Specification and Construction of Control Flow Semantics

a generic approach using graph transformations

R.M. Smelik

Master of Science Thesis

Enschede, 26th January 2006

Twente Research and Education on Software Engineering (TRESE)
Dept. of Electrical Engineering, Mathematics and Computer Science
University of Twente, the Netherlands

Committee	Dr. ir. A. Rensink
	Prof. dr. ir. M. Akşit
	ir. H. Kastenberg

Abstract

The semantics of programming languages lack a formal, standardized specification language. We focus on control flow semantics and propose a graphical specification framework for these semantics, consisting of three elements: a graphical control flow specification language, a rule-based approach for constructing flow graphs and transformations from the former to the latter.

In this thesis we introduce a control flow specification language (CFSL) with which a language designer can specify the control flow semantics of all constructs that are featured in the programming language he or she designs. A control flow specification in CFSL consists of a set of specification graphs that adhere to the CFSL meta-model.

We also presents a structured, rule-based approach for constructing a flow graph (FG) for a program written in a particular programming language. In this approach, we use graph transformations to transform an abstract syntax graph representation (ASG) of the program into a FG. Such a graph transformation system consists of a set of programming language specific FG construction rules.

Transformations between the two models are performed by another set of graph production rules: the FG meta-rules. These meta-rules generate the FG construction rules for a programming language from a control flow specification of that language in CFSL, thereby eliminating the need for hand designing the FG construction rules.

Contents

Chapter 1

Introduction

Like natural languages, all programming languages have a grammar, which specifies the legal syntactical structures of the language. This grammar is typically specified in a formal notation called Extended Backus-Naur Form (EBNF [9]). Besides a syntactical structure, a programming language has semantics. The semantics of a programming language, contrary to the syntax, is typically specified informally, using natural language. The absence of a formal specification language for the semantics of programming languages might in some cases introduce ambiguity in the interpretation of the semantics of a programming construct. But the main problem is that it makes automated reasoning or correctness proving of, for instance, refactoring operations more difficult.

These problems also play a role in Model Driven Architecture (MDA) [12]. In the MDA approach several, preferably automatic, transformation steps are developed in order to transform a platform independent specification model of a system to a model which depends more on a specific platform or technology. This process ideally ends at the level of an executable model. In order to assure the correctness of any of these transformations, we need a formal specification of the semantics of both the source language and the target language of the transformation step. These are then used to prove that the transformation step preserves the semantics of the source model.

It is therefore clear that it is desirable to have a standard, formal specification language for the semantics of programming languages, comparable to the EBNF standard that we have for the syntax of languages.

This thesis will introduce a specification language that focusses on a subset of the semantics of programming languages, namely the *control flow* semantics. To put it simply, control flow defines the *order* in which individual operations in a program are executed. The control flow semantics of a programming construct is described in terms of the influence this construct has on this execution order in a program.

1.1 Goal

The goal of this thesis is:

> The development of a formal specification language for the control flow semantics of programming languages, using graphs and graph transformations.

1.2 Approach

Our research has resulted in a framework for formal specification of control flow semantics, consisting of:

1. A control flow specification language (the main goal of this research project);
2. A flow graph construction approach;
3. Transformations between the above two elements.

All the elements of this framework have been implemented, using graphs and graph transformations. Below, we give a brief introduction to each element. The coming chapters will treat them in far more detail and present an instantiation of this framework for our example programming language.

1.2.1 Control flow specification language

The purpose of our control flow specification language (to which we will often refer as CFSL) is for a language designer to be able to formally specify the control flow semantics of the programming language he or she is designing. The most important requirements for CFSL are the following:

1. The language should have formal semantics;
2. The language should be programming language independent, i.e. generic;
3. A control flow specification for a particular programming language in CFSL should have a close relation to the syntax of the programming language;
4. The language should be powerful enough to be able to specify all control flow semantics for any *imperative* programming language (including object-oriented languages with imperative programming constructs).

When we specify the control flow semantics of a particular programming language in CFSL, we design a *control flow specification* for each individual programming construct in the programming language. Such a control flow specification states the influence the construct has on the flow of control of any program that features this construct. Together, this set of control flow specifications forms a specification of the control flow semantics of the entire programming language.

Regarding requirement 1, our CFSL is based on the mathematical *graph* datatype: control flow specifications in CFSL are graphs which adhere to the CFSL meta-model. The CFSL meta-model consists of programming language independent (requirement 2) graph elements (i.e. nodes and edges) we use to denote control flow in a specification.

We have chosen graphs as the base of our specifications because of a number of reasons. The most important reason is that we do not want our specifications to be limited to a tree-structure, as is present in a parsed syntax tree or, more implicitly, present in a (E)BNF grammar rule. Also, by using graphs we are able to apply *graph transformations* to CFSL specifications, as we will explain later on.

Regarding requirement 3, the close relation with programming language syntax is realised by the fact that we use a graph representation of abstracted syntax of a particular programming language construct as the base of a control flow specification for this construct. For this base graph we specify the control flow of the programming construct with (graph) elements present in the CFSL meta-model.

It is not straightforward to show that requirement 4 is fulfilled by CFSL. At the end of this thesis, we present a critical evaluation of the applicability of CFSL to imperative programming languages.

1.2.2 Flow graph construction

In the previous section we introduced CFSL in which we can specify the control flow semantics of a *programming language*. We now consider constructing the control flow of a *program* written in some programming language.

In our framework, we represent the control flow of a program as a *flow graph*. For program analysis and measurements of the structural complexity of a code fragment, programs are often represented as flow diagrams or flow graphs [5]. The properties of these graphs (like the number of independent paths in a flow graph) can provide the necessary information for this kind of analysis [20].

There are many possible ways of modeling flow graphs, ranging from detailed and accurate (with respect to real program execution) representations to abstract representations that are more suitable for (mathematical) analysis of complexity properties. We use a detailed flow graph model, which is based on an abstract graph representation of syntax to which we introduce control flow information. We present an elaborate flow graph meta-model that accommodates for the different types of control flow we discern.

We present a structured, rule-based way of introducing control flow information. For each type of programming construct, we design a rule that decorates its abstract syntax representation with our representation of its control flow semantics. We apply a set of these rules in order to transform an abstract syntax graph representation into our flow graph representation.

The abstract syntax representation we use is a graph, i.e. an *abstract syntax graph* and, of course, our flow graph is also a graph. Because of this we are able to use a transformation technique known as *graph transformations* [15]. When using graph transformations, we have a graph grammar or graph production system, which consists of a set of *graph production rules* and a *start graph*. The start graph is, in this case, an abstract syntax graph representation of some program (fragment) for which we want to construct a flow graph. This program is written in a specific programming language. For each programming construct that is featured in this language, the set of production rules provides a rule that transforms a *matching* part of the abstract syntax representation into a partial flow graph. In other words, for each type of statement we have a rule that introduces the control flow information of this statement to the abstract syntax graph of the program that features this statement. After applying all matching rules to the abstract syntax start graph, we end up with a completed flow graph.

Even when we are committed to use graphs and graph transformations, there are still many possible ways to represent control flow (i.e. our flow graph meta-model), to construct a flow graph, to abstract from concrete syntax, etc. The flow graph construction approach and the related meta-models we use in this thesis are all inspired by from an extensive case study that we performed on constructing flow graphs, using graph transformations. During

this case study, we have developed and applied several different approaches, before coming to the final approach and meta-models that we use in this thesis.

1.2.3 From specifications to construction rules

As we mentioned earlier, we present CFSL as a means for a language designer to formally specify the control flow semantics of a programming language. A language designer specifies for each programming construct of the language he or she is designing the control flow semantics as a control flow specification. This results in a set of formal specifications that can accompany the formal syntax specification of the programming language.

We also mentioned the purpose and benefits of representing a program as a flow graph and having a structured way for constructing such graphs. As follows from the case study, designing flow graph construction rules by hand is time-consuming and not a trivial task. Fortunately, graph transformations can aid us here (again).

Our control flow specification framework includes a set of *flow graph meta-rules*: graph production rules that transform graphs intro graph production rules. The source graphs of these meta-rules are a set of hand-designed control flow specifications in CFSL. The production rules that result from the application of these meta-rules are the corresponding flow graph construction rules. Therefore, there is no need to design these flow graph construction rules by hand.

1.2.4 Running example: the Java programming language

To provide concrete examples for both CFSL as our flow graph construction approach, we have chosen for the Java programming language [19], a modern and widespread programming language. We have composed an abstract grammar which features most Java constructs. For these constructs we present a set of control flow specifications in CFSL. We also present a corresponding set of flow graph construction rules for these constructs in this thesis, which were generated by our flow graph meta-rules.

1.3 The big picture

It is time to present an overview of all elements of this research mentioned above (the big picture). For this, we look at Figure 1.1.

This figure features three meta-levels. Level 1 is at the level of program defined in a specific programming language. Level 2 is at the level of a specific programming language, of which a program of course is an instance. Level 3 is at the level of meta-languages, the level in which parts of a specific programming language are defined.

Three types of actors play a role in this figure. We have a *researcher* who designs meta-languages and meta-models. Next we have a *language designer*; he or she designs the syntax and semantics of a specific programming language. And last, we have a *developer*; he or she develops a program in this specific language.

We follow the chains, starting with the role of the language designer. The language designer specifies the legal syntax of a new programming language (6) in EBNF (1). This grammar is abstracted by the language designer to an abstract syntax graph meta-model (7) specific to this language.

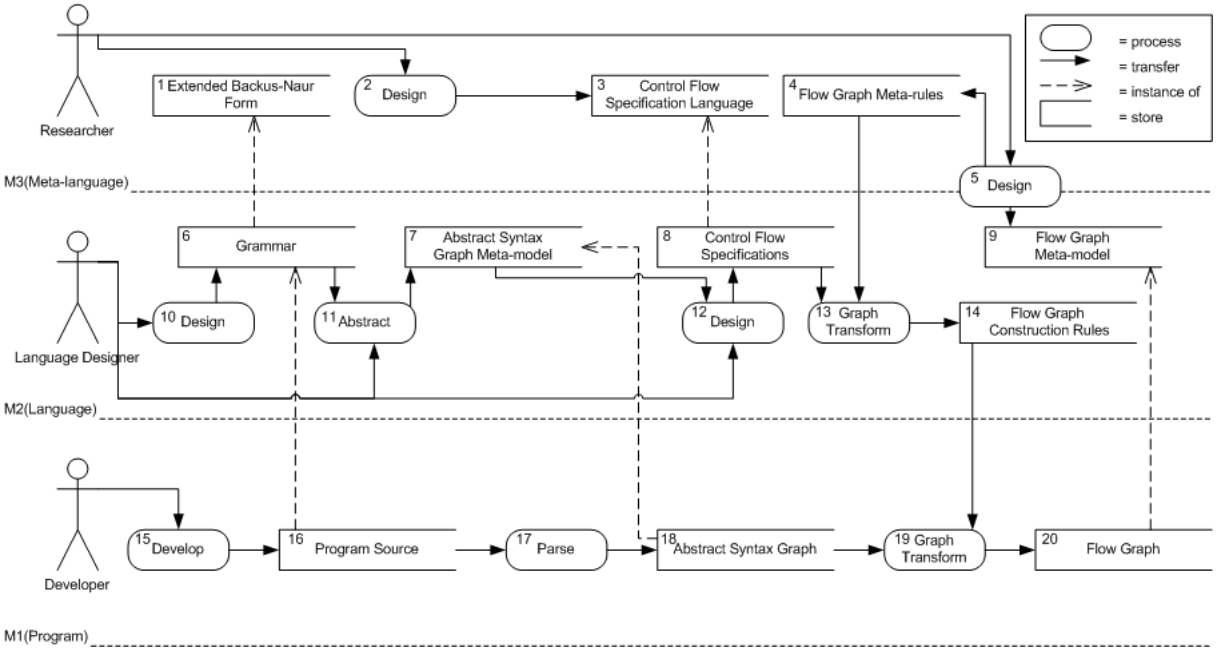


Figure 1.1: Overview of the chain of the elements involved in this thesis and the levels on which they reside.

Next he or she specifies the control flow semantics of the language, not in natural language, but in formal specifications (8) in our control flow specification language (3).

Graph transformations (13) are performed on these specifications using the GROOVE tool set [14], which applies our designed set of flow graph meta-rules (4). The performed graph transformations result in a set of corresponding flow graph construction rules (14).

Now we examine the role of the developer. The developer writes (15) some program (16) in this new programming language, conforming to the grammar of the language (6). The program is parsed (17), resulting in an abstract syntax graph (18) that conforms to the abstract syntax graph meta-model that is specific for this programming language (7). The abstract syntax graph is input for the GROOVE tool (19), were the generated set of flow graph construction rules (14) are applied. The resulting transformations result in flow graphs (20) that have been attached to the input abstract syntax graph and adhere to the flow graph meta-model (9).

1.4 Overview

We now give a brief overview of the chapters in this thesis.
Chapter 2 introduces the formal notion of graphs and explains the aspects related to the graph transformation technique and production rule application.

Chapter 3 introduces two types of graphs that play a major role in this thesis: abstract syntax graphs and flow graphs. It presents both meta-models and illustrates these graphs by some examples. Chapter 4 presents our flow graph construction approach and the example flow graph construction rules we have developed for the Java programming language.

Chapter 5 introduces our control flow specification language CFSL, the main result of this thesis. As an example for our specification language, we present a large number of example control flow specifications of Java statements.

In Chapter 6 we present our set of flow graph meta-rules that transform control flow specifications into corresponding flow graph construction rules.

Chapter 7 evaluates our control flow specification language and identifies its applicability and limitations.

Finally, Chapter 8 presents the conclusions of this thesis and summarizes related and further work.

Chapter 2

Graph Transformations

This chapter gives an introduction to graph transformations as they are used in this thesis. In the big picture (Figure 1.1) of this thesis, presented in Chapter 1, we have seen that graph transformations are applied to transform control flow specifications into flow graph construction rules (13) [1] and to construct a flow graph given an abstract syntax graph (19).

Graph transformations are a systematic, rule-based transformation technique. They have a solid research foundation [17] and have many areas in computer science for which they are found suitable to apply (e.g. [15, 3, 7]).

In the next section we introduce the formal definitions of the graphs we use. In the last section, we introduce graph transformations and conclude this chapter with an introduction example to graph transformations.

2.1 Graphs

Graphs are our mainly used datastructure, in fact we represent all components of the framework we introduce as graphs. All our graphs conform to the following formal definition.

Definition 2.1.1. [15] A graph is a tuple $\langle Nod, Edg \rangle$ where

- Graphs are specified over a global, finite set *Lab* of labels;
- *Nod* is a finite set of nodes;
- *Edg* is a subset of $Nod \times Lab \times Nod$, i.e. a (finite) set of edges.

Our graphs are *directed* graphs, meaning that we discern for each edge a *source* and a *target* node. As a result, our edges can only represent *binary* relations (i.e. we have no *hyperedges*).

As follows from the definition, edges are labeled. Nodes are not labeled by definition, but we can have edges with the same source and target node. These edges are called *self-edges* of a node and, for practical purposes, can be considered as the labels of a node (as a result, a node can have several labels). As we use a *set* of edges, we do not have multiple edges with the same source, target and label (i.e. no *parallel* edges).

Graphically, nodes are represented as black rectangles and edges as black arrows. Self-edges can be represented as labels of nodes (graphically depicted inside the rectangle), or as arrows with the same start and end node.

Figure 2.1 shows an example graph. This graph is part of the Ferryman Problem example that will be elaborated in Section 2.2.4. Note that we have represented self-edges as labels of nodes in this figure.

[1] These numbers refer to one of the number elements of Figure 1.1.

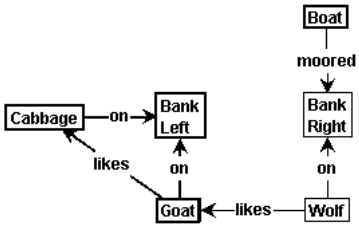


Figure 2.1: An example graph that conforms to our definitions.

2.2 Graph transformations

Now that we have made clear what type of graph we use in this thesis, we introduce the graph transformations technique and show how we can apply this technique to our graphs.

A *graph transformation system*, or graph grammar [17], consists of a set of *graph productions rules* and a start source graph.

A graph transformation system can transform a graph called the *source graph*, into another graph called the *target graph*. The target graph is a transformed version of the source graph. What the changes to this source graph are depends on what is specified in the graph production rules that were applied to this source graph.

2.2.1 Graph production rules

A graph production rule consist of two graphs, a *left hand side* L and a *right hand side* R. A production rule p has the form: $L \xrightarrow{r} R$. The left hand side L is matched to (a part of) the source graph and the occurrence of L in the source graph is replaced by R, resulting in the target graph. Thus, the left hand side L is matched on the source graph and the right hand side specifies elements (nodes or edges) of the matching sub graph (of the source graph) to be preserved, elements to be removed and new elements to be introduced in the target graph. r is a *partial graph morphism*, which identifies which elements (nodes or edges) in L correspond to which elements in R. Note that we use the Single Pushout Approach, were in the Double Pushout Approach the correspondence of elements is, among others, identified using a *common interface graph* or *gluing graph* [17].

Matching of a left hand side L of a graph production rule on a source graph G is NP-complete.

The technical details of our production rules and how they are applied are treated more thoroughly in [17] and, more specifically, in [15].

The production rules are made up from four different kinds of elements:

Readers The readers are elements that are present in both L and R. They have to be present in the source graph in order for L to match and are preserved in the target graph.

Erasers The erasers are elements present in L but not in R. Thus, they have been matched in the source graph but are not found in the target graph, i.e. they are removed.

Creators The creators are elements absent in L but introduced in R. They are added to the target graph.

Embargoes The embargoes (or negative application conditions) are neither present in L and R but are in fact an extension to standard graph transformations [15]. These elements have to be absent in the source graph and in a matching of L on the source graph for the production rule to apply.

We give an example application (Pushout) of a graph production rule to a graph in Figure 2.2. The matched nodes and edges in the source graph are depicted bold. By comparing L and R we can see that a node and two edges are erased by this production rule. Note that the representation of the negative application condition moored in L is technically incorrect. In practice, after a match of L has been found in the source graph, the negative application elements are added to this matching graph. Only when this extended matching graph does not (again) match the source graph the rule applies.

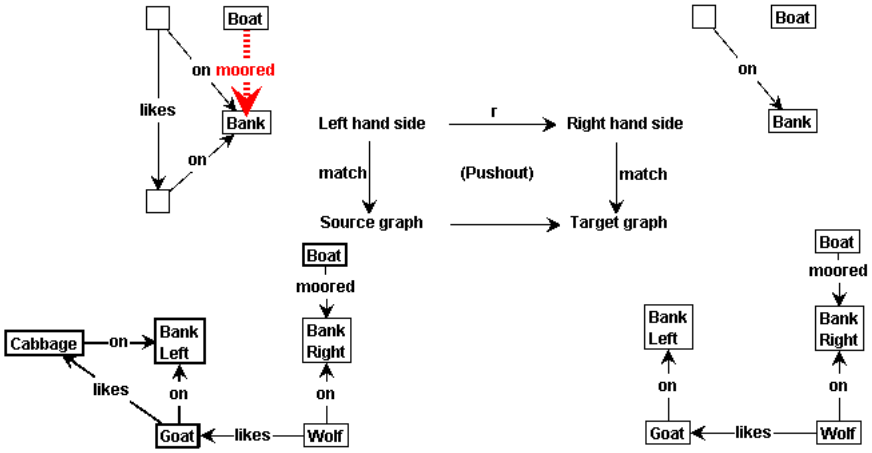


Figure 2.2: An application of a graph production rule to a graph.

In our graphical presentation of graph production rules, the left and right hand side are combined in a single graph. To discern the four types of rule elements, each element has a distinct color and form, listed below. Figure 2.3 shows the combined version of the graph production rule presented in Figure 2.2.

Readers Readers are presented as black rectangles and arrows;

Erasers Erasers are presented as dashed, blue (darker gray in black and white presentations) rectangles and arrows;

Creators Creators are presented as bold, green (light gray in black and white presentations) rectangles and arrows;

Embargoes Graphically, they are represented by bold, dashed, red (dark gray in black and white presentations) rectangles and arrows.

The matchings of readers and embargoes can be refined by extending the labels of the elements with path expressions. The path expressions used in this thesis are (a subset of) regular expressions supported by the GROOVE tool. We list these regular path expression operators in a grammar:

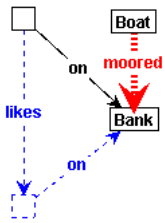


Figure 2.3: An example of the graph production rule depicted with L and R combined.

$$x ::= a \mid x \cdot x \mid x \,.\, x \mid x * \mid ?$$

The Kleene star ($*$) matches any number of subsequent edges that match the path expression that is left to the operator (the direction of the edges is of course relevant). The choice-operator (|) matches a path that matches the path expression left of the operator or the path expression right of the operator. The path construction operator (.), matches a path that matches the path expression on the left side of the operator followed by the path expression on the right side (the direction of the edges is again relevant). The wilcard operator (?) matches an edge with any label.

Another operator we often use is the equality operator = which can be the label of a creator or embargo. When it is used as the label of a creator, it *merges* the two nodes it connects to one node, which receives all incident edges of the original nodes. When it is used as the label of an embargo, it states that the two nodes it connects may not be matched to the same node for a matching to be valid. This is sometimes useful, as the matching of elements in the production rule on elements in the source graph can be non-injective (e.g. two nodes in a production rule may be matched onto one node in the source graph).

2.2.2 Rule applications

In a graph transformation system, a set of graph production rules are applied to a start source graph. After each application, the original start graph will be somewhat changed (i.e. transformed). This transformation process typically continues until none of the production rules are applicable anymore to the changed intermediate graph; we can then say that the transformation is complete.

However, at any moment during the transformation process, several rules may apply to the intermediate graph. We can choose to apply one of these rules arbitrarily or explore all applications. This results in a tree-like structure of rule applications and resulting intermediate graphs. Because we check each intermediate graph on isomorphism with all other intermediate graphs and connect rule application paths when they result in isomorph resulting graphs, a tree-structure is not suitable. We represent rule applications using a Labeled Transition System (LTS), in which each node is an intermediate graph (the root node is that start graph) and each edge represents a rule application (and is labeled with the name of the rule).

As said, several graph production rules may apply to the same intermediate graph at the same time. In some cases, each production rule application could eventually be leading

to a different final graph. When this is the case, the order in which the applicable rules are applied to the intermediate graph influences the resulting final graph.

In our case this is not desirable. We normally design production rules, which will eventually lead to the same target graph, independent of the order in which the rules are applied. Such rules are called *confluent*.

As the order in which applicable production rules is not important when the rules are confluent, we can explore the rule applications in a *linear* fashion: for each intermediate graph, one applicable rule is chosen arbitrarily. When the order of the applicable production rules is important, we have to perform a *full* exploration of the rule applications state space to end up with all possible final graphs.

We have another (primitive) mechanism for guiding the applications of production rules: rule *priorities*. When a production rule has been given a higher application priority than other rules, the other applicable rules can only be applied in case the rule with higher priority is not applicable.

2.2.3 Graph transformation tool

For graph transformation, we us the GROOVE [14] tool, which consists of an editor for creating graphs and graph production rules and a simlator for performing graph transformations.

2.2.4 Example: The Ferryman Problem

As an introduction example to graph transformations we consider a classical problem called the Ferryman Problem. We have a ferryman who wants to transport his wolf, his goat, and his cabbage to the other side of a river. The problem is that he has a boat which is only large enough for himself and one other animal or vegetable. Matters are complicated even more for this ferryman by the fact that his possessions have created their own small food chain: his wolf would very much like to devour his goat, while his goat has an interest in his cabbage vegetable. Both the wolf and the goat behave while the ferryman is there to watch them, but will start eating that what they like the moment the ferryman rows away.

We represent this problem as a graph transformation system. Such a system consists, as mentioned above, of a start graph and a set of graph production rules. The start graph (Figure 2.4) is the representation of the start scenario of the Ferryman Problem. The river to cross has two Banks: one Left and one Right (remember, nodes can have several labels which are actually labeled self-edges of the node).

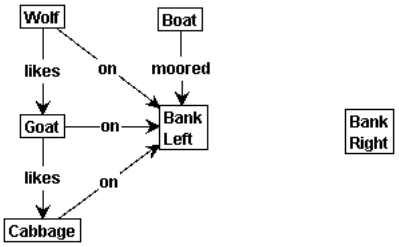


Figure 2.4: The start graph of the Ferryman problem.

The ferryman itself has no explicit representation in our graph, as we are only concerned with the position of the Boat. The Boat can be moored on either Bank or be going to the other side of the river.

The two animals and the vegetable are represented as nodes labeled Wolf, Goat and Cabbage. As we know, the Wolf likes the Goat and the Goat likes the Cabbage. All three can be either on one of both Banks or loaded in the Boat. The start graph states that all three are on the left bank and the Boat is moored on this bank too.

The goal of the ferryman is to transport the wolf, goat and cabbage to the other side of the river. The corresponding goal state graph (Figure 2.5) is the mirrored version of the start graph. There are many possible states which are considered a failure. All fail states have in common that either the Goat or the Cabbage or both have been devoured by some animal. An example fail state graph is shown in Figure 2.6. Here we see that, although the Wolf has been transported across the river, both the Goat and the Cabbage are no more.

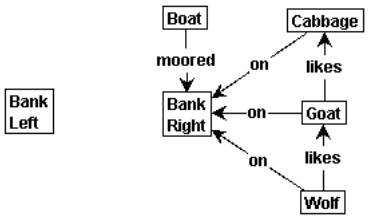


Figure 2.5: The goal state of the Ferryman problem.

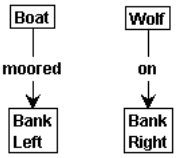


Figure 2.6: A fail state of the Ferryman problem.

Now that we have treated the graph representations of states of the Ferryman Problem, we treat the set of graph production rules that are part of the graph transformation system for this problem.

