for the occurrence of operation or variable calls. If found, they will be analyzed first. In the concrete syntax the keyword super may precede an operation call. When this is the case, the OperCallExp instance is transformed into an OperSuperCallInstance. The search for a matching signature will in this case start from the supertype of the enclosing type.

In order to find the VarDecl instance to which a VarCallExp refers, first the enclosing OperDecl, if present, is searched for formal parameters and local variable declarations, next the enclosing TypeDecl is searched for attributes.

If the ParsedPropCallExp from which analysis is started, has an appliedProperty, then this property is analyzed as well. In this case the TypeDecl instance being searched is the type of the ParsedPropCallExp from which analysis started, i.e. the source of the appliedProperty. If an OperCallExp does not have a source, a new SelfExp is created that acts as source to these expressions. The same holds for VarCallExps that refer to an attribute.

This pass sets the values of the following associations:

- OperCallExp.actualPars
- OperCallExp.referredSig
- OperCallExp.resultType
- SelfExp.resultType
- VarCallExp.referredVar
- VarCallExp.resultType

Pass 6 The source of every OperCallExp or VarCallExp is set in pass 6. Furthermore, every link to an expression is set. In most abstract syntax trees the top node of an expression is the rightmost element of the expression. For instance, in the expression `aa.bb().cc` the VarCallExp that refers to `cc` is the top of the subtree representing the total expression. The OperCallExp that refers to `bb()` is its child, and the VarCallExp that refers to `aa` is again a child of this OperCallExp. However, the Concrete Syntax Tree is build the other way around. The same example expression has a ParsedVarCall with name `aa` as its top. This pass solves the reversal by setting each expression to the ASG counterpart of the last applied property: lastAppliedProp().getMyAsElem.

Finally, the last link of each AssignStat instance is set: assignedVar. Note that this can only be done after pass 5, when the referred variable of every VarCallExp has been set. The value of the following associations is set in this pass:

- AssignStat.assignedVar

- AssignStat.source
- AssignStat.rightHandSide
- ConditionalStat.condition
- ExpStat.expression
- PropCallExp.source
- ReturnStat.value
- VarDecl.initExp
- WhileStat.condition

Pass 7 In the last pass, no changes are made to the ASG. The pass merely performs a check on the created ASG. Any errors found are reported. The following questions are answered:

- Are all obligatory associations present?
- Is the number of signatures in a type equal to the number of operations?
- Does the number of parameter types in an operations signature equal the number of parameters in the operation?
- Is the resultType of a NullLitExp indeed equal to the singleton instance of class NullType?
- Is the value of a NullLitExp indeed equal to the unique instance of class LiteralVal that represent a null value?
- Is the expression in an ExpStat executable? I.e. it must not be a VarCallExp, a LitExp, or a SelfExp.

5 Flat Abstract Syntax Graphs

In this section we discuss the "flattening" process of Fig. 2.1. This is of special interest because it is the only transformation that crosses the boundary between modelling languages: whereas the (meta)models discussed so far were all formulated in UML and lie on the left hand side of Fig. 2.1, and the subsequent sections only deal with graphs and lie on the right hand side of the figure, in the flattening we have to bridge the gap between the two.

As we argue in more detail below (Sect. 5.3), there is a limit to the degree in which the flattening step can be made precise formally, essentially because, where the flattened graphs are already mathematical objects, the models that are flattened into them are only available as run-time data structures. It is outside the scope of this report to formulate (and validate) the appropriate corresponding mathematical model for the run-time data structures and to formalize the relationship with the graphs. What we can (and do) do, however, is to formalize the relationship between graphs and (UML) meta-models, both of which can be captured mathematically. This is described at some length in Sections 5.1–5.2.

5.1 UML models versus graphs

In this study we bring together the worlds of modeling and of graphs. The UML, a relevant part of the modeling world, does a great job in visualizing models and meta-models. On the other hand, graphs are very good in visualizing instances and their relationships. Whereas the power of the UML lies on the level of types, graphs are stronger on the level of instances. In OMG terminology, UML rules the levels M1, M2 and M3, but graphs rule the level M0. To specify the semantics of TAAL by means of graph transformations, we have to link the TAAL metamodel (most naturally thought of as residing on level M2) to graphs, which, depending on the context, represent structures on level M1 or M0.

In fact, because UML is good at capturing structural aspects of models, in the subsequent sections we will continue to use it on the type level to extend the TAAL abstract syntax metamodel of Figures 4.1, 4.2 and 4.3, even though the corresponding instances will now be graphs. We prefer this to using the (comparable) concept of a *type graph* (see, e.g., [24]) since those do not provide the same capabilities. For instance, type graphs do not include multiplicities or inheritance (at least in the standard theory; see, however, [4, 9, 33] for some proposals for extension).

This implies, however, that it has to be be clear what it means for a graph to be an instance of a UML model. We clarify this below by providing formal definitions, both for graphs (which is straightforward) and for the fragment of UML we use in this report. In both cases this results in a mathematical structure (formally: an algebra) consisting of sets and operations on them. We then define a class of mappings, called *typings*, from graph structures to UML structures, and we say that a graph is (or, more properly, can be regarded as) an instance of a UML model if a typing exists between their respective mathematical structures.

5.1.1 Graphs

In the research reported here we use graphs of a particularly straightforward kind, consisting of *nodes* (which typically represent particular instances of some concept, such as a statement, a class, an object or a data value) and *edges* (which typically represent relations between the nodes). Each edge has a *label*, which indicates the kind of relation it represents.

This gives rise to the following mathematical structure:

Definition 5.1 (graph) *A graph* G *is a tuple* $\langle Lab, Nod, Edg \rangle$ *where*

- Lab *is a finite set of* labels*;*
- Nod *is a finite set of* nodes*;*
- $Edg \subseteq Nod \times Lab \times Nod$ *is a (finite) set of* edges*.*

It follows from the definition that, formally, an edge is a triple consisting of a node, a label, and another node. The first node is called the *source* and the second the *target* node of the edge. To manipulate and reason about graphs, we often use the following auxiliary functions, which project edges onto their constituent parts:

- The *label* function $lab\colon Edg \to Lab$;
- The *source* function $src\colon Edg \to Nod$;
- The *target* function $tgt\colon Edg \to Nod$.

This gives rise to another view, also frequently encountered, namely to regard a graph as a special kind of many-sorted algebra with signature $\langle S, F\rangle$, where $S = \{Lab, Nod, Edg\}$ is the set of carrier types and $F = \{lab, src, tgt\}$ is the set of operations (all of which are unary). We mention this here especially because this signature (and indeed any many-sorted signature with only unary operations) can itself be seen as a graph, with nodes S and edges F. This "signature graph," which in MDA terms would be called the meta-model of graphs, is depicted in Fig. 5.1 (left hand side).

As this example shows, it is convenient to represent graphs pictorially (as usual) by drawing nodes as boxes and edges as arrows between the boxes. Since a graph, in its above definition, does not contain any layout information, or (in other words) the layout is not considered to be part of the graph, in a pictorial representation we are free to layout a graph in any way convenient for the application at hand.

In addition to these representation choices we also use labels written *inside* nodes to stand for labels of self-edges, i.e., edges with that node as both their source and target. (Note that this does not give rise to confusion since such inscribed labels are not used for any other purpose.)

5.1.2 UML models

The formalization of UML class diagrams proceeds along the same lines as that of graphs; however, the richer structure of class diagrams is reflected in its formalization. We start by providing some notation for multiplicities: the set of natural numbers will be denoted Nat, and Nat_* denotes Nat together with the special symbol $*$, which stands for "arbitrarily many" — formally, this can be treated as infinity. Indeed, we pose $i < *$ for any $i \in Nat$. We use $Mult = [i, j] \in Nat \times Nat_* i \leq j$ as the set of multiplicities; that is, all pairs of numbers from Nat and Nat_* of which the second is no smaller than the first.

The following definition covers all the aspects of class diagrams that we use.

Definition 5.2 (UML model) *A class diagram* CD *is a tuple* $\langle Id, Cls, Dat, Ass, Att, Ext, name\rangle$ *where*

- *Id is a finite set of* identifiers*;*
- *Cls is a finite set of* classes*;*

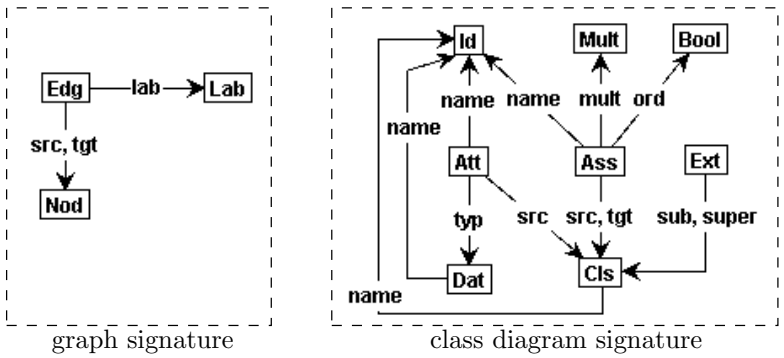


Figure 5.1: The signatures of graphs and class diagrams, depicted as graphs.

- *Dat is a finite set of* data types*, disjoint from Cls such that each* $T \in Dat$ *is a disjoint set of values;*
- $Ass\colon (Cls \times Id) \rightharpoonup (Mult \times Bool \times Cls)$ *is a partial* association mapping*;*
- $Att\colon (Cls \times Id) \rightharpoonup Dat$ *is a partial* attribute mapping*, with a domain of definition disjoint from that of Ass;*
- $Ext \subseteq Cls \times Cls$ *is an acyclic* extension relation*, with reflexive and transitive closure* Ext^**;*
- $name\colon (Cls \cup Dat) \to Id$ *is an injective* naming function*.*

The association and attribute mappings might need some clarification. Both are partial functions from $Cls \times Id$, meaning that for any combination of class and (association/attribute) name, they may either yield a value (explained below) or be undefined — namely, if there is no association, respectively attribute, with that name in that class. Where defined, Ass yield a triple of multiplicity, ordering indication and target class of the association, whereas Att yields the type of the attribute. The constraint on the disjointness of the domains of definition of Ass and Att means that there cannot be an association and an attribute with the same name in the same class.

Just as with graphs, there is an alternative (and equivalent) algebraic view on these structures. To understand this, it should be clear first that any function (total or partial) can be seen as a set of pairs of values from its domain and codomain: each such pair represents the fact that the first component is mapped by the function to the second. If the domain or codomain are themselves Cartesian products, as in the case of Ass and Att, we can also flatten the resulting pairs of tuples into one bigger tuple. Thus, for instance, Ass and Att can both be viewed as sets, of 5-tuples and triples, respectively. From these sets we can define auxiliary functions that project the tuples to their constituent components, much like *lab*, *src* and *tgt* for graphs. This gives rise to the following auxiliary functions:

- $src\colon (Ass \cup Att) \to Cls$, projecting each association or attribute tuple to its first (source) component;
- $name\colon (Ass \cup Att) \to Id$, projecting each association or attribute tuple to its second (name) component;
- $mult\colon Ass \to Mult$, projecting each association tuple to its third (multiplicity) component;
- $ord\colon Ass \to Bool$, projecting each association tuple to the fourth (orderedness) component;
- $tgt\colon Ass \to Cls$, projecting each association tuple to the fifth (target) component;
- $typ\colon Att \to Dat$, projecting each attribute tuple to its third (type) component;
- $sub\colon Ext \to Cls$, returning the subclass (specialized) type of an extension pair;
- $super\colon Ext \to Cls$, returning the superclass (generalized) type of an extension pair.

Like in the case of graphs, the resulting signature can itself be depicted in the form of a graph; this is also given in Fig. 5.1 (right hand side).

5.2 Pre-typings and typings

We will now define what it means for a graph to be an *instance* of a class diagram. This hinges on the existence of a mapping *map* from the graph to the class diagram that

- maps the nodes of the graph to the classes of the class diagram, in such a way that the image of this mapping for any one node can be considered as the type of the node, and
- maps the edges of the graph to associations and attributes, and in some case also classes, of the class diagram.

Formally, *map* is given by a pair of functions

$$\begin{aligned} map.Nod : \mathsf{G}.Nod &\to \mathsf{CD}.Cls \\ map.Edg : \mathsf{G}.Edg &\to \mathsf{CD}.Ass \cup \mathsf{CD}.Att \cup \mathsf{CD}.Cls \end{aligned}$$

but since no confusion can arise (due to the fact that the domains of definition are disjoint) we will denote both of these by *map*.

5.2.1 Pre-typings

We will call any such mapping *map* (without further constraints) a *pre-typing* from G to CD. In the remainder of this sub-section we discuss constraints under which a pre-typing is considered to be a *typing* — which in turn will be the evidence we will use to justify the claim that the graph G is an instance of the class diagram CD.

Some further terminology first. Given a pre-typing *map* from G to CD, we recognize the following five categories of edges in G. Let $e = (v, a, w) \in \mathsf{G}.Edg$ in the explanation below.

1. *Association edges*, which are instances of some association in CD defined between generalizations of the classes of the edge's source and target. Formally, e is an association edge if $map(e) \in \mathsf{CD}.Ass$ such that

   - $(map(v), src(map(e))) \in Ext^*$, meaning that the source of the association generalizes the type of the source of e;
   - $(map(w), tgt(map(e))) \in Ext^*$, meaning that the target of the association generalizes the type of the target of e;
   - $name(map(e)) = a$, meaning that the name of the association coincides with the name of the edge.

2. *Attribute edges*, which are self-edges modelling instances of some attribute in CD defined in a class that generalizes the type of the edge's source. Formally, e is an attribute edge if

   - $v = w$, meaning that e is a self-edge (and in our figures the label will be written on the node);
   - $map(e) \in \mathsf{CD}.Att$ and $(map(v), src(map(e))) \in Ext^*$, meaning that e is mapped to an attribute defined in a class that generalizes the type of the source of e;
   - $a =$ "$name(map(\mathrm{e})) = A$" for some data value $A \in typ(map(e))$, meaning that the label of the edge is a string consisting of the name of the attribute, the equals sign, and some data value in the type of the attribute.

   This encoding of attributes is quite different from that of associations, since we are not relying on the graph structure here: instead the attribute name and the instance value are both encoded in the edge label. This means that we cannot manipulate attribute values in any way. Although this is satisfactory in the context of this paper, and avoids the somewhat messy issue of graph attribution, it sharply limits the generality of the flattening.

3. *Type edges*, which are self-edges bearing the name of one of the super-classes of the node type. Formally, e is a type name edge if

   - $v = w$, meaning that e is a self-edge (and in our figures the label will be written on the node);
   - $map(e) = C \in \mathsf{CD}.Cls$ such that $(map(v), C) \in Ext^*$ and $name(C) = a$, meaning that e is mapped to a supertype of v's type, and the name of that type coincides with the label of e.

   We use type edges as a kind of "poor man's type annotation:" in this way, we can formulate the graph transformation rules on the proper level of type abstraction. A richer graph model, with node labels or explicit typing, would obviate the need for type edges.

4. *Head edges*, which are edges pointing to the first target element of some ordered association instance. Formally, e is a head edge if

   - $a = orderFirst$, meaning that e has a special label indicating its role;

- $map(e) \in \mathsf{CD}.Ass$ such that $ord(map(e))$, meaning that e corresponds to an ordered association;
- there is a unique association edge $e' = (v, name(map(e)), w) \in \mathsf{G}.Edg$ such that $map(e') = map(e)$, meaning that e uniquely identifies one of the instances of that ordered association.

5. *Successor edges*, which are edges between two successive target elements of some ordered association instance. Formally, e is a successor edge if

   - $a = orderNext$, meaning that e has a special label indicating its role;
   - $map(e) \in \mathsf{CD}.Ass$ such that $ord(map(e))$, meaning that e corresponds to an ordered association;
   - there are association edges $e' = (v', name(map(e)), v), e'' = (v', name(map(e)), w) \in \mathsf{G}.Edg$ such that $map(e') = map(e'') = map(e)$, meaning that e connects the targets of two instances of that ordered association.

   The successor edges are actually the trickiest part of the flattening, since the identity of the association instance source (v' above) cannot be derived from the edge e. For that reason, our encoding of ordered associations is only good enough if no ordered association targets are shared in the instance models. In the context of our paper this is ensured by the fact that we only use ordered *aggregations*: absence of sharing is one of the characteristics of aggregates. In a definition of graph instances for arbitrary class diagrams, the encoding of the ordering relationship needs to be more elaborate, involving for instance special list nodes.

An example can be found in Fig. 5.2.

Summarizing, there are five types of edge in the graph, roughly corresponding to, respectively, the associations of the class diagram, the attributes, the subtyping relation imposed by inheritance, and (for the last two) the ordering of ordered multiple associations. Multiplicities are only modelled in the instance graph through further constraints on the edges.

5.2.2 Typings

To formulate the remaining constraints, which hammer down the details of the subtyping, ordering and multiplicities, we introduce the concept of the *value set* of a given association, say $A \in \mathsf{CD}.Ass$, for a given node, say $v \in \mathsf{G}.Nod$:

$$map.Val(v, A) = \{tgt(e) \mid e \in \mathsf{G}.Edg, src(e) = v, map(e) = A\} \ .$$

Definition 5.3 (typing) *Let* G *be a graph and let* CD *be a class diagram. A pre-typing map from* G *to* CD *is called a* typing *if the following constraints are satisfied for all* $v \in \mathsf{G}.Nod$*:*

1. Association multiplicity. *For all* $A \in \mathsf{CD}.Ass$ *such that* $(map(v), src(A)) \in \mathsf{CD}.Ext^*$*, if* $mult(A) = [i, j]$ *then* $i \leq |map.Val(v, A)| \leq j$ *(meaning that the number of* A*-values for* v *is between* i *and* j*);*

2. Attribute completeness. *For all* $A \in \mathsf{CD}.Att$ *such that* $(map(v), src(A)) \in \mathsf{CD}.Ext^*$*,* $|map.Val(v, A)| = 1$ *(meaning that* v *has a value for* A*);*

3. Subtyping. *For all* $C \in \mathsf{CD}.Cls$ *such that* $(map(v), C) \in \mathsf{CD}.Ext^*$*, there is a type edge* $e \in \mathsf{G}.Edg$ *with* $src(e) = v$ *and* $map(e) = C$ *(meaning that* v *has type edges for all generalizations of its actual type);*

4. Association ordering. *For all* $A \in \mathsf{CD}.Ass$ *such that* $ord(A)$ *and* $(map(v), src(A)) \in \mathsf{CD}.Ext^*$*, if* $|map.Val(v, A)| > 0$ *then there is exactly one head edge* $e \in \mathsf{G}.Edg$ *with* $src(e) = v$ *and* $map(e) = A$*, and furthermore,* $map.Val(v, A)$ *is totally ordered by the successor edges* $\{e' \in \mathsf{G}.Edg \mid src(e') = v, map(e') = A\}$*, with least (i.e., first) element* $tgt(e)$*.*

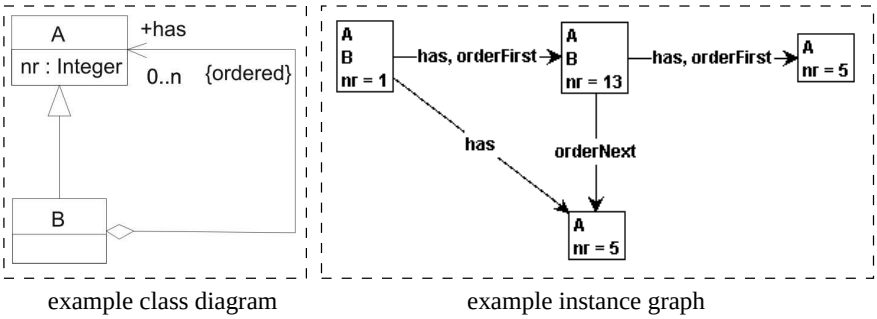


Figure 5.2: An example flattening.

This finally gives rise to the notion of instances we were after.

Definition 5.4 *A a graph* G *is called an* instance *of a class diagram* CD *if there exists a typing map from* G *to* CD*.*

For instance, the graph on the right hand side of Fig. 5.2 is an instance of the class diagram of the left hand side of the figure. Fig. 5.3 shows a fragment of the flat abstract syntax graph of our example program.

5.2.3 Discussion

Note that Definition 5.4 does *not* require that the typing is explicitly given, merely that it exists. This is motivated by the chosen graph transformation tool, GROOVE (see Sect. 8), which does not support explicit typing. On the other hand, for our purpose it is not desirable that the typing is ambiguous: if a graph is an instance of a class diagram, we want to be able to derive beyond a doubt what class a given node has, and what association a given edge is an instance of. Under Definition 5.4, this is not possible for arbitrary graph instances and class diagrams. For instance, if an attribute or association is overloaded in a subclass, then it is not necessarily clear from an edge label which of the associations the edge should be mapped to. Another possible source of ambiguity is the encoding of association ordering, especially when the ordered nodes occur as targets of different associations.