The first production rule we treat is the eat rule (this rule was already depicted in Figure 2.3). This rule performs eating actions according to the food chain. We use a simple and standard technique to make this rule more general: we do not match the pairs (Wolf, Goat) and (Goat, Vegetable), but instead only match on the likes relation between two unlabeled nodes. As the nodes have no labels, the rule matches on any two nodes that are connected with a likes edge. This way, one rule suffices for both pairs in the food chain. Both elements should of course be on the same Bank in order for the eating process to commence. And, another important condition is that the Boat is *not* moored on this Bank, we therefore have an embargo. The result of the eat rule is that one animal or vegetable will be devoured when applying this rule, i.e. erased.

Figure 2.2 showed a sample application of the eat rule (with the left and right hand side

shown as separate graphs). In the *source graph* for the rule application, the matching nodes and edges are depicted bold. We can see that the eat rule applies to this source graph, as the Boat is moored on the other side of the river, while the Cabbage and Goat are left unattended on the left bank of the river. The figure also shows the *target graph*, the state graph that results from the eating of the Cabbage. We see that the Cabbage node and his outgoing on and incoming likes edges have been deleted.

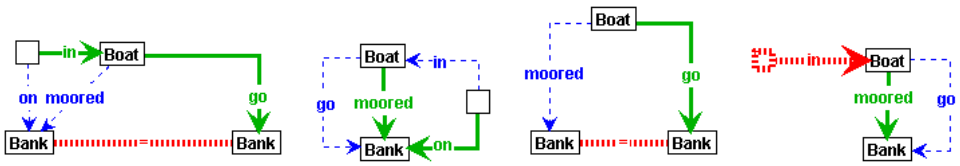
Figure 2.7 shows production rules for the four actions the ferryman can perform:

Load This rule loads one possession that is present on the same bank as the Boat is moored in the Boat (its on edge is deleted and a new in edge is created) and starts the rowing (by creating a go edge) to the other side (an embargo = specifies that the banks may not be the same node);

Unload This rule unloads one possession on to the Bank to which was rowed; the Boat is moored there too;

Go empty This rule rows the Boat to the other side (again this cannot be the same Bank node) without loading any cargo;

Arrive empty This rule mores the Boat without unloading any cargo (the Boat must be empty, i.e. there cannot be any node in the Boat).



(a) Load (b) Unload (c) Go - empty (d) Arrive - empty

Figure 2.7: The graph production rules for loading and moving the boat.

Figure 2.8 features the somewhat peculiar rule final: it consists only of readers, meaning that the source and target graph of an application of this rule will be equal. This rule is used to indicate that the goal has been reached, i.e. if this rule applies to the current state graph, the Ferryman Problem has been solved.

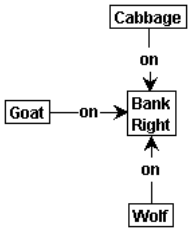


Figure 2.8: The graph production rule that applies when the success-state is reached.

When we simulate the transformations in this graph transformation system, most sequences of rule applications lead to a fail state, i.e. a state from which a state in which the

final rule applies cannot be reached. Rule applications sequences that lead to a state graph in which the final rule applies are solutions to the Ferryman Problem. Figure 2.9 shows a (partial) rule application LTS in which a sequence of rule applications lead to a graph to which the final rule applies. This sequence is a solution for the Ferryman Problem. Exploring any sequence from a state graph labeled open might lead to other solutions or failures.

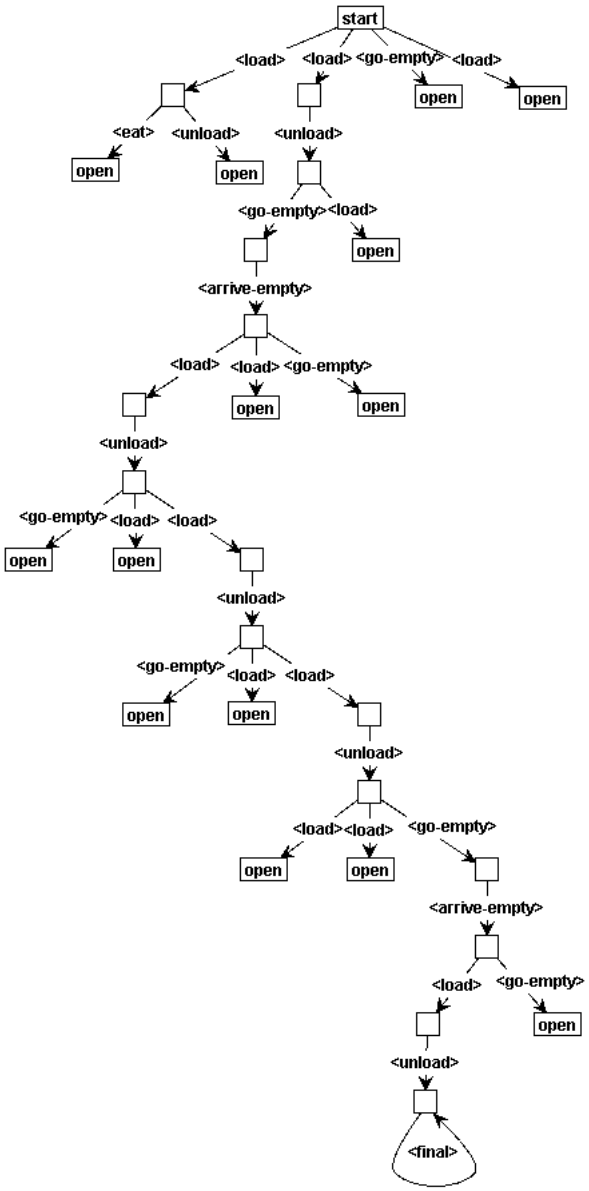


Figure 2.9: The partial rule application LTS leading to a solution of the Ferryman problem.

Chapter 3

Graph Representations

This chapter treats two important graph representations in this thesis: abstract syntax graphs (ASG's) and the flow graphs (FG's). Related to the big picture of this thesis (Figure 1.1), this chapter presents the abstract syntax meta-model (7) and the flow graph meta-model (9) and gives examples of an abstract syntax graph (18) and flow graph (20) for a Java program.

3.1 Abstract syntax graphs

The (context-free) grammar of a programming language is typically specified in (extended) Backus-Naur Form ((E)BNF) [9]. When parsing the source code of a program written in this language using the EBNF grammar, the result is a concrete syntax tree: a tree structure representing the program's source, containing all syntactic elements (terminals) of the source as leaf nodes and the non-terminals as intermediate nodes.

Syntactic details are often irrelevant to the control flow semantics of the parsed program. Therefore, as is often done in compilers [21], we use an abstract representation of the concrete syntax. In this representation most syntactic tokens (terminals) can be omitted.

A well-known abstract version of the concrete syntax tree is the abstract syntax tree (AST). This tree-structure is often enhanced in another pass with extra context information: bindings of used variables to their declarations, unification of labels for labeled statements, etc. These enhanced (or "decorated" [21]) AST's are in fact no longer trees, but *graphs*.

Our abstract syntax representation is also a graph, hence we call it an *abstract syntax graph* (ASG). Yet the original tree-structure is still clearly visible in these ASG's. The non-terminal symbols in the production rules of a programming language grammar appear as nodes of our ASG's. The syntax tree structure is represented by edges labeled child between parent and child nodes.

Unlike most compiler AST's, our ASG's are not based on a programming language's EBNF grammar, but on an adapted BNF version thereof.

An issue with an EBNF production rule is that the right hand side may feature elements that are optional (enclosed in square brackets). The presence or absence of these elements in a concrete program fragment may affect the control flow semantics of that particular statement. As a concrete example, consider the if-statement that is featured in the Java programming language. The control flow semantics of this statement depends on whether the optional else-part is present. Therefore, when considering the control flow semantics of the if-statement, we actually discern two different statements: an if-then-statement and an if-then-else-statement.

In our abstract syntax, we rewrite rules with optional parts to several new rules, which

we combine using the standard or-operator (denoted by |). In other words, we introduce new non-terminals for each variation of a statement due to optional parts. This actually means that we use plain BNF without extensions for our abstract syntax.

When considering the if-statement in Java again, our abstract syntax has the following BNF production rules (we use < and > to indicate terminals):

```
Statement ::= IfStatement | .. | ..
IfStatement ::= IfThen | IfThenElse
IfThenElse ::= <IF> <LPAR> Expression <RPAR> Statement <ELSE> Statement
```

Appendix A presents an abstract, plain BNF, grammar we have composed for a large portion of the Java programming language (based on the EBNF grammar presented in [6]).

We represent the non-terminals in our abstract syntax as nodes in our abstract syntax graph. Thus, the name of a non-terminal is represented as the label of the corresponding node.

Often, the left hand side of a BNF rule evaluates to one or more non-terminals. There are two possible ways of representing this in our abstract syntax graph, which are similar to two concepts that feature object-oriented programming: *composition* or *inheritance*. If we use composition, the left hand side non-terminal is represented as the parent node and the right hand side non-terminal(s) as child node(s). If we use inheritance, we use the fact that a node can have several labels (represented as self-edges of the node), as we saw in Chapter 2. The left hand side and right hand side non-terminal both appear as labels of the node. Table 3.1 compares both concepts in ASG's to the corresponding UML concepts.

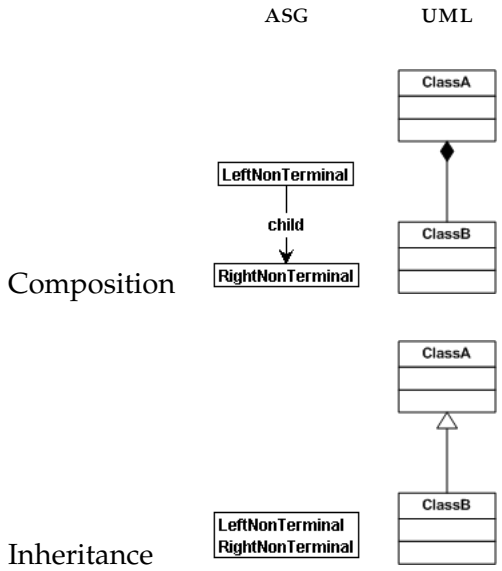


Table 3.1: Composition and inheritance for representing abstract syntax, compared to the corresponding UML concepts.

Inheritance in this context is only possible if the left hand side of the grammar rule evaluates to one non-terminal or several, separated by or-operators (e.g. Statement and IfStatement

```
1 if (x >= 5)
2     x = 0;
3 else
4     x = x + 1;
```

Listing 3.1: Example of an if-then-else-statement in Java.

in the grammar above). In this thesis, we make arbitrary choices for using composition or inheritance.

As an example, the IfStatement BNF rule for Java, presented above, is represented using inheritance: an abstract syntax node, labeled IfStatement, also features the label IfThen or IfThenElse (depending on whether it features an else-part).

Beside the tree-structure parent-child relations, represented by child edges, the syntax representation is decorated with additional information. We annotate the relations between parent and child syntax nodes with edges with labels that explain the role of a child syntax node with respect to its parent. We introduce these annotations to our abstract syntax BNF rules using the following notation: `label:NonTerminal`. Taking Java as an example again, our BNF rule for the if-then-else-statement in Java therefore becomes:

```
IfThenElse ::= <IF> <LPAR> condition:Expression <RPAR> thenPart:Statement
               <ELSE> elsePart:Statement
```

In our graph representation, an IfThenElse node has an edge labeled condition to its Expression child syntax node, an edge labeled thenPart to its then-part Statement syntax node and an edge labeled elsePart to its else-part Statement syntax node.

Figure 3.1 shows the abstract syntax graph representation for if-then-else statements in Java. Note that in this representation, terminals like `if` and `else` and parentheses are not present.

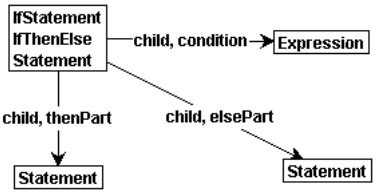


Figure 3.1: The abstract syntax graph representation of the IfThenElse statement.

To give an example of an abstract syntax graph that actually represents a specific source code fragment, we look at Listing 3.1. This listing is a Java code fragment that features an if-then-else-statement. The abstract syntax graph representation of this code fragment is depicted in Figure 3.2. We see here that the Expression and Statement child nodes of the IfThenElse represent actual expression and assignment statements, conforming to the source code. This figure also shows some terminals (in this case, literal values) that have been preserved in the ASG. The precise details of this abstract syntax graph are not important at this point.

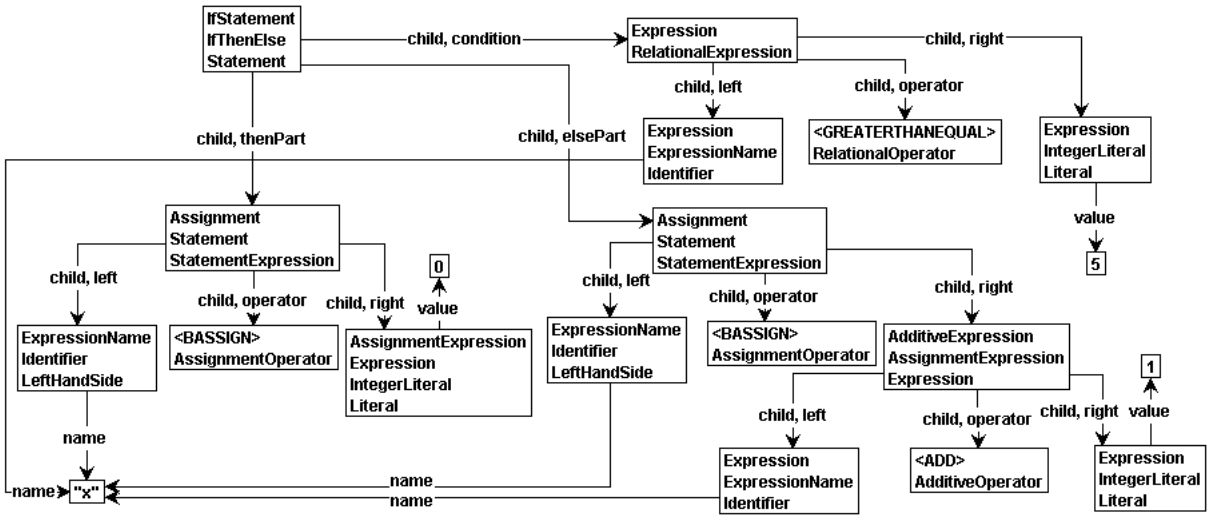


Figure 3.2: An abstract syntax graph of the code fragment in Listing 3.1.

3.1.1 Generic meta-model

As we saw in the big picture (Figure 1.1), an abstract syntax graph conforms to a language-specific abstract syntax meta-model (7). All language specific abstract syntax graph meta-models adhere to a prescribed structure. We have specified this structure in a *generic* abstract syntax meta-model, specific abstract syntax graph meta-models are *specializations* of this meta-model.

The generic abstract syntax meta-model (Figure 3.3) specifies that an AbstractSyntaxElement (i.e. any node representing a non-terminal) can have any number of children (child) and any number of annotated role edges for these children. The number of child edges and annotated role edges need not coincide: a child edge is mandatory for any child of an abstract syntax element, but the annotated roles are optional.

An AbstractSyntaxElement can optionally have a value relation with a literal value terminal. For instance, a BooleanLiteral in Java has a value relation with either the True or the False node.

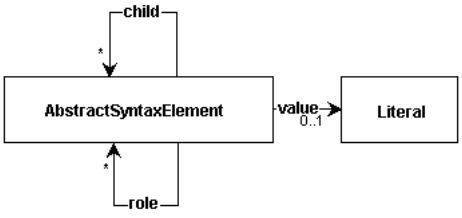


Figure 3.3: The generic abstract syntax meta-model.

A language specific abstract syntax graph meta-model depends, of course, on the programming language under consideration. It is actually a set of graphs representing the abstract syntax BNF grammar rules composed for that language. Figure 3.1 is an example of

such graphs for the abstract syntax grammar we composed for Java (see Appendix A).

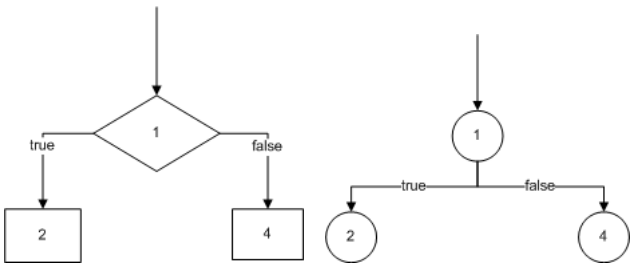
3.2 Flow graphs

Control flow information describes the order in which the individual, atomic, instructions of a program are executed. Most statements are executed in the order they appear in the program's source code, i.e. *sequential* control flow.

However, some statements *change* this sequential flow of control. These statements are often called *control statements*. They do not perform calculations or change the program's state, but determine the order of execution for a group of sequential statements. Most control statements depend in their execution on the value of an associated condition. These statements thereby introduce branches in the program's execution. Depending on the value of the condition one of these branches is executed, i.e. *conditional branching* control flow.

Most languages feature another group of statements that *disrupt* the control flow of a program. The statements perform local (within the method) or in some cases non-local, unconditional jumps. The most (in)famous example is the `goto` statement [4], that is featured in many programming languages. In Java, these statements are referred to as abrupt completion statements. We refer to this type of control flow as *abrupt completion* control flow.

The flow of control in a program is often shown graphically in control flow diagrams or graphs, where arrows or edges indicate how the control is transferred between statements, which are often presented as rectangles or nodes (e.g. [5]). There are many possible ways of modeling flow graphs, ranging from detailed to abstract. Figure 3.4 shows two examples of abstract models of the control flow of Listing 3.1: a flow diagram and a control graph (see [20]). In these figures, the numbers are related to the line numbers of the corresponding listing.



(a) Flow diagram (b) Control graph

Figure 3.4: Examples of control flow diagrams and graphs of Listing 3.1.

We use a far more detailed graphical representation of control flow in the form of a flow graph (FG). Our flow graph is actually an abstract syntax graph which is decorated with control flow information in the form of special-purpose edges and nodes. The semantics and constraints on these control flow elements are described in the following section, in which we treat the flow graph meta-model.

3.2.1 Meta-model

Figure 3.5 presents our flow graph meta-model. We have developed this meta-model during the case study we have performed on designing flow graph construction rules for Java.

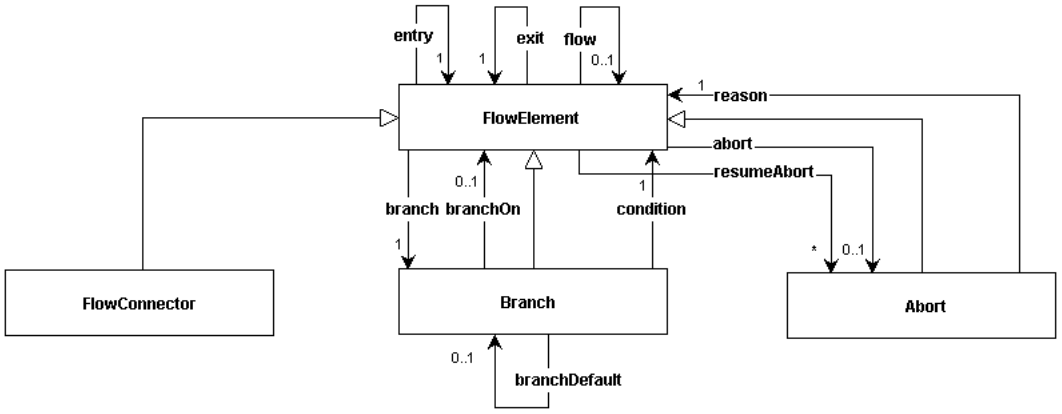


Figure 3.5: The flow graph meta-model.

FlowElement

We consider all abstract syntax elements with control flow semantics to be FlowElements. For each FlowElement, we denote which element is executed when control is transferred to this FlowElement (i.e. its *entry*) and which element represents the end of the FlowElement's execution (i.e. its *exit*). Thus, each FlowElement has one entry and one exit edge. The target of these edges can be the FlowElement itself, another FlowElement (most likely some sub statement) or a FlowConnector.

The auxiliary nodes FlowConnector, Branch and Abort inherit from FlowElement. As we have seen in Table 3.1, this means that these auxiliary nodes also feature a FlowElement edge and have an entry and exit, which are defined as self-edges of the auxiliary nodes.

FlowConnector

FlowConnectors serve as connection points for control flow in flow graphs and have no semantics of their own.

Sequential control flow

Above, we identified three types of control flow: sequential, conditional branching and abrupt completion control flow. Sequential control flow is represented by an edge labeled flow. As sequential flow does not branch, a FlowElement can have at most one outgoing flow edge, but it can be the target of any number of flow edges.

Conditional branching control flow

Conditional branching has a more complex representation in our flow graphs. We represent each possible branch with an auxiliary Branch node. A Branch has several possible edges:

branch Indicates at which statement the conditional branching originates;

condition Indicates the condition FlowElement that results in the actual value for the conditional branching;

branchOn Indicates the literal value to which the condition should evaluate if this branch is to be taken;

branchDefault Indicates that this branch is taken if no other branch can be taken;

flow Indicates to which FlowElement control is transferred upon taking this branch.

A branchDefault edge is used when we have the special `default` case with, for instance, switch-statements in Java, it excludes a branchOn edge. From a FlowElement that features conditional branching we have a branch edge to each possible branch (Branch). For example, branching on a Boolean condition is represented by two Branches, one on `true` and one on `false`. The statement from which this branching originates has a branch edge to each of the Branches.

Conditional branching and sequential control flow are mutually exclusive; a constraint on FlowElements is that they can have an outgoing flow edge or one or more outgoing branch edges, but not both types of edges.

Abrupt completion control flow

Abrupt completion is the most complex form of control flow. The precise details on abrupt completion are treated in Chapter 4, for now it suffices to explain the representations and constraints for abrupt completion in flow graphs. Abrupt completion is represented by an auxiliary node labeled Abort. An Abort has several possible edges:

abort Indicates at which statement the abrupt completion control flow originates;

reason Indicates the FlowElement that causes the abrupt completion control flow;

resumeAbort Indicates at which statement the abrupt completion control flow resumes;

flow Indicates to which FlowElement control is transferred (analogous to Branch).

The edges abort and resumeAbort are mutually exclusive. The FlowElement from which the abrupt completion control flow is followed features an abort edge to an Abort. In some cases, however, abrupt completion is resumed (see Chapter 4). When this is the case, the FlowElement features one (or several) resumeAbort edge(s) to Abort(s) instead.

Example flowgraph

As an example flow graph, we consider the Java code fragment in Listing 3.1 again. The abstract syntax graph of this code fragment was presented in Figure 3.2. The flow graph is shown in Figure 3.6, some syntax elements have been grayed out to make the graph more readable. We can clearly see in this graph that the original ASG has been decorated by control flow elements. Compared to the models in Figure 3.4 it is clear that our FG's are detailed models of the control flow of a program.

Again, the precise details of this flow graph are not important yet. Still, one may wonder where the control flow in this graph starts. Normally, the if-then-else statement is placed in the context of some method that features an ordered list of statements. After execution of the statement ordered before this if-then-else statement, control is transferred to the entry of the if-then-else, in this case the Identifier node that is part of the condition. Thus, the statement ordered before the if-then-else would feature flow edge from its exit to the Identifier node.

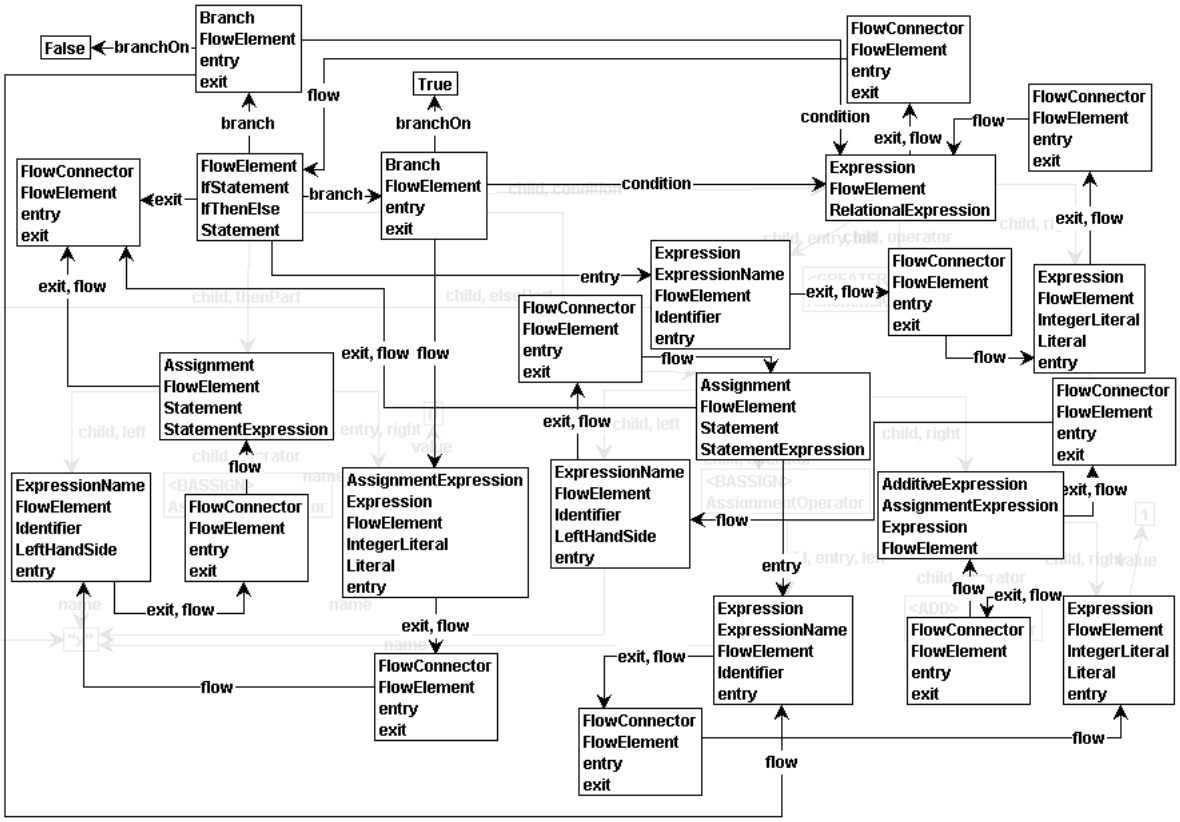


Figure 3.6: A flow graph of the code fragment in Listing 3.1.

Chapter 4

Flow Graph Construction Rules

This chapter introduces flow graph construction rules. In the big picture (Figure 1.1) we have seen that flow graph construction rules (14) are applied to an abstract syntax graph (18, see Section 3.1) representation of a program (16) developed in some programming language. The flow graph construction rules are graph production rules (see Chapter 2) that attach a flow graph (20, see Section 3.2) to such an ASG.

Flow graph construction rules construct a flow graph from an abstract syntax graph by introducing elements (nodes and edges) present in the flow graph meta-model (Figure 3.5). There are many possible approaches for this flow graph construction process. The approach we use for flow graph construction and our flow graph meta-model were both developed during an extensive case study on flow graph construction for the Java programming language.

We first present our flow graph construction approach. Next we illustrate our approach by presenting a large set of example flow graph construction rules for the Java programming language.

4.1 Flow graph construction approach

This section presents our flow graph construction approach. Our approach consist of a number of design choices we have made (during our case study on flow graph construction for Java). We review these choices briefly.

Our approach mainly consist of the following principles:

1. For each type of abstract syntax element, we design one (preferred) or several flow graph construction rules that introduce the necessary control flow elements;

2. Our flow graph construction process operates top-down, starting from the root-node of the flow graph under construction and ending at the level of primitive statements;

3. For each flow element in the graph, we create its *entry* and *exit* (with respect to control flow). Initially, we provide auxiliary flow connectors for these entries and exits;

4. We remove superfluous flow connectors while the flow graph is being constructed;

5. We resolve abrupt completion control flow using a bottom-up resolution process.

We explain these principles in more detail below.

4.1.1 One construction rule per programming construct

Were possible, we design *one* rule per programming construct. This means that for each construct in a programming language, we have a separate production rule that introduces control flow elements to an abstract syntax graph that features that particular type of statement.

We believe that adhering to this standard results in readable and understandable flow graph construction rules, as these rules come close to being a *specification* of the control flow semantics of a particular statement type (for real control flow specifications, we refer to Chapter 5). Most of the example construction rules for Java (Section 4.2) adhere to this principle and are considered by us to be quite intuitive. In some cases though, we have to break from this principle. An example of this are abrupt completion resolution rules (see Section 4.1.4).

4.1.2 Top-down construction process

Flow graph construction is, in our case, a *top-down* process. By top-down we mean that we start at an abstract syntax node that is defined to be the *root* of the flow graph we are going to construct. We start flow graph construction at this root node and continue along its abstract syntax children. More concrete, we start by marking the root node as eligible for flow graph construction. The associated flow graph construction rule for the root node removes the marker from this syntax node and marks all its abstract syntax child nodes as eligible. We represent this marking by introducing or removing a self-edge labeled build to a FlowElement. Flow graph construction rules for children of the root node match on this build to be present. This way, the application of the flow graph construction rules is ordered top-down.

Figure 4.1 shows how in a flow graph construction rule the build marker is passed on. We have a parent syntax node, for which the marker is removed, and n child nodes, for which markers are introduced.

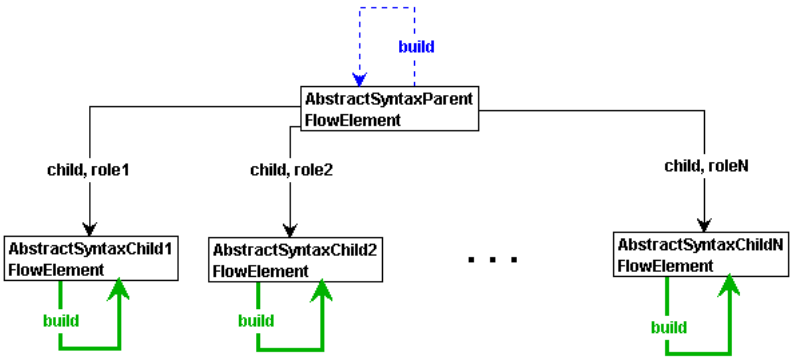


Figure 4.1: The top-down flow graph construction process illustrated.

4.1.3 Flow connectors

As we have seen in the flow graph meta-model (Figure 3.5), all flow elements feature an explicit *entry* and *exit* (with respect to control flow). These are represented by two auxiliary edges, labeled entry and exit.

The control flow of a parent flow element is, among others, defined as the execution of its flow element children. A result of using a top-down approach is that when a parent flow element is under construction, the control flow of its children has not yet been determined. This fact introduces a problem: the parent flow element production rule introduces control flow that states that one of its child flow elements will be executed upon executing the parent. But, at that time in the construction process, it is unknown where the execution of the child flow element starts (i.e. its *entry* is unknown).

Our solution is to provide control flow connector nodes as the (initial) targets for the entry and exit edges of each statement. Parent flow elements can connect their control flow elements to these connector nodes and the child flow elements introduce their internal control flow, starting and ending at their own flow connectors.

During flow graph construction, we introduce entries and exits to each flow element in the flow graph under construction. We connect these edges to auxiliary control flow connectors, represented as nodes labeled FlowConnector. Figure 4.4 shows the generic flow graph construction rule for introducing these elements.

If we were to preserve for every abstract syntax node in a completed flow graph the entry and exit flow connector node, the flow graphs would be somewhat crowded and as a result less readable. Also, in most cases, control flow edges between these flow connectors do not represent actual control flow transfer. In a simulation run of the flow graph these edges would simply be skipped.

We therefore introduce entry and exit flow connectors uniformly to all elements, but we remove superfluous flow connectors during flow graph construction. These flow connectors are merged (see Chapter 2) with other flow connectors or abstract syntax elements, thereby preserving all control flow elements that are connected to the redundant connectors. This merging is performed when the internal control flow structure is known, i.e. when the construction rule associated with the type of flow element is applied.

There are four scenarios in which we consider a FlowConnector obsolete and decide to merge it:

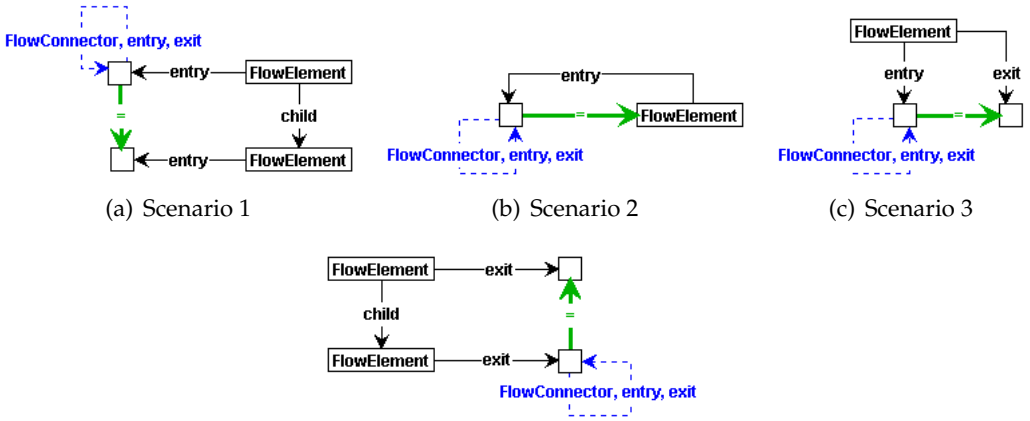
1. A parent flow element's execution may be defined to start at one of it child elements. In this case, the entry flow connector node is merged with the entry node of the child.

2. A flow element may feature no children (a primitive statement) and define execution to start at the syntax node itself. In this case, the entry flow connector node is merged with the syntax node.

3. A flow element may feature no control flow and therefore can be skipped. In this case, the entry flow connector of the flow element is merged with the exit flow connector.

4. A child flow element may define its exit to be the exit of the parent node. In this case, the exit flow connector of the flow element child is merged with the exit of the parent.

We present an illustration for each of the merging processes in Figure 4.2 (remember, a creator with a label = is a merging operation). When merging a node with another node, all edges of the one node, including self-edges, are moved to the other node. The self-edges of the FlowConnector must not be moved to the target node, therefore we delete those edges before the merging.



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

(d) Scenario 4

Figure 4.2: The different merging operations on flow connectors, corresponding to the four scenarios.

4.1.4 Abrupt completion resolution

Any programming language can feature specific statements that introduce control flow that is not defined with respect to the statement itself or one of its sub statements, but that depends on the context the statement is contained in. The transfer of control they introduce is not a sequential transfer, but a jump to some other statement within the context, thereby terminating prematurely one or, possibly, more statements that enclose this jump statement. We call these statements *abrupt completion statements*, as they abruptly complete the execution of one or more enclosing statements.

As these statements disrupt the sequential control flow, flow graph construction rules for these statements are somewhat more involved. The main issue here is that the construction rules have to examine the flow graph context of the abrupt completion statement in order to determine the target statement of the control flow jump that is to be performed. What types of flow elements are eligible to be targets depends on the type of abrupt completion statement.

We refer to the process of finding the correct target flow element for an abrupt completion statement and introducing the abrupt completion control flow to this statement as *abrupt completion resolution*. The abrupt completion resolution process can be approached in many different ways, as we learned during the case study. As we mentioned, we choose to use a bottom-up resolution process.

When an abrupt completion statement is marked for construction, a corresponding flow graph construction rule introduces abrupt completion flow. As in most cases the target flow element is not immediately known, we use a stepwise resolution process: we propagate an abrupt completion marker element upward in the syntax tree (hence bottom-up) until the target flow element is reached. Next we apply another flow graph construction rule that corresponds both to the type of abrupt completion statement and the type of the target flow element. This rule connects the introduced abrupt completion control flow to the flow element to which control should be transferred.

When the target flow element is known at the moment we introduce abrupt completion

control flow, we immediately connect the abrupt completion control flow to the correct flow element.

In some cases, after control is transferred to a flow element because of some abrupt completion statement, after execution of the flow element abrupt completion is reintroduced because of the same abrupt completion statement and another control flow jump is performed. We refer to this as *abrupt completion resumption* (represented by resumeAbort edges).

In our flow graph meta-model (Figure 3.5) we represent abrupt completion control flow with an auxiliary node labeled Abort. As explained in Section 3.2.1, this Abort has an abrupt completion statement as its reason. Abrupt completion starts at the flow element that features an abort edge to this Abort and next the control is transferred to the flow element to which the outgoing flow edge leads. Thus, abrupt completion resolution in flow graphs is finding the correct target node for the Abort's flow edge.

We mentioned that we propagate an abrupt completion marker bottom-up. This marker is an edge labeled resolving that is propagated from child to parent syntax nodes, starting at the origin of the abrupt completion control flow. The propagation of the edge ends when an enclosing statement is reached that terminated prematurely for this reason (i.e. this type of abrupt completion statement). An associated construction rule introduces the flow edge to the flow element to which control should be transferred upon abrupt completion of this enclosing statement and removes the auxiliary resolving edge.

As mentioned, in some cases we can immediately resolve abrupt completion. We then directly connect the newly created Abort with a flow edge to the target flow element.

In other cases, we resume abrupt completion. If this is the case, we create a new Abort with the same reason and start a new resolution process from thereon.

4.1.5 Flow graph construction auxiliaries meta-model

We have two elements that are not present in a completed flow graph, but that we need during its construction: the build edge, used for regulating the top-down construction order, and the resolving edge, introduced for abrupt completion resolution. Figure 4.3 shows the type-graph for these temporary edges.

A FlowElement can have at most one build self-edge, but can have any number of incoming resolving edges. The source of a resolving edge always is an Abort.

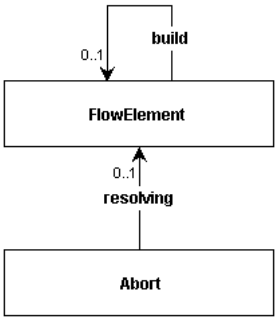


Figure 4.3: The meta-model of auxiliary edges used for flow graph construction.

4.1.6 Auxiliary production rules for flow graph construction

This section presents several auxiliary rules for flow graph construction that are not specific to a particular programming language, but can be used in flow graph construction systems of any (supported) language.

We have already explained the purpose of the construction rule that introduces entry and exit edges and their initial targets, the FlowConnectors (Figure 4.4).

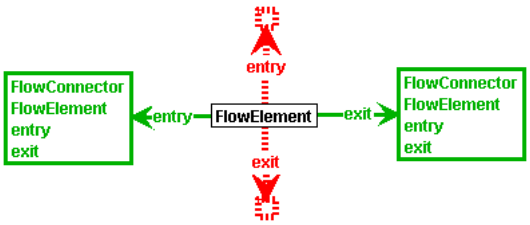


Figure 4.4: Rule for adding entries and exits to all flow elements.

For conditional branching flow, we often specify that a branch is followed if some condition evaluates to `true` or `false`. We assume primitive values to be implicitly present in a flow graph, but due to the fact that the GROOVE tool [14] does not (yet) support this, they have to be created explicitly by a graph production rule (which is not shown here).

The top-down flow graph construction process has to start at the root-node of a flow graph under construction (for example a MethodBody in Java). The rule in Figure 4.5 initiates the construction process by introducing a build edge to the root-node (which we label ContextNode). This rule has the highest application priority.

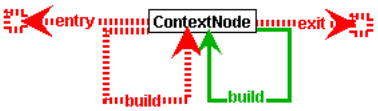


Figure 4.5: Flow graph construction rule for initiating the top-down flow graph construction.

Our abrupt completion resolution process operates bottom-up by propagating the resolving edge from child to parent nodes. Figure 4.6 shows the construction rule that propagates this edge. This rule has the lowest application priority (see Chapter 2), to enable abrupt completion resolution rules to (possibly) match before the propagation is resumed.

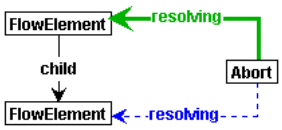


Figure 4.6: Flow graph construction rule for propagating abrupt completion bottom-up.

4.2 Flow graph construction rules for Java

As an example of our flow graph meta-model and our approach for designing flow graph construction rules, we have worked out a significant portion of the statements that feature the Java programming language [19].

The statements we have considered are method bodies, blocks of statements, the `while`, `do` and `for` loop statements, the `if` and `switch` statement, assignments and several types of expressions, the abrupt completion statements `break`, `continue`, `return`, `throw` and the `try`, `catch`, `finally` exception handling statements.

For this we have used an abstract graph representation of a partial BNF Java grammar. In Appendix A, this adapted grammar is given. The changes we made to the original Java grammar in the language specification [6] are described in Section 3.1.

In the following sections we present the flow graph construction rules for Java. All construction rules presented below were actually generated by our flow graph meta-rules, which are treated in Chapter 6, from the control flow specifications for Java we will present in Section 5.2.

4.2.1 Method bodies

Method declarations in Java (see [6]) consist of a method signature and a method body. A method body consist of a block of ordered statements. After execution of the method body, control is transferred back to the calling statement (i.e. a method invocation) of the method.

The MethodBody node is the *start* and *end* point of the flow of control in our flow graphs. For each MethodBody in an ASG, a flow graph is constructed. The MethodBody node is the start point of the top-down flow graph construction process.

The flow graph construction rule of the MethodBody (Figure 4.7) introduces a build edge to the method's body. From thereon, this build edge is passed on to sub statements. The entry of the MethodBody is merged with the entry of the body Block and this Block shares its exit with the MethodBody. This exit is the end point in the flow graph.

When merging a FlowConnector with a FlowElement the edges of the FlowConnector node (i.e. its label FlowConnector, its entry and its exit) should be removed (or end up being self-edges of the FlowElement, see Chapter 2). All rules that merge FlowConnectors feature this eraser edge.

Note that although the Block in this rule has two entry edges, these edges will be matched onto the single entry edge of the Block in a flow graph under construction, because the matching can be non-injective.

4.2.2 Blocks of statements

Like most programming languages, Java features sequentially ordered blocks of statements (see [6] p. 361). These statements are executed in order first to last. For convenience, we repeat the relevant BNF rules from our Java grammar in Appendix A.

```
Block ::= BlockFull | BlockEmpty
BlockFull ::= <LCUR> orderFirst:BlockStatements <RCUR>
BlockEmpty ::= <LCUR> <RCUR>
BlockStatements ::= BlockStatementsNext | BlockStatementsLast
```

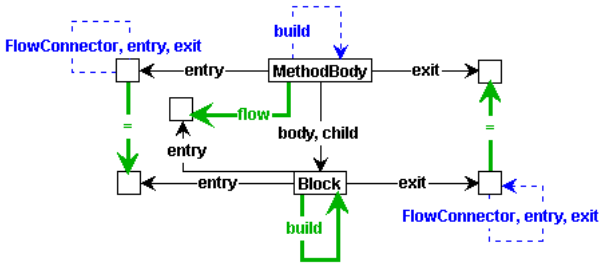


Figure 4.7: Flow graph construction rule for method-bodies.

```
BlockStatementsNext ::= Statement orderNext:BlockStatements
BlockStatementLast ::= Statement
```

Ordered lists of statements are handled by four rules (Figure 4.8). The statement ordered first in a (non-empty) statement block receives a build edge from the block (Figure 4.8(a)). The entry of the block is defined as the entry of the first statement in the block, and the exit of the statements in a block is defined as the exit of the block.

Figure 4.8(b) shows the rule for a block of statements that is empty, i.e. it does not contain sub statements. If this is the case, the block is simply skipped (recall Figure 4.2(c)).

A block can contain any number of statements. The recursive BNF rule is represented in the abstract syntax as a number of BlockStatements nodes. A BlockStatements node has a Statement child, the actual statement, and either another BlockStatements node (if it is a BlockStatementsNext), or none (BlockStatementsLast).

We propagate the flow graph construction indicator (the build edge) among these BlockStatements, by matching the ordering of statements represented by the orderNext edge (Figure 4.8(c)).

From the exit of the statement we transfer control to the next statement by introducing a flow edge to the entry of the BlockStatements ordered next, which will be the next Statement, due to the merging of the entry of the BlockStatements node with the entry of the Statement node.

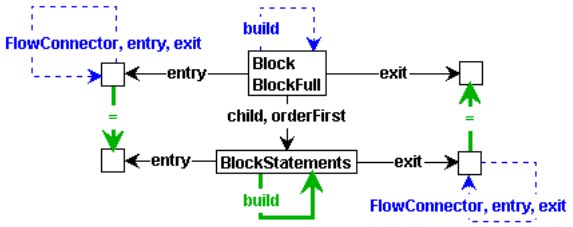
In both the BlockStatementsNextRule and the BlockStatementsLastRule (Figure 4.8(d)) we merge the exit of the child statement or BlockStatements node with the parent BlockStatements node. The exit of the statement ordered last will therefore be the same node as the exit of the Block.

4.2.3 Conditional statements

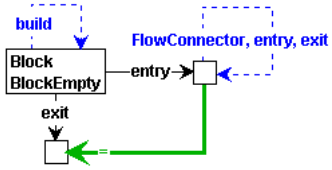
Java features two types of conditional statements: the `if` and the `switch`. The if-statement introduces two branches on the value of a Boolean condition, the switch-statement can have any number of branches on the value of a condition of type `char`, `short`, `int`, `byte` or an `enum` type.

If-statement

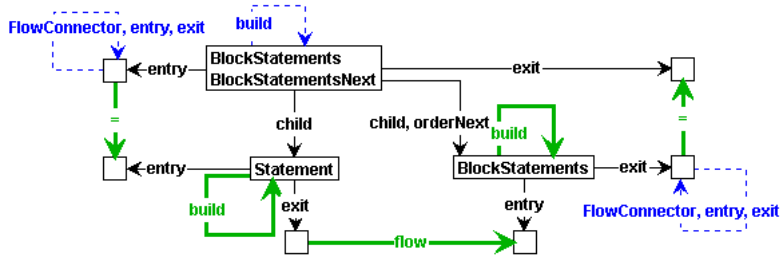
The if-statement in Java (see [6] p. 372) is the common conditional branching statement. The if-statement features a then-part, which is executed when the condition evaluates to `true`,



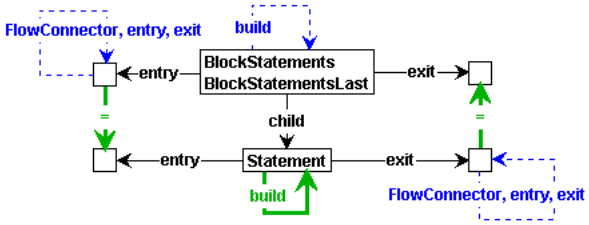
(a) Block - first statement



(b) Block - empty



(c) Block-statements - statement ordered next



(d) Block-statements - statement ordered last

Figure 4.8: Flow graph construction rules for blocks of statements.

and an optional else-part, which is executed when the condition evaluates to `false`. If the if-statement lacks a then-part, and the condition does not hold, the if-statement is finished.

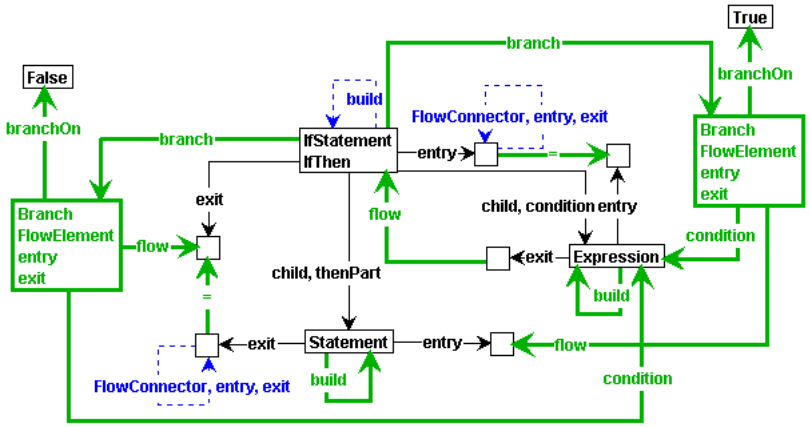
Figure 4.9(a) shows the flow graph construction rule for an if-statement without an else-part. As one would expect, the rule introduces two Branches: one branch, taken when the condition evaluates to `true`, leads to the then-part Statement. The other leads to the exit of the if-statement, and is taken when the condition does not hold.

Figure 4.9(b) shows the rule for an if-statement with an else-part, which is executed when the condition does not hold.

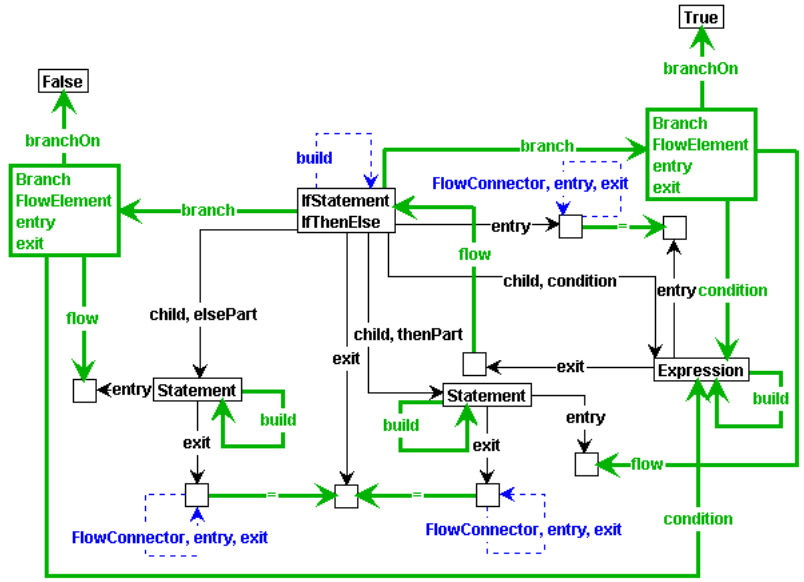
After execution of the thenPart or the elsePart of an if-statement, the control flow leads to the exit of the if-statement.

Switch-statement

The switch-statement in Java (see [6] p. 377) introduces any number of branches on literal values of some of the primitive types to several different statements. The switch-statement can be seen as a table of labels (the *cases*) and associated statements. The case to be chosen is determined by the condition expression of the switch-statement. Cases feature *fall-through* in



(a) If-then



(b) If-then-else

Figure 4.9: Flow graph construction rules for if-statement with or without else-part.

Java, this means that after execution of the statements associated with one case, the statements of the case ordered next will be executed. A switch-statement can be terminated prematurely by a break-statement (typically used to prevent this fall-through).

The production rules of the switch-statement are shown in Figure 4.11. Both the SwitchBlock and the SwitchBlockStatementGroups nodes are containers in the syntax representation that have no actual control flow semantics.

The ASG structure of the switch-block-statements-groups is similar to sequentially ordered blocks of statements (Section 4.2.2). Again do the orderFirst and orderNext edges indicate the ordering of switch-block-statements-groups. Because of the fall-through, the groups are connected with flow edges in this order (Figure 4.11(b)). The exit of the last group is defined as the exit of the switch-statement (Figure 4.11(c)).

The production rule in Figure 4.2.3 shows that upon entering a switch-statement, the expression is first executed, next the flow is transferred to the SwitchStatement node.

The branching behavior of the switch-statement is represented by a Branch for every case label present. The rule in Figure 4.12(a) adds a branch from the SwitchStatement decision node, on the SwitchLabel's value, to the entry of the Statement associated with the case label. As can be concluded when considering our flow graph meta-model (Figure 3.5), we have a special representation for the `default` case (Figure 4.12(b)).

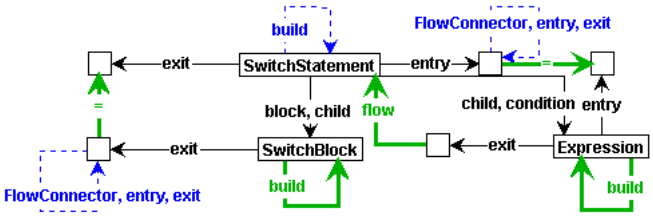


Figure 4.10: Flow graph construction rule for the switch-statement.

4.2.4 Loop statements

Java features three kinds of loop-statements: the `while`, `do` and `for`. The while-statement and do-statement are minor variations of eachother. The for-statement has eight variations, all having their own control flow semantics.