However, if we rule out the possible sources of ambiguity, by restricting the graphs to those where ordered nodes are unshared and the class diagrams to those where attributes and associations are not overloaded, and imposing some further (weak) restrictions on the uniqueness of names of associations and attributes, then we can prove the following result, which states the uniqueness of typings.

Theorem 5.5 *Let* G *be a graph and* CD *a class diagram. If the following conditions are met, then there is at most one typing map from* G *to* CD*.*

1. CD *contains no classes, attributes or associations named orderFirst or orderNext;*
2. CD *contains no class* C *and association* A *such that* $name(C) = name(A)$*;*
3. CD *contains no two distinct attributes or associations* A_1, A_2 *with* $name(A_1) = name(A_2)$ *and* $(src(A_1), src(A_2)) \in Ext^*$*;*
4. G *contains no two distinct pairs of edges* $(v_1, orderFirst, w)$,$(v_2, orderFirst, w)$ *or* $(v, orderNext, w_1)$,$(v, orderNext, w_2)$ *or* $(v_1, orderNext, w)$,$(v_2, orderNext, w)$*.*

Proof sketch. The node mapping part of *map* is unambiguous because *name* is injective on *Cls* and therefore the target class of any node can be derived from its type edges. Regarding the edges, we have to argue that the category of any edge (association edge, attribute edge etc.) is uniquely

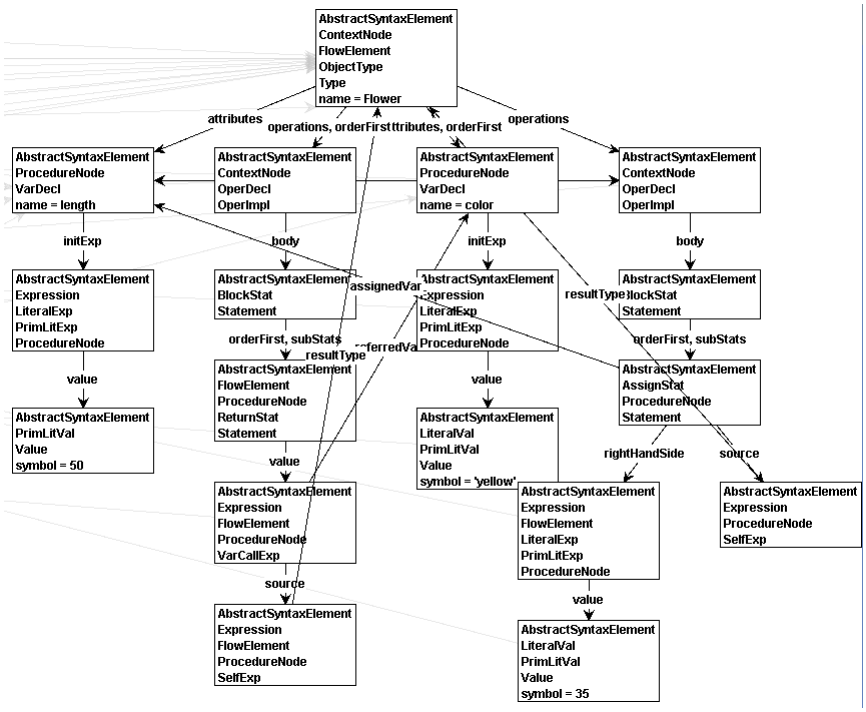


Figure 5.3: Fragment of the flat ASG for the vase *program.*

determined, and that for each edge from any of these categories, the image under the edge part of *map* is unambiguous.

Regarding the uniqueness of the categorization of the edges, conditions 1 and 2 in the theorem ensure that the label of an edge uniquely determines its category. The non-ambiguity within each of the categories can be argued as follows:

1. *Association edges.* Follows from Condition 3 of the theorem;
2. *Attribute edges.* Follows from Condition 3 of the theorem;
3. *Type edges.* Follows from the injectivity of *name* on *Cls*;
4. *Head edges.* Follows from Condition 4 of the theorem;
5. *Successor edges.* Follows from Condition 4 of the theorem. □

It is interesting, but for our study of no further relevance, to observe that here we encounter a conceptual clash between the worlds of MDA-type modelling and graphs. In the OMG terminology, class diagrams would be considered as M1-level models, and so the graphs that instantiate them are instance-level and belong to M0. In particular, this is true of any concrete flat abstract syntax graph that is an instance of the combined class diagrams in Sect. 4. Yet we will see in the remainder of this report (Sect. 7) that the simulation of a program is defined in terms of so-called *execution graphs*, which are conceptually a meta-level lower still. (One way to resolve this terminology clash is to point out that, in the context of this work, the abstract syntax class diagram actually plays the role of a *meta*-model and so really belongs to M2.)

5.3 Flattening

We have set up the theory above to support the notion of flattening. However, the flattening process itself transforms an *instance* of a class diagram into a Flat ASG, where the notion of instance is that of the Java programming language. In our transformation process (Fig. 2.1), such instances are not available in diagrammatic or mathematical form but as run-time data structures in the plug-in developed for the purpose of this work (see Sect. 8). Yet we believe that by using

the graph encoding discussed above as a guideline, flattening is in principle well-defined. Our implementation of this step is in fact fully model-driven: it does not only work for the abstract syntax graph model but for arbitrary class diagrams.

In the context of this report we do not see an advantage in providing a mathematical model for the Java data structures used in the plug-in, let alone for the flattening transformation itself. (In fact, the status of such models would be disputable in any case: first of all, it should be somehow established that the data structure model really captures the semantics of Java, and second, even when formalized, the flattening transformation would still have to be implemented, with again the discrepancy between the mathematical and programming worlds.)

5.4 Graph Transformation

In the next two sections we will describe how the Flat ASG is used in specifying the semantics of the program being modelled by it. For this purpose we apply *graph transformations*, specified by so-called *graph production rules*. Here we briefly discuss the principles behind graph transformation, in the form implemented in the GROOVE tool set (see Sect. 8.2).

5.4.1 Graph Production Rules

Traditionally, graph production rules consist of a *left hand side* (LHS) and a *right hand side* (RHS). Applying a graph production rule, say $r : \mathsf{LHS} \rightarrow \mathsf{RHS}$, transforms a *source graph* G into a *target graph* H by matching LHS in G, which comes down to finding an image in G of the structure described by LHS, and replacing this LHS-image by RHS. The replacement, however, is not complete: wherever LHS and RHS overlap the structure is preserved.

In this paper, production rules are extended by *negative application conditions* (NACs, see [20]), which are extensions of LHS that serve to limit the applicability of a rule: r will only be applied to a source graph G, for a given image of LHS in G, if that image *cannot be extended* to a matching of any of r's NACs.

Rather than giving LHS, RHS and the NAC's of a rule as separate (overlapping) graphs, GROOVE offers a representation of all these elements in a single graph, where the roles of the different elements are distinguished by a color and shape coding. Four different types of elements can occur in such rule graphs, namely:[1]

- *reader*-elements, being the elements that occur in both LHS and RHS. Reader-elements are represented by solid black arrows and rectangles.
- *eraser*-elements, being the elements that only occur in LHS. Eraser-elements are represented by dashed blue arrows and rectangles.
- *creator*-elements, being the elements that only occur in RHS. Creator-elements are represented by solid green arrows and rectangles.
- *embargo*-elements, making up the NACs. These elements prohibit the application of r when they exist at the prospective matching in G. Embargo-elements are represented by dashed red arrows and rectangles.

Fig. 5.4 (i) shows an example modelling a circular buffer and Fig. 5.4 (ii) shows a graph production rule that puts an element into such a buffer. The rule requires an empty slot. After applying this rule, this first empty slot will be assigned an object and will become the last non-empty slot of the buffer.

A *graph production system*, finally, is a set of graph production rules, each of which describes a step of a more complex transformation. To distinguish between the rules in a graph production system, we assume all the rules to be uniquely named; we use N_r to refer to the name of a rule r.

[1] Obviously the colours do not show up in a black-and-white presentation: there the red and green are, respectively, dark and slightly lighter gray, whereas the blue is almost indistinguishable from black.

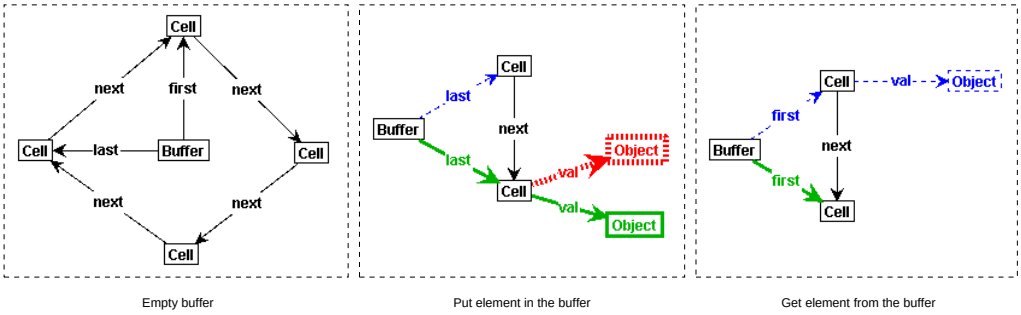


Figure 5.4: Initial graph and production rules for a circular buffer.

5.4.2 Graph Transition Systems

When a production rule is applied to a graph, obviously this results in another graph. We write $G \xrightarrow{r,m} H$ to denote that graph G has been transformed into graph H by applying production rule r; the component m stands for the matching that identifies the occurrence of r's LHS in G. We refer to the literature on graph transformation (e.g., [13]) to give the formal definition of H's construction from G using r and m.

Given a graph production system $\mathcal{R}$ and a start graph G, by recursively applying all rules in all possible ways, we obtain a set of derived graphs: formally, the smallest set $\mathcal{S}$ such that

- $G \in \mathcal{S}$, i.e., the start graph is there;
- If $H_1 \in \mathcal{S}$ and $H_1 \xrightarrow{r,m} H_2$ for some rule $r \in \mathcal{R}$ and matching m of r's LHS in H_1, then there is a $H_3 \in \mathcal{S}$ that is *isomorphic* to H_2.

The set $\mathcal{S}$ together with the indexed relation$\longrightarrow$ gives rise to what we call a *graph transition system*. We will display graph transition systems as graphs themselves by taking the elements of $\mathcal{S}$ as nodes, and adding an edge (H_1, N_r, H_2) whenever $H_1 \xrightarrow{r,m} H_3$ for some m and some H_3 isomorphic to H_2. An example is given in Fig. 5.5 which shows the graph transition system resulting from the initial graph and the rules in Fig. 5.4. That is, the initial graph corresponds to the node marked "start" in Fig. 5.5, and the other nodes in the figure represent graphs in which respectively 1, 2, 3 and 4 cell nodes of the buffer contain a val-edge to an Object-node. Two of those states are also shown explicitly in the picture. The transitions correspond to applications of the rules in Fig. 5.4.

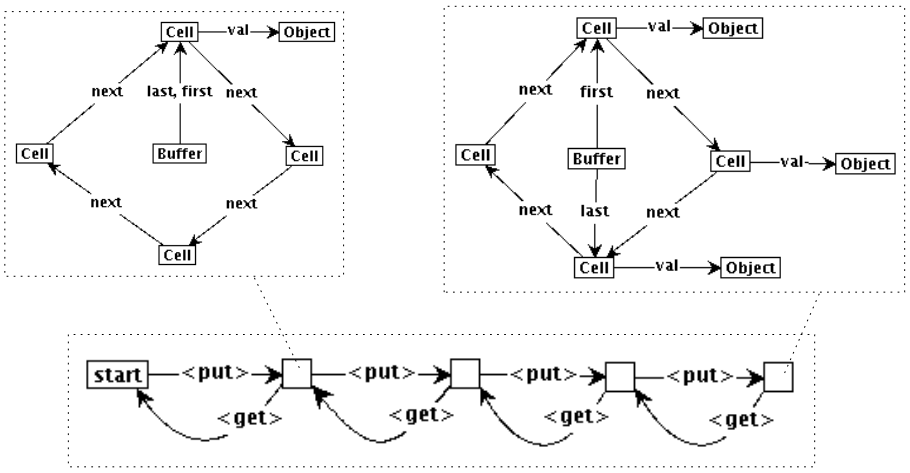


Figure 5.5: A graph transformation system, with some example states.

6 Program Graphs

A program graph is the start point for correctly simulating the execution of the system. Therefore, as described in Sect. 2, a program graph contains the following information:

- the *flat abstract syntax graph*, described in Sect. 5, modelling the required elements of the concrete syntax in terms of nodes and edges representing abstract syntax elements;
- *flow graphs*, modelling the sequential execution relation between executable statements.

Given the Flat ASG, we construct the corresponding Program Graph by applying graph production rules which add control flow information to the Flat ASG. In this section we will firstly define the concept of flow graphs and then discuss their structure as they appear in this report. Next, the general principles underlying flow graph construction will be discussed. Then a number of graph production rules concerning flow graph construction are shown and explained. The last part of this section demonstrates what the flow graph construction process results in, by applying the rules on the Flat ASG of a small part of the example program from Sect. 3.

6.1 Flow Graphs

We will start with a definition of what we call flow graph in this report.

Definition 6.1 (flow graph) *A* flow graph *is a directed graph consisting of three types of nodes (also called* flow elements*), namely* procedure, predicate *and* context nodes*, connected by different types of* successor-*edges. Procedure and predicate nodes represent executable statements, the context nodes represent the start and end point of each flow graph; the successor-edges represent the sequential relation between between statements. For every flow graph the following properties hold:*

- *a flow graph has exactly one context node which has one outgoing and one incoming successor-edge;*
- *every procedure node has exactly one outgoing successor-edge;*
- *every predicate node has exactly two outgoing successor-edges.*

A flow graph can best be understood in relation to a *locus of control*, which is a node of the execution graph (discussed in detail in the Sect. 7) standing for a thread of execution. Control is said to be *at* a node of the flow graph. The successor-edges indicate where control should go after it leaves the current flow graph node.

The structure of flow graphs, as they will eventually appear in Program Graphs, is shown in Fig. 6.1. Flow graphs are built up from flow elements that are connected by successor-edges. Flow elements represent either executable statements or abstract syntax elements that manage control flow. From this figure it also becomes clear what kind of successor-edges can appear between different flow elements. Successor-edges have one of the labels flowNext, flowTrue or flowFalse. The FlowTmp-node with its outgoing flowIn-edge will be discussed in Sect. 6.2.

Flow graphs, traditionally [16], have two special distinct nodes: the *start node* and the *end node*. In our approach we collapse these two nodes in the context node of a single flow graph. This means that every context node represents the start node of a flow graph as well as its end node. Using this approach, every flow graph is cyclic. This cyclic property will be discussed in more detail in Sect. 7.

In the next paragraphs we will discuss the three different types of flow elements in more detail. In Sect. 6.2 we will say more about the structure of flow graphs during the construction process.

Procedure Nodes. Procedure nodes represent statements after which it is deterministic which statement to execute next: control flow does not depend on the evaluation of a condition. The syntax elements that can serve as a procedure node are shown in Fig. 6.2 (i). One element that seems a bit out of place here is the VarDecl-element since it is neither a Statement nor an Expression. When discussing the ContextNode it will become clear why we model it as a procedure node.

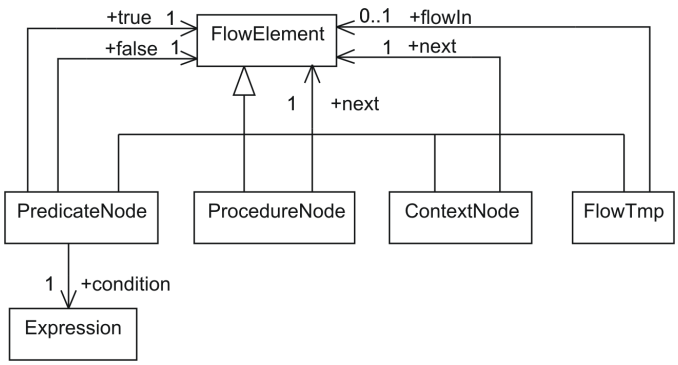


Figure 6.1: Flow graph meta-model.

Predicate Nodes. Statements that are represented by predicate nodes are related to a condition, which will be evaluated to either true or false. The actual value of the condition determines which branch will be taken. In this paper we only consider predicate nodes with two outgoing successor-edges: one for the case the condition is evaluated to true and one for the case the condition is false. As Fig. 6.2 (ii) shows, in TAAL there are only two kinds of statements for which a conditional expression needs to be evaluated: the WhileStat and the ConditionalStat (if-then and if-then-else).

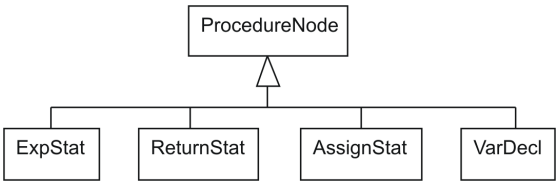
Context Nodes. Flow graphs, in this report, appear in three different contexts, namely:

- Program context. Program flow graphs control the startup of the program being modelled. In TAAL, program startup is modelled by the execution of the initial expression of the program. This expression may require other kinds of statements to be executed. A program's initial expression will typically be build up from an expression which instantiates an object from which a method will then be called (this is the case in the example shown in Sect. 4). The Program flow graph will therefore be quite small in most cases. A program graph always contains exactly one flow graph at Program context.

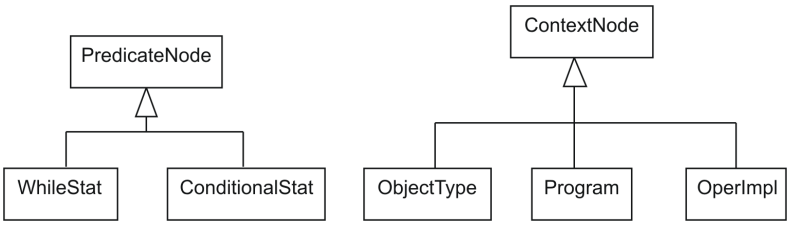
- ObjectType context. ObjectType flow graphs are traversed when an object is instantiated. Object creation will be discussed in detail in Sect. 7. Now it becomes clear why the VarDecl-element is a subclass of ProcedureNode: instantiating an object means that its instance variables need to be created and assigned their initial value. Instance variables declarations are represented by VarDecl-elements and therefore VarDecl-elements are part of ObjectType flow graphs. A program graph contains a ObjectType flow graph for each ObjectType being specified in the original program.

- OperImpl context. OperImpl flow graphs control the execution of the body of operations. The execution of an operation can be split up into a number of phases: (1) evaluating the parameters, (2) calling the operation, (3) dynamically looking up the corresponding implementation, (4) passing the actual parameters, (5) instantiating the local variables and (6) executing the body. An OperImpl flow graph only takes care of the last two phases. The first and second phase are part of another flow graph in which the operation is called. The third and fourth phase will be discussed in detail in Sect. 7.4.2. A program graph contains a OperImpl flow graph for each operation that has been implemented in the original program, including the operation implemented in the standard library.

Context nodes are important when discussing the flow graphs in the different contexts mentioned before. Fig. 6.2 (iii) shows what types of nodes can appear as context nodes. In special cases, such as an ObjectType without attributes, a flow graph can exist of *only* the context node. In those cases, the context node still has a successor-edge, which in this case points back to the context node, making the cyclic property even clearer.



(i) Procedure nodes



(ii) Predicate nodes (iii) Context nodes

Figure 6.2: Flow elements and their subclasses.

Although you would expect that program execution is modelled by one single flow graph, a program graph contains several flow graphs at the different contexts. The different flow graphs are not directly connected to each other. Fig. 6.3 gives an impression of the occurrences of different flow graphs not being connected to each other. Each highlighted cycle represents a single flow graph at a specific context. The different flow graphs are 'connected' to each other when simulating the program. The Program flow graph is connected to an ObjectType flow graph by executing a CreateExp element. The Program flow graph will also be connected to an OperImpl flow graph, when a specific operation of the instantiated ObjectType is called. OperImpl flow graphs can also be connected to each other, when one OperImpl flow graph contains an OperCallExp element calling another operation.