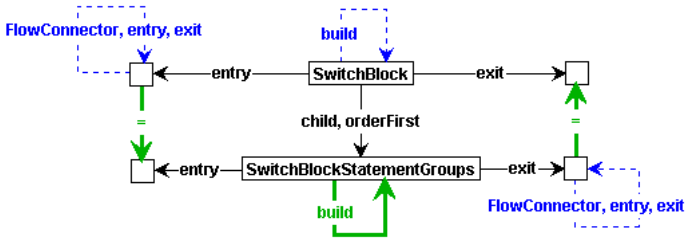
While-statement

The while-statement in Java (see [6] p. 380) executes a statement (the body) repeatedly as long as the loop condition holds. The body can ofcourse be a block of statements. The loop condition is a boolean expression.

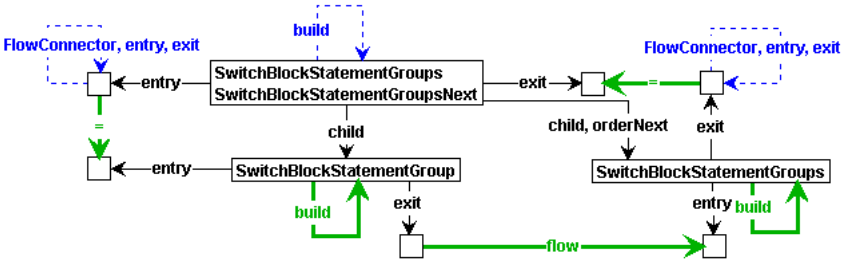
Control flow enters the while-statement (Figure 4.13) at the loop condition Expression. At the WhileStatement node, a decision is made. One Branch enters the body of the while-statement and the other proceeds directly to the exit of the while-statement. The former branch is taken when the loop-condition evaluates to `true` and the latter branch is taken when the condition evaluates to `false`. After execution of the body Statement, the loop condition Expression is evaluated again (i.e. a flow edge is introduced from the exit of the Statement to the entry of the Expression).

Do-statement

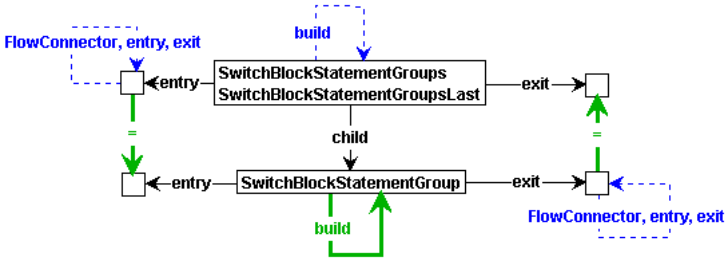
The do-statement in Java (see [6] p. 382) operates identical to the while-statement, except that the control flow enters the do-statement at the body Statement (Figure 4.14). After execution of the body, control is transferred to the loop condition. The do-statement thus performs at least one iteration of its body.



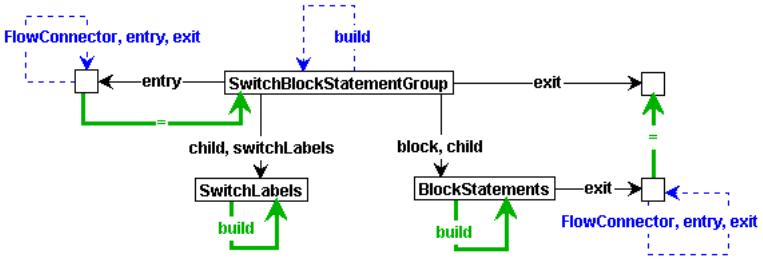
(a) Switch-block



(b) Switch-block-statement-groups - ordered next



(c) Switch-block-statement-groups - ordered last

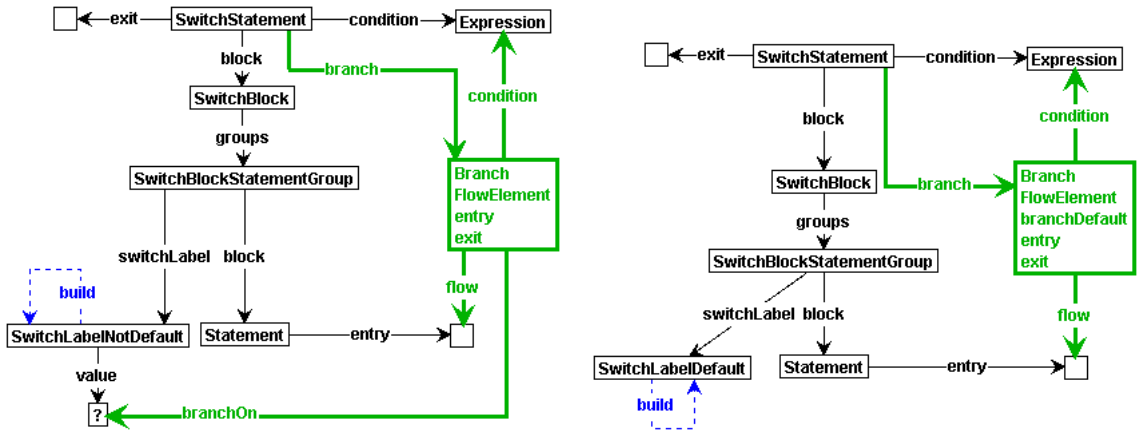


(d) Switch-block-statement-group

Figure 4.11: Flow graph construction rules for switch-block-statements-groups.

For-statement

The for-statement in Java (see [6] p. 384) has eight variants, depending on the presence or absence of the loop counter initialization section, the loop condition and the loop counter update section. As follows from Section 3.1, our abstract syntax representation enumerates all variants of the for-statement explicitly:



(a) Switch-label (b) Switch-label - default

Figure 4.12: Flow graph construction rules for switch-labels.

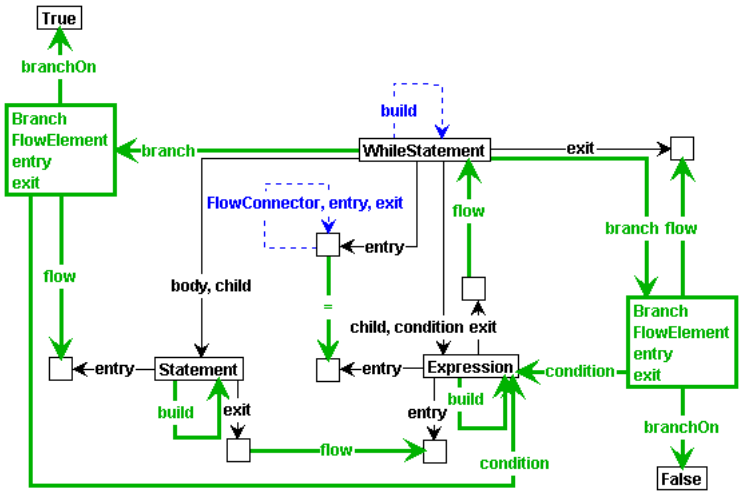


Figure 4.13: Flow graph construction rule for while-statements.

1. ForEver: for-statement without init-, condition- or update-part (Figure 4.15);
2. ForWithInit: for-statement with init-part only (not shown);
3. ForWithInitCondition: for-statement with init- and condition-part (not shown);
4. ForWithInitConditionUpdate: for-statement with init-, condition- and update-part (Figure 4.16);
5. ForWithInitUpdate: for-statement with init- and update-part (not shown);
6. ForWithCondition: for-statement with condition-part only (not shown);
7. ForWithConditionUpdate: for-statement with condition and update-part (not shown);

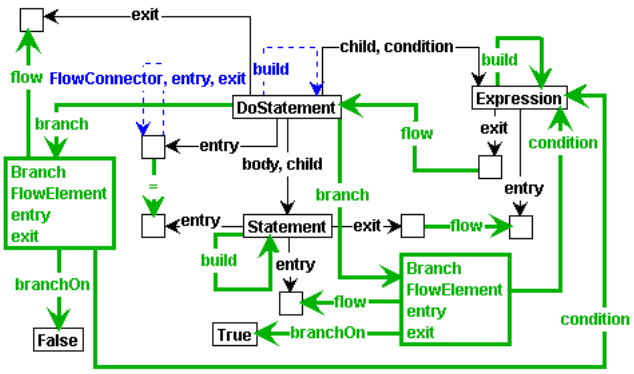


Figure 4.14: Flow graph construction rule for do-statements.

8. ForWithUpdate: for-statement with update-part only (Figure 4.17).

The init-part of a for-statement performs initialization of the loop counter(s). The condition-part is the loop-condition and equivalent to the condition in the while-statement and do-statement. The update-part updates the loop counter(s) after each iteration of the body of the for-statement.

The ForEver variant (Figure 4.15) iterates its body continuously, i.e. forever. This is very similar to a while-statement that features the Boolean literal `true` as loop condition, although in the case of a `while(true)` the literal expression is evaluated each iteration and the Branch on `true` is followed each iteration.

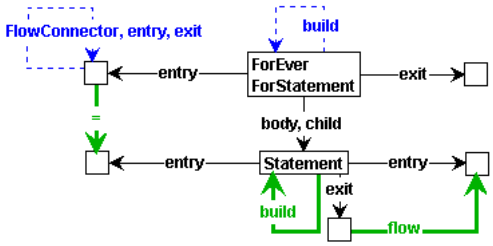


Figure 4.15: Flow graph construction rule for the for-statement without init-, condition- or update-part.

The ForWithInit variant (rule omitted here)performs some initialization of loop counters and then starts to iterate its body continuously, i.e. forever.

The ForWithInitCondition variant (rule omitted here)performs some initialization of loop counters, evaluates its loop condition and executes its body if its condition evaluated to `true`. Like the while-statement, the expression is evaluated again after each iteration of the body to decide whether to iterate the body again.

The ForWithInitConditionUpdate variant (Figure 4.16) is the "full" version of the for-statement. Loop counters are initialized before the first evaluation of the condition and after each iteration the counters are updated before re-evaluating the condition.

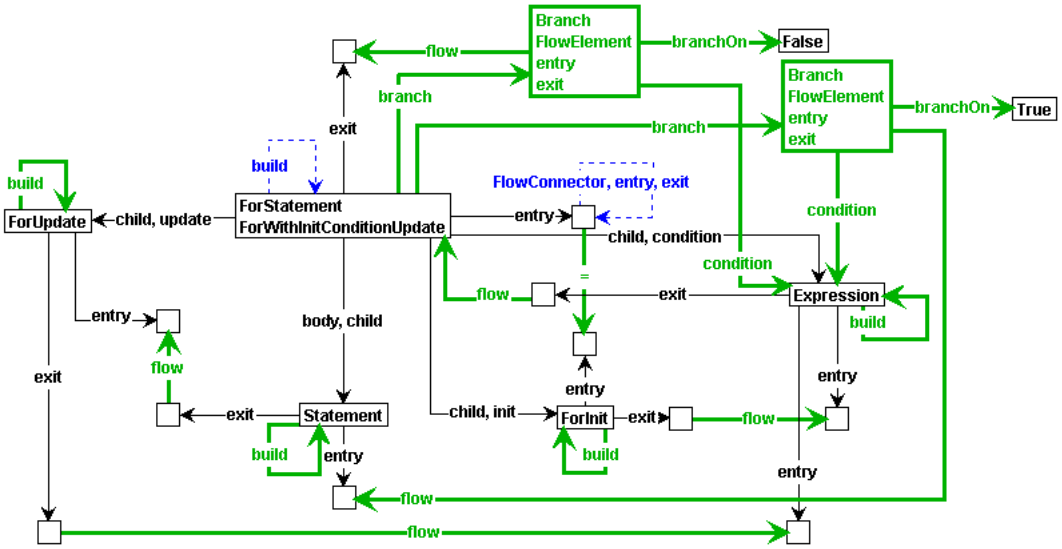


Figure 4.16: Flow graph construction rule for the for-statement with init-, condition- and update-part.

The ForWithInitUpdate variant (rule omitted here)performs loop counter initialization and then iterates forever, but updates the counters after each iteration of the body.

The ForWithCondition variant (rule omitted here)is equivalent to a while-statement.

The ForWithConditionUpdate variant (rule omitted here)evaluates the loop condition before each iteration and updates the loop counters after each iteration.

The ForWithUpdate variant (Figure 4.17) executes the body forever but updates the loop counters after each iteration of the body.

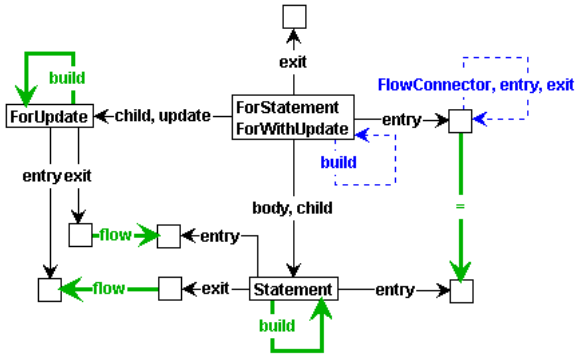


Figure 4.17: Flow graph construction rule for the for-statement with update-part.

4.2.5 Primitive statements and expressions

Java features many types of primitive statements and expressions. Due to space considerations, we treat a limited number of them to illustrate how expressions are treated with respect

to flow graph construction. The (statement) expression types we treat here are:

- Empty statements;
- Assignments;
- Local variable declarations;
- Literals;
- Identifiers;
- Binary operators;
- Method invocations.

Empty statement

The empty-statement ; in Java (see [6] p. 370) obviously is the most trivial statement of Java, as it does nothing and is simply skipped. Figure 4.18 shows the rule that merges the entry of the EmptyStatement with its exit.

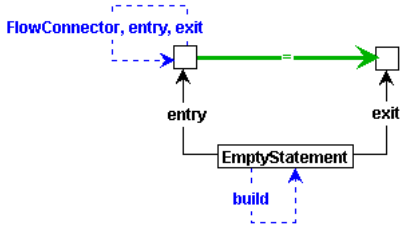


Figure 4.18: Flow graph construction rule for the empty-statement.

Assignment

The assignment (expression) statement in Java (see [6] p. 512) stores a value to which an expression evaluates in one or several declared variables. This is possibly the most important statement for any programming language.

Figure 4.19 shows the production rule for assignments. The AssignmentExpression that results in the value that is to be stored in the variable is first evaluated, next the assignment to the variable is performed. As the AssignmentExpression can be another Assignment a value can be stored in several variables, which are thus evaluated right to left.

The left hand side is an ExpressionName, but as we have not elaborated field selectors (i.e. `variable.field`) this is in our case always a primitive Identifier.

Local variable declaration

Within method bodies, variables can be declared that are local to the method (or some scope within the method). In Java, several local variables of the same type can be declared in one statement (see [6] p. 363).

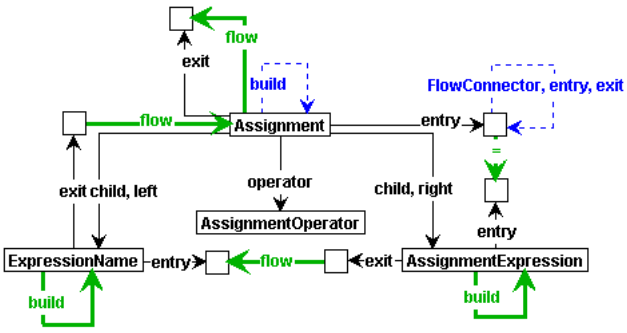


Figure 4.19: Flow graph construction rule for the assignment-statement.

Figure 4.20 shows our construction rule for a LocalVariableDeclarationStatement. Control is first transferred to the VariableDeclarators and next the declaration statement itself is executed.

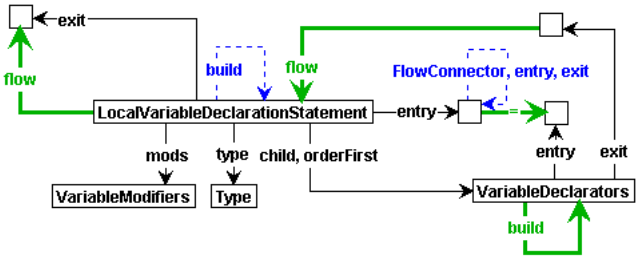


Figure 4.20: Flow graph construction rule for declarations of local variables.

As one can declare several local variables in one variable declaration statement, the VariableDeclarators are represented in an ordered list in which the declarators are evaluated left to right. The construction rules are therefore very similar to any other ordered list (i.e. with orderFirst and orderNext) like BlockStatements (see Section 4.2.2) and are omitted here.

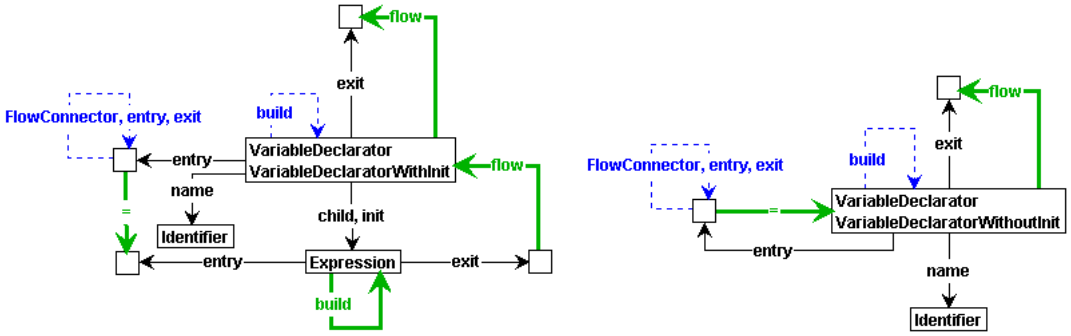
An indiviual VariableDeclarator can feature an initialization expression or not. If the VariableDeclarator features an init expression (Figure 4.21(a)) this expression is evaluated first and next control is transferred to the VariableDeclarator. The construction rule for a VariableDeclarator without init is shown in Figure 4.21(b).

Literal

A literal value in Java can be a value of any primitive type or `null`. Figure 4.2.5 shows our construction rule for a Literal.

Identifier

An identifier in Java can refer to a declared variable (in can also be for instance a label for a labeled statement). In our grammar (Appendix A) such an Identifier is an ExpressionName. Figure 4.2.5 shows the construction rules for these identifiers.



(a) Variable declarator - init (b) Variable declarator - no init

Figure 4.21: Flow graph construction rules for variable declarators with or withour init-expressions.

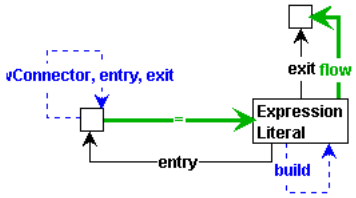


Figure 4.22: Flow graph construction rule for a literal value.

Binary operator

We treat one example binary operator expression type: the relational operator (i.e. >, >=, <, <=). Figure 4.2.5 shows the construction rule for the RelationalOperator. First the left side and right side of the operator are evaluated, next control is transferred to the RelationalExpression. Other binary operators are treated in an equivalent manner, for example the AdditiveOperator which features Figure 3.6.

Method invocation

Method invocations in Java (see [6] p. 440) are used to invoke (static or non-static) methods. The method to invoke is determined at run-time through some elaborate method name

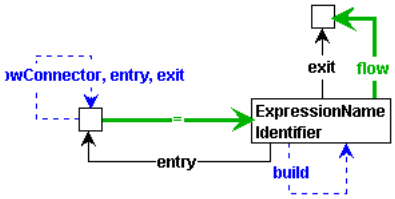


Figure 4.23: Flow graph construction rule for an identifier.

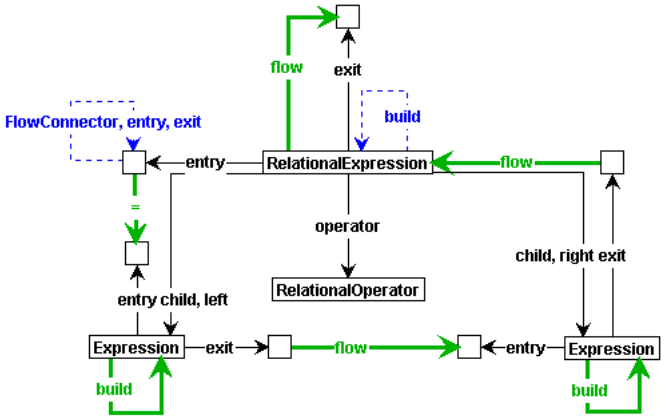


Figure 4.24: Flow graph construction rule for a relational expression.

resolution procedure.

MethodInvocations come in two variants: with or without arguments. These arguments are expressions that result in values (or references to objects) that are passed to the invoked method. These arguments are evaluated before the invocation process begins.

Figure 4.25 shows the construction rules for MethodInvocations with or without arguments. If a MethodInvocation features arguments, control is transferred first to the ArgumentList. Next the MethodName is evaluated (which we have not elaborated) and the actual invocation is executed (which method exactly to invoke is a run-time decision). A fter control is transferred back to this calling statement, the method invocation is completed.

The ArgumentList in a method invocation is a typically ordered list in which the arguments are evaluated left to right and each argument is an Expression. The construction rules are therefore very similar to any other ordered list (i.e. with orderFirst and orderNext) like BlockStatements (see Section 4.2.2) and are omitted here.

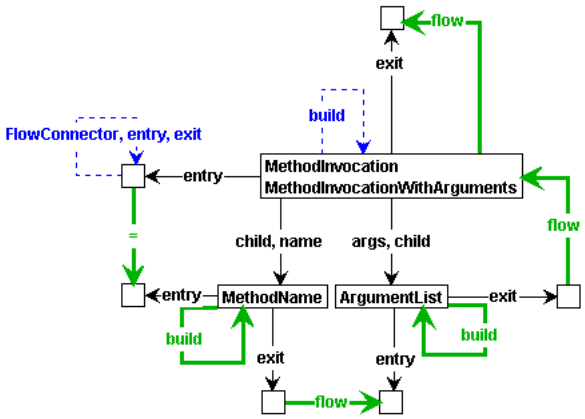
4.2.6 Abrupt completion statements

Now that we have treated how some of the common Java statements are handled by flow graph construction rules, we look at the abrupt completion statements that feature Java. These statements we have studied extensively in the case study we have performed on flow graph construction for Java.

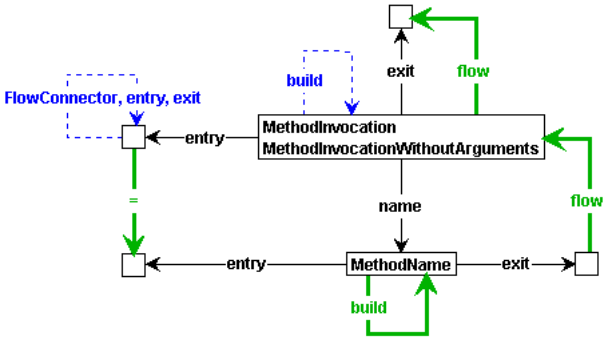
Break-statement

The break-statement in Java (see [6] p. 388) is a statement that introduces abrupt completion: it aborts the innermost enclosing loop (`for`, `while`, `do`) or `switch` statement, in other words, it will transfer control to the statement that is sequentially ordered next to this enclosing statement. When its used in a `switch` statement, it prevents the fall-through into the next case.

Like most abrupt completion statements, the production rule for the break-statement without label introduces an auxiliary abrupt completion (Abort) node, with an edge labeled



(a) Method invocation - arguments



(b) Method invocation - no arguments

Figure 4.25: Flow graph construction rules for method-invocations.

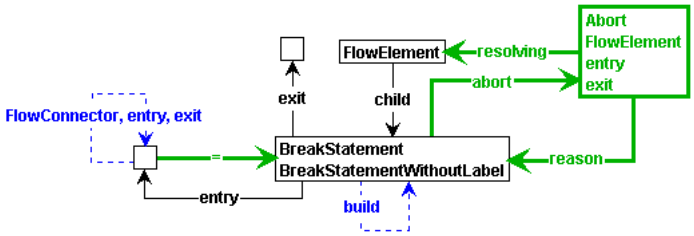
resolving that is propagated upwards, until the first enclosing loop or switch-statement is reached (Figure 4.26(a)).

When the propagated resolving edge reaches one of the abrupt completion *targets* of the break-statement, the abrupt completion is resolved by removing the resolving and introducing a flow edge to the element to which control is transferred by the abrupt completion.

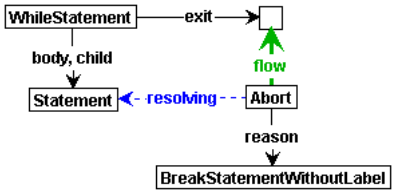
Figure 4.26(b) shows the rule that resolves abrupt completion when a while-statement is the target of a break-statement. This while-statement is terminated by the break-statement. Figure 4.26(c) shows the rule that resolves abrupt completion when a do-statement is the target. This do-statement is also terminated by the break-statement. Figure 4.26(d) shows the rule that resolves abrupt completion when a for-statement is targetted. Again, this for-statement is terminated by the break-statement. Figure 4.26(e) shows the rule that resolves abrupt completion when a switch-statement is targetted. Like the other break-targets, this switch-statement is terminated by the break-statement.

Continue-statement

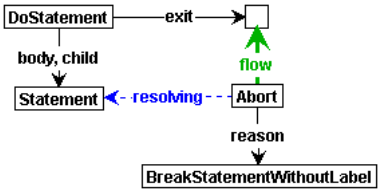
The continue-statement in Java (see [6] p. 390) is another statement that introduces abrupt completion. The continue-statement terminates the current iteration and continues the next



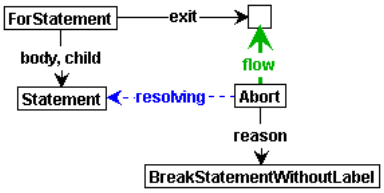
(a) Break-statement



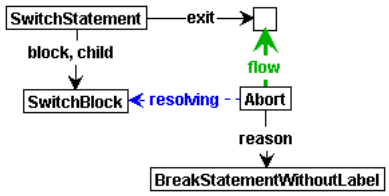
(b) Break-statement - abort WhileStatement



(c) Break-statement - abort DoStatement



(d) Break-statement - abort ForStatement



(e) Break-statement - abort SwitchStatement

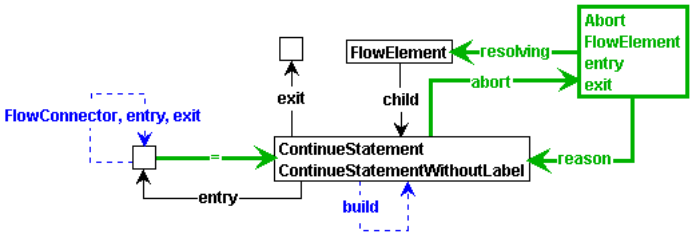
Figure 4.26: Flow graph construction rules for the break-statement, introducing abrupt completion, and for resolving abrupt completion because of a break-statement.

iteration of the of the loop-statement it targets.