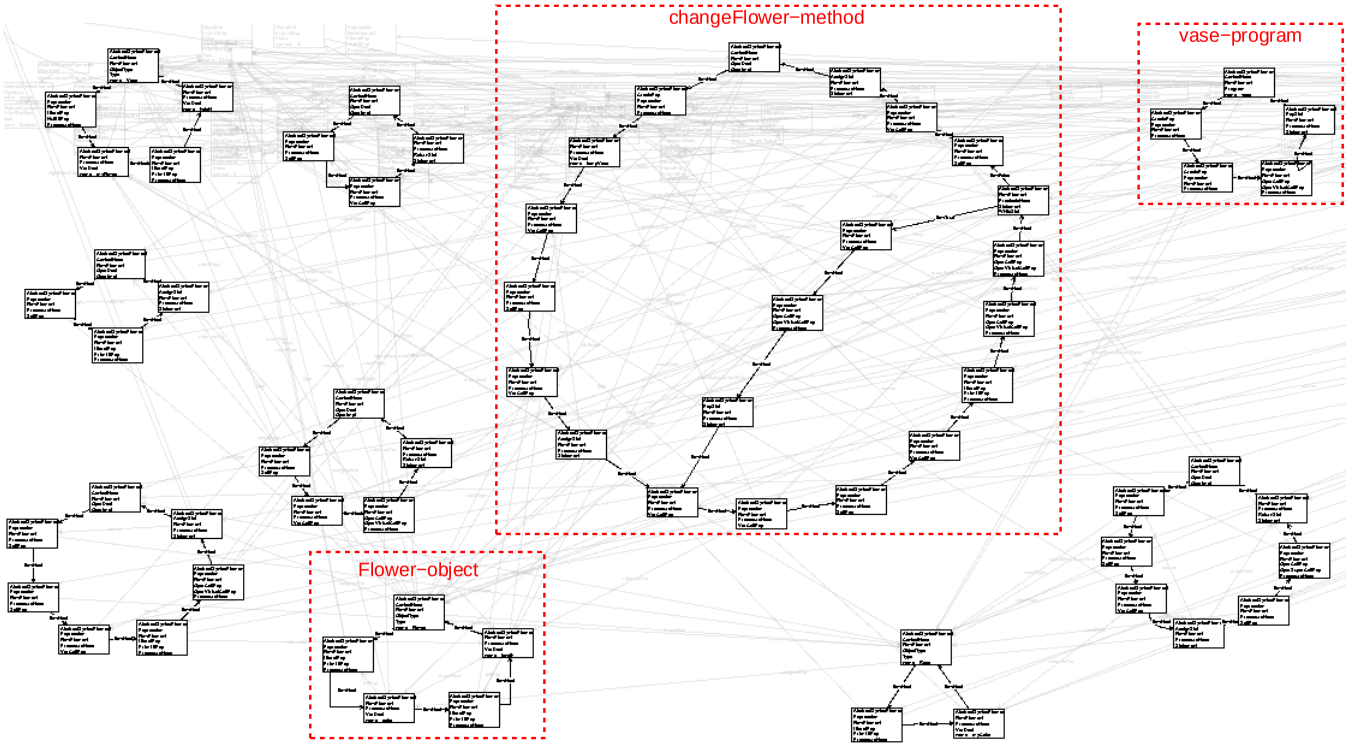


Figure 6.3: Part of the program graph from the example showing the separated flow graphs.

In the rest of this section we will refer to the nodes of flow graphs by writing 'procedure node(s)', 'predicate node(s)' or 'context node(s)' in case there is a difference. When discussing properties that hold for all three types of nodes we write 'flow element(s)'.

6.2 Flow Graph Construction

The flow graph is implicit in the abstract syntax graph; it can be derived from the abstract syntax graph by appropriately interpreting the various kinds of statements and expressions. It follows that some of the statements originally in a program do not give rise to flow graph nodes at all; in particular, this is true for block statements. There are a number of general principles underlying the generation of flow graphs.

To extract the flow information from the abstract syntax graph, we apply a graph production system that traverses the syntax graph in a top-down fashion. The general approach is that for every type of statement we specify a graph production rule. Those various transformation rules have a lot in common. Each of these rules contains a node representing the statement type involved. These nodes may or may not have subparts being statements or expressions. If a statement has subparts, the relation with its subparts has one of the following characteristics:

- *containment*-relation. In this case, the statement itself does not affect the program state, but its subparts do. This means that the execution of such a statement is defined in terms of the execution of its subparts. An example of such a statement is the block statement: a block statement itself can not be executed, but its substatements can.
- *dependence*-relation. In case of a dependence-relation, the result of the execution of the main statement depends on the results of its subparts. This means that the substatements need to be executed before the main statement. An example of such a statement is expression statement representing a method call with a number of parameters. The result of such a method call depends on the values of the parameters that will be passed to that called method.

During flow graph construction, four types of flow edges occur, namely flowIn, flowNext, flowTrue and flowFalse. Eventually, the last three types of flow edges, together with their source and target nodes, represent flow graphs. The flowIn-edges are temporary edges that direct flow graph construction. The flow graph construction process starts by creating a flowIn-edge from a ContextNode to its 'sub flow element' and a flowNext-edge backwards as shown in Fig. 6.4 (i). The label <relation> in this figure can be one of the three labels startExp, attributes or body. The graph production rules for each of the abstract syntax elements then specify how and which flow elements must be created and how to pass on the flowIn-edge to its subelements (if there are any). If a flowIn-edge points to the first element of an ordered list (e.g. the Statement's contained in a BlockStator the Expression's serving as parameters for an OperCallExp), the flowNext-edge will be propagated along the ordered list as shown in Fig. 6.4 (ii). This rule contains a number of negative application conditions. These are needed because propagation of the flowNext-edge along an ordered list is only allowed if the element that has both an outgoing flowNext-edge and an outgoing orderNext-edge is not a ContextNode. Otherwise this rule would also apply on the ordered relationship of ObjectType's being declared in a Program or on the ordered list of OperImpl's of an ObjectType. Another restriction is that that the next FlowElement of the ordered list may not yet have an outgoing successor-edge, otherwise this rule will be applicable infinite many times.

In some cases we need an additional flow element type in order to construct correct flow graphs. This node will be called FlowTmp. The reason for this is that sometimes it is not at forehand known what FlowElement will eventually be the next to execute, because we still need to construct the sub flow graph of that particular FlowElement. A FlowTmp element can have three different types of incoming edges, namely flowNext, flowTrue and flowFalse. When a FlowTmp element is created it has an outgoing flowIn-edge which enables other graph production rules. Eventually, the flowIn-edge will be replaced by a flowNext-edge. At that time you know what FlowElement succeeds the FlowTmp and therefore the FlowTmp element can then be deleted. This deletion must

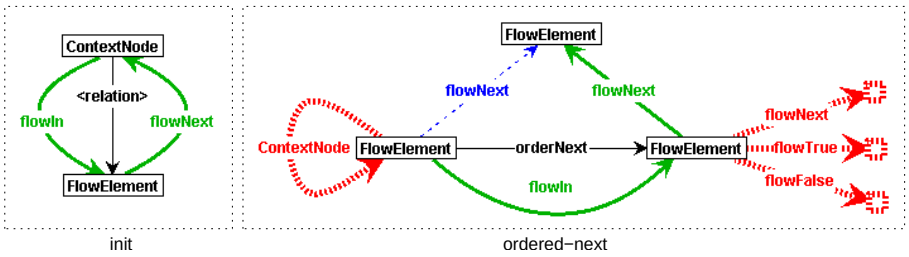


Figure 6.4: Flow graph construction for subparts.

be properly specified since in special situation, the FlowTmp element can have multiple incoming edges which, after deletion, all need to point to the FlowTmp's successive FlowElement. This proper deletion is specified by Fig. 6.5. Basically, the FlowTmp element and its successive FlowElement will be merged. Merging two nodes is specified as follows:

- all incoming edges of the first node will be incoming edges of the new node, including self-edges of the first node
- all outgoing edges of the second node will be outgoing edges of the new node, including self-edges of the second node
- all edges between both nodes will be self-edges of the new node

Since the FlowTmp-label (of the self-edge of the FlowTmp element) and the flowNext-label (of the edge between the two successive elements) may not be preserved they are deleted explicitly. The FlowTmp element will be used in the flow graph construction rules of the WhileStat, the ConditionalStat and the OperImpl.

6.2.1 Constraints.

At this point we have discussed all the elements from the flow graph meta-model from Fig. 6.1. There are still some additional constraints.

1. a FlowElement cannot have multiple incoming flowNext-edges together with one or more incoming flowTrue-edges. The reason for this is because if a FlowElement has an incoming flowTrue-edge this means that this FlowElement is part of a BlockStat which will only be executed if a particular condition evaluated to true. In such a case, control can never reach the first executable substatement or expression through a flowNext-edge;
2. a FlowTmp element has either a single outgoing flowIn-edge or a single outgoing flowNext-edge.

6.3 Graph Production Rules

In this section we will discuss a number of graph production rules that give a good impression of how flow graphs are constructed.

Program. The graph production rule that triggers the creation of the flow graph in Program context, as described in Sect. 6.2, is shown in Fig. 6.6. In this figure the node labelled Program is the root of the program. From this node, we can access the initial expression of the program pointed to by an edge labelled startExp. Generating the flow graph at program level is done by creating a flowIn-edge pointing to the node representing the initial expression and creating a flowNext-edge pointing back to the Program-node indicating that the Program-node is the end node of this flow graph. This rule is only applicable if the process for generating the flow graph for this Program-node has not yet started (i.e. the flowIn-edge is not present) or finished (i.e. the flowNext-edge is not present).

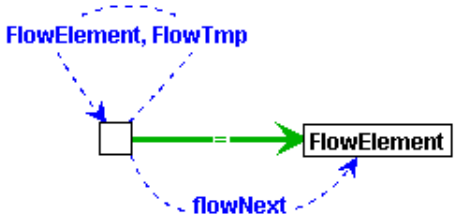


Figure 6.5: Graph production rule removing the FlowTmp *element.*

ObjectType. Flow graph construction for ObjectType elements is twofold. We need to check whether the object has attributes or not. The left hand side rule of Fig. 6.6 shows what to do in the former case; the right hand side rule specifies flow graph construction for the latter case.

OperImpl. Flow graph generation at operation-level is also twofold: an operation may or may not have local variables. In the former case, control must first be passed along the declarations of the local variables before entering the body of the operation; in the latter case, control can directly be passed on to the first statement in the body. Fig. 6.6 shows how we take care of operations that do have locally declared variables. The rule on the left hand side of the figure shows that a new element called FlowTmp is created which controls the sequencing of the two flow graphs. Eventually, the rule on right hand side will remove this element again.

WhileStat. The flow graph of a WhileStat element contains a loop: after each time the body of the WhileStat is executed the condition must be evaluated. The outgoing successor-edges of a WhileStat will be labelled flowTrue and flowFalse. The rules for creating the flow graph of the body of the WhileStat, however, require a flowIn-edge. This contradiction is solved by creating the flowTrue-edge pointing to a FlowTmp element which will eventually (i.e. when it has an outgoing flowNext-edge) be removed like shown in Fig. 6.5. Fig. 6.7 shows how to construct the flow graph for a WhileStat properly. In order for this rule to be applicable there must already be a flowNext-edge pointing from the WhileStat element to the flow element to which control must flow in case the condition evaluates to false. This flowNext-edge can then be replaced by a flowFalse-edge. Evaluation of the condition will be performed when the WhileStat is reached for the first time, but also when the body of the WhileStat has been executed. This means that control flows to the conditional expression from two distinct FlowElement's. Construction of the flow graph of the conditional expression requires only a single flowIn-edge. Therefore we also create a FlowTmp element at this point which has two incoming flowNext-edges and one outgoing flowIn-edge. This flowIn-edge will eventually turn into a flowNext-edge which makes the rule shown in Fig. 6.5 applicable for properly removing the FlowTmp element.

ConditionalStat. When construction the flow graph for a ConditionalStat we encounter similar problems as for the WhileStat. Fig. 6.7 shows how to construct the flow graph for a ConditionalStat having both a thenPart and an elsePart. This rule also requires a flowNext-edge to exists between the ConditionalStat and the FlowElement that will be reached after executing either the thenPart or the elsePart.

BlockStat. The flow graph of the BlockStat element is generated by generating subsequently generating the flow graphs of all its substatements. This is done by passing on the flowIn-edge to its first substatement (if it has any), as shown in Fig. 6.8. The rules shown in Fig. 6.4 (ii) then take care of adding all other substatements to the flow graph in the right order. The BlockStat element itself cannot be executed and therefore it can be stepped over. This is specified by the rule by replacing the outgoing flowNext-edge from the BlockStat element by a flowNext-edge from its substatements pointing to the same next FlowElement.

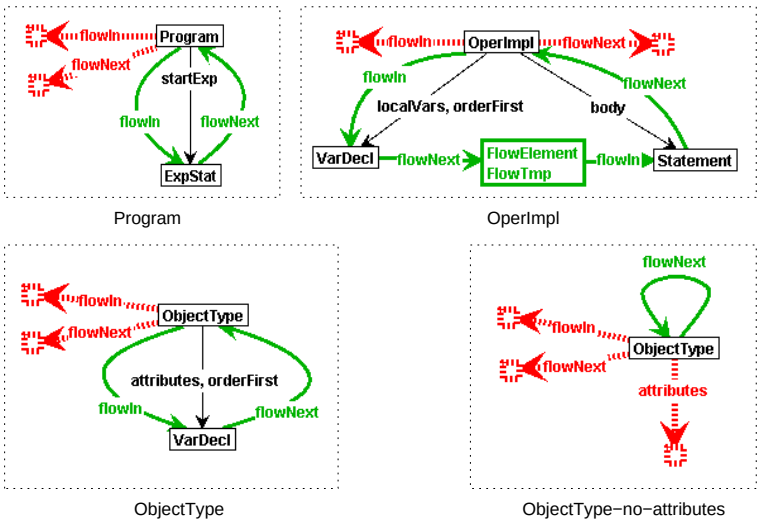


Figure 6.6: Flow graph construction rules (I).

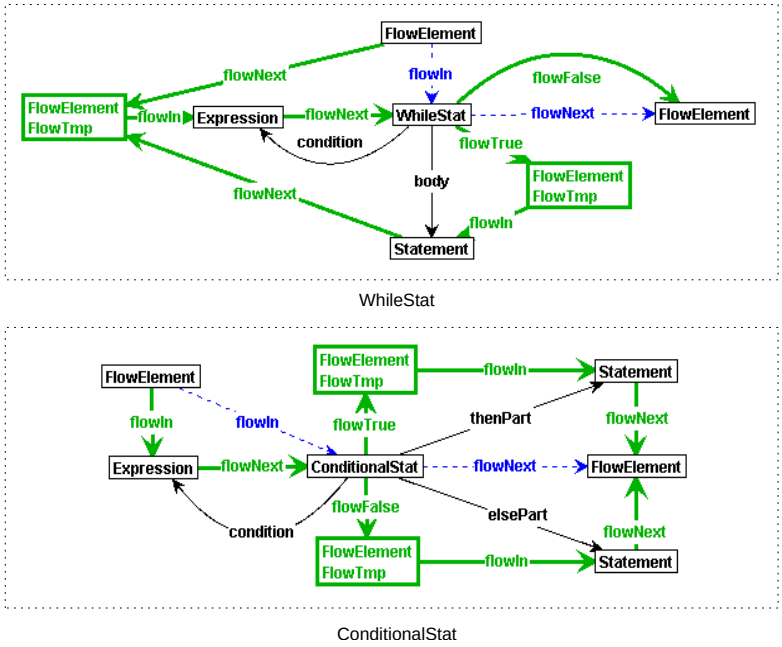


Figure 6.7: Flow graph construction rules (II).

OperCallExp. The rule shown in Fig. 6.8 specifies how to generate flow graph elements for a call to an operation given a number of parameters. The source of the operation needs first to be created, then the values for the parameters must be evaluated and eventually the operation can be called.

VarDecl. The graph production rule that specifies what graph elements to generate for variable declarations is shown in Fig. 6.8. We already discussed that, at this point, every VarDecl has an initial expression. If the variable declaration in the original program does not specify its initial value, we assign the null-value to it in case of a reference type and in case of a primitive type we assign the default value of the corresponding primitive type. Before being able to assign that value to the variable, we still have to evaluate that initial value since it can also be a more complex expression which reads other variables or calls specific operations. Therefore, control is first passed on to the initial expression after which control flows to the VarDecl element itself.

PrimLitExp. An expression that is of primitive type simply holds a specific value of that type. It does not depend on or contain other statements. In those cases, the flowIn-edge can simply be replaced by a flowNext-edge like specified by the rule shown in Fig. 6.8. This figure shows the graph production rule for a LiteralExp but since the rules for PrimLitExp and NullLitExp this has been abstracted one level up by specifying the rule for their common superclass LiteralExp. An analogue rule also applies for CreateExp, NullLitExp and SelfExp elements.

The discussed graph production rules together with the remaining ones are shown in Appendix B.1 in alphabetical order.

6.4 Example: From Flat ASG to Program Graph

In this section we will give a short impression of how the flow graph construction rules transform a Flat ASG into the corresponding Program Graph by applying them on a small example. We will show the result of the flow graph construction process. We will focus on the construction of a flow graph in an ObjectType context. The figures will only show those elements that are part of that single flow graph since showing the entire Flat ASG and Program Graph would not make any sense.

Listing 1 specifies that for every flower in the vase we store the flower's color and length; the color-attribute being of type String and having an initial value 'yellow', the length-attribute being of type Integer assigned an initial value of 50.

```
program vase
...
class Flower
  color: String := 'yellow';
  length: Integer := 50;
  ...
endclass
...
endprogram
```

Listing 2: Part of a TAAL-program specifying the data-structure of type Flower.

In Fig. 6.9 the elements of the Flat ASG concerning the data-type Flower are shown. You can easily recognize the two attributes and their ordered relationship. Starting from this graph, the labelled transition system (see Sect. 5.4.2) in Fig. 6.10 shows which graph production rules are applied in sequence to construct the corresponding Program Graph.

Fig. 6.11 again shows the elements of the Flat ASG concerning the data-structure of type Flower, but also the successor-edges that are added during the flow graph construction process, shown as fat, red arrows. In this figure it becomes clear that for every instance variable, control first reaches its initial expression and then the node representing its declaration.

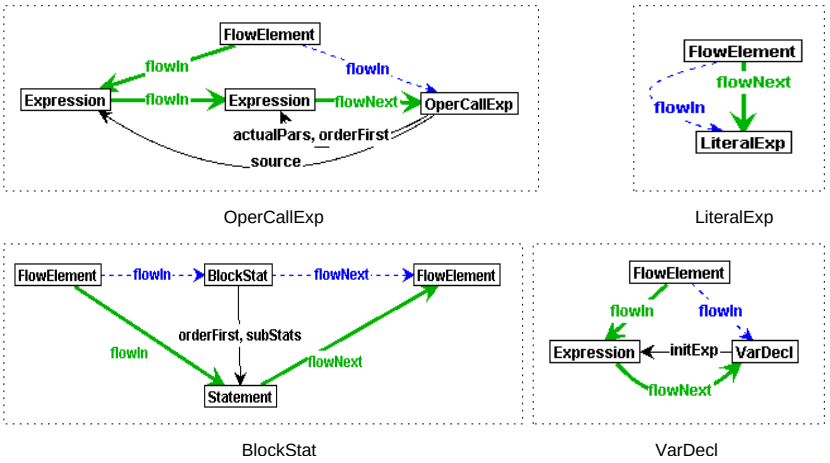


Figure 6.8: Flow graph construction rules - III.

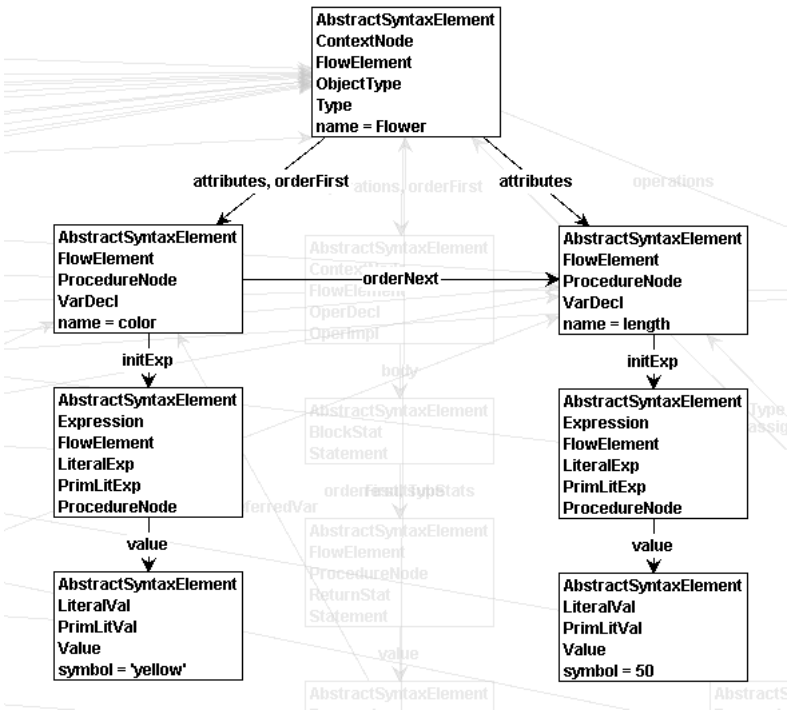


Figure 6.9: Flat ASG of the data-structure of type Flower.