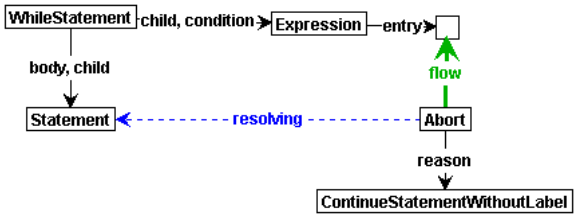
The construction rule in Figure 4.27(a) introduces abrupt completion because of a continue-statement. Figure 4.27(b) and 4.27(c) show how the current iterations of a while-statement or do-statement are skipped (control is transferred to the loop-condition). When we apply the continue-statement to a for-loop, the control flow semantics depend on the presence of a loop-condition and loop-counter update-part. If the ForUpdate part is present, control is transferred to this section (Figure 4.27(e)). Else if the condition is present, control is transferred to this loop-condition (Figure 4.27(d)). Else, the body of the for-statement is simply re-entered (Figure 4.27(f)).

Break-statement with label

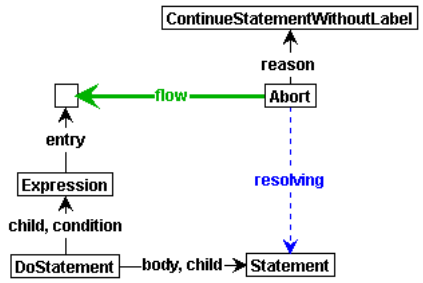
The break-statement can optionally feature a label that refers to another (labeled) statement that encloses the break-statement directly or indirectly, this statement can be any type of statement that features sub statements (e.g. a WhileStatement but even a Block is possible). The labeled statement is the abrupt completion target of the break-statement with the identical label and is terminated by it.



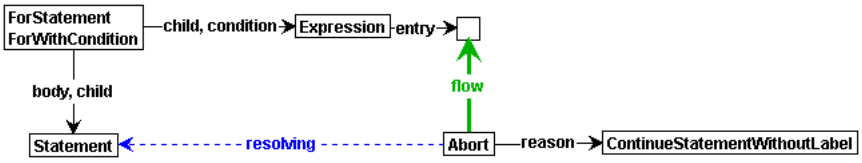
(a) Continue-statement



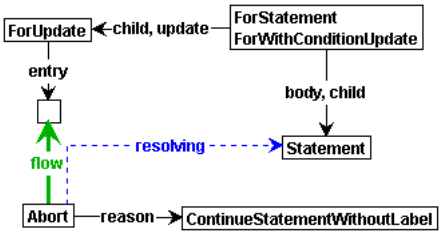
(b) Continue-statement - abort WhileStatement



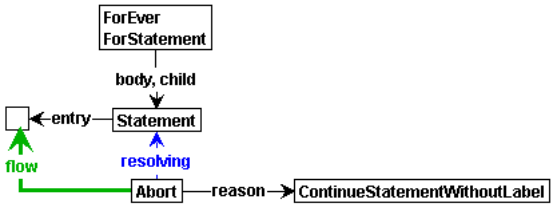
(c) Continue-statement - abort DoStatement



(d) Continue-statement - abort ForStatement with condition



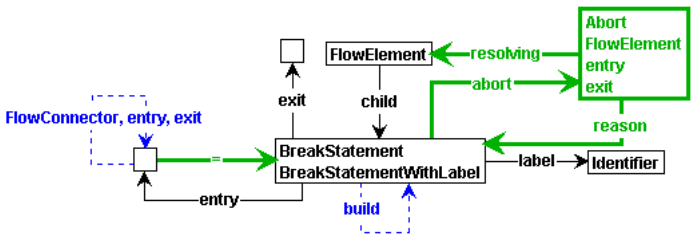
(e) Continue-statement - abort ForStatement with condition and update



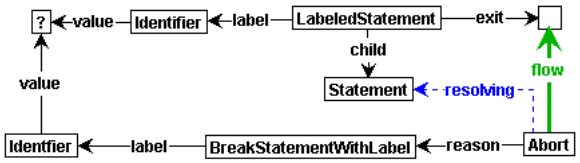
(f) Continue-statement - abort ForStatement without condition and update

Figure 4.27: Flow graph construction rules for the continue-statement, introducing abrupt completion, and for resolving abrupt completion because of a continue-statement.

The production rule for the break-statement with label is shown in Figure 4.28(a)). The resolving edge is deleted and replaced by a flow edge, leading to the exit, when an identically labeled enclosing flow element has been found (Figure 4.28(b)).



(a) Break-statement with label



(b) Break-statement with label - abort labeled statement

Figure 4.28: Flow graph construction rules for the break-statement with label, introducing abrupt completion, and for resolving abrupt completion because of a break-statement with label.

Continue-statement with label

The continue-statement can also optionally feature a label that refers to another (labeled) loop-statement that encloses the continue-statement directly or indirectly. The labeled loop-statement's current iteration is terminated by the continue-statement with an identical label.

The production rule for the continue-statement with label is shown in Figure 4.29(a)). The resolving edge is deleted and replaced by a flow edge, leading to the exit of the body of an identically labeled enclosing loop statement (Figure 4.29(b)).

Return-statement

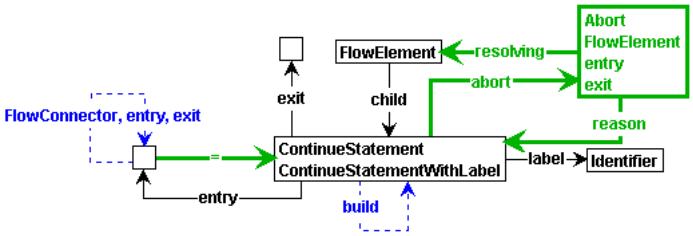
The return-statement in Java (see [6] p. 392) introduces abrupt completion: it terminates the execution of the method it is contained in.

The return-statements comes in two variants (with or without return-value), therefore two different construction rules are used. Figure 4.30(a) shows the construction rule for the ReturnStatement with value. First the returnValue expression is evaluated, next abrupt completion is introduced. Figure 4.30(b) shows the construction rule for ReturnStatement without value.

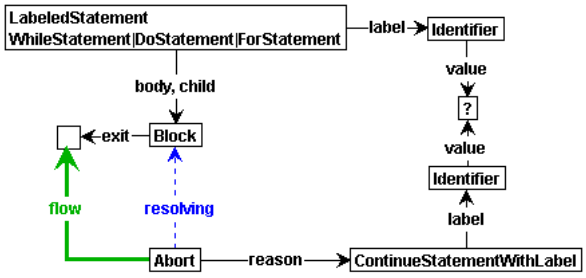
The abrupt completion of both variants of the return-statement is resolved by transferring control to the exit of the MethodBody context node (Figure 4.30(c)).

Throw-statement

The throw-statement in Java (see [6] p. 393) introduces abrupt completion by throwing an exception that causes control to be transferred through several enclosing statements and possible even several method invocations on the call-stack until it is caught by an enclosing try-statement that features a catch-clauses for a corresponding exception type.



(a) Continue-statement with label



(b) Continue-statement with label - abort labeled statement

Figure 4.29: Flow graph construction rules for the continue-statement with label, introducing abrupt completion, and for resolving abrupt completion because of a continue-statement with label.

The throw-statement has an expression that results in its exception object, that should be evaluated first. Besides this, again an abrupt completion node is introduced (Figure 4.31(a)). When the throw-statement terminates the execution of the method it is contained in (i.e. is not enclosed by a try-statement), the rule in Figure 4.31(b) applies.

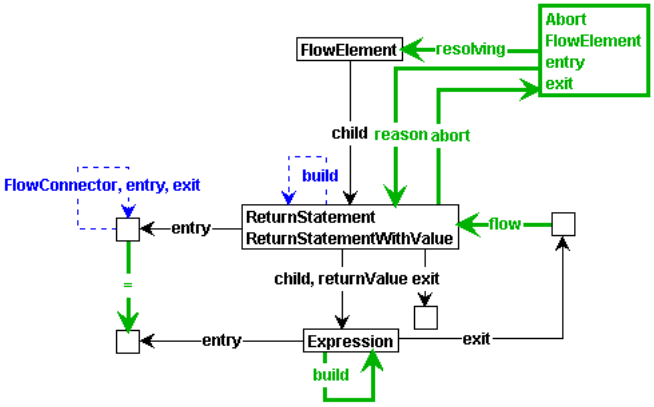
4.2.7 Exception-handling statements

The statement in Java that performs exception handling is called the try-statement (see [6] p. 396). The try-statement comes in three variants: try-finally, try-catch and try-catch-finally.

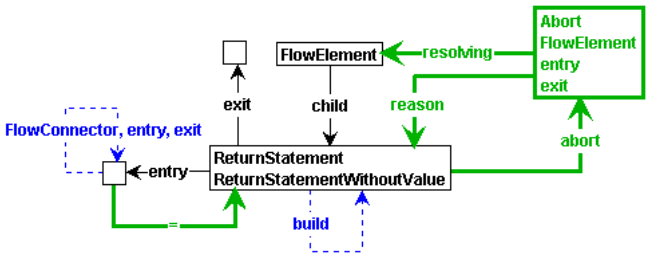
TryCatch

The try-catch-statement features a block of statements that are executed upon execution of the try-catch-statement. It also features one or more catch clauses with associated blocks of statements, which are not executed when the body of the try-catch-statement completes normally.

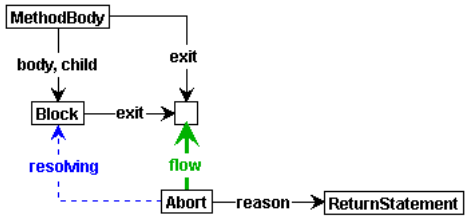
However, if an exception occurs in the body of the try-catch statement, control is transferred to the catch-clauses. These clauses are considered in order from first to last. When one of the clauses matches the (run-time) type of the thrown exception, control is transferred to the block of statements associated to this clause. If (at run-time) the exception is not caught by any of the clauses, the abrupt completion because of this exception is resumed. After execution of the body or a block of statements associated with one of the catch-clauses,



(a) Return-statement with value

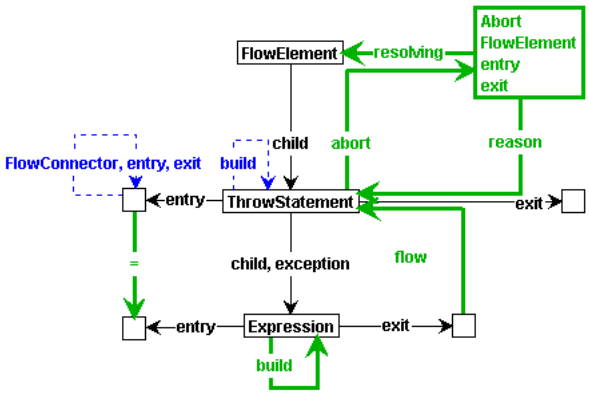


(b) Return-statement without value

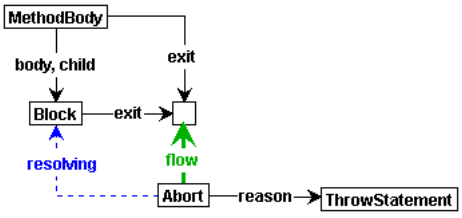


(c) Return-statement - abort method-body

Figure 4.30: Flow graph construction rules for the return-statement, introducing abrupt completion, and for resolving abrupt completion because of a return-statement.



(a) Throw-statement



(b) Throw-statement - abort method-body

Figure 4.31: Flow graph construction rules for the throw-statement, introducing abrupt completion, and for resolving abrupt completion because of a return-statement.

the try-statement is completed normally (unless, of course, the body of the catch-clause completed abruptly, in that case the try-statement completes abruptly).

The construction rule for the try-catch statement is presented in Figure 4.32.

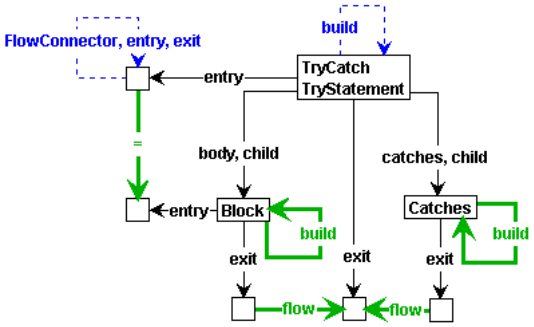


Figure 4.32: Flow graph construction rule for the try-catch-statement.

Figure 4.33 shows our rule for resolving abrupt completion because of a throw-statement in the body of a try-catch statement. In this figure we can see that control is first transferred to the Catches (thereby, perhaps only temporarily, resolving the abrupt completion). If, at run-time, none of the clauses match, abrupt completion is resumed. This is represented by another Abort node with the same reason, namely the throw-statement that threw the exception.

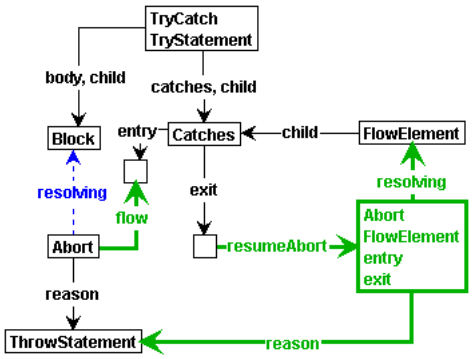


Figure 4.33: Flow graph construction rule for resolving throw-statements for the try-catch-statement.

TryFinally

The try-finally-statement features a finally-statement that consists of a block of statements that are guaranteed to be executed. Whether the body of the try-finally-statement completes normally or abruptly, control is transferred to this finally-statement next.

The flow graph construction rule in Figure 4.34 shows that after execution of the body of a try-finally statement, control is transferred to the finally-statement. After execution of this statement, the try-statement completes like the finally-statement completed.

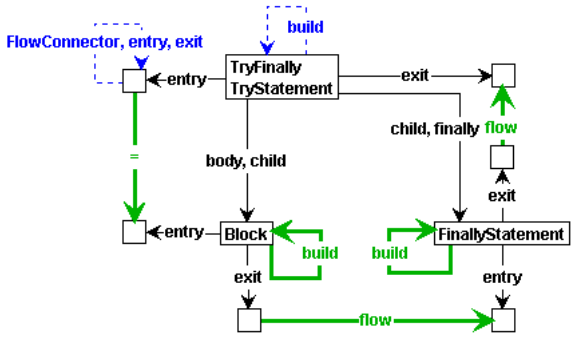


Figure 4.34: Flow graph construction rule for the try-finally-statement.

When the body of the try-finally statement completes abruptly because of some abrupt completion statement, the control is first transferred to the finally-statement. From the exit of the finally-statement, a new abrupt completion node is introduced that will abort some statement enclosing the finally-statement, because of the same abrupt completion statement. We say that abrupt completion has been *resumed*. This holds for all four types of abrupt completion in Java: `break`, `continue`, `return` and `throw`. Figure 4.35 shows the abrupt completion resolution and resumption rule for a try-finally-statement that features a break-statement. The other three abrupt completion statement types are treated in an equivalent manner, therefore we omit the corresponding flow graph construction rules.

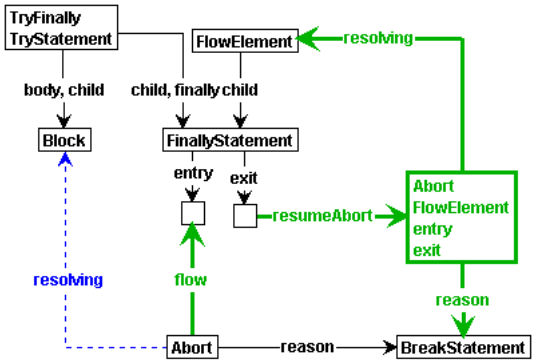


Figure 4.35: Flow graph construction rule for resolving and resuming abrupt completion of the break-statement for the try-finally-statement.

TryCatchFinally

As follows from its name, the try-catch-finally-statement combines the features of both the try-catch-statement as the try-finally-statement.

Figure 4.36 shows our flow graph construction rule for the try-catch-finally statement. As we can see in this figure, after execution of the body control is transferred to the finally-statement, and after execution of a block of statements associated with one of the catch-

clauses control is also transferred to the finally-statement. When following the normal flow of control, the catch-clauses are not reached.

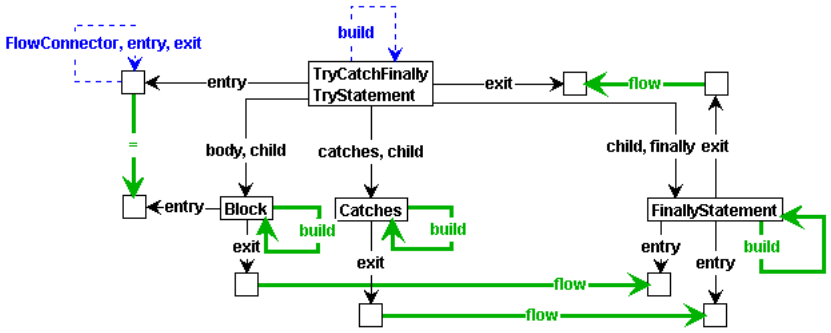


Figure 4.36: Flow graph construction rule for the try-catch-finally-statement.

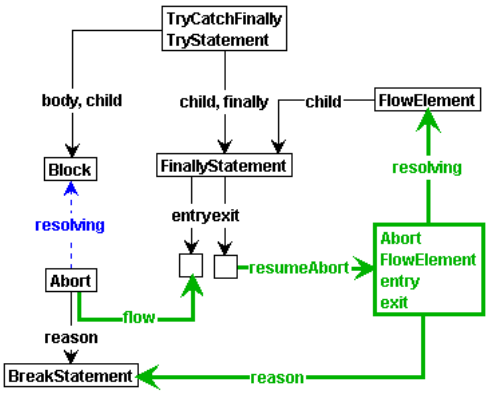
When the body of the try-catch-finally statement completes abruptly because of some abrupt completion statement other than a throw-statement, control is first transferred to the finally-statement. From the exit of the finally-statement, a new abrupt completion node is introduced that will terminate some statement enclosing the finally-statement, because of the same abrupt completion statement. We again say that abrupt completion has been resumed. This holds for three of the four types of abrupt completion in Java: `break` (Figure 4.37(a)), `continue` and `return`. As the rules of resolving and resuming continue- and return-statements are equivalent to the rule for the break-statement, these rules have been omitted here.

A throw-statement present in the body of the try-catch-finally-statement is treated differently, because of the presence of catch-clauses. Control is first transferred to the Catches section. Whether or not the thrown exception is caught by one of these clauses, control is transferred to the finally-statement next. Depending on whether the exception was caught by a clause (which is a run-time decision), the exception travels further (abrupt completion resumption) or not.

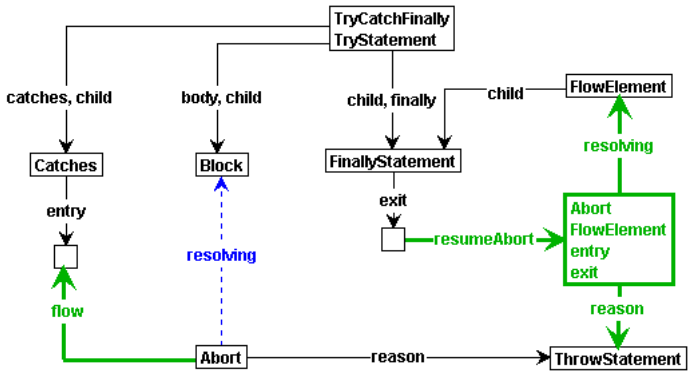
Catch-clauses

The construction rule for the catch-clauses is shown in Figure 4.38. We represent the run-time decision whether a thrown exception matches one of the clauses of the catch-statement by a Boolean decision, for which the condition is the catch-clause. When the condition (i.e. the run-time `instanceof` check performed at the clause) evaluates to `true`, the block of statements associated with the clause is executed, else the exception is tested on the next clause. If the last clause (CatchClausesLast) is not chosen, the `false`-branch leads to the exit of the catch-block, from which the exception will travel further (to another catch-block, finally-statement or method-exit). If the one of the clauses is executed, the exception will not have to travel further, but a finally-statement might have to be executed.

The Catches part in a try-statement is a typically ordered list of clauses that are examined first to last when handling a thrown exception. The construction rules are therefore very similar to any other ordered list (i.e. with orderFirst and orderNext) like BlockStatements (see Section 4.2.2) and are omitted here.



(a) Try-catch-finally-statement - abort break



(b) Try-catch-finally-statement - abort throw

Figure 4.37: Flow graph construction rules for resolving and resuming abrupt completion for the try-catch-finally-statement.

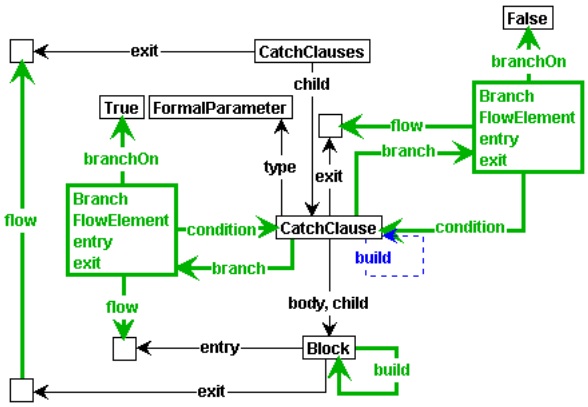


Figure 4.38: Flow graph construction rules for catch-clauses.

```
1  public static int indexOf(int[] list, int x) {
2      int i = 0;
3      while (i < list.length) {
4          if (list[i] == x) {
5              // found x, break the while loop
6              break;
7          }
8          i = i + 1;
9      }
10     if (i < list.length)
11         return i;
12     else
13         return -1;
14 }
```

Listing 4.1: Example of an linear array search in Java.

Finally-statement

The finally-statement features a block of statements that is executed upon execution of the finally-statement. The associated flow graph construction rule merges the entry of the FinallyStatement with the entry of the Block and shares their exits.

4.2.8 Example of Java flow graph construction

As a conclusion of this chapter on flow graph construction, we treat a fairly large Java example. For an example Java code listing, we show how a corresponding flow graph is constructed based on the abstract syntax graph representation of the source code. The code listing is shown in Listing 4.1. The `indexOf()` method performs a linear search in array `list` to find a given integer element `x` and returns the position of `x` in this array. If `x` is not found in the array, the value `-1` is returned.

Although there obviously is a more elegant version of the depicted algorithm, this algorithm is interesting in this form because it features both a `break` and two `return` statements. The flow graph construction process will therefore feature three instances of abrupt completion resolution.

The translation of the source code in Listing 4.1 to the abstract syntax graph depicted in Figure 4.39 is straightforward, although there are some syntax elements present in this graph that have not been treated in the previous sections. A new syntax element is the EqualityExpression, this is a binary expression that checks the left and right side on (in)equality) (i.e. `==` and `!=`). Another new syntax element is the FieldAccess (`list.length` in the code listing), consisting of a reference type on which a field (Identifier) is accessed. One more new syntax element is the ArrayAccess (`list[i]` in the code listing), consisting of a reference to an array and an Expression that determines the (integer) index in the array.

The ASG in Figure 4.39 becomes the start graph of the flow graph construction system for Java, which consists of a set of (84) Java flow graph construction rules. These rules are applied in a linear fashion, resulting in a (partial) rule application LTS (see Section 2.2.2). The partial rule application LTS for this example is shown in Figure 4.40. The flow graph construction process is initiated by creating a build self-edge for the ContextNode, the method body. Next for each flow element we uniformly add an entry and exit FlowConnector (part of

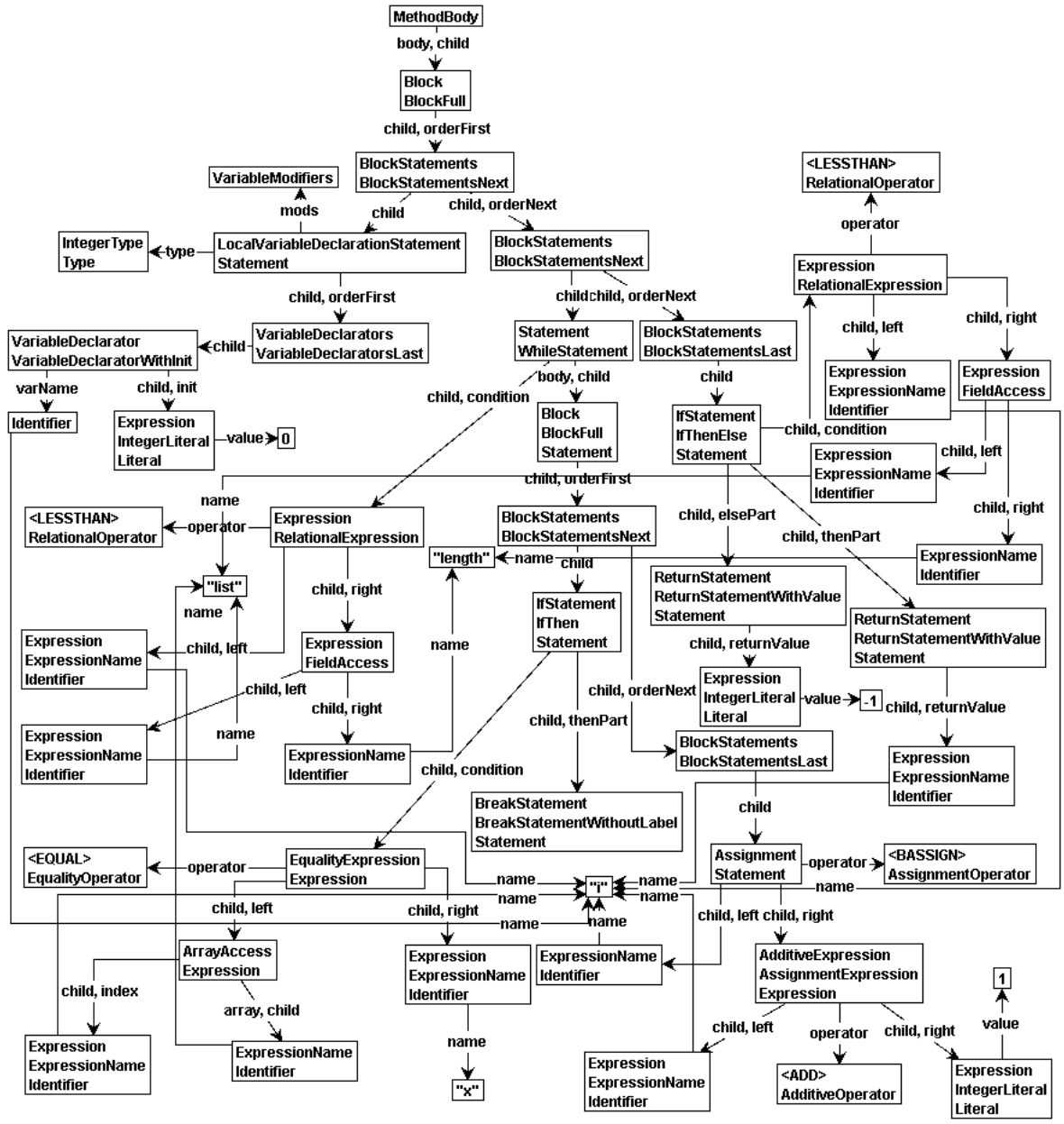


Figure 4.39: Abstract syntax graph of Listing 4.1.

this section of the LTS has been left out). Now the rules for the individual FlowElements apply, starting with the MethodBodyRule.

It is interesting to see that after the BreakStatementRule or ReturnStatementWithValueRule have been applied, we have several sequential applications of the AbortPropagationRule. This means that the abrupt completion of the `break` or `return` is being resolved, and the correct target is searched in a bottom-up way. When the correct abrupt completion target has been found, the WhileStatementAbortBreakStatementWithoutLabelRule or the MethodBodyAbortReturnStatementRule applies to resolve the abrupt completion.

The newly introduced syntax elements EqualityExpression and FieldAccess have similar control flow to the RelationalExpression and the ArrayAccess has similar control flow to Assignments.

The resulting completed flow graph is shown in Figure 4.41. To increase the readability of this large graph, some sacrifices had to be made. The auxiliary elements FlowConnector, Branch and Abort do not feature their inherited labels FlowElement, entry and exit. Furthermore, many irrelevant syntax nodes and edges have been grayed out. And last, the entry and exit edges of each FlowElement have been grayed out too, else the graph would have been unreadable.

We can see in this graph that the starting flow edge (and the entry) of the MethodBody is propagated all the way to the initializing expression (the integer literal `0`) of the declaration of the variable `i`. A reoccurring pattern is that control flow starts at the left side (a literal of identifier) of some primitive expression and works its way up back to the control statements like `if` and `while`.

Also interesting to note are the six branches (for the `while` and two `if`'s) and the three cases of abrupt completion flow. We see that the `break` statement terminates the `while` and that both `return`'s terminate the execution of the method body.

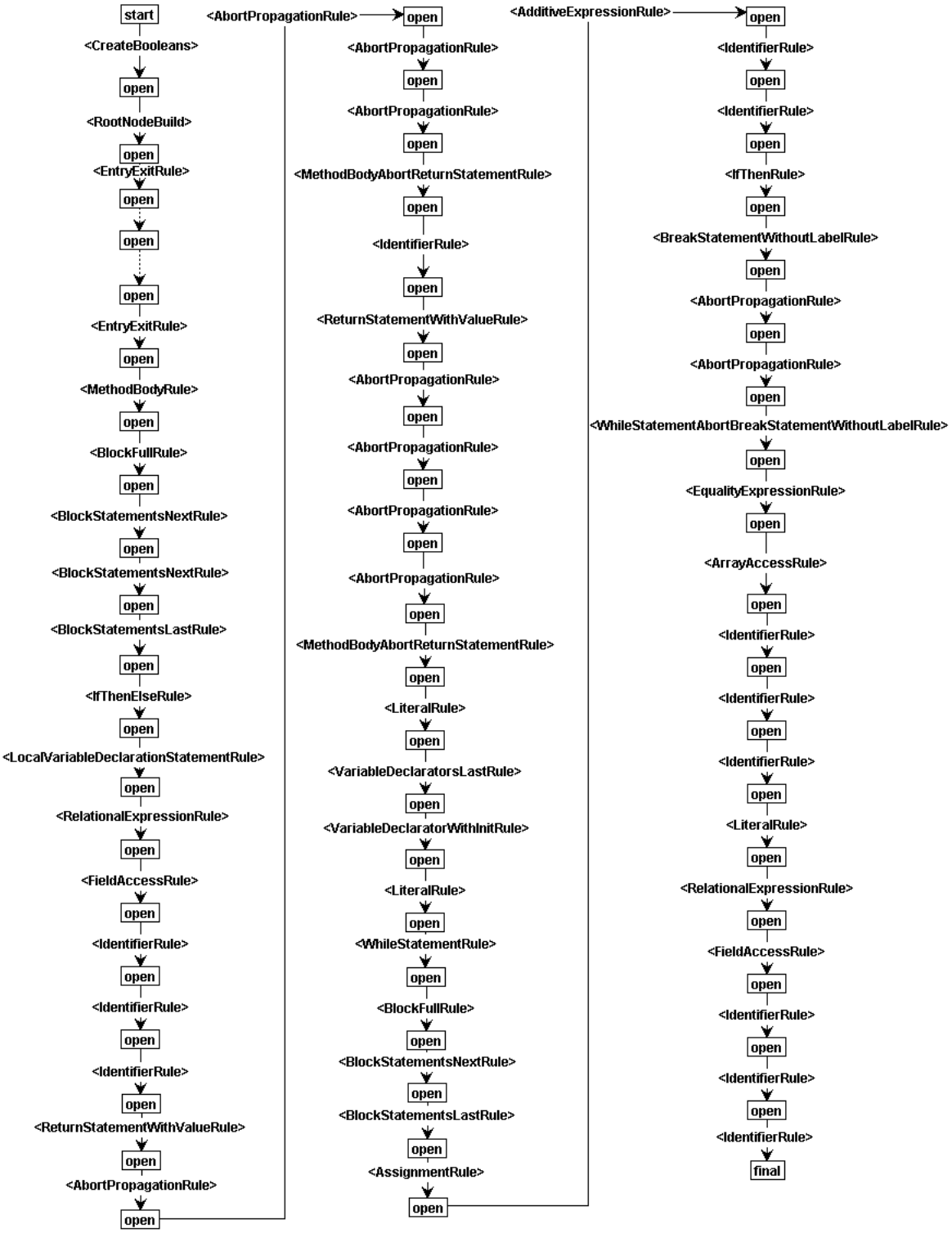


Figure 4.40: Partial rule application LTS of the flow graph construction of Listing 4.1.

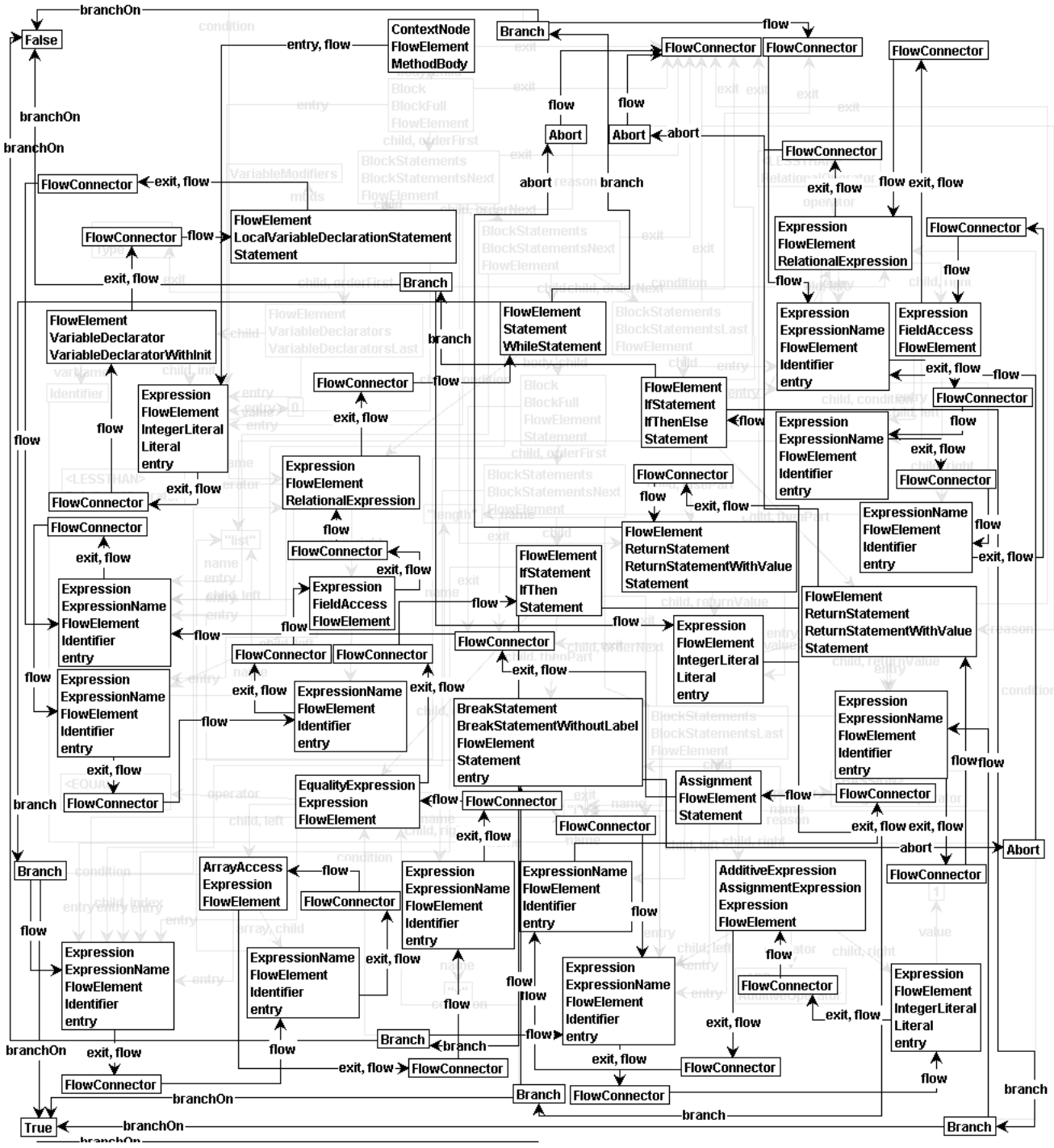


Figure 4.41: Constructed flow graph of Listing 4.1.

Chapter 5

Control Flow Specification Language

This chapter treats our formal specification language for the control flow semantics of programming languages (to which we refer as CFSL), the main result of this thesis.

Let us again review the big picture of this thesis (Figure 1.1). We see that the researcher has designed (2) the control flow specification language (3), which resides at the meta-language level (the level of languages in which parts of programming languages are defined).

We also see that a language designer uses CFSL to design (12) a set of formal control flow specifications (8) for each construct that features the programming language he or she designs. The starting point of this design process is the language specific abstract syntax meta-model (7) he or she has designed.

In this chapter we present a manual for designing specifications in CFSL, including the CFSL graph meta-model, further constraints on specifications and a stepwise approach for specification design. We then present a large number of specifications for Java constructs to serve as examples for CFSL.

5.1 Control flow specifications design manual

This section presents a manual for designing correct control flow specifications in CFSL. The CFSL meta-model is discussed and additional constraints are introduced along with a graph transformation system for checking constraint violations in control flow specifications. Next we present a recipe for designing control flow specifications.

The basis of a control flow specification for some programming language in CFSL is the language-specific abstract syntax graph representation (Section 3.1, e.g. Figure 3.1) of an abstracted version of the grammar of the language (for instance, the abstract Java grammar in Appendix A). To such a representation we specify the corresponding control flow by introducing elements from the CFSL meta-model where necessarily.

As a rule of thumb, we can say that for every non-terminal in the abstracted grammar of a programming language, in other words, for every left hand side of the BNF production rules we have to design a control flow specification in CFSL. The rule of thumb "one specification per construct" applies even more than the "one rule per construct" principle to which we tried to adhere when designing flow graph construction rules (Section 4.1.1). Summarizing, for each construct of a specific programming language we specify in CFSL all (possible) control flow semantics (including, for example, possible occurences of abrupt completion resolution).

5.1.1 CFSL Meta-model

This section presents the meta-model to which control flow specification graphs should adhere. The meta-model of CFSL (Figure 5.1) is on some points similar to the meta-model of our flow graphs (Figure 3.5). But since we want our control flow specifications to be readable and want to make it easy for a language designer to use the CFSL, this meta-model requires fewer control flow elements. A design principle for the CFSL meta-model was that it should allow a language designer to only specify control flow in as far as it is interesting and not trivial.

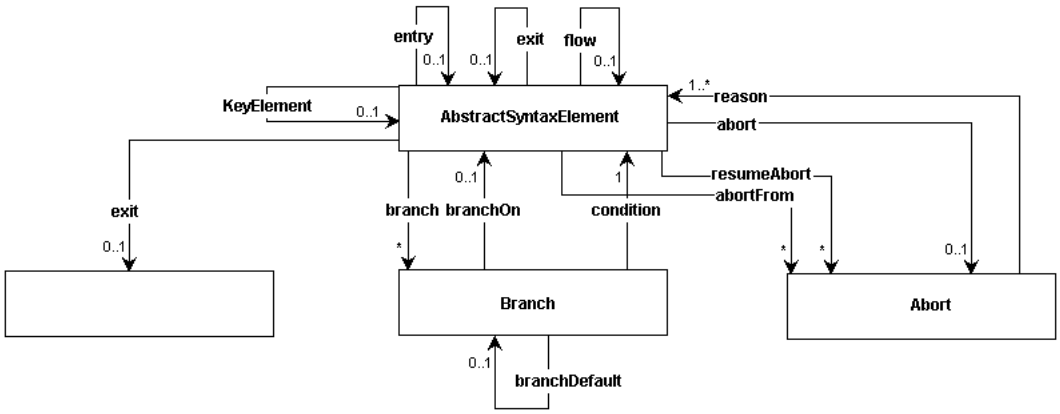


Figure 5.1: Meta-model of control flow specifications in CFSL.

A good example of a simplification is the fact that all control flow edges are implicitly assumed to leave from the exit of a syntax node and enter the target syntax node at its entry, thereby eliminating the need of specifying all entry and exit edges.

We can explicitly specify the entry or exit of any syntax node, but we are not required to. In practice, one will only specify entries and exits as far as they are relevant and not obvious.

A new self-edge of a syntax node is KeyElement; this edge indicates that the specification specifies the control flow for this statement type (remember our "one specification per construct" principle). As a result a specification can have at most one KeyElement.

CFSL does not feature flow connectors (FlowConnectors) as our flow graphs do, as these mainly are auxiliary, and in most cases temporary, nodes necessary for facilitating top-down flow graph construction. However, it is possible to specify the, unknown, control flow context as the target of the exit edge of a syntax node, represented as an empty node. A common scenario is to specify the exit of the KeyElement using this empty node.

We again use auxiliary nodes for branches (Branch) and abrupt completion (Abort). The multiplicities of the edges associated with Branch and Abort in most cases match with the multiplicities of the edges associated with their flow graph counterparts.

The representation of abrupt completion needs further explanation. Depending on the associated edges, an Abort node can represent the following four types of abrupt completion:

Introduction The introduction of new abrupt completion control flow is represented by an abort edge to the Abort node and the absence of an outgoing flow edge;

Resolution The resolution of abrupt completion is represented by an Abort node with an outgoing flow edge, the syntax node from which the abrupt completion originates is indicated by an edge labeled abortFrom;

Immediate resolution Abrupt completion that can be resolved on introduction (e.g. an unconditional jump to a labeled statement) is represented by an Abort node that features an incoming abort and outgoing flow edge;

Resumption The resumption of abrupt completion is represented by an Abort node that features an incoming resumeAbort edge.

All variants of Abort nodes have at least one reason. An Abort node can have several reasons: for readability we can combine abrupt completion control flow that behaves the same for a set of different reasons in one Abort node.

Not present in this meta-model, but present in control flow specifications in the CFSL due to the fact that our abstract syntax representation forms the base of our specifications, are the child edges and the syntax role annotation edges. Another relaxation is made for the CFSL: we may use child edges or the language specific syntax role edges, or both, whatever suits us.

5.1.2 Additional constraints

When we consider all possible specification graphs that conform to the meta-model of CFSL, introduced above, a significant portion of these graphs are still considered not valid. The meta-model is not strict enough in the sense that it allows illegal combinations of elements, as there is no way for denoting that some element excludes another element in a meta-model. To provide more accurate guidelines to what we consider valid and invalid specifications we introduce a set of additional constraints on control flow specifications in CFSL.

To provide means for checking whether a specification adheres to the additional constraints, we provide yet another graph transformation system, in which the specification to be checked is the start graph. For each constraint we have a graph production rule that attempts to match a structure that violates that constraint. If a match is found in the control flow specification graph, we introduce a ConstraintViolation node with another label that indicates the constraint that was violated.

We have identified the following additional constraints for specifications that conform to the CFSL meta-model:

1. A specification can contain at most one node labeled KeyElement (Figure 5.2(a));
2. A node can have at most one exit (Figure 5.2(b));
3. From the exit to the context, no control flow can originate (Figure 5.2(c));
4. Sequential control flow originating from a node excludes branching control flow from that node and vice versa (Figure 5.2(d));
5. Branching control flow originating from a node can feature at most one default case (Figure 5.2(e));
6. All Branch nodes have a branchOn edge or a branchDefault edge (Figure 5.2(f));

7. A Branch node can not feature both a branchOn and a branchDefault edge (Figure 5.2(g));
8. The branchOn values of branches that originate from the same node should all be different (Figure 5.2(h));
9. All branches originating from a node should share the same condition (Figure 5.2(i));
10. Between Branch and Abort nodes, flow edges are not allowed (Figure 5.2(j));
11. An abort edge from a node excludes resumeAbort from that node and vice versa (Figure 5.2(k));
12. Abrupt completion resumption must feature the same reason as an instance of abrupt completion resolution (Figure 5.2(l)).

We have two additional rules: OK (lowest priority, Figure 5.3(a)) and INVALID (highest priority, Figure 5.3(b)). These are primitive indicators of the fact that a specification does or does not violate any constraints (compare to the final indicator (Figure 2.8) for the example in Chapter 2).

The rule application LTS of a specification start graph without constraint violations is shown in Figure 5.5(a). An example of a constraint violation (constraint 3) and the detection thereof by the corresponding constraint checking rule is shown in Figure 5.4. The corresponding LTS is also shown in Figure 5.5.

5.1.3 Design steps for specifications

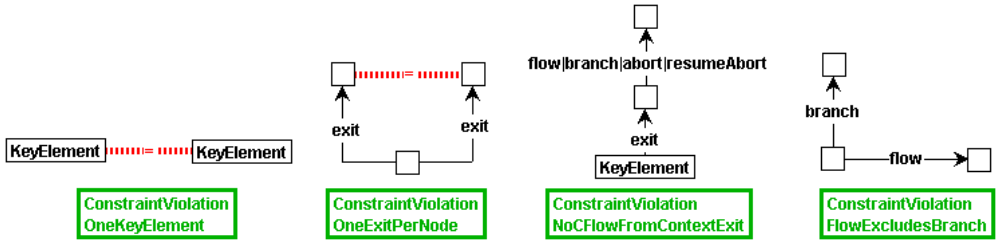
To specify the control flow semantics of a programming language in CFSL one has to perform several design steps:

1. Compose abstract syntax grammar;
2. Extract abstract syntax graph representations;
3. Specify control flow.

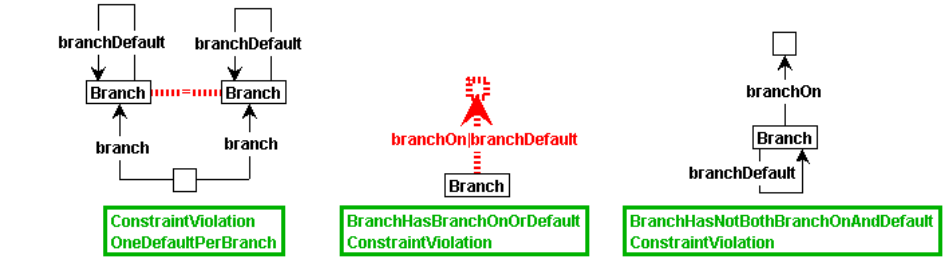
Abstract grammar composition

The first step is to compose an abstract grammar. The programming language probably features a syntax grammar in EBNF. From this syntax grammar an abstract, pure BNF grammar has to be composed (for example, see Appendix A). We summarize the rewrite steps for this abstract grammar (see Section 3.1):

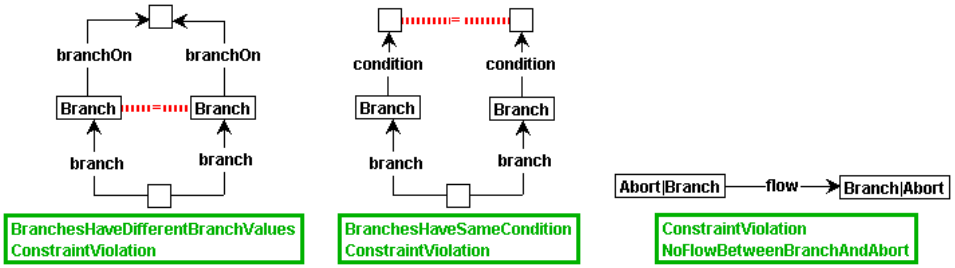
1. Plain BNF rules are not allowed to have optional elements; rewrite such EBNF rules to plain BNF rules by introducing new non-terminals for the variations and combining these with the or-operator;
2. Rules that feature the Kleene Star (*) should be rewritten to recursive BNF rules;
3. When the abstract syntax graph representation should represent a rule using *inheritance* (Figure 3.1), the right hand side cannot feature terminals;
4. Specific syntax role annotations in the syntax graph representation are denoted in this grammar with the notation: `role:NonTerminal`.



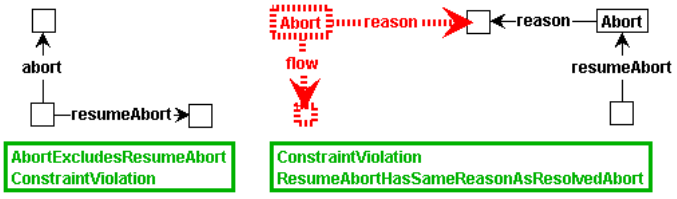
(a) Constraint 1 (b) Constraint 2 (c) Constraint 3 (d) Constraint 4



(e) Constraint 5 (f) Constraint 6 (g) Constraint 7



(h) Constraint 8 (i) Constraint 9 (j) Constraint 10



(k) Constraint 11 (l) Constraint 12

Figure 5.2: The additional constraints on specifications in CFSL.

(a) Ok (b) Invalid

Figure 5.3: Two additional rules for constraint checking.

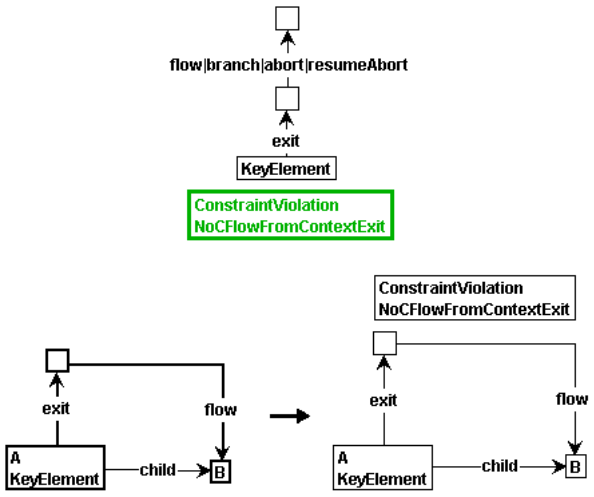
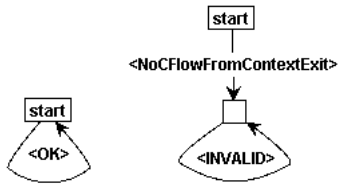


Figure 5.4: Example of detecting a constraint violation.



(a) LTS - Ok (b) LTS - Invalid

Figure 5.5: Examples of rule application LTS's for valid and invalid specifications.

Abstract syntax graph extraction

The second step is to extract the abstract syntax graph representations. From an abstract grammar it is easy to derive the corresponding graph representations. In an abstract syntax graph representation of a BNF rule terminals are omitted, the left hand side is represented as the parent node and the right hand side non-terminals as the children. The syntax roles are represented as edges from the parent node to one of its children. Some relations between the left hand side non-terminal and a right hand side non-terminal may be represented using *inheritance*.

Specification of control flow

The third step is to create the set of control flow specifications. For each left-hand-side non-terminal that represents a construct that has control flow semantics, we create a corresponding control flow specification in several steps:

1. Start with the graph representation of the abstract BNF syntax rule;

2. Label the syntax node corresponding to the left-hand-side non-terminal as the KeyElement;

3. For this KeyElement, specify the exit edge to a new, unlabeled node (the exit to the context);

4. Also, for this KeyElement, specify an entry edge either as self-edge (no sub statements) or to one of the sub statements or the unlabeled exit node (the construct is skipped);

5. Specify all control flow for the KeyElement and its sub statements, adhering to the CFSL meta-model and the additional constraints. Keep in mind that all control flow edges are implicitly assumed to leave from the exit of a syntax node and enter the target syntax node at its entry;

6. Exits of sub statements may be connected to the unlabeled exit node of the KeyElement to imply sharing of the exit;

7. If necessary, the exit of a sub statement can be represented explicitly using again an unlabeled node;

8. If absolutely necessary, create additional helper specifications that do not directly correspond to a syntax rule (i.e. do not feature a KeyElement).