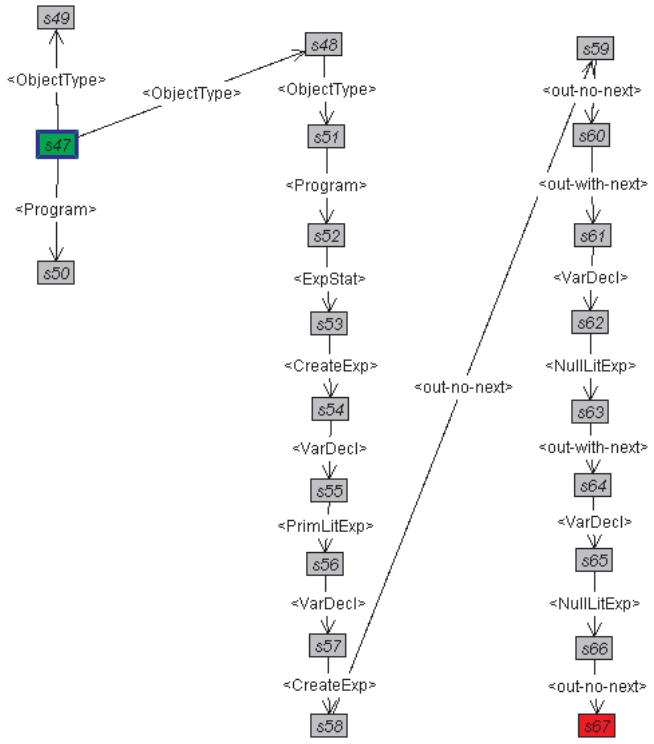


Figure 6.10: Transition steps from Flat ASG to Program Graph.

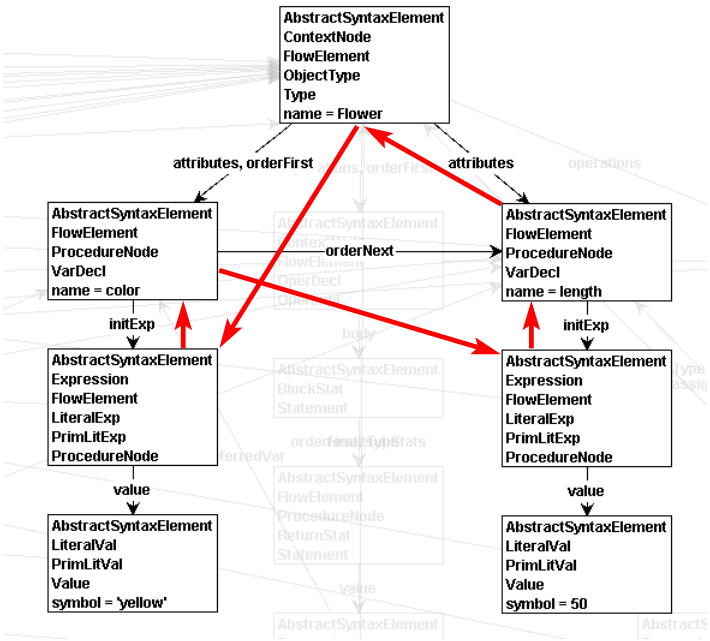


Figure 6.11: Flat ASG of the data-structure of type Flower *added with successor-edges that direct control when this type is instantiated.*

7 Execution Graphs

An execution graph essentially represents a snapshot of the state of the system, together with static context information about the program itself. Concretely, as discussed in Sect. 2, an execution graph combines three kinds of information:

- The *program graph* described in Sect. 6, which provides the required static context information;
- A *value graph*, modelling the data part of the current state. In compiler terms, this roughly corresponds to the *heap*;
- A *frame graph*, modelling the process part of the current state. In compiler terms, this corresponds to the *stack*.

It is important to realize that a given program gives rise not to a single execution graph but to a sequence of consecutive graphs, each representing the next state during execution. We use graph transformations to generate this sequence; they constitute the *simulation* referred to in Fig. 2.1.

In this section we first present meta-models of the value and frame graphs, and subsequently give an overview of the simulation rules.

7.1 Value Graphs

A value graph contains elements representing the objects that will be created and referred to during executing the program. A meta-model for the value graph is shown in Fig. 7.1.

We briefly discuss the new concepts in this meta-model.

Value. This stands for a concrete value of an arbitrary type. The type of each value is fixed and indicated by an instanceOf-edge leading from the Value-node to the corresponding Type-node (already introduced in Sect. 4). Note that this refers to primitive values as well as the null-value and object-instances; in fact the instanceOf-edge could be specialized to lead from ObjectVal to ObjectType, etcetera. This class is abstract.

ObjectVal. This is a specialization of Value used to model values of ObjectType. An object value may have attributes, which are modelled by VarSlot nodes referenced through the attributes-association. Each ObjectVal-node has precisely one VarSlot (see below) for each attribute of its ObjectType (which are given by the attributes-association of the ObjectType). In other words, for each ObjectVal-node there is a one-to-one correspondence between the VarSlots appearing as attributes-targets, and the VarDecl-nodes appearing as attributes-targets of the the corresponding ObjectType.

NullLitVal. This stands for the one and only null value, i.e., the unique value of the NullType. In other words, there will always be precisely one instance in the execution graph.

PrimLitVal. This stands for all the values of the primitive types. Conceptually there are infinitely many of them. Of course we cannot represent all those; instead, an execution graph will contain precisely those primitive values that actually occur as values anywhere in the current state.

Slot. This stands for a container of a value, occurring anywhere in the state snapshot. There are two kinds of slots: VarSlots, which correspond to variables declared in the program, and AuxSlots, which are temporary slots used to store intermediate values during evaluation of an expression. This class is abstract.

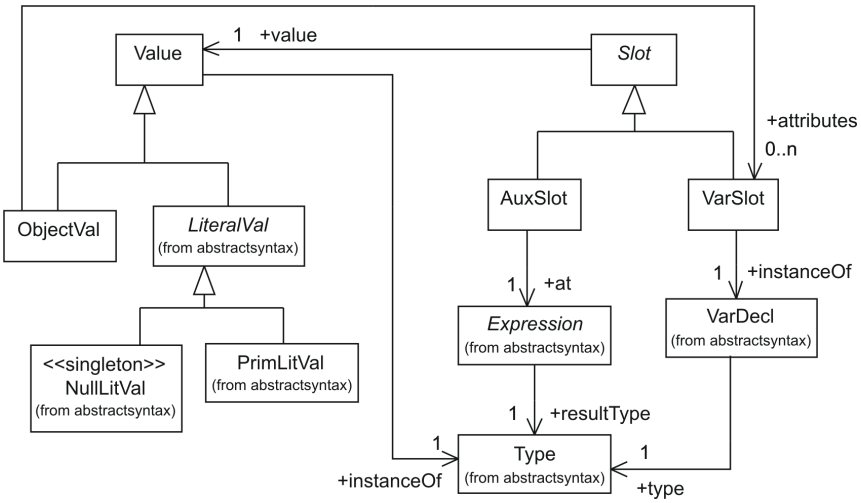


Figure 7.1: Value graph meta-model.

VarSlot. This is a specialization of Slot, viz. a container corresponding to a variable in the program. Above we have already seen that objects have corresponding VarSlot-instances for all their attributes. Similarly, in the frame graph we will see that formal parameters and local variables also give rise to VarSlot-instances. For each instance of these kinds of VarSlot, there is a corresponding VarDecl.

AuxSlot. This is a specialization of Slot, viz. a container holding a temporary value during evaluation of an expression. These traditionally correspond to stack locations. AuxSlots have no corresponding VarDecl; instead, each AuxSlot refers to the sub-expressions for which it holds a value, via an at-edge.

An example value graph is given in Fig. 7.2. This shows three objects with attributes, one of static type Flower and dynamic type Rose and two of (static and dynamic) type Vase. One of the Vases holds a reference to the Rose in its myFlower attribute. The example does not contain PrimLitVals or AuxSlots; see, however, Fig. 7.4 below.

7.2 Frame Graphs

The frame graph meta-model is shown in Fig. 7.3. It essentially introduces only one type of new node: the Frame. This stands for a dynamically created instance of a ContextNode (see Fig. 6.1), with a pointer to the current FlowElement. In fact, for each sub-type of ContextNode there is one Frame sub-type.

Frame. This is the main type for execution frame nodes. In general, a Frame controls the execution of the code at a particular ContextNode. In terms of the program graph, that code can be found as the control flow graph appended to the relevant ContextNode (depending on the kind of ContextNode, it can represent the program's initial expression, the initialization code for objects of a particular class, or the body of an operation). Each Frame has an executes-reference to the corresponding ContextNode.

A Frame controls the execution of the corresponding code by maintaining a pc-edge (where pc stands for "program counter") to the current FlowElement in the flow graph of the ContextNode. The pc-edge is moved to a successor in the flow graph at every execution step. When a method is called or a new object is constructed, a new, "nested" frame is created for it and the pc-edge is (temporarily) removed, indicating that the calling frame is passive while the nested frame is running.

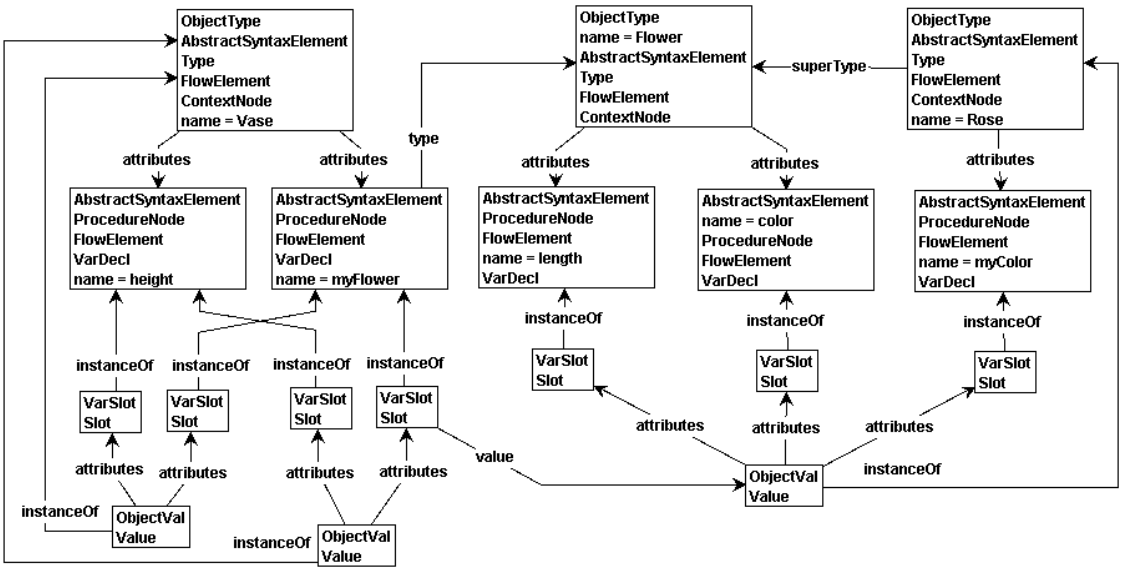


Figure 7.2: Fragment of the value graph reached after executing the vase *program (see Listing 1).*

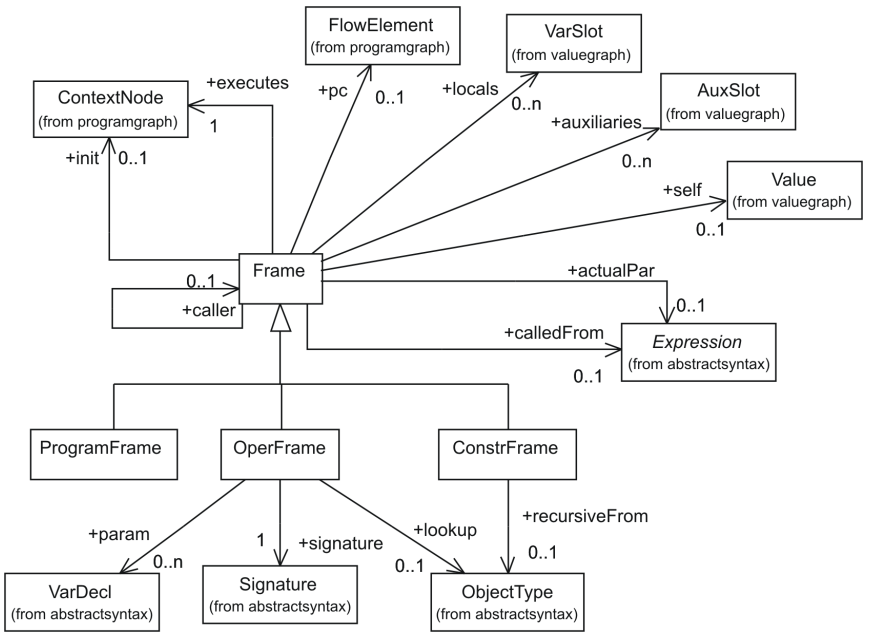


Figure 7.3: Frame graph meta-model.

In fact, each frame that does not correspond to a Program has a calling frame, referenced by a caller-edge. Moreover, where the caller gives the semantic calling context, there is also a *syntactic* calling context, viz. the location in the flow graph of the caller where its pc-edge was pointing when the current Frame was invoked. This syntactic context is stored in a calledFrom-reference. When the nested frame has finished, it is deleted and a pc-edge is re-created in the calling frame, using the calledFrom-reference to determine the correct location.

A further important aspect of Frame-instances is that they can have local and auxiliary *variables*. The former are instances of the node type VarSlot, discussed above as part of the value graph; they are referenced through a one-to-many locals-association. (Actually, the current version of TAAL has been defined such that not all types of Frame have local variables — in fact,

only OperFrames do — but we regard this as a coincidence.) Auxiliary variables are instances of AuxSlot, also discussed above. A frame may have a self-edge to the value that is the context of the operation being executed. This is usually an ObjectVal, but for built-in operations it may be a primitive value; hence the type is Value. (Again, in TAAL self-edges only occur for OperFrames and ConstrFrames.)

Finally, there is an auxiliary edge actualPar that is used for the purpose of passing parameters at operation calls; see Sect. 7.4.2 below.

ProgramFrame. This is the sub-type of Frame modelling the execution of the entire program. To denote that fact, every instance has an executes-edge to the (unique) Program-node. A ProgramFrame is only active when the program starts; the initial statement usually creates an object and invokes a method upon it. When control returns to the ProgramFrame, the program stops.

OperFrame. This is the sub-type of Frame modelling the execution of an operation. The signature of the operation being executed is known at compile time for every operation; in this graph this is indicated by a signature-edge. The actual operation implementation being executed (which is looked up dynamically, see Sect. 7.4.2 below) is indicated by an executes-edge to the corresponding OperImpl-instance. Furthermore, an OperFrame is always *called* from another frame, and hence has a caller-edge to its calling frame. Furthermore, in order to be able to reconstruct the pc-edge in the calling frame after this one terminates, the OperFrame also records the FlowElement from which it was called, through a calledFrom-edge.

In addition, there are a number of auxiliary edges, which are used during the process of method invocation; see Sect. 7.4.2 below.

- lookup, used to signal the phase of dynamic method lookup, from the start of the method invocation until the appropriate method implementation has been found (see Sect. 7.4.2);
- init, used to signal the phase of parameter passing, which takes place after method lookup but before the actual execution of the method body (see Sect. 7.4.2);
- param, which is used during the parameter passing phase to point at the formal parameter being initialized (see Sect. 7.4.2). (The corresponding actual parameter is indicated by the actualPar-edge at the calling frame; see above.)

There are several consistency requirements on the local graph structure of an OperFrame:

- all the VarSlot-instances reachable through locals have an instanceOf-edge pointing to a VarDecl that is one of the localVars or params of the corresponding OperImpl;
- the FlowElement pointed to by calledFrom is reachable through a chain of flow-edges from the ContextNode to which the caller-Frame has an instanceOf-edge.

ConstrFrame. This is the sub-type of Frame modelling the initialization of an object — or, in Java terms, the execution of a constructor. It shares most of the characteristics of an OperFrame, except that its executes-edge is not pointing to an OperImpl but to an ObjectType. This implies that there are actually no locals-edges, since constructors in TAAL cannot have local variables or formal parameters.

In addition, a ConstrFrame can also have an auxiliary init-edge, used at object creation time — see Sect. 7.4.1 below.

7.3 Program simulation

The construction of execution graphs for a given program graph is done through another graph transformation system that *simulates* the dynamic semantics of the programming language. This essentially comes down to mimicking the effect of the individual statements and expressions of the program in terms of the value and frame graph. For instance, object creation and assignment to

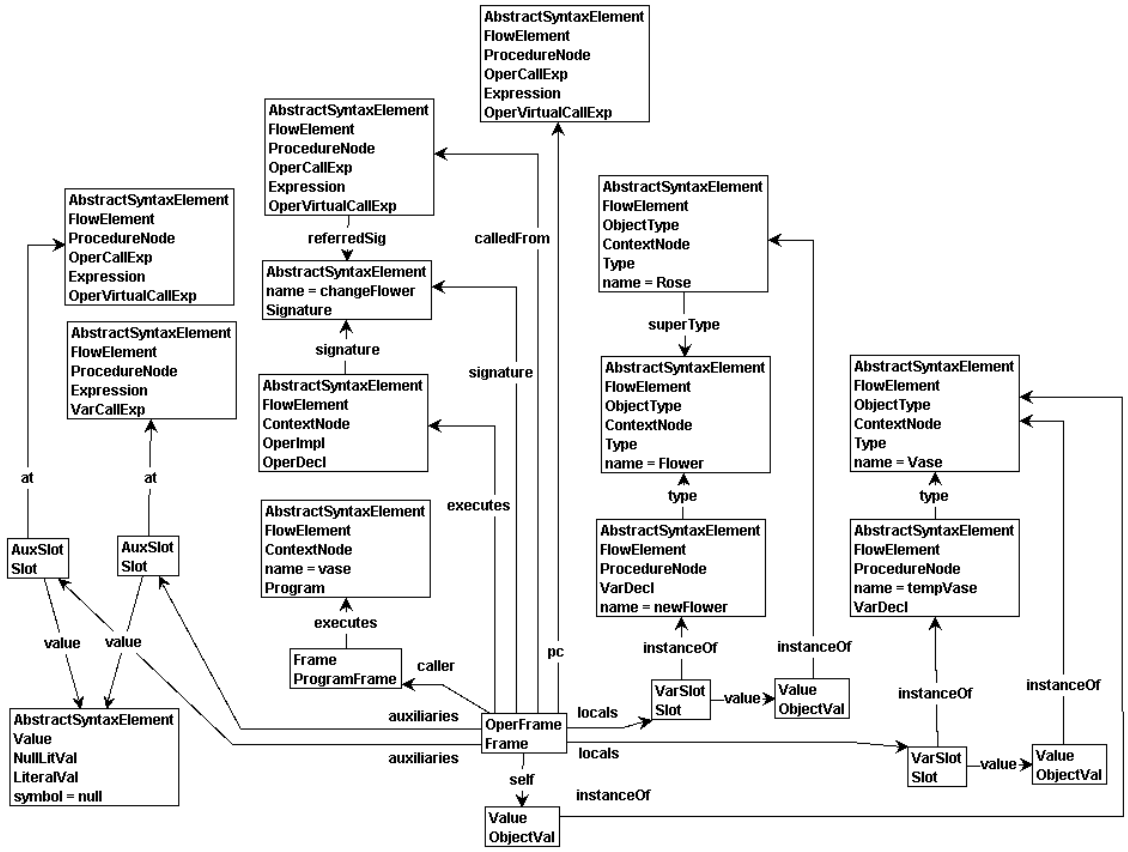


Figure 7.4: Fragment of the frame graph reached during simulation of the while-*condition in* changeFlower.

attributes are reflected in the value graph, whereas method invocation is reflected mainly in the frame graph.

This means that, when we apply the resulting transformation system to the start graph of the program, being the program graph resulting from the transformation described in the previous section, each rule application corresponds to the execution of a small step in the program. The GROOVE tool records those individual steps in the form of a transition systems — which is another graph, where, however, the nodes now stand for state snapshots and the edges for transitions, i.e., small execution steps. For instance, the transition system resulting from the simulation of the vase program of Listing 1 is shown in Fig. 7.5.

We divide the issues arising in simulation into the following categories, depending on the type of node at which control currently resides, i.e., what the pc-edge is pointing to:

- *Expression evaluation*, which occurs if the pc-edge points to an Expression-node. Expression evaluation typically involves the removal and creation of AuxSlot-nodes in a frame. We discuss each of the expression types of TAAL in turn.
- *Statement execution*, which occurs if the pc-edge points to a Statement-node. Statement execution typically has to do with control flow. We discuss each of the statement types of TAAL in turn.
- *Termination*, which occurs if the pc-edge points to a ContextNode of the flow graph. We discuss each of the context node types occurring in TAAL.

This section especially addresses those language elements whose simulation is more or less straightforward. Several more complicated issues are deferred to Sect. 7.4. Furthermore, we have decided

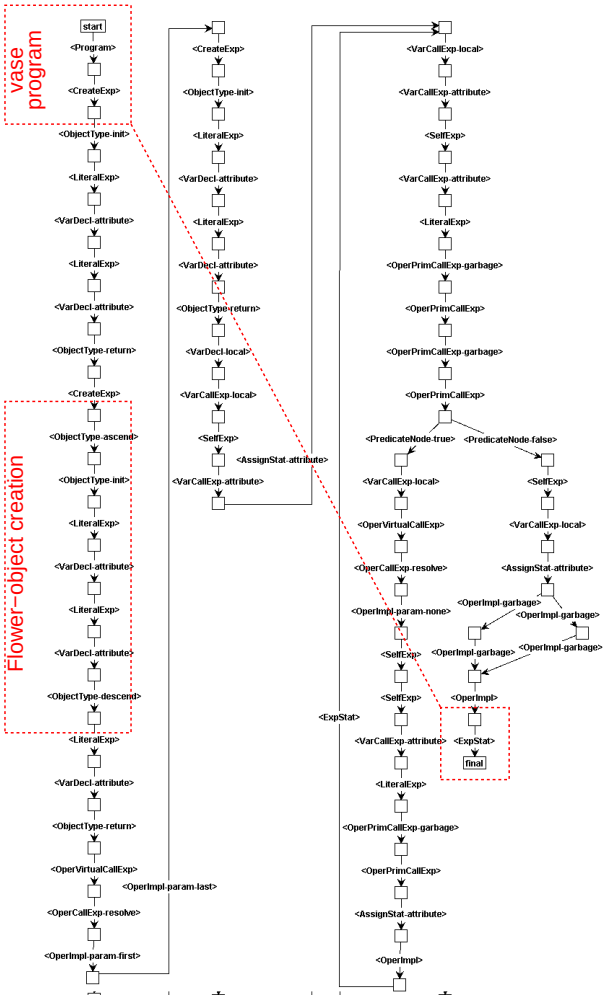


Figure 7.5: The transition system resulting from the simulation of the vase *program.*

not to simulate computations on primitive values. We will elaborate on this choice first, before continuing with the concrete rules.

7.3.1 Abstract interpretation for primitive values

Graph transformation is good for modelling references and their manipulation, but not primitive values and their operations. Although there has been a certain amount of research on combining the strengths of graph transformation and traditional algebra [11, 14], this has not yet led to a commonly accepted model; and moreover, no support for primitive values and operations is currently provided in GROOVE, the tool we have been using for formulating and executing the graph transformation rules.

For these reasons, at this stage we have chosen a pragmatic way of dealing with primitive values, namely to abstract their domain into a *single* default value. In fact, since we had already chosen the NullType to *conform to* every other type (in the sense of Sect. 4.1), including primitive types, we can use the unique NullLitVal for this purpose. Therefore, in the rules below, we will use NullLitVal whenever a PrimLitVal of any kind is expected, and we will not simulate primitive operations.

One interesting consequence is that we cannot simulate the effect of conditional statements precisely (since the value of the boolean expression cannot be computed); therefore the simulation becomes *non-deterministic* at this point. This will be demonstrated on an example below.

7.3.2 Expression evaluation

Most of the business of simulation deals with more or less ordinary statements and expressions, the effect of which is mainly local and uncontroversial. An important choice is to store intermediate values, resulting from the evaluation of sub-expressions, in AuxSlot-instances; these are connected on the one hand to the sub-expression in question (through an at-edge) and on the other hand to the intermediate value (through a value-edge). Apart from this, whenever an expression is evaluated, the pc-edge is moved to the next FlowElement in the control flow graph.

The description of the functionality is structured according to the FlowElement-instance that the pc-edge is pointing to. The simpler rules are shown in Fig. 7.6; some more involved rules are deferred.

CreateExp. This is described in detail in Sect. 7.4.1 below.

LiteralExp. Execution consists of creating a fresh AuxSlot for the expression and assigning the value identified by the LiteralExp to it. However, as discussed above, in the current rule we do not distinguish between primitive values, and instead always attach the NullLitVal. Furthermore, the pc-pointer is moved forward.

OperCallExp. This is described in detail in Sect. 7.4.2 below.

VarCallExp. There are two cases: either the relevant variable is an instance variable (if there is a source) or it is a local variable or parameter. In either case the referredVar identifies a unique VarSlot; execution of the VarCallExp then consists of creating a fresh AuxSlot for the expression and assigning the value of the referredVar to it. Furthermore, the pc-pointer is moved forward.

SelfExp. Execution consists of creating a fresh AuxSlot for the expression and assigning the ObjectVal referenced by self to it. Furthermore, the pc-pointer is moved forward.

7.3.3 Statement execution

We now come to the pc-targets that model statements rather than expressions. Again, we discuss them in alphabetical order, deferring those whose effect requires some more discussion to a later section. The rules discussed here are shown in Fig. 7.7.

AssignStat. The effect of an AssignStat is to make a variable (modelled by a VarSlot) point to a pre-computed value — the rightHandSide of the assignment. Just as for VarCallExp, we have to distinguish the cases of local and instance variables. In either case, the assignedVar (possibly together with the AuxSlot at the source-referenced Expression) uniquely identifies a VarSlot-instance; this receives the value of the AuxSlot at the rightHandSide. The AuxSlot-instances involved are subsequently discarded.

ExpStat. Execution consists of discarding the AuxSlot at the Expression pointed to by the expression-edge, and moving the pc-pointer forward.

PredicateNode. On the level of the simulation, the effect of a ConditionalStat is indistinguishable from that of a WhileStat, and can be captured by turning to the flow graph classification, recognizing that both statements are instances of PredicateNode. The execution in principle consists of inspecting the value of the AuxSlot at the condition, and moving the pc-edge to the FlowElement referenced by either flowTrue or flowFalse. The AuxSlot is discarded. However, as discussed in Sect. 7.3.1, here we do not model primitive values precisely, and hence there is no basis to decide between flowTrue and flowFalse. Hence we model the statement by disregarding the condition and non-deterministically choosing between the two branches — which in the simulation model comes down to simulating *both* choices, as we will see below. Note the similarity with the rule for ExpStat.

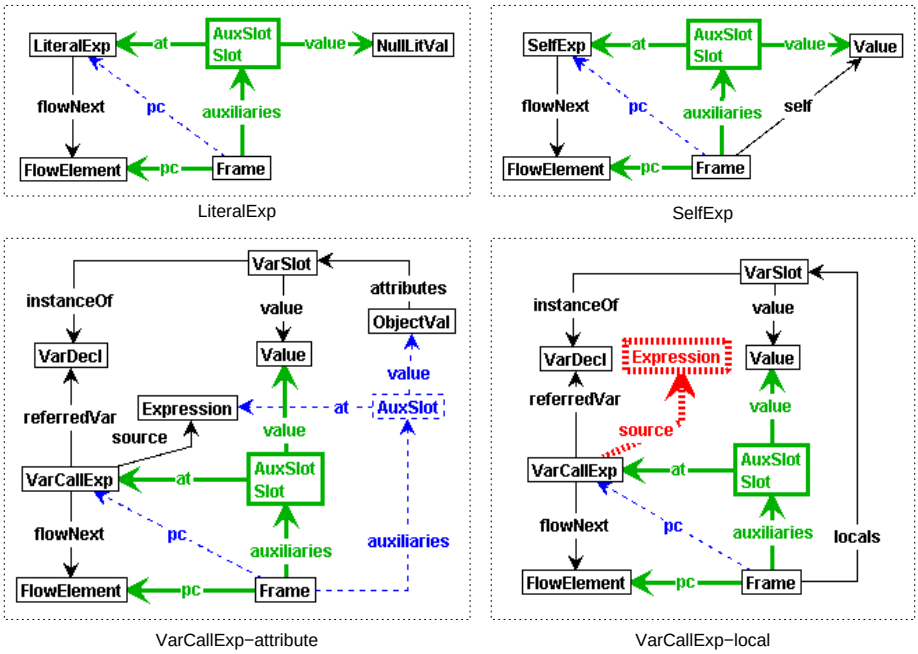


Figure 7.6: Expression simulation rules.

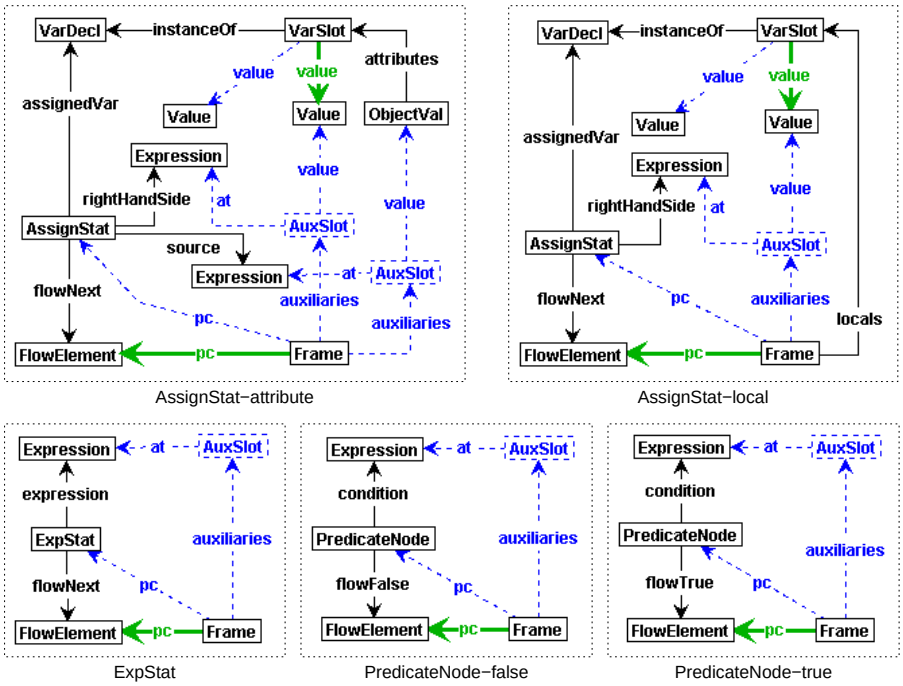


Figure 7.7: Some statement simulation rules.

ReturnStat. Execution terminates the current frame — which actually is always an OperFrame. (A ConstrFrame implicitly terminates upon the end of initialization, and a ProgramFrame upon reaching the end of the program; both are signalled by the pc-edge reaching the corresponding ContextNode, and not by an explicit ReturnStat.) We combine the detailed description with the treatment of method invocation; see Sect. 7.4.2 below.

7.3.4 Handling of Structural Elements

We have treated expressions and statements; from Fig. 6.1 and Fig. 6.2 (i) it can be seen that the only remaining FlowElement-nodes are VarDecls and ContextNodes. We discuss these now.

VarDecl. Essentially, when execution reaches a VarDecl, it means that a VarSlot-instance has to be created and the value of an initial expression has to be assigned to it. Like with the VarCallExp and AssignStat, we have to distinguish between instance variables and local variables. This can be detected by investigating the context: if the VarDecl is among the localVars of the OperImpl associated with this Frame, it is a local variable; otherwise it must be one of the attributes of the ObjectType of the Value referenced by self. (Note that in either case there is guaranteed to be an initial expression, which indeed has been evaluated before control reaches the VarDecl.) The rules are shown in Fig. 7.8.

Program. When control reaches the Program-instance, this indicates that the program has terminated. Nothing happens any more, and we need no transformation rule. Note that we have not implemented any form of garbage collection. On the other hand, since the Program itself has no local variables, and all auxiliary expression values have been consumed, the only execution graph nodes are Value-nodes with, in the case of ObjectVals, VarSlot-attributes. See also Fig. 7.2.

OperImpl. The only case in which control (in the form of a pc-edge) can reach an OperImpl-node is if the operation does not contain any explicit ReturnStat. This, in turn, is only possible if the type of the operation is NullType; so we are safe in treating this situation as a kind of implicit ReturnStat that returns NullLitVal. We show the rule in Sect. 7.4.2 below, where we discuss operation invocation.

ObjectType. Control reaches an ObjectType-instance after a new object of that type has been created and initialized; it signals the termination of a ConstrFrame. There are actually two cases: the ObjectType in question may be the actual type of the ObjectVal just created, or it may be a supertype. We discuss the details as part of the procedure of object creation; see Sect. 7.4.1 below.

7.4 Constructors and Methods

In the description above we have deferred two more complex issues that deserve a more extensive discussion: *object creation* (which occurs as a result of constructor invocation) and *method invocation*. We discuss these below.

7.4.1 Object Creation

Object creation is one important point where the execution of programs with and without inheritance differ. The following paragraphs discuss how we define the process of object creation. Inheritance also influences the protocol for method lookup. This will be discussed in more detail in Sect. 7.4.2.

Traditionally, in object-oriented languages, object construction is a two-pass affair: first, space is allocated for the object and its instance variables (or in other words, the object and its variables are *created*), and subsequently the instance variables are initialized. However, to avoid the need for distinguishing the state of a variable in between its creation and initialization, it is usually specified that at the time of its creation, an instance variable already receives a *default value*.

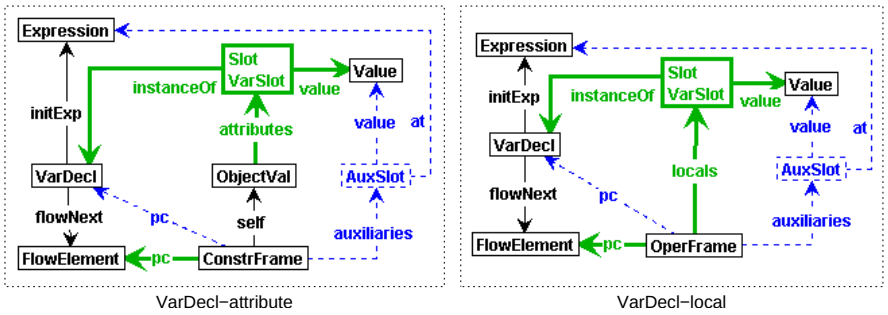


Figure 7.8: Creation and initialization of variables.

(Note that it is not decidable whether a variable is accessed before being explicitly initialized.) In turn, the fact that variables receive default initial values turns their explicit (re-)initialization from an absolute necessity into a practical convenience: it is, in principle, always possible to defer initialization to an ordinary method. Indeed, the Java Language Specification [32] states that the execution of a constructor body is internally implemented as a special < init > method.

In TAAL, on the other hand, we have taken a more simplistic approach, which avoids both the need for default initial values and the need for two consecutive passes. All attributes have an initializing expression; forward references to uninitialized variables are forbidden.[2] In this setting, we can at the same time construct locations for the variables and assign initial values to those locations, provided we take care that this process starts at the top of the inheritance hierarchy. This results in the following procedure:

Allocation. The actual object creation occurs when control reaches a CreateExp-instance. A ConstrFrame and an ObjectVal are created straight away. The ObjectVal is referenced through self from the ConstrFrame. Moreover, the fresh ConstrFrame has an init-pointer to the ObjectType, to indicate the fact that we are initializing an instance of this type. This is shown as rule CreateExp in Fig. 7.9.

Initialization. *Initialization:* A ConstrFrame-instance with an init-edge to an ObjectType is treated in either of two ways, depending on whether the ObjectType has a super type. If it has a super type, then a new ConstrFrame is created recursively for that, but with the same self. If it has no super type, then execution is started, by replacing the init-edge with a pc-edge pointing to the first FlowElement reachable from the ObjectType. The subsequent simulation rules will compute initial values and assign them to newly instantiated AuxSlot-instances for the ObjectVal.

Termination. A ConstrFrame terminates when the pc-edge has arrived (back) at the ObjectType. The frame is discarded and a pc-edge is (re)created at the caller frame. Just as for initialization, there is a case distinction, depending on whether the current frame was called recursively from a sub-type or directly from a CreateExp. Both cases are depicted in Fig. 7.9.

- If the current ConstrFrame was invoked recursively, there is a recursiveFrom-edge to the ObjectType that models the sub-type from which it was called. This means that initialization now proceeds back down the inheritance hierarchy to that sub-type, and the calling Frame is also a ConstrFrame, which already has a self-edge to the underlying object. In this case no return value is required. This is shown in rule ObjectType – descend.

- If the current ConstrFrame was called directly, there is a calledFrom-edge to a CreateExp. This means that the underlying object, pointed to by self, should be returned to the caller, in the same way as in a ReturnStat (see below). This is shown in rule ObjectType – return.

[2]Note that this is a constraint that is generally undecidable and hence cannot be enforced at compile time.

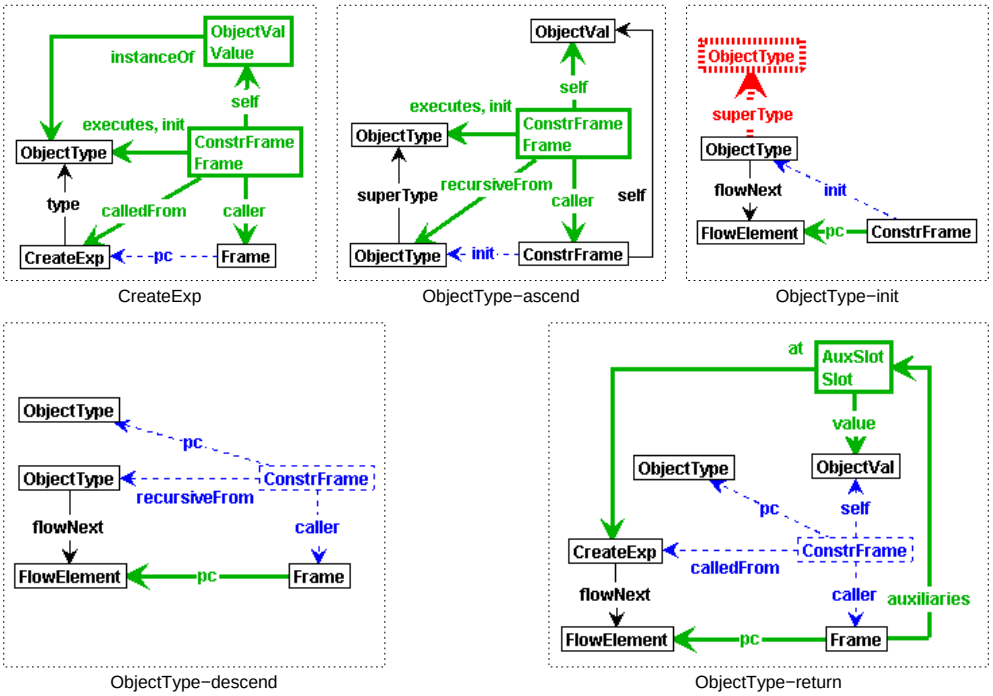


Figure 7.9: Rules for object creation and initialization.

7.4.2 Method Invocation and Return

Method calls are among the basic expressions of a program. Their effect is entirely within the frame graph: the execution of a method call creates a new OperFrame-node, associates it with an OperImpl-instance (which is part of the program graph), passes the actual to the formal parameters, and finally turns over control to it.

Frame creation. The creation of the OperFrame itself is straightforward: the more difficult parts follow only later. Once the frame is created, control is transferred to it, but not yet in the form of a pc-edge; instead the corresponding method implementation has to be found (see below) first. This phase is modelled by an outgoing lookup-edge to the class in which the method is sought — with the assumption, checked at compile-time, that a suitable method implementation, i.e., with the correct signature, actually exists either in the class or in one of its super-classes.

There are two versions of method invocation: virtual and super. In the first case the lookup starts at the dynamic type of the target object; in the second case it starts at the super-type of the type in which the current method is located. The two cases are shown in Fig. 7.10 (rules OperVirtualCallExp and OperSuperCallExp, respectively).

The other rules in Fig. 7.10 deal with the case of primitive operations. As discussed above, we abstract from primitive values by using only NullLitVal; consequently, for operations located in a primitive type we do not have to look up the implementation but rather can insert the return value immediately. However, we must take care that any actual parameters of the operation call (present as AuxSlots in the value graph) are discarded first. This process is taken care of by rules OperPrimCallExp−garbage and OperPrimCallExp.