Example control flow specification design

As an example for designing control flow specifications in CFSL, we show how we design a specification for the while-statement in Java.

As a first step, we introduce two syntax role labels (condition and body) to the BNF rule of the WhileStatement:

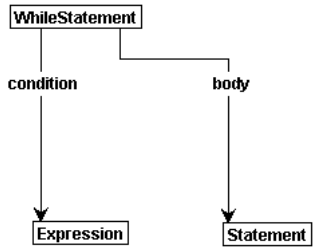
```
WhileStatement ::= <WHILE> <LPAR> condition:Expression <RPAR> body:Statement
```

We extract the abstract syntax graph representation shown in Figure 5.6(a). Note that we do not use the child edges here, only the syntax roles we introduced to the BNF rule.

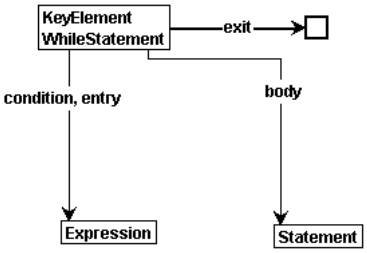
Now we start with the actual control flow specification. In Figure 5.6(b), we specify the WhileStatement to be the KeyElement, the condition Expression to be the entry point of the while-statement (execution of the while-statement starts by evaluating the condition) and denote an unlabeled node to be the exit of the while-statement.

We specify sequential control flow (Figure 5.6(c)): a flow edge from the Expression to the WhileStatement and a flow edge from the Block (implicitly from its *exit*) to the Expression (implicitly to its *entry*). The first flow edge indicates that after evaluating the condition, control is transferred to the while-statement node, where the decision whether to continue iterating the body is made. The second flow edge indicates that after execution of the body, the condition is re-evaluated.

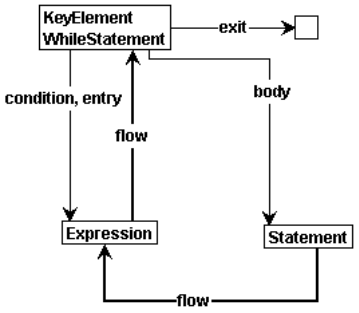
Two branches are specified (Figure 5.6(d)): on `false` the while-statement is completed (normally) by transferring control to the exit node, on `true` the body is entered for another iteration.



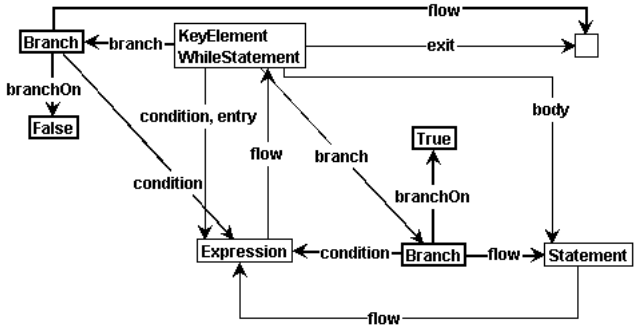
(a) While-statement - abstract syntax



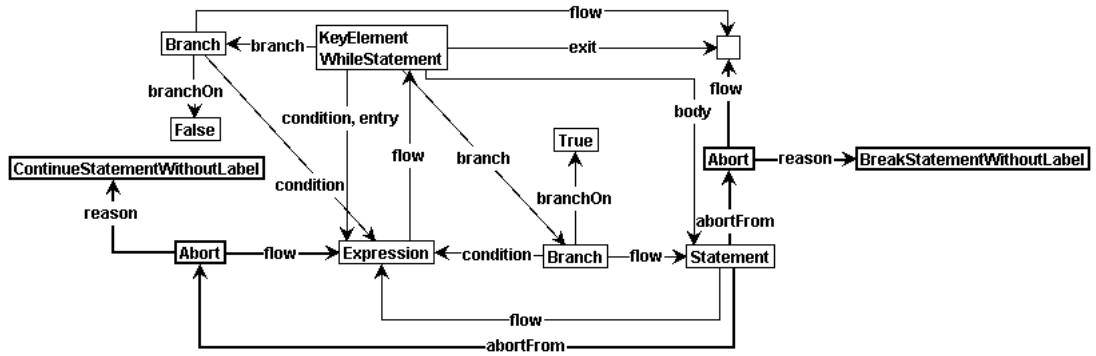
(b) While-statement - key element



(c) While-statement - sequential control flow



(d) While-statement - conditional branching



(e) While-statement - abrupt completion

Figure 5.6: The control flow specification for blocks of statements.

Our control flow specification is nearly finished, but we still have to specify two cases of abrupt completion resolution (Figure 5.6(e)). Because of a break-statement in the while-statement's body, the while-statement will be terminated. We specify an Abort node with a reason edge to a BreakStatementWithoutLabel node and a flow edge to the exit of the while-statement. The abrupt completion originates from somewhere in the while-statement's body, we indicate this with an abortFrom edge from the Block.

In case of a continue-statement in the while-statement's body, the condition of the while-

statement will be re-evaluated. We specify an Abort node with a reason edge to a ContinueStatementWithoutLabel node and a flow edge to the condition. We again indicate the origin of the abrupt completion flow with an abortFrom edge from the Block.

Labeled break- and continue-statements are not considered here; abrupt completion of labeled statements is treated uniformly, as we will see in the next section.

5.2 Control flow specifications for Java

In this section, we present the control flow specifications of the Java constructs that were also presented as examples in the chapter on flow graph construction rules (Section 4.2). The abstracted grammar of these constructs can again be found in Appendix A.

5.2.1 Method bodies

Method-bodies in Java are described in Section 4.2.1. Related flow graph construction rules are shown in Figure 4.7, 4.30(c) and 4.31(b).

The CFSL specification is shown in Figure 5.7. The control flow of a method-body is specified to start at the (first statement of the) body block of statements. After execution of the body, the control transfers to the exit of the MethodBody.

The specification also specifies possible abrupt completion resolution for return- or throw-statements, represented by an Abort node with the combined reasons: `return` and `throw`. The resolution is defined as a transfer of control to the exit of the the MethodBody.

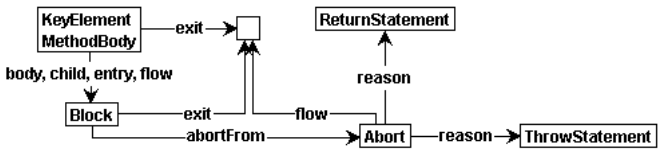


Figure 5.7: The control flow specification of a method-body.

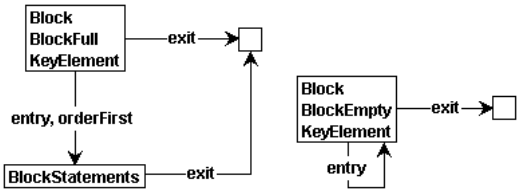
5.2.2 Blocks of statements

Blocks of statements in Java are described in Section 4.2.2. Related flow graph construction rules are shown in Figure 4.8(a), 4.8(b), 4.8(c) and 4.8(d).

A Block can be empty or can contain a number of sequentially ordered statements. If the block of statements is empty, it is specified to be skipped (Figure 5.8(b)). If it contains statements, the execution of the block is defined as the execution of the statements it contains from left to right (Figure 5.8(a)). The exit of the BlockStatements coincides with the exit of the Block.

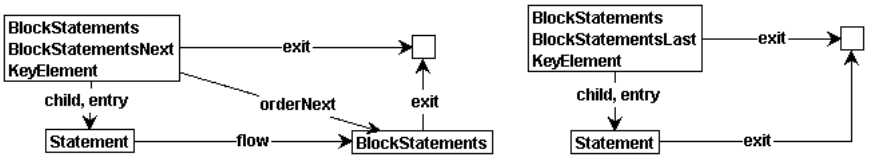
A BlockStatementsNext node features a single Statement child that is specified to be executed upon entering the parent BlockStatementsNext node and a BlockStatements node that is defined to be executed next (Figure 5.9(a)).

The exit of the last statement coincides with the exit of the parent BlockStatementsLast node and, indirectly, with the exit of the containing Block node (Figure 5.9(b)).



(a) Block (b) Block - empty

Figure 5.8: The control flow specification for blocks of statements.



(a) BlockStatements - next (b) BlockStatements - last

Figure 5.9: The control flow specification for the ordering of statements in a block.

5.2.3 Conditional statements

As is described in Section 4.2.3, Java features to types of conditional statements: the if- and the switch-statement.

If-statement

If-statements in Java are described in Section 4.2.3. The related flow graph construction rules are shown in Figure 4.9(a) and 4.9(b).

The control flow specifications of the if-statement with or without else-part are shown in Figure 5.10.

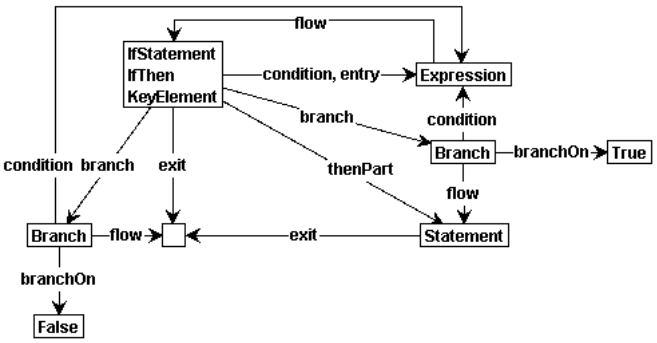
Switch-statement

Switch-statements in Java are described in Section 4.2.3. The related flow graph construction rules are shown in Figure 4.2.3, 4.11(a), 4.11(b), 4.11(c), 4.11(d), 4.12(a), 4.12(b) and 4.26(e).

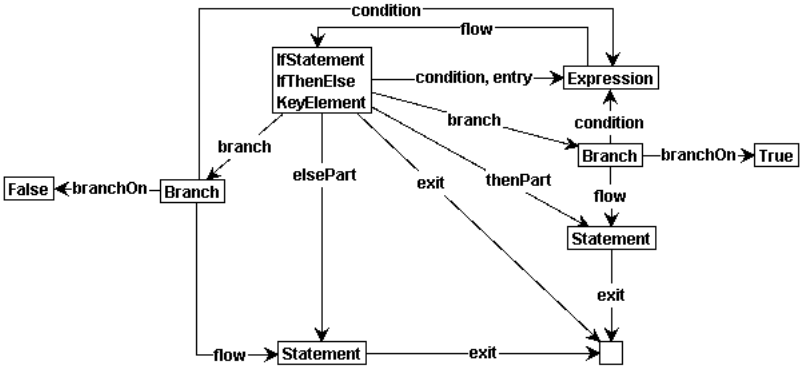
We specify the control flow semantics of the switch-statement in a number of specifications. In Figure 5.11 we specify that the switch-statements starts by evaluating its condition and that the complete switch-statement is terminated by any break-statement present in the statements of one of its cases.

In Figure 5.12(a), we specify that a conditional branch, based on the value of the switch-statement's condition, is present for any of the labels in the block of cases, leading to the statement(s) associated to the label. In Figure 5.12(b), we treat the special `default` label.

We specify that the groups are connected to each other in their sequential ordering to facilitate the "fall-through". We use a similar approach as for statements in a block (Figure 5.9 and 5.8): Figure 5.13 specifies the control flow for the block of switch-statement-block-groups. Figure 5.14 specifies that after executing the statements associated to a group, the



(a) IfThen



(b) IfThenElse

Figure 5.10: The flow specification of the if-statement with or without else-part.

control flow is transferred to the next group or to the exit of the switch-statement, depending on whether there is another group ordered next.

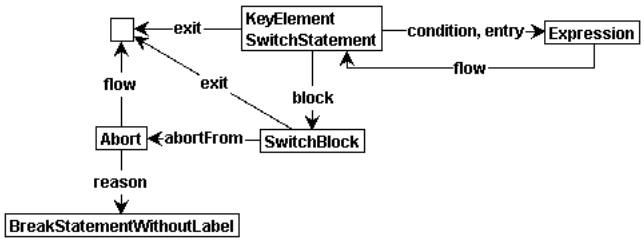
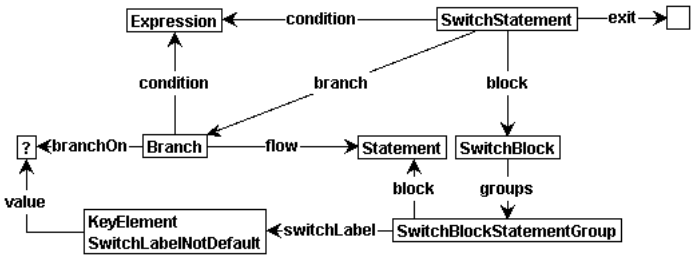


Figure 5.11: The flow specification of the switch-statement.

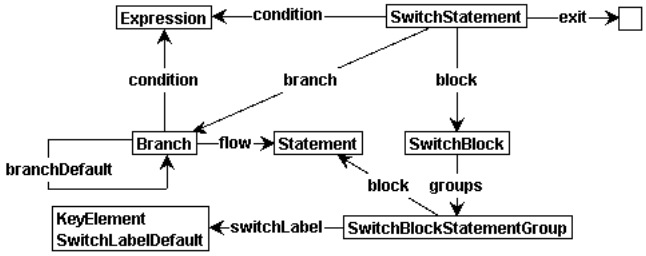
5.2.4 Loop statements

Loop statements in Java are described in Section 4.2.4. Related flow graph construction rules are shown in Figure 4.7, 4.30(c) and 4.31(b).

Java features three types of loop-statements: the `while`, `do` and `for` loops. The control flow



(a) Switch label



(b) Switch label - default

Figure 5.12: The flow specification of the switch-label.

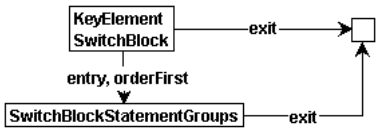
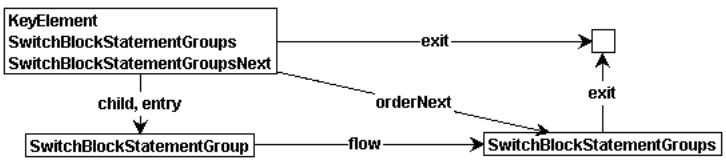
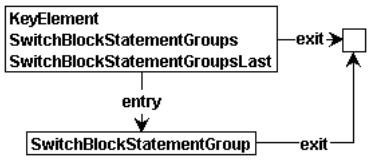


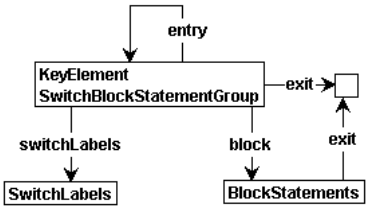
Figure 5.13: The control flow specification of the groups in a switch-block.



(a) Switch groups - next



(b) Switch groups - last



(c) Switch group

Figure 5.14: The control flow specification of the switch-statement-blocks-groups.

specification for the while-statement was given in Figure 5.6(e). The control flow specification of the do-statement (Figure 5.15) is a minor variation on this specification.

The for-statement has eight variants, which are described in Section 4.2.4:

1. ForEver: for-statement without init-, condition- or update-part (Figure 5.16);
2. ForWithInit: for-statement with init-part only (specification omitted here);
3. ForWithInitCondition: for-statement with init- and condition-part (specification omitted here);
4. ForWithInitConditionUpdate: for-statement with init-, condition- and update-part (Figure 5.17);
5. ForWithInitUpdate: for-statement with init- and update-part (specification omitted here);
6. ForWithCondition: for-statement with condition-part only (specification omitted here);
7. ForWithConditionUpdate: for-statement with condition and update-part (specification omitted here);
8. ForWithUpdate: for-statement with update-part only (specification omitted here);

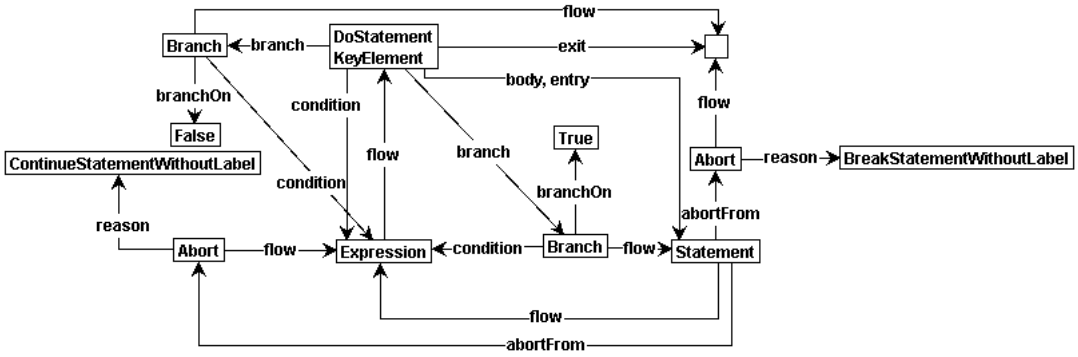


Figure 5.15: The flow specification of the do-statement.

5.2.5 Primitive statements and expressions

As in Section 4.2.5, we treat a limited number of primitive statements and expression types. The (statement) expression types we treat are:

- Empty statements;
- Assignments;
- Local variable declarations;
- Literals;

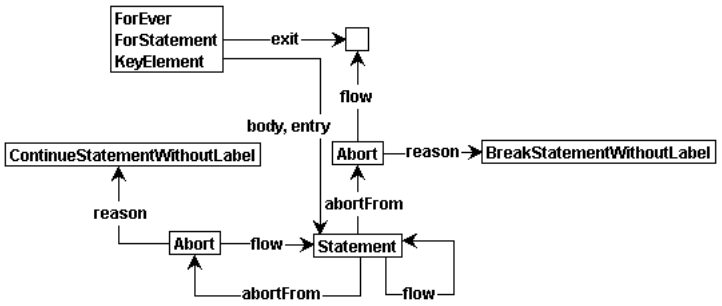


Figure 5.16: The flow specification of the for-statement without init-, condition- or update-part.

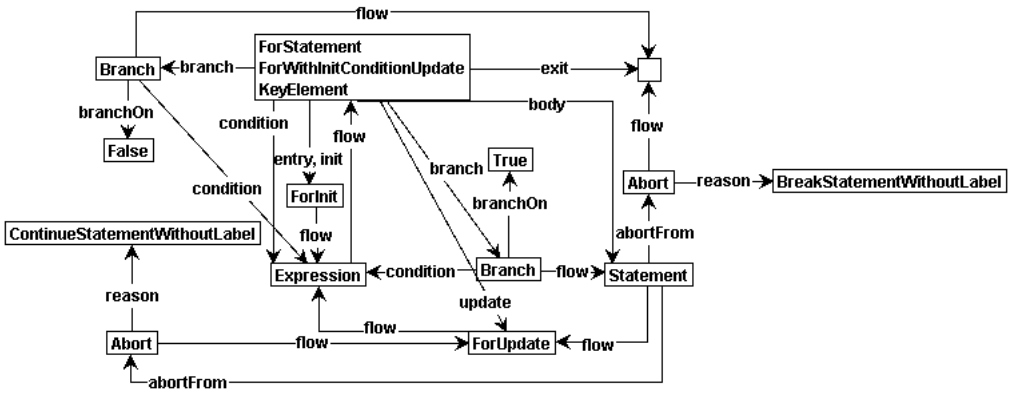


Figure 5.17: The flow specification of the for-statement with init-, condition- and update-part.

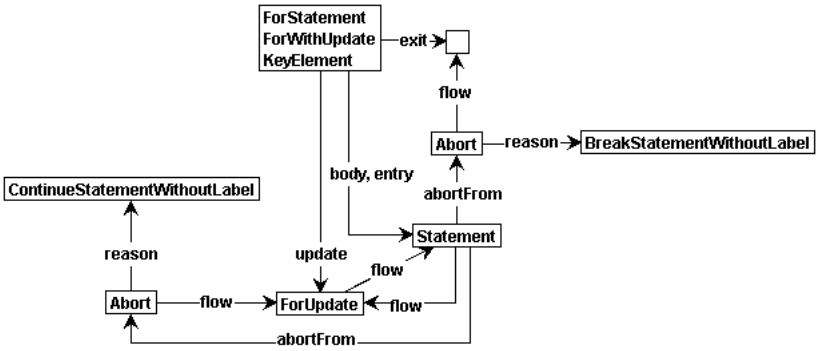


Figure 5.18: The flow specification of the for-statement with update-part.

- Identifiers;
- Binary operators;
- Method invocations.

Empty statement

Empty statements in Java are described in Section 4.2.5. The related flow graph construction rule is shown in Figure 4.18. Figure 5.19 shows the control flow specification of empty statements.

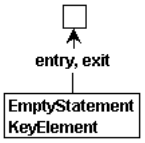


Figure 5.19: The control flow specification of empty statements.

Assignment

Assignments in Java are described in Section 4.2.5. The related flow graph construction rule is shown in Figure 4.19. Figure 5.20 shows the control flow specification of assignments.

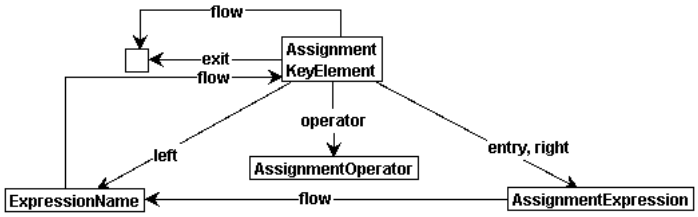


Figure 5.20: The control flow specification of assignments.

Local variable declaration statement

Declarations of local variables in Java are described in Section 4.2.5. Recall that several variables of the same type can be declared in one LocalVariableDeclarationStatement. Related flow graph construction rules are shown in Figure 4.20, 4.21(a) and 4.21(a).

Figure 5.21 shows the control flow specification of local variables declaration statements.

As one declaration can declare several variables, we have an ordering of VariableDeclarators that are evaluated left to right. Control flow specification of this ordering is similar to specifying the control flow of statements ordered in a block (Figure 5.9(a) and 5.9(b)), therefore the corresponding specifications are omitted here.

An individual VariableDeclarator can feature an initializing expression or not. In Figure 5.22 we specify the control flow of both variants.

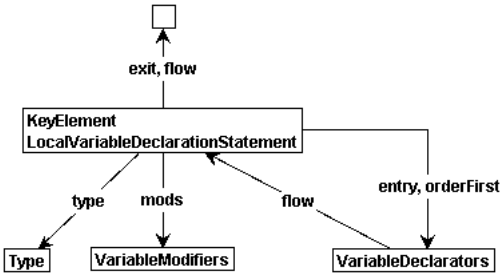
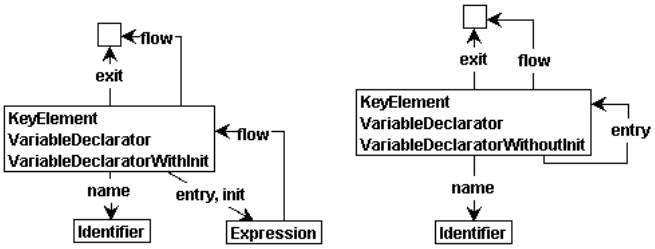


Figure 5.21: The control flow specification of local variable declaration statements.



(a) Variable declarator - init (b) Variable declarator - no init

Figure 5.22: The control flow specification of variable declarators with or without init.

Literal

Literal values in Java are described in Section 4.2.5. The related flow graph construction rule is shown in Figure 4.2.5. Figure 5.23 shows the control flow specification of literal values.

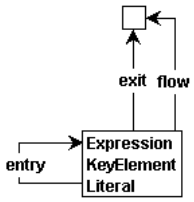


Figure 5.23: The control flow specification of literal values.

Identifier

Identifiers in Java that refer to declared variables are described in Section 4.2.5. The related flow graph construction rule is shown in Figure 4.2.5. Figure 5.24 shows the control flow specification of identifiers.

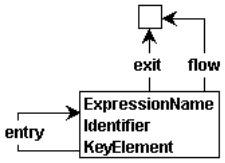


Figure 5.24: The control flow specification of identifiers.

Binary operator

Binary operator expressions in Java are described in Section 4.2.5. Again we examine relational expressions as an example of binary operators, the related flow graph construction rule is shown in Figure 4.2.5. Figure 5.25 shows the control flow specification of relational expressions.

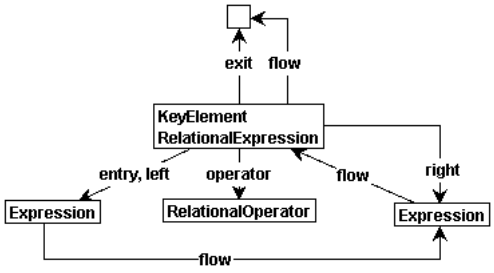


Figure 5.25: The control flow specification of relational expressions.

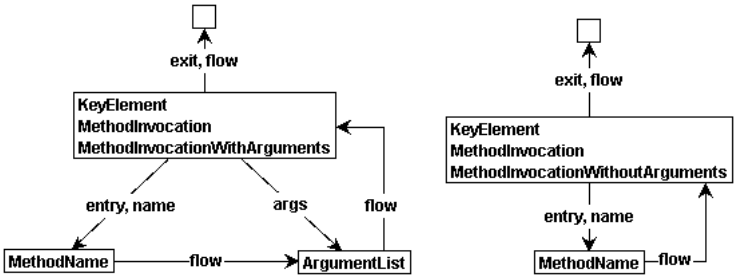
Method invocation

Invocations of methods in Java are described in Section 4.2.5. Method invocations can feature an ArgumentList or not. The related flow graph construction rules are shown in Figure 4.25(a) and 4.25(b). Figure 5.26 shows the control flow specifications of method invocations with or without arguments.

A method invocation with arguments features an ordered list of Arguments that are evaluated left to right, each individual argument being an Expression. Control flow specification of this ordering is similar to specifying the control flow of statements ordered in a block (Figure 5.9(a) and 5.9(b)), therefore the corresponding specifications are omitted here.