Method lookup. The association of an OperImpl-instance with a freshly created OperFrame is called *method lookup*. The protocol for method lookup is, as mentioned before, one important point where the execution of programs with and without inheritance differ. Furthermore, we strongly believe that it is also the only point where changes will have to be made in order to accommodate aspect orientation.

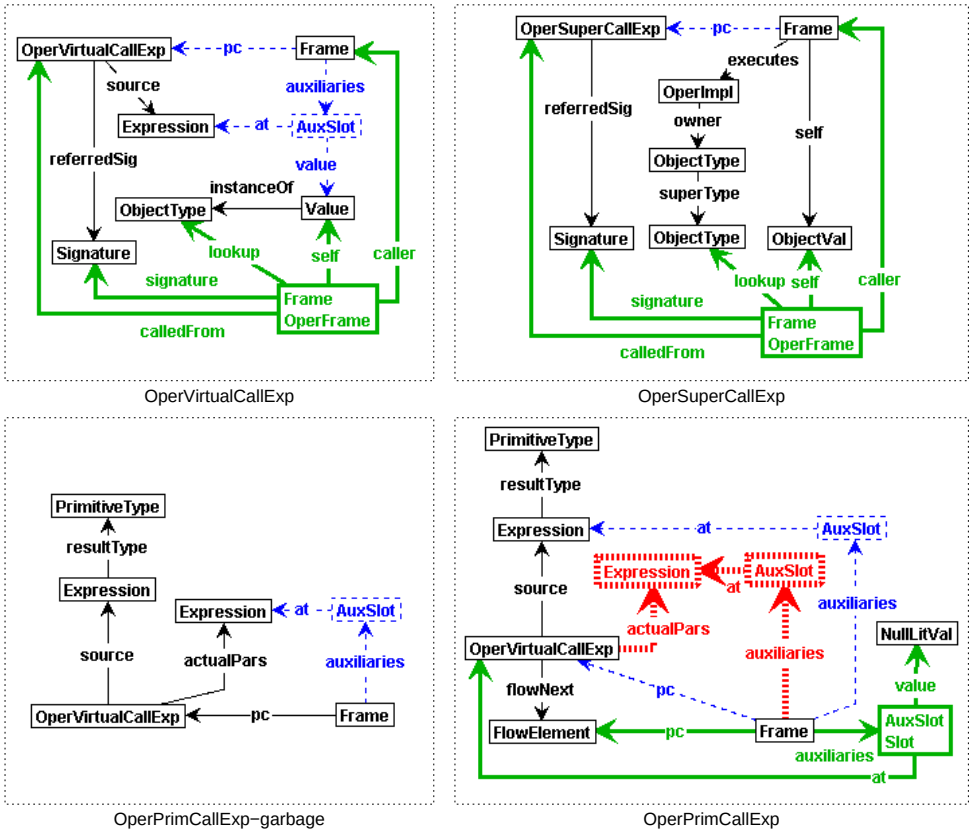


Figure 7.10: Rules for operation calls.

In programs without inheritance, method lookup is static. That is, the compiler is able to decide which OperImpl is to be associated with a given method call, purely on the basis of the Signature. Not so, however, in the presence of inheritance, where the dynamic type of the "target" (the object asked to execute a method) is a factor in determining the relevant method implementation. All the compiler is able to determine statically is the *signature* of the method to be executed (see Sect. 4). This is complemented by a dynamic method lookup protocol whereby classes are queried (as it were) in succession whether they have implemented a method with the given signature, in which case that definition becomes the one to execute — if not, the query is passed on to the next higher class in the inheritance hierarchy. (This part of the protocol could be optimized statically, since each class can store a map from those method signatures implemented somewhere in their super-classes to the "most concrete" implementation; this obviates the need for recursively passing on queries.) The rules for method lookup propagation and resolution are shown in Fig. 7.11.

Parameter passing. Another complication is the need to pass the parameters from the calling Frame to the called, newly created OperFrame. This essentially involves an assignment of all actual parameter values (which are stored in AuxSlot-instances associated with auxiliaries-edges to the calling frame) to the corresponding formal parameters (which are VarSlot-instances associated with locals-edges to the called frame). The correspondence of actual to formal parameters is given only through the *ordering* of the parameters. For that reason, the parameter passing phase is modelled by *four* transformation rules, shown in Fig. 7.12: for starting, continuing and ending the phase, as well as for the case where the number of parameters is zero. To keep track of the current actual and formal parameters during propagation, we have introduced special actualPar- and param-edges pointing to, respectively, the current actual parameter (an Expression) and the current formal parameter (a VarDecl).

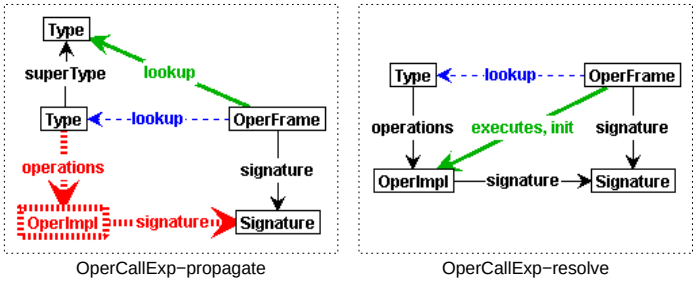


Figure 7.11: Rules for method lookup.

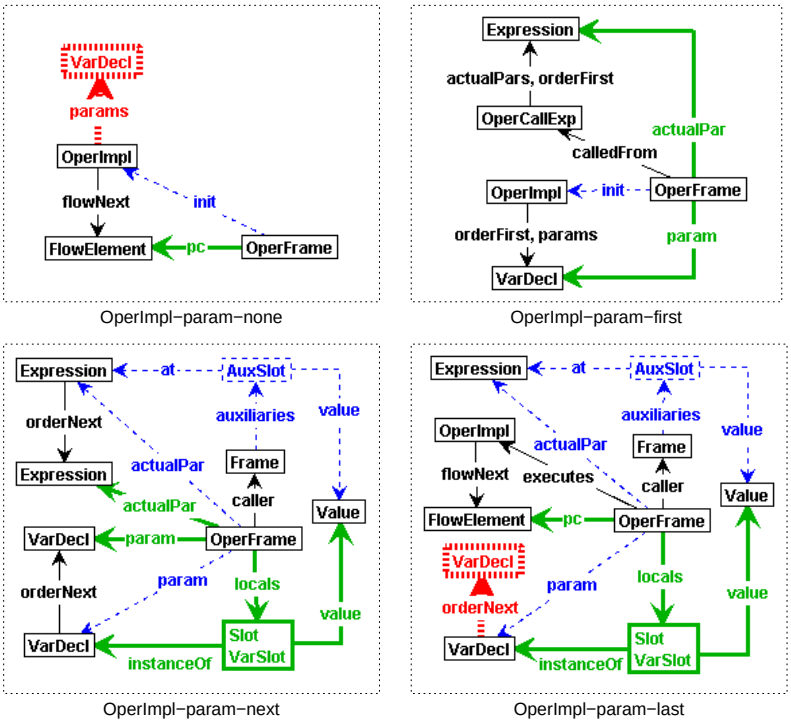


Figure 7.12: Parameter passing rules.

Return Here we describe the execution of the ReturnStat, which we omitted from Sect. 7.3.3. There are several steps involved

- The value of the AuxSlot attached to the value-referenced Expression is assigned to a new AuxSlot at the calledFrom-referenced OperCallExp of the calling frame;
- The AuxSlot itself is discarded;
- All VarSlot-instances pointed to by locals are discarded;
- A pc-edge is created from the calling Frame to the next FlowElement with respect to the calledFrom-referenced node;
- The current OperFrame-instance is discarded.

Since discarding the locals is a repetitive process, it requires a separate rule; the main rule is only applicable if no more locals-edges exist. The two rules are shown in Fig. 7.13.

A special case, also shown in Fig. 7.13, occurs if the method has type NullType and no explicit ReturnStat: then instead control eventually reaches the OperImpl-node. The actions necessary to deal with this are analogous to those for the ReturnStat, except that this time the value to be returned is not taken from some expression; instead, it is always NullLitVal.

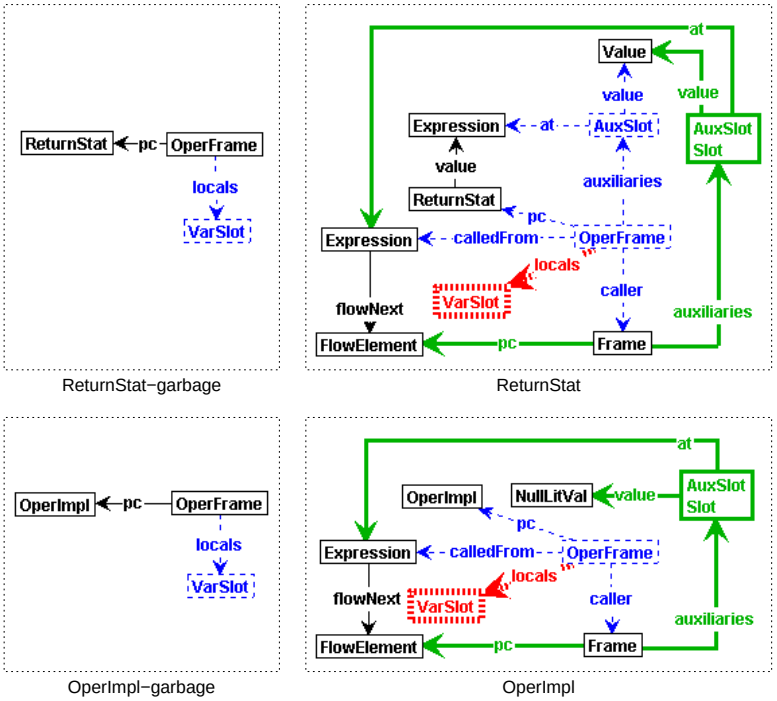


Figure 7.13: Rules for method return.

8 Tool Support

The tool support that was developed during our research can clearly be divided into two products: the TAAL Eclipse plug-in and the Groove tool set. This section covers some of the more interesting facts of both tools.

8.1 The TAAL Eclipse Plug-in

The parsing and static analysis phases of TAAL are implemented in an Eclipse plug-in [18, 31, 10]. Eclipse is, according to the website, 'an open extensible IDE for anything and nothing in particular', and is, therefore, very suitable to build a small IDE for a small language like TAAL. Because of the particular support for Java, the plug-in was developed in that language.

The core of the plug-in is formed by the implementation of both the CST and the ASG meta-models. Furthermore the plug-in contains a parser, and separate classes that each executes one of the passes in the static analysis. For sake of clarity, each of the static analysis passes were implemented separately in the TAAL plug-in, although this is not a very efficient implementation.

Next to the implementation of the bare functionality of parsing and analysis, the plug-in offers an editor, and two actions that can be performed on a TAAL program. The first action shows a model of the ASG in a separate window, the second action generates Groove input; a flat ASG. The editor has simple syntax highlighting.

The effort of implementing this plug-in was kept minimal. Much of the Java code was generated.

1. The parser was generated using JavaCC [21], a parser generator for Java applications.
2. The implementation of the CST and ASG meta-models was generated by Octopus [Octopus], including the visitor interfaces used in the static analysis.
3. The implementation of the Flattening was generated by an Octopus add-on.
4. The implementation of the static analysis was the only part that was done completely by hand.

The TAAL Eclipse plug-in can be downloaded from www.klasse.nl/taal.

8.1.1 The Implementation of the Meta-models

The implementation of the meta-models follows directly from the UML model. Each attribute becomes a Java field, with get and set operations, and each operation becomes an operation. Each navigable association-end becomes a Java field, with get and set operations, similar to the attributes.

For every (meta)class in the meta-models three Java units were created. First, an interface is created that captures the public structure and functionality. The name of the interface is always the name of the meta-class prefixed with 'I'. The set of all interfaces for the ASG is placed in the package com.klasse.taal.rasg, and the set of all interfaces for the CST is placed in the package com.klasse.taal.cst.

Second, a class is made that contains a complete implementation of this interface. Some of the bodies of the operations are defined by OCL [35] expressions. If this is the case, this implementation will contain the corresponding Java code generated by Octopus. Appendix B contains the OCL expressions that were implemented in this manner, and a number of invariants on the ASG meta-model. This Java class implements the corresponding interface, and it is abstract. The name of this class is the name of the meta-class postfixed with 'GEN'. The set of all GEN-classes for a meta-model is placed in a sub-package of the package where the interfaces are placed: com.klasse.taal.rasg.internal.generated, and com.klasse.taal.cst.internal.generated, respectively.

Third, because not all operation bodies could be defined using OCL expressions, a Java class is created that contains some handcoded implementations of the operations defined in the interface. This Java class inherits from the corresponding GEN-class. The name of this class is always equal

to the name of the meta-class. Whenever instances of the meta-classes are created, either during parsing or during static analysis, they are instances of a Java class from this category. The set of all instantiable classes for a metamodel is placed in a subpackage of the package where the interfaces are placed: com.klasse.taal.rasg.internal, and com.klasse.taal.cst.internal, respectively.

8.2 The Groove Tool Set

The Groove tool set [30] supports editing and displaying graphs and graph production rules, and (especially) applying those rules to concrete graphs. It consists of a number of tools, the main three being:

- The Editor: a graphical environment which enables you to specify graphs and graph transformation.
- The Simulator: a graphical environment in which graph transformations can be performed and tracked.
- The Imager: a tool which can be used to generate figures from graphs or graph transformation rules as stored in graph production rule files (GPR-files) or graph state files (GST-files). It supports exporting to PNG, JPEG and FSM extensions.

The tool set has been created especially for the generation and storage of state spaces, in the form of graph transition systems as discussed in Sect. 5.4.2.

For this project we have specified two graph production systems: one for constructing control flow graphs (described in Sect. 6) and another one for simulating the actual programs (Sect. 7). Both graph transformation systems can be downloaded from the Groove project's website [30].

To use these graph production systems, the output of the TAAL Eclipse Plug-in (i.e. the Flat Abstract Syntax Graph) should be used as the start graph of the flow graph construction graph transformation system. The final state of this GPS (i.e. the Program Graph) is the starting state for simulating the program.

For more information about the Groove-project, we refer to the project's website. From there you can download the tool and some small example of graph transformation systems.

9 Conclusion

The work described in this report shows a complete example of how programming languages can be defined using graphs and graph transformation rules. The language definition of TAAL includes all necessary parts of a language definition: concrete syntax, abstract syntax and semantics, which all three have been defined using this single formalism. Although other work has been presented that uses graphs and graph transformation rules (e.g., [7, 15]) for (parts of) language definitions, none of this work reaches the same level of completeness.

We have defined the operational semantics of an imperative, object-oriented language. The use of graph transformation rules to specify the semantic rules offers a number of advantages. First, the visual representation of the graph transformation rules provides an intuitive understanding of the semantics. Second, formal verification techniques become available.

Furthermore, the graph transformation rules offer the possibility to include in one mathematical structure, the graph, information on both the run-time system and the program that is being executed. Traditional approaches to operational semantics (e.g. [1, 36, 29, 5, 6, 2, 8]) often need to revert to inclusion in the syntax definition of run-time concepts, e.g. inclusion of the concept of location to indicate a value that may possibly change over time. This seems to be an artificial manner of integrating parts of the language definition, i.e. of the abstract syntax and the semantic domain, that can be avoided using graph transformation rules. Finally, in graph transformation rules, context information can be included more naturally and uniformly than for example when using SOS-rules [36].

The example language that we have chosen comprises some of the fundamental aspects of object-oriented programming languages, like inheritance, including dynamic method look-up, and object creation. The structure of our solution makes us confident that the approach can be extended to real-life software languages in the object-oriented paradigm:

- All the transformation steps (parsing, static analysis, flow generation and simulation) are structured according to the concepts in the abstract syntax. This lends a modularity to the definitions that is independent of the language being defined.
- The structure of the Flow and Execution Graphs is generic, in the sense that the elements therein are not specific to TAAL; rather, they capture the essential aspects of imperative, object-oriented languages.

As mentioned above, work that is closely related to ours is by Corradini et al. [7]. They use graph transformations to formalize the semantics of a realistic programming language: they address a fairly large fragment of Java. Technically, the difference is that they interpret method invocation *unfolding* — meaning that the *program graph* changes dynamically. This obviates the need for the frame graph, at the price of having program-dependent rules (namely, one per method implementation). Another difference is that they provide no tool support, and in that sense theirs is a more theoretical exercise.

Another, less directly related source of research is on defining dynamic semantics of (UML-type) design models, where also the idea of using graph transformations has been proposed, e.g. in [15, 25, 34]. Furthermore, in Engels et al. [15] ideas are presented on how to use collaboration diagrams, interpreted as graph transformation rules, for defining SL semantics.

In this work, graph transformation rules were used for defining the semantic mapping of a language, but not for the mapping of the concrete to abstract syntax (the syntax mapping). We are convinced that graph transformations can also be used for the syntax mapping.

A separate contribution of this work is that we define a mapping from a UML class diagram to a graph. The mapping is not a general one, i.e. it cannot map any UML model to a graph. Still, it is good enough to do the work for not only the metamodel of TAAL, but for any UML model that adheres to the given restrictions. These restrictions are:

- all associations are one-way, directed;
- all association multiplicities are either 1, 0..1, 1..n, or 0..n;

- all association ends that are navigable, have role names;
- only ordered is allowed as constraint to an association end, and only when the other end is an aggregation.

A final aspect of the work reported here is that we have not only developed the TAAL language definition but supporting tools as well. This means that we can actually compile and simulate any TAAL-program and store the resulting transition system so, for instance, all the ingredients for verification are there. Both tool sets as well as the full sets of transformation rules defining the flow generation and simulation phases are available for downloading [23, 30].

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. Inductive proof outlines for monitors in java. In Najm et al. [27], pages 155–169.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[4] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE'04)*, number 2984 in Lecture Notes in Computer Science, pages 214–228. Springer, 2004.

[5] K. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 603–626. Springer, 1994.

[6] K. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.

[7] A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating Java code to graph transformation systems. In Ehrig et al. [12], pages 383–398.

[8] F. S. de Boer and C. Pierik. How to cook a complete hoare logic for your pet OO language. In *Formal Methods for Components and Objects (FMCO)*, volume 3188 of *Lecture Notes in Computer Science*, pages 111–133. Springer, 2004.

[9] F. L. Dotti, L. Foss, L. Ribeiro, and O. M. dos Santos. Verification of distributed object-based systems. In Najm et al. [27], pages 261–275.

[10] Eclipse. Eclipse website, 2006. http://www.eclispe.org/.

[11] H. Ehrig. Attributed graphs and typing: Relationship between different representations. Technical report, Technische Universität Berlin, November 2003.

[12] H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors. *Proceedings of the 2nd International Conference on Graph Transformation (ICGT'04)*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.

[13] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation, Part II: Single pushout approach and comparison with double pushout approach. volume I: Foundations, pages 247–312. World Scientific, Singapore, 1997.

[14] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed graph transformation. In Ehrig et al. [12], pages 161–177.

[15] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of the Third International Conference on the Unified Modeling Language (UML'00)*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[16] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thomsen Publishing Inc., 2nd edition, 1997.

[17] D. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. New York: John Wiley & Sons, 2003.

[18] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns and Plug-ins*. The Eclipse Series. Addison-Wesley, 2004.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[20] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[21] JavaCC. Javacc project, 2006. https://javacc.dev.java.net/.

[22] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained; The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[23] A. G. Kleppe. TAAL Eclipse plugin. Available from http://www.klasse.nl/english/research/taal-install.html, 2005.

[24] B. König. A general framework for types in graph rewriting. In S. Kapoor and S. Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'00)*, volume 1974 of *Lecture Notes in Computer Science*, pages 373–384. Springer, 2000.

[25] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In M. J. Butler, L. Petre, and K. Sere, editors, *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.

[26] S. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled, Priciples of Model-Driven Architecture*. Addison-Wesley, 2004.

[27] E. Najm, U. Nestmann, and P. Stevens, editors. *Proceedings of the 6th International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'03)*, volume 2884 of *Lecture Notes in Computer Science*. Springer, 2003.

[28] OMG. MDA guide version 1.0.1, 2005. http://www.omg.org/.

[29] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[30] A. Rensink. The GROOVE tool set. Available from http://groove.sf.net, 2006.

[31] S. Shavor, J. D´Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer´s Guide to Eclipse*. Addison-Wesley, 2003.

[32] SUN. The Java language specification, 2006. http://java.sun.com/.

[33] G. Taentzer and A. Rensink. Ensuring structural constraints in graph-based models with type inheritance. In M. Cerioli, editor, *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'05)*, volume 3442 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2005.

[34] D. Varró. A formal semantics of UML statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the 1st International Conference on Graph Transformation (ICGT'02)*, Lecture Notes in Computer Science, pages 378–392. Springer, 2002.

[35] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

[36] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.

A Concrete Syntax Grammar in BNF

```
ParsedProgram ::=
    <PROGRAM_START> <STRING>
    <CURLY_OPEN> ParsedExpression <CURLY_CLOSE>
    ( ParsedTypeDecl )*
    <PROGRAM_END>

ParsedTypeDecl ::=
    <TYPE_START> <STRING> [ <EXTENDS> ParsedTypeRef ]
    ( ParsedVarDecl <SEMICOLON> | ParsedOperDecl )*
    <TYPE_END>

ParsedVarDecl ::=
    <STRING> <COLON> ParsedTypeRef [ <ASSIGN> ParsedExpression ]

ParsedOperDecl ::=
    <STRING>
    <BRACKET_OPEN> [ ParsedVarDecl ] ( <COMMA> ParsedVarDecl )* <BRACKET_CLOSE>
    [ <COLON> ParsedTypeRef ]
    [ <LOCALS> ( ParsedVarDecl <SEMICOLON> )* ]
    [ <CURLY_OPEN> ( ParsedStatement )* <CURLY_CLOSE> ]

ParsedBlockStat ::=
    <CURLY_OPEN> ( ParsedStatement )* <CURLY_CLOSE>

ParsedStatement ::=
    ParsedExpression [ <ASSIGN> ParsedExpression ] <SEMICOLON>
    | ParsedReturnStat <SEMICOLON>
    | ParsedConditionalStat
    | ParsedWhileStat
    | ParsedBlockStat

ParsedExpression ::=
    ( ParsedLitExp | ParsedCreateExp | ParsedPropCallExp )
    ( <DOT> ParsedPropCallExp )*

ParsedConditionalStat ::=
    <IF> condition = ParsedExpression
    <THEN> thenParParsedStatement
    [ <ELSE> elseParParsedStatement ]
    <ENDIF>

ParsedWhileStat ::=
    <WHILE> ParsedExpression <DO>
    ( ParsedStatement )*
    <ENDWHILE>

ParsedReturnStat ::=
    <RETURN> ParsedExpression

ParsedCreateExp ::=
    <NEW> ParsedTypeRef <BRACKET_OPEN> <BRACKET_CLOSE>

ParsedTypeRef ::=
    <STRING>

ParsedPropCallExp ::=
```

```
    <STRING>
    <BRACKET_OPEN>
    [ ParsedExpression ] ( <COMMA> actual = ParsedExpression )*
    <BRACKET_CLOSE>
    | <STRING>

ParsedLitExp ::=
    <STRINGLITERAL>
    | <NUMBERLITERAL>
    | <TRUE>
    | <FALSE>
    | <NULLLITERAL>
```

Listing 3: The BNF concrete syntax specification of TAAL - the non-terminals.

```
PROGRAM_START   ::= "program"
PROGRAM_END     ::= "endprogram"
TYPE_START      ::= "class"
TYPE_END        ::= "endclass"
EXTENDS         ::= "extends"
NEW             ::= "new"
ACTION          ::= "action"
LOCALS          ::= "locals"
IF              ::= "if"
THEN            ::= "then"
ELSE            ::= "else"
ENDIF           ::= "endif"
TRUE            ::= "true" | "TRUE"
FALSE           ::= "false" | "FALSE"
BRACKET_OPEN    ::= "("
BRACKET_CLOSE   ::= ")"
CURLY_OPEN      ::= "{"
CURLY_CLOSE     ::= "}"
COLON           ::= ":"
SEMICOLON       ::= ";"
COMMA           ::= ","
DOT             ::= "."
WHILE           ::= "while"
DO              ::= "do"
ENDWHILE        ::= "endwhile"
RETURN          ::= "return"
ASSIGN          ::= ":="
NULLLITERAL     ::= "null"
STRING          ::=
                    ["a"-"z", "A"-"Z", "_"]
                    ( ["a"-"z", "A"-"Z", "0"-"9", "_" ] )*
NUMBERLITERAL   ::=
                    ["0"-"9"] (["0"-"9"])*
                    ( "." ["0"-"9"] (["0"-"9"])* )?
                    ( ("e" | "E") ( "+" | "-" )?
                    ["0"-"9"] (["0"-"9"])* )?
STRINGLITERAL   ::=
                    // from Java 1.1 grammar
                    "\'"
                    (
                      (~["\'","\\","\n","\r"])
                        |
                        ("\\"
```

```
                        ( ["n","t","b","r","f","\\","'","\""]
                          | ["0"-"7"] ( ["0"-"7"] (["0"-"7"])?
                        )?
                      )
                    )
                  )*
                  "\'"
```

Listing 4: The BNF concrete syntax specification of TAAL - the terminals.

B Graph Production Rules

In this appendix we list all the graph production rules in alphabetical order, including the ones that have already been discussed in the paper. We will first list all the rules concerning the flow graph construction and then the rules that define the semantics.

B.1 Flow Graph Construction

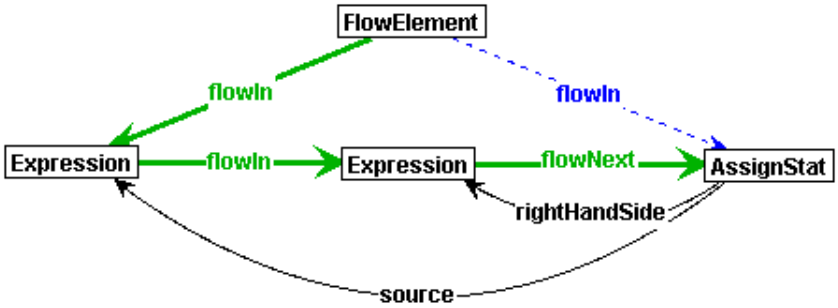


Figure B.1: AssignStat-attribute.gpr

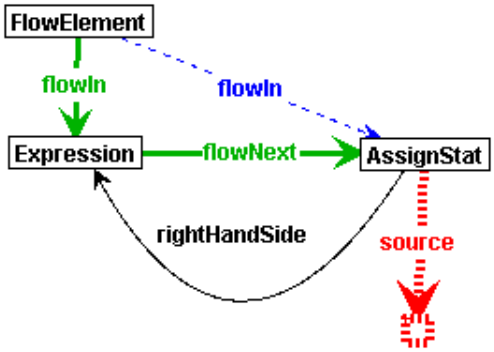


Figure B.2: AssignStat-local.gpr

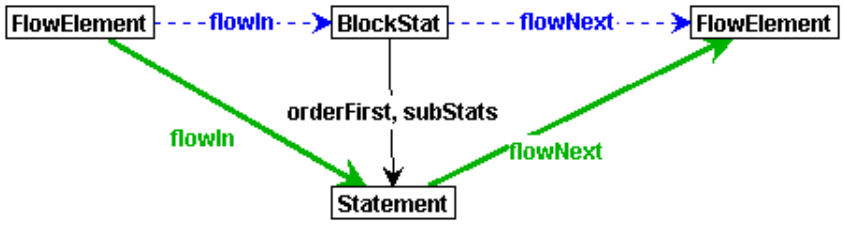


Figure B.3: BlockStat.gpr (see Sect. 6.3)

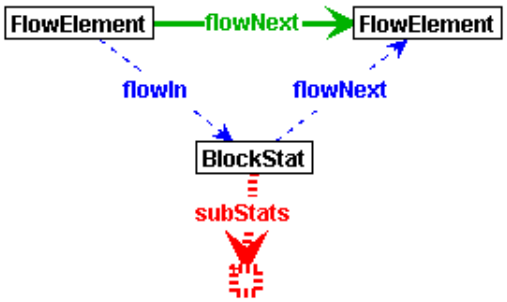


Figure B.4: BlockStat-empty.gpr

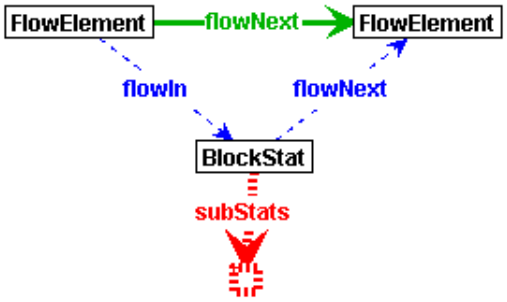


Figure B.5: BlockStat-empty.gpr

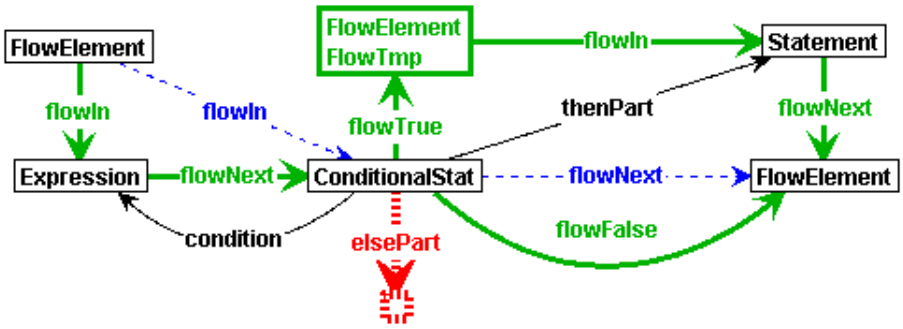


Figure B.6: ConditionalStat-ifthen.gpr

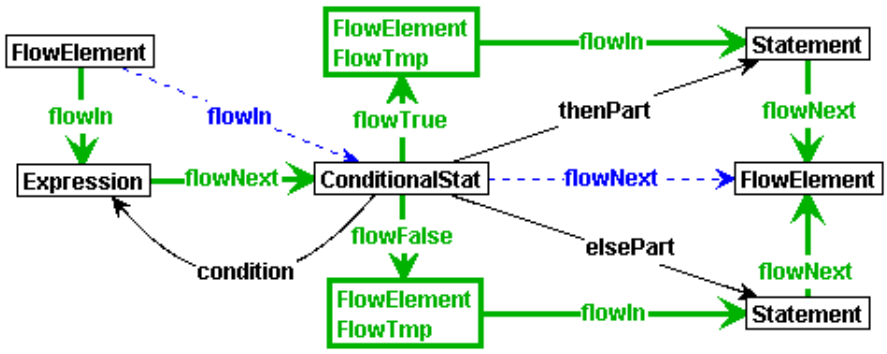


Figure B.7: ConditionalStat-ifthenelse.gpr (see Sect. 6.3)

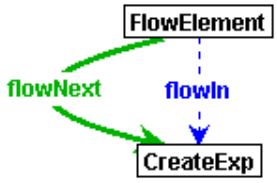


Figure B.8: CreateExp.gpr

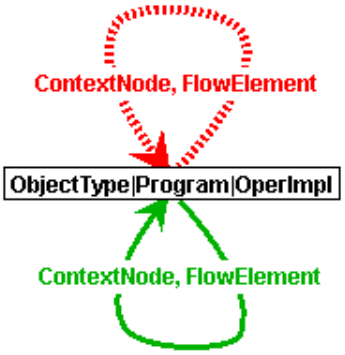


Figure B.9: ContextNode.gpr

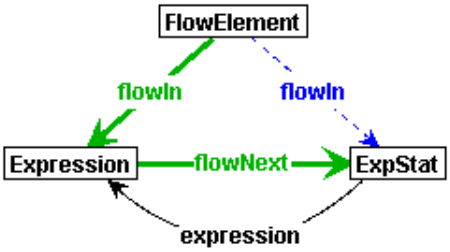


Figure B.10: ExpStat.gpr

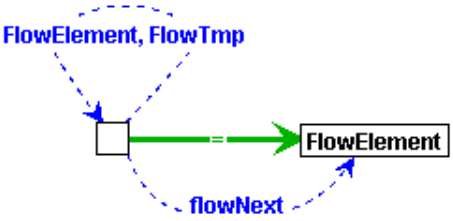


Figure B.11: FlowTmp-merge.gpr (see Sect. 6.2)

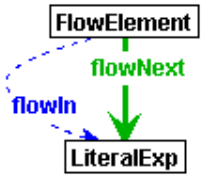


Figure B.12: LiteralExp.gpr (see Sect. 6.3)

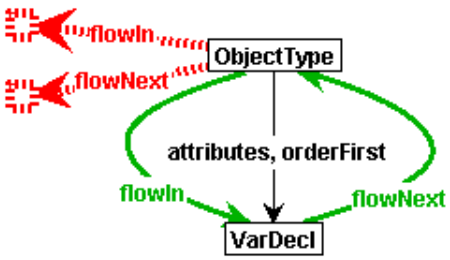


Figure B.13: ObjectType.gpr (see Sect. 6.3)

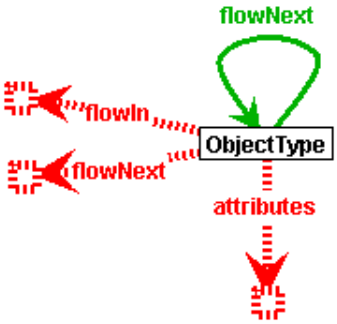


Figure B.14: ObjectType-no-attributes.gpr (see Sect. 6.3)

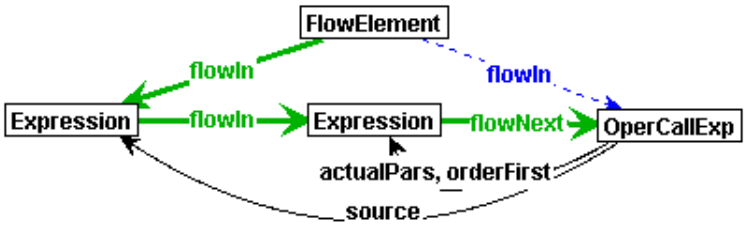


Figure B.15: OperCallExp.gpr (see Sect. 6.3)

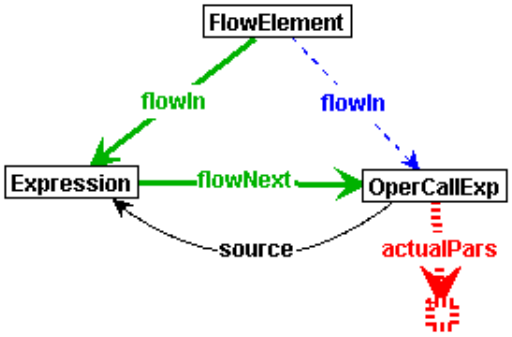


Figure B.16: OperCallExp-no-actualPars.gpr

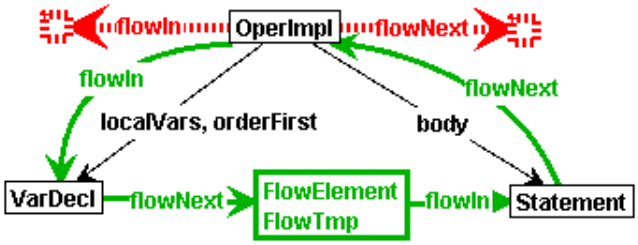


Figure B.17: OperImpl.gpr (see Sect. 6.3)

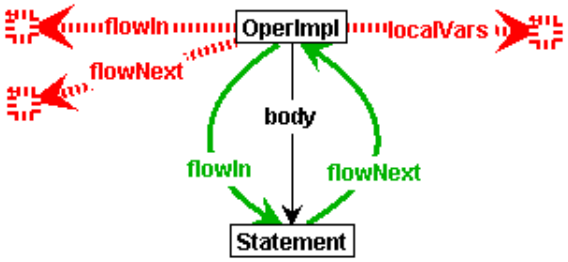


Figure B.18: OperImpl-no-localVars.gpr

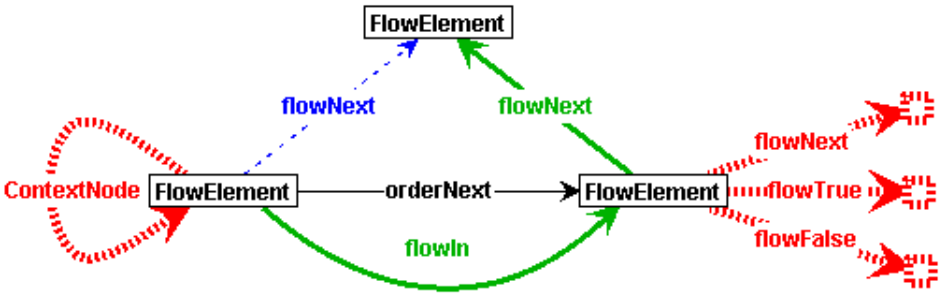


Figure B.19: ordered-next.gpr (see Sect. 6.2)

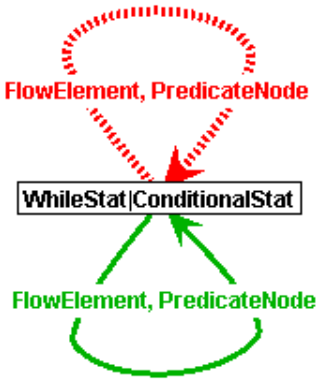


Figure B.20: PredicateNode.gpr

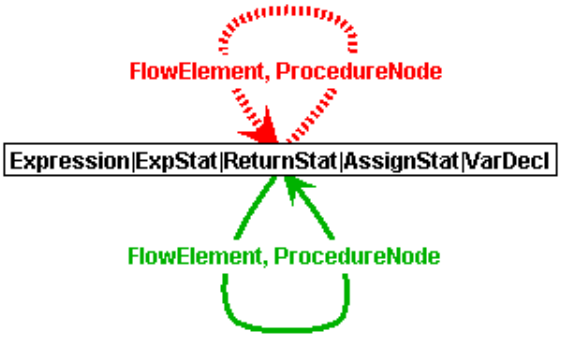


Figure B.21: ProcedureNode.gpr

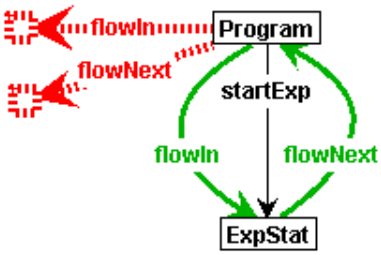


Figure B.22: Program.gpr (see Sect. 6.3)

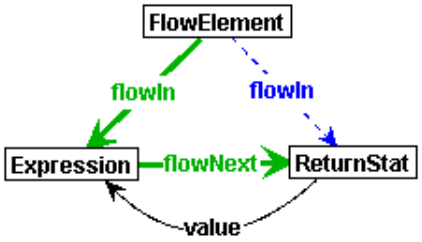


Figure B.23: ReturnStat.gpr

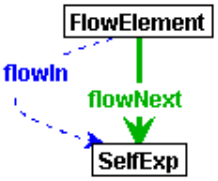


Figure B.24: SelfExp.gpr

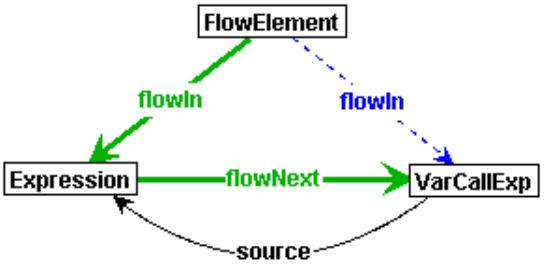


Figure B.25: VarCallExp-attribute.gpr

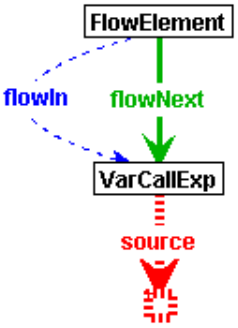


Figure B.26: VarCallExp-local.gpr

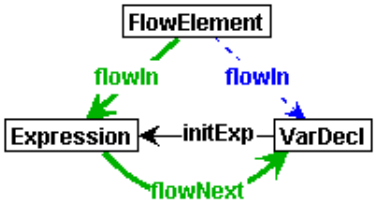


Figure B.27: VarDecl.gpr (see Sect. 6.3)

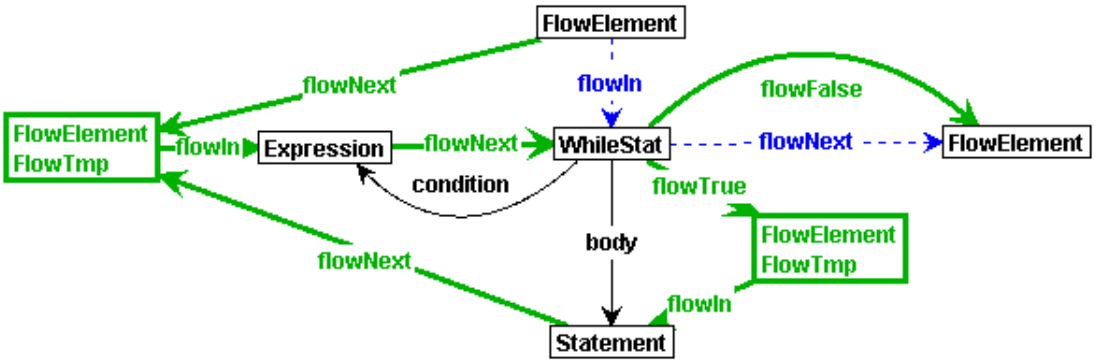


Figure B.28: WhileStat.gpr (see Sect. 6.3)

B.2 Simulation

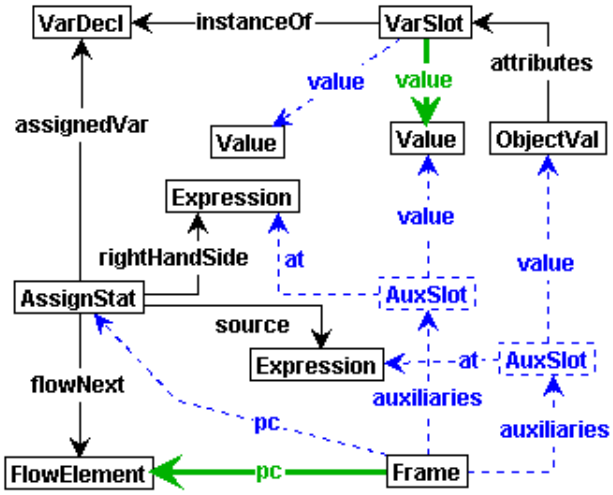


Figure B.29: AssignStat-attribute.gpr

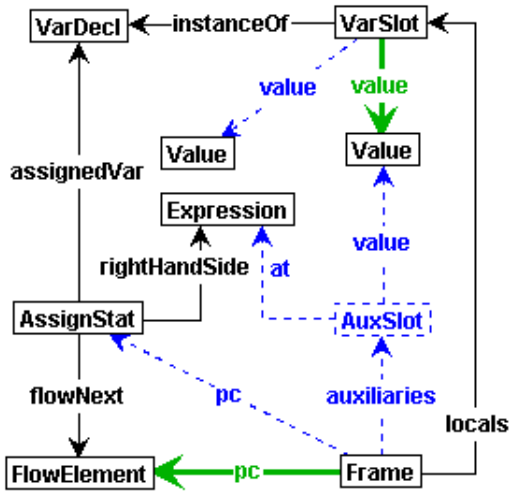


Figure B.30: AssignStat-local.gpr

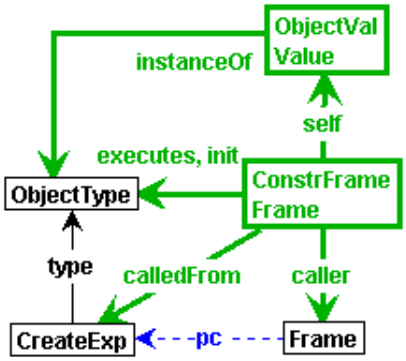


Figure B.31: CreateExp.gpr

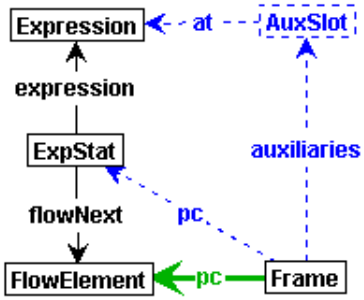


Figure B.32: ExpStat.gpr

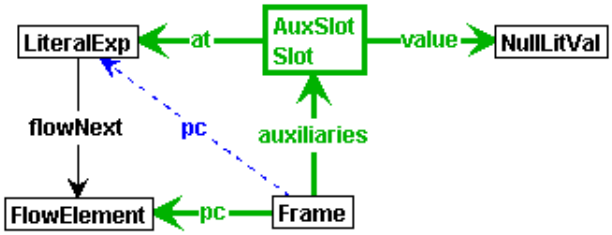


Figure B.33: LiteralExp.gpr

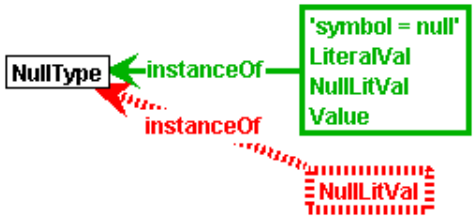


Figure B.34: NullType-init.gpr

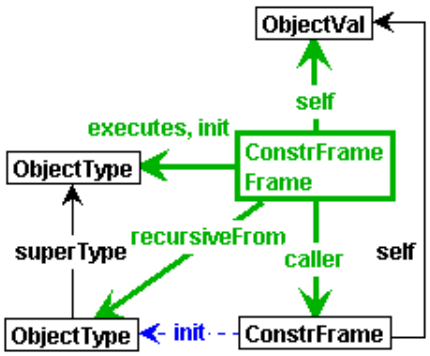


Figure B.35: ObjectType-ascend.gpr

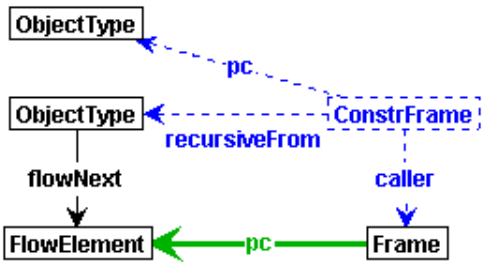


Figure B.36: ObjectType-descend.gpr

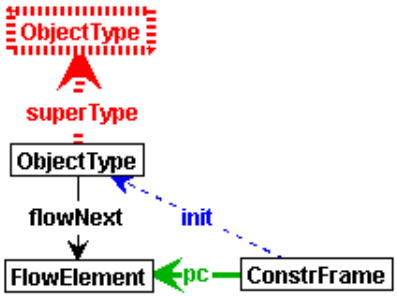


Figure B.37: ObjectType-init.gpr

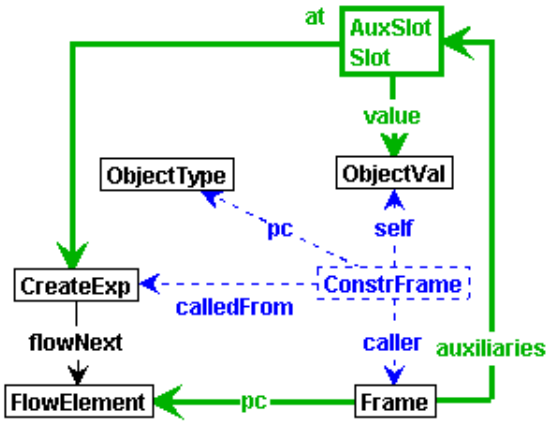


Figure B.38: ObjectType-return.gpr

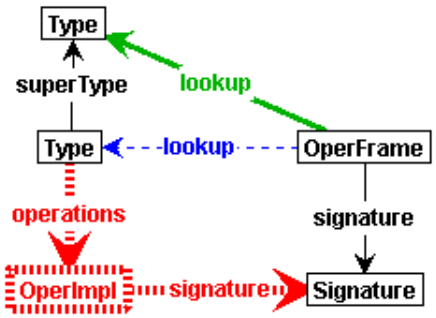


Figure B.39: OperCallExp-propagate.gpr

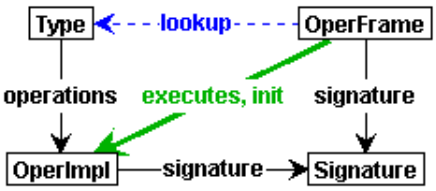


Figure B.40: OperCallExp-resolve.gpr

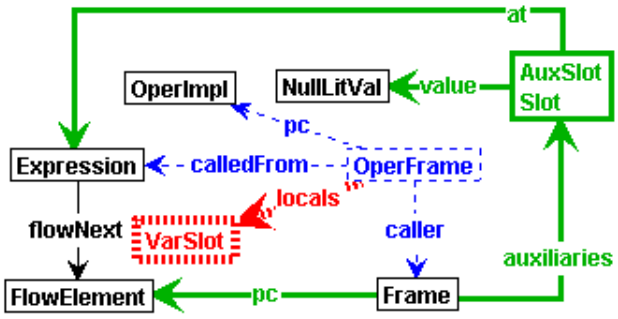


Figure B.41: OperImpl.gpr

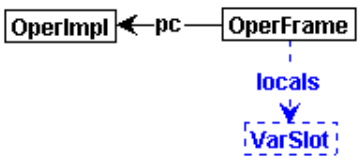


Figure B.42: OperImpl-garbage.gpr

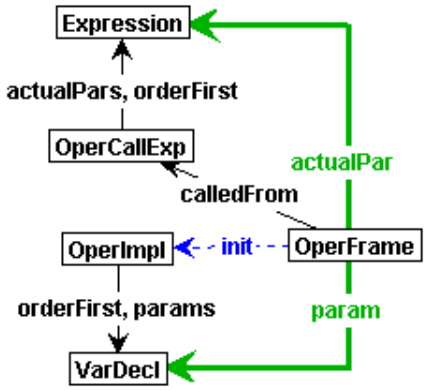


Figure B.43: OperImpl-param-first.gpr

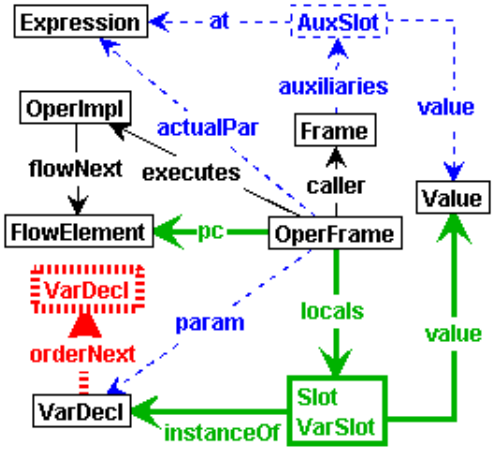


Figure B.44: OperImpl-param-last.gpr

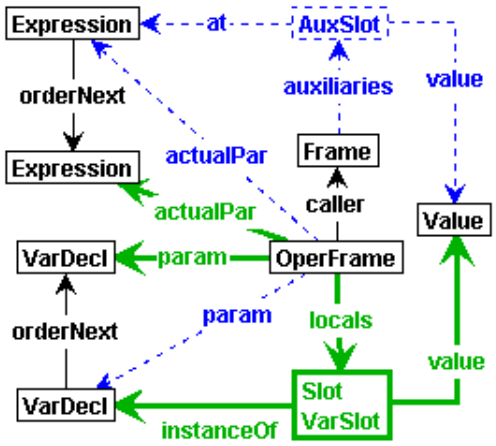


Figure B.45: OperImpl-param-next.gpr

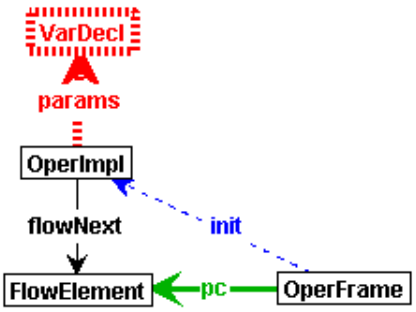


Figure B.46: OperImpl-param-none.gpr

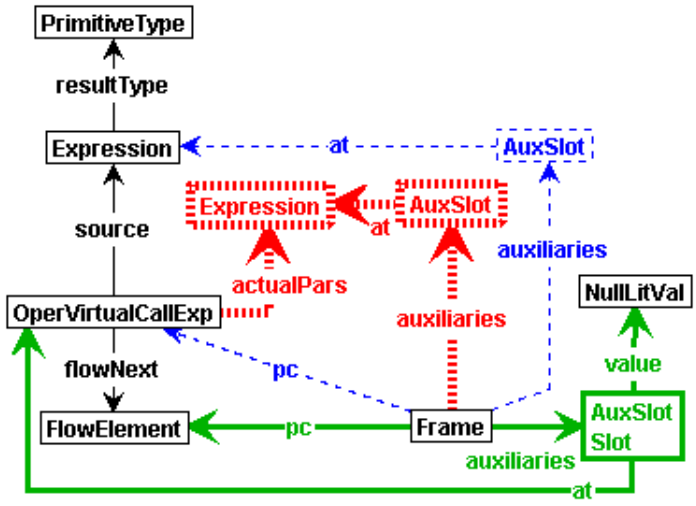


Figure B.47: OperPrimCallExp.gpr

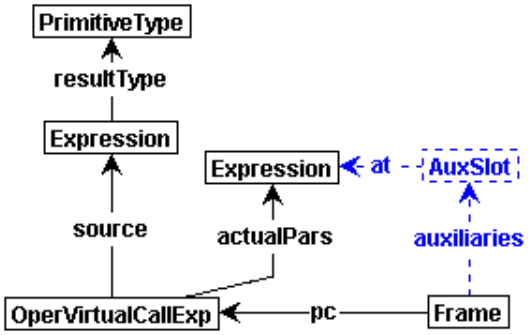


Figure B.48: OperPrimCallExp-garbage.gpr

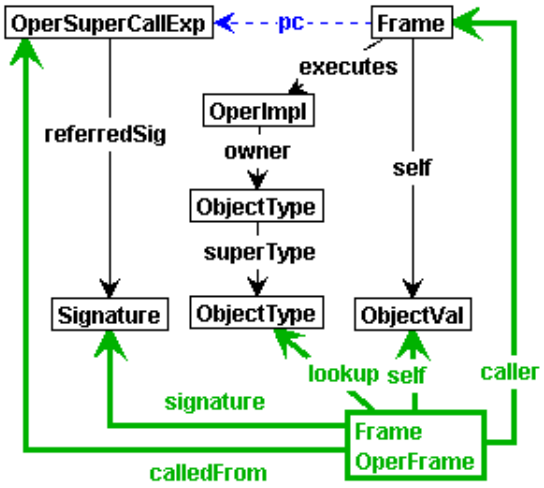


Figure B.49: OperSuperCallExp.gpr

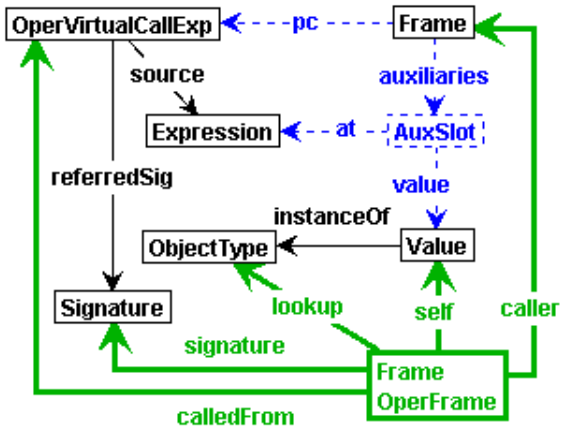


Figure B.50: OperVirtualCallExp.gpr

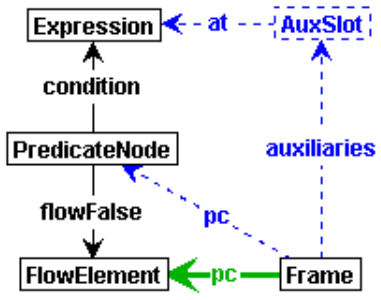


Figure B.51: PredicateNode-false.gpr

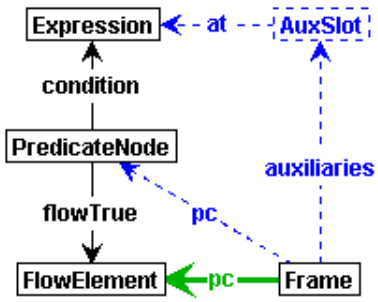


Figure B.52: PredicateNode-true.gpr

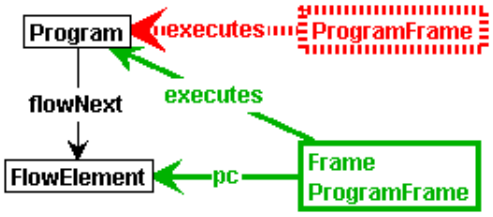


Figure B.53: Program.gpr

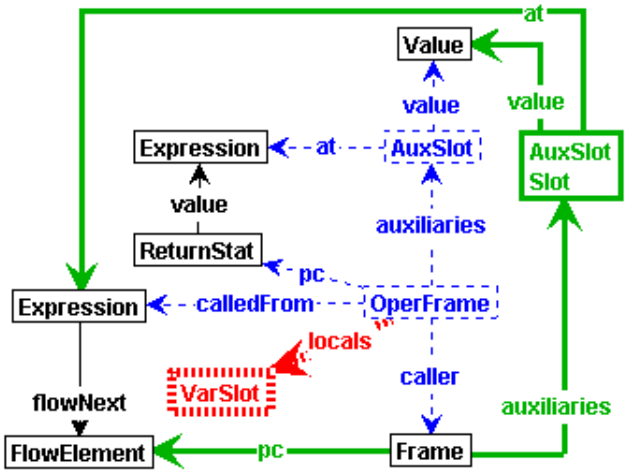


Figure B.54: ReturnStat.gpr

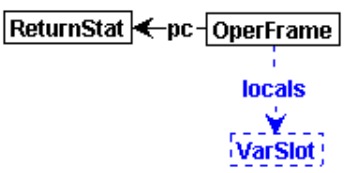


Figure B.55: ReturnStat-garbage.gpr

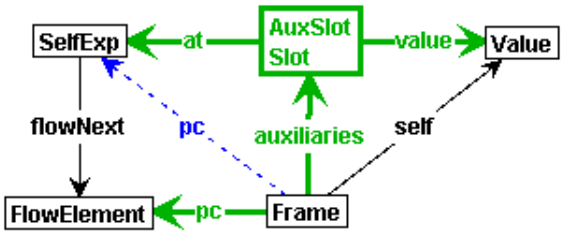


Figure B.56: SelfExp.gpr

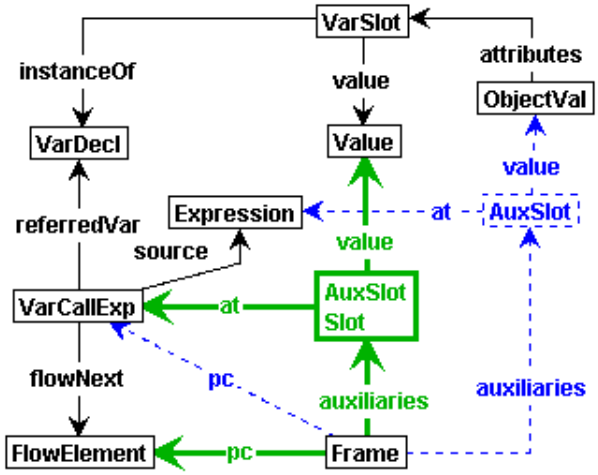


Figure B.57: VarCallExp-attribute.gpr

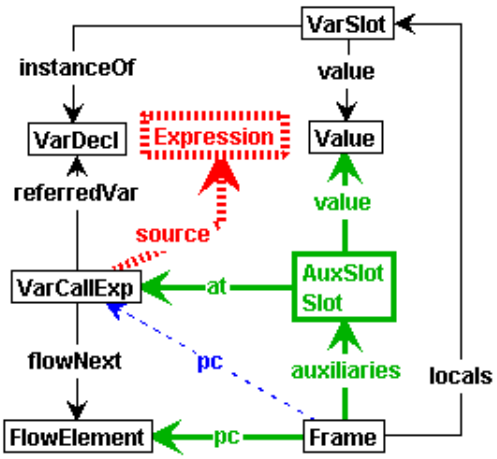


Figure B.58: VarCallExp-local.gpr

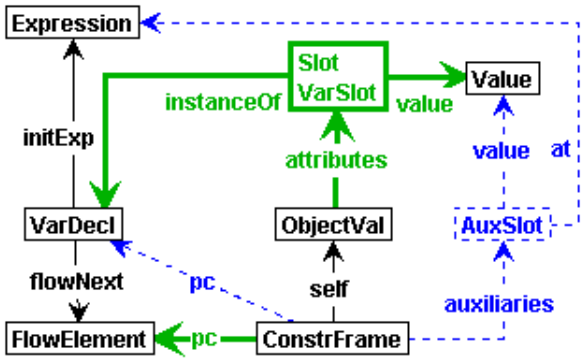


Figure B.59: VarDecl-attribute.gpr

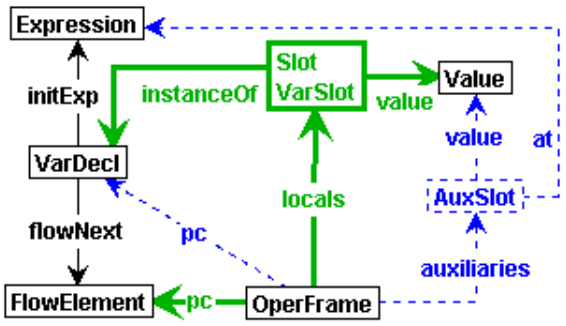


Figure B.60: VarDecl-local.gpr