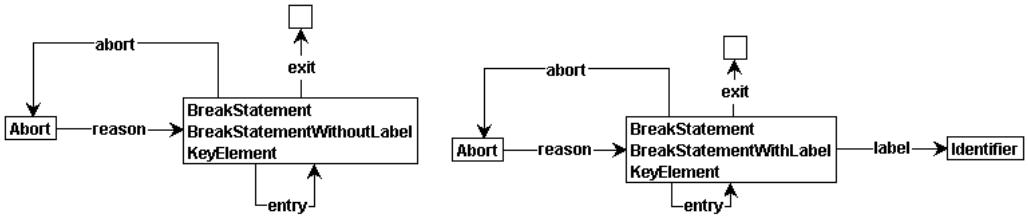
5.2.6 Abrupt completion statements

The abrupt completion statements in Java are described in Section 4.2.6. The specifications for the abrupt completion statements indicate that they terminate some enclosing statement, i.e. that they introduce unresolved abrupt completion. Figure 5.27 specifies the control flow of the break-statement. Figure 5.28 specifies the control flow of the continue-statement. Figure 5.30 shows the specification for the return-statement with or without value. Figure 5.31 shows the specification of the throw-statement with its exception expression.



(a) Method invocation - arguments (b) Method invocation - no arguments

Figure 5.26: The control flow specification of method invocations with or without arguments.



(a) Break - no label (b) Break - label

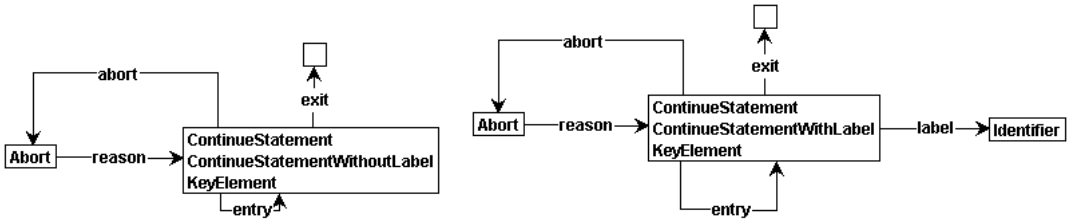
Figure 5.27: The control flow specification of the break-statement with or without label.

5.2.7 Exception handling statements

The exception-handling try-statement is described in Section 4.2.7.

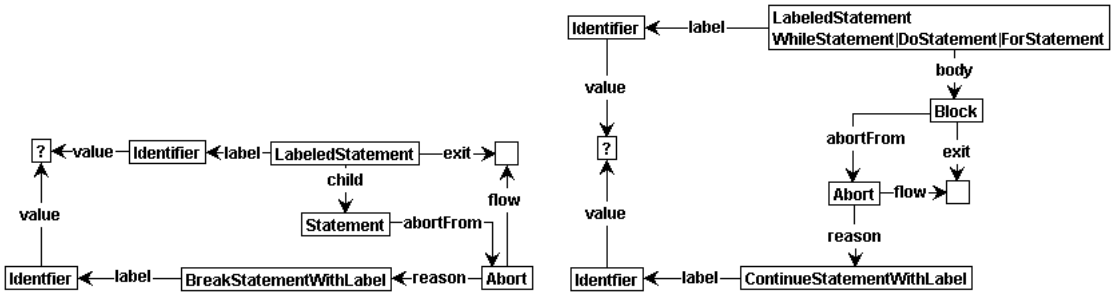
We have three variants of the try-statement: the try-catch, the try-finally and the try-catch-finally.

The specification for the try-catch specifies that any thrown exception in the body of the try-statement causes control to be transferred to the catches-block first, and that after evaluation of the catch-clauses the exception might travel further in a bottom-up manner (indicated by the resumeAbort edge from the Catches). But when the exception is caught



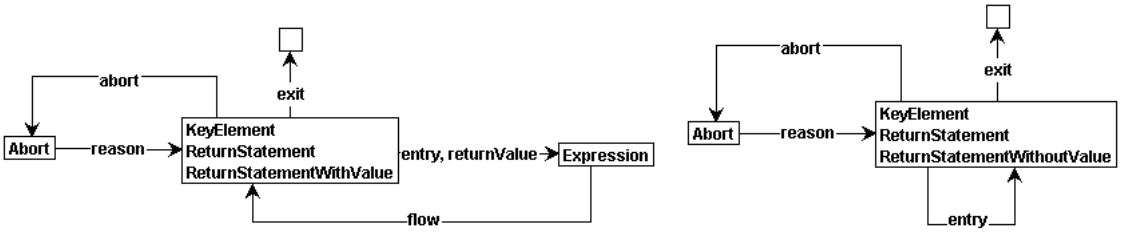
(a) Continue - no label (b) Continue - label

Figure 5.28: The control flow specification of the continue-statement with or without label.



(a) Labeled statement - break (b) Labeled statement - continue

Figure 5.29: The control flow specification of labeled statements aborted by a break- or continue-statement.



(a) Return - value (b) Return - no value

Figure 5.30: The control flow specification of the return-statement.

in the catches-block, the control is defined to be transferred to the exit of the try-statement (which as a result completes normally).

The try-finally specification specifies that any type of abrupt completion of the body of the try-statement causes the control to be transferred to the finally-statement. After execution of the try-statement, the abrupt completion might be resumed in the bottom-up way (indicated by the resumeAbort edge from the FinallyStatement).

The specification for the try-catch-finally combines both specifications. After normal or abrupt completion of either the try-body or the catches-block, the finally-statement is executed. After the finally-statement, the bottom-up abrupt completion might resume.

The specifications of the sequentially ordered catch-clauses are comparable to the statements in a block (Figure 5.9).

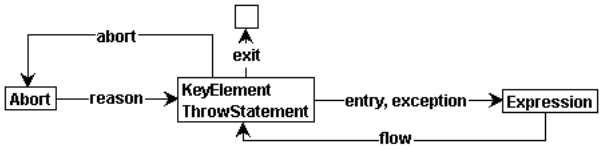
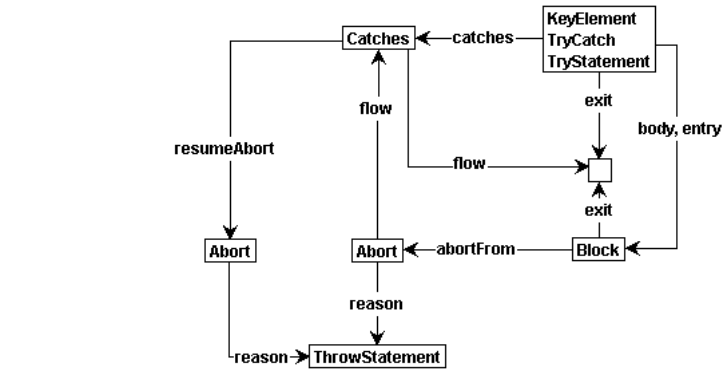
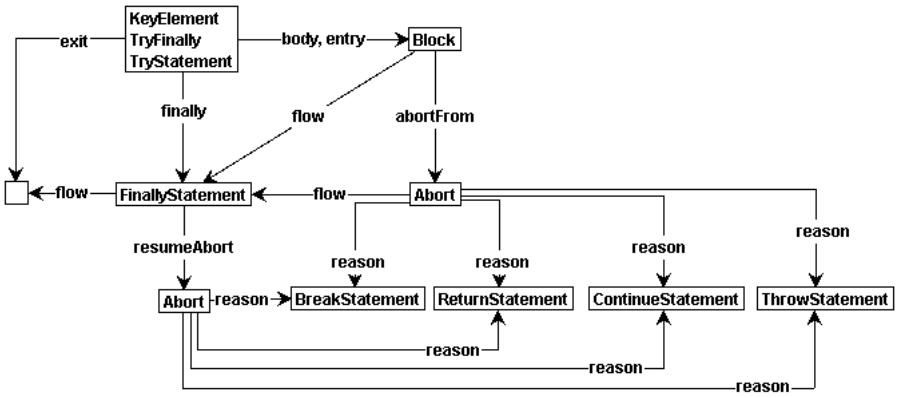


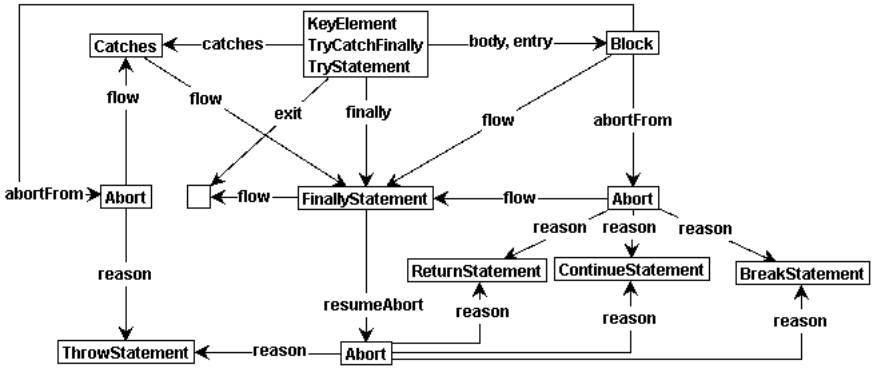
Figure 5.31: The control flow specification of the throw-statement.



(a) TryCatch



(b) TryFinally



(c) TryCatchFinally

Figure 5.32: The control flow specifications of the try-statement.

Figure 5.33 shows the specification of a catch-clause. As in the associated production rule (Figure 4.38), we have a branch on the thrown exception's (runtime) type. The specification denotes that if the type-comparison evaluates to `false`, the exit (leading to the next clause) is to be taken, otherwise the body is to be executed. After executed of the body, the exit of the catches-block is followed.

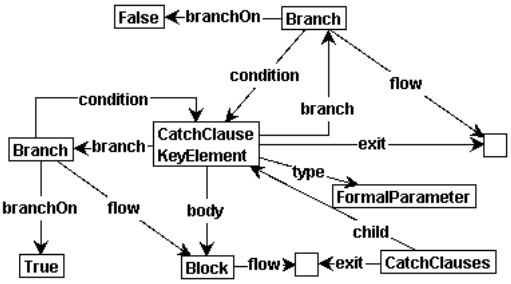


Figure 5.33: The control flow specification of the catch-clause.

We omit the (straightforward) control flow specification of the FinallyStatement.

Chapter 6

Flow Graph Meta-rules

This chapter treats the flow graph meta-rules we provide with our control flow specification framework.

We briefly review the purpose of these flow graph meta-rules. In Chapter 4 we showed how we can use graph transformations to construct control flow graphs, given an abstract syntax graph representation of the source code. The case study we have performed on flow graph construction for Java showed us that designing flow graph construction rules for some programming language (in this case, Java) is not a trivial task and involves a lot of work, when designing these rules by hand.

We can do better: the meta-rules we present here can *transform* control flow specifications of a given programming language to the corresponding flow graph construction rules for that language. When we review the big picture of this thesis presented in Chapter 1, we see (Figure 1.1) that the flow graph meta-rules (5), designed by us, transform a set of control flow specifications (10), designed by a language designer, into a set of flow graph construction rules (16).

Summarizing:

> Our flow graph meta-rules transform a set of control flow specifications of a given language, denoted in CFSL, into a set of graph production rules that can construct flow graphs of programs written in this language.

6.1 Design of the meta-transformation

Both the control flow specifications (Chapter 5) and the flow graph construction rules (Chapter 4) are graphs. As we saw in the big picture (Figure 1.1), we again apply graph transformations in order to transform control flow specifications into control flow graph construction rules.

The rules that guide these transformations are *meta-rules* since they are graph production rules that generate graph production rules.

These flow graph meta-rules are applicable to any control flow specification of any programming language, specified in our control flow specification language. This means that they do not depend on language- (for example, Java-) specific constructs. Thus, the rules only match elements present in the control flow specification language meta-model (Figure 5.1) and introduce only elements present in the flow graph meta-model (Figure 3.5) and the auxiliary and temporary elements needed for flow graph construction (Figure 4.3).

6.1.1 Flow graph meta-rules

We have designed a set of flow graph meta-rules. These flow graph meta-rules share the assumptions and design choices made for the control flow specification language and the flow graph construction rules, in particular:

- Flow graph construction uses a top-down approach;
- Abrupt completion resolution operates bottom-up.

There are some clear differences between control flow specifications and flow graph construction rules, besides the fact that the former are specifications and the latter are graph production rules that can be applied to a source graph. In our flow specifications, we combine the specification of all control flow semantics of one type of statement (denoted as the KeyElement) into a single specification, thereby combining the top-down control flow and the bottom-up abrupt completion resolution (represented by the abortFrom edge).

We can make this more clear by considering our flow specification of the while-statement in Java (Figure 5.6(e)). This specification specifies the sequential and branching control flow with respect to the sub statements of the while-statement. It also specifies how certain types of abrupt completion that originate from the body of the while-statement are resolved.

In our flow graph construction rules, we split up the top-down construction and the bottom-up abrupt completion resolution into separate rules (Figure 4.13, 4.26(b) and 4.27(b)).

This means that we in some cases generate several production rules, given one specification. Given a control flow specification, we first transform it into a top-down production rule. Next, we apply a different set of meta-rules to the original specification to transform it into, perhaps several, additional bottom-up abrupt completion resolution rules.

The two sets of meta-rules differ in the way that they are applied to a flow specification graph. The rules for construction top-down production rules are *confluent* (see Section 2.2.2), meaning that we can apply these rules in a *linear* fashion. The end result (the final state in the transformation LTS) of this linear transformation process is the flow graph construction rule corresponding to the input control flow specification.

The meta-rules for the construction of abrupt completion resolution rules have to be applied differently. A flow specification can feature several instances of abrupt completion resolution. For each instance, we generate a different production rule. If we for example have two instances of abrupt completion resolution for the while-statement (the `break` and the `continue`), we select one of these instances for which we generate a rule at a time. The meta-rules can now be applied in linear fashion and the end result is the flow graph construction rule that resolves this instance of abrupt completion.

To make this more concrete, suppose we have designed a set of control flow specifications for some programming language, say Java, and now want to use the provided meta-rules to generate the corresponding flow graph construction rules. We use two graph production systems: one for creating top-down flow graph construction rules and one for creating bottom-up abrupt completion resolution rules. We introduce each control flow specification graph subsequently as the start graph of these production systems.

Among others, we introduce a WhileStatementSpecification graph (the control flow specification for a Java while-statement) as the start graph of our flow graph construction graph production system. After linear exploration of the applications of the meta-rules in this

production system, we end up with a flow graph construction rule: the WhileStatementRule. This rule application process is illustrated in Figure 6.1.

Next we present this specification graph as the start graph of the abrupt completion resolution graph production system. Two different linear explorations are performed, resulting in the WhileStatementAbortBreakWithoutLabelRule and the WhileStatementAbortContinueWithoutLabelRule. This somewhat more complicated rule application process is illustrated in Figure 6.2.

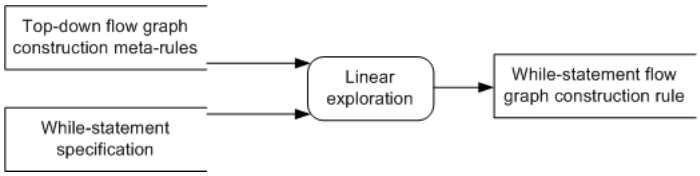


Figure 6.1: The application of the top-down flow graph construction meta-rules to an example control flow specification.

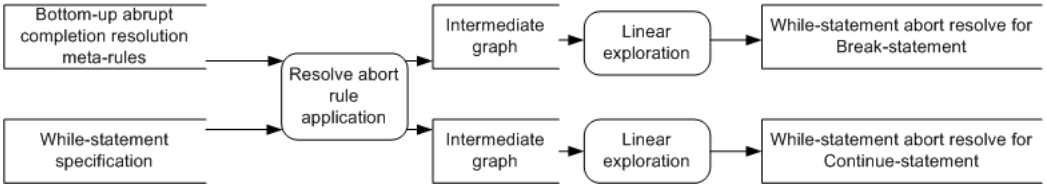


Figure 6.2: The application of the bottom-up abrupt completion resolution meta-rules to an example control flow specification.

Now that we have treated the purpose and *modus operandi* of our sets of meta-rules, we delve into more detail on the individual flow graph meta-rules.

As the flow graph meta-rules are production rules that create production rules, we have many cases in which we, for instance, have to specify the "creation an edge creator". For example, when we want to create a rule that introduces a flow edge to some node, our meta-rule has to create a new edge that will create the flow edge when the resulting production rule is applied.

How do we represent such a meta-transformation graphically? The creation of an edge is, as we know, specified using a bold, green, edge. But, for the creation of creating an edge we have no graphical representation available in the GROOVE tool. Our solution is simple: we prefix the label of the edge that is created with the type of transformation it has to perform if the resulting production rule is applied.

We have, of course, four types of prefixes:

new The creation of an element, i.e. a *creator*;

del The deletion of an element, i.e. an *eraser*;

not The required absence of an element, i.e. an *embargo*;

use The required presence of an element, i.e. a *reader*.

For instance, the creation of a flow edge creator is thus denoted graphically as a bold, green line with the label new_flow (Figure 6.3).

Figure 6.3: Example of creating a flow edge creator.

The following sections treat the two sets of meta-rules in detail.

6.1.2 Meta-rules for top-down flow graph construction rules

We first list the main tasks for the set of production rules that construct top-down flow graph construction rules:

1. Introduce the creation and removal of build edges for top-down construction;
2. Replace auxiliary elements in the specifications with the creation of their corresponding elements in the flow graphs;
3. Introduce the creation of flow, branch or abort edges at the appropriate places;
4. Introduce the merging of some of the FlowConnectors with the elements to which they belong;
5. Introduce the creation of the temporary resolving edge;
6. Remove Abort nodes that resolve abrupt completion.

For each of these tasks, we have designed one or more meta-rules. These meta-rules are listed in Table 6.1. Each meta-rule is related to one or more of the mentioned tasks and has an application priority (see Chapter 2). We treat these rules in more detail in the order in which they are presented in this table.

Create child edges

As mentioned in Chapter 5, to improve the readability and understandability of the flow specifications, we use abstract syntax edges that are annotated with the role of the relation. To have generic meta-rules, we cannot match on the language-specific abstract syntax roles.

Therefore, for each sub statement of the KeyElement of a specification, we add a child edge (Figure 6.4), resembling a syntax tree structure. Now our meta-rules can match sub statements using this edge.

Introduce build edge propagation

For any specification that features a KeyElement, we introduce the removal of the build edge for the KeyElement (Figure 6.5(a)) and the creation of the build edge for the sub statements of the KeyElement (Figure 6.5(b)), matching on the child edge introduced by the rule in Figure 6.4.

Name	Task	Priority	Figure
ChildCreate	-	4	6.4
BuildKeyElementDelIntro	1	3	6.5(a)
BuildChildCreateIntro	1	3	6.5(b)
AbortCreateIntro	2, 5	3	6.6(a)
AbortWithResolveCreateIntro	2, 5	3	6.6(b)
BranchCreateIntro	2	3	6.7(a)
BranchDefaultCreateIntro	2	3	6.7(b)
AbortWithExitCreateIntro	3	2	6.8(a)
AbortWithoutExitCreateIntro	3	2	6.8(b)
BranchWithExitCreateIntro	3	2	6.9(a)
BranchWithoutExitCreateIntro	3	2	6.9(b)
FlowWithEntryCreateIntro	3	2	6.10(a)
FlowWithExitCreateIntro	3	2	6.10(b)
FlowWithEntryExitCreateIntro	3	2	6.10(c)
FlowWithoutEntryExitCreateIntro	3	2	6.10(d)
FlowWithExitToExitNodeCreateIntro	3	2	6.10(e)
FlowWithoutExitToExitNodeCreateIntro	3	2	6.10(f)
EntryFlowConnectorMergeIntro	4	1	6.11(a)
EntrySelfEdgeFlowConnectorMergeIntro	4	1	6.11(b)
SkipFlowConnectorMergeIntro	4	1	6.11(c)
SharedExitMergeIntro	4	1	6.11(d)
AbortResolveResumeReasonDelo	6	4	6.12(a)
AbortResumeDel	6	3	6.12(b)
AbortResolveDel	6	3	6.12(c)
KeyElementDel	-	0	6.13

Table 6.1: The meta-rules for top-down flow graph construction rules with their related tasks and priorities.

Map auxiliary specification elements to flow graph elements

To generate a production rule that introduces abrupt completion from a specification that features an Abort node without flow edge, we, among other things, introduce the creation of a resolving edge to the parent node (for which we introduce a reader) of the node from which the abrupt completion originates.

We also make sure that the specification specifies abrupt completion introduction and not the immediate resolution of abrupt completion, using a negative application condition on the presence of a flow edge in the specification. Figure 6.6(a) shows our meta-rule for creating a rule that introduces abrupt completion.

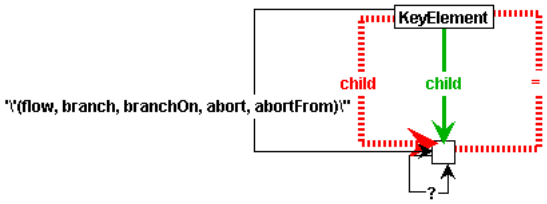
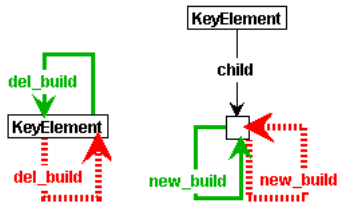


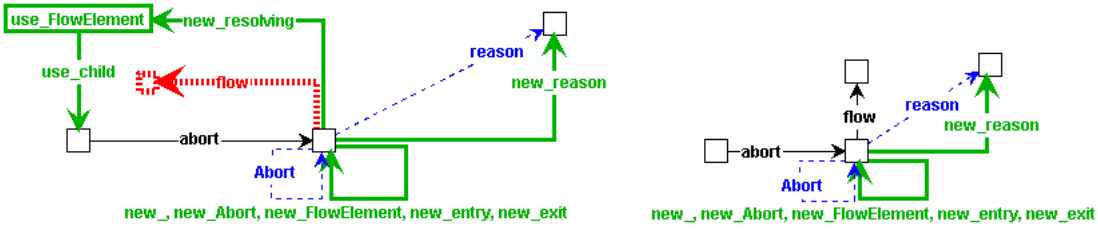
Figure 6.4: The meta-rule that adds a child edge to any sub statement of the specification element.



(a) Build - erase (b) Build - create

Figure 6.5: The meta-rules for creating the top-down construction using build edges.

In case of immediate resolution of abrupt completion, we do not have to introduce the bottom resolution procedure using the resolving edge, as the abrupt completion target is already known at the moment that the abrupt completion is introduced (Figure 6.6(b)).



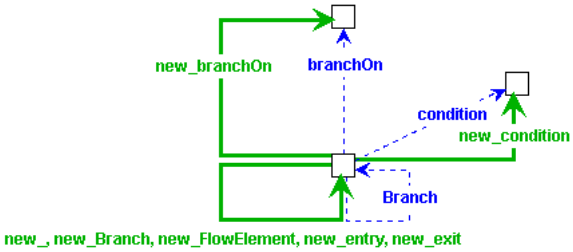
(a) Abrupt completion introduction (b) Abrupt completion immediate resolution

Figure 6.6: The meta-rule that creates the introduction of abrupt completion with or without immediate resolution.

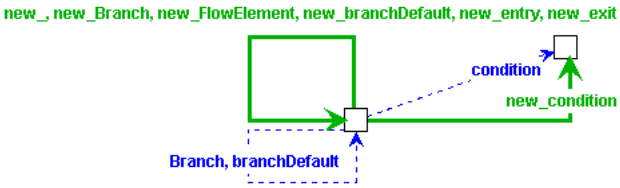
For creating rules that feature Branch introduction, we have two meta-rules: one for Branches with a branchOn edge (Figure 6.7(a)) and one for Branches with a branchDefault edge (Figure 6.7(b)).

Introduce flow, branch and abort edges

For the creation of rules that introduce flow, branch or abort edges, we have to deal with the assumption made in our specification language (see Section 5.1.1) that, unless otherwise



(a) Branch

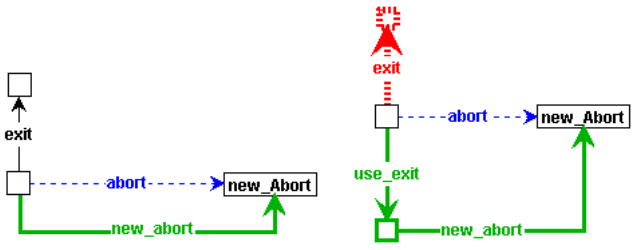


(b) Branch - default

Figure 6.7: The meta-rules for creating the introduction of Branchs for Branches in the specifications.

specified, these edges leave the abstract syntax node from their *exit* and enter an abstract syntax node through its *entry*.

As a result, for each of the three control flow edges we have to consider a number of variants. As the destination of a branch or abort edge always is a Branch or Abort, respectively, for these edges we only discern two alternatives: the edge leaves from the exit of an abstract syntax node or directly from the node itself. This is visible in the rules for the branch edge (Figure 6.9) and the abort edge (Figure 6.8).

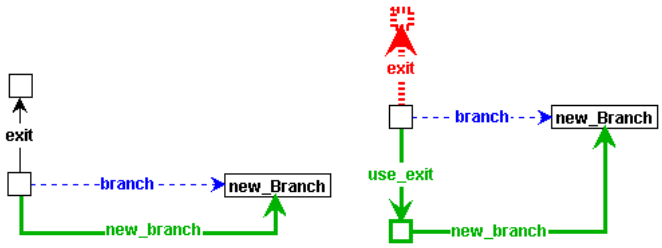


(a) abort - with exit (b) abort - without exit

Figure 6.8: The meta-rules for creating the introduction of abort edges for abort in the specifications.

For the flow edge, we have to discern more variants. Depending on whether the specification has specified an exit for the source of the sequential flow and an entry for the target, we create a different rule that introduces this flow edge. This results in four different meta-rules.

Another possibility is that the target of a flow in a specification is an unlabeled exit node. In this case, we of course do not want to create a rule that connects a new flow edge to the *entry*



(a) branch - with exit (b) branch - without exit

Figure 6.9: The meta-rules for creating the introduction of branch edges for branch in the specifications.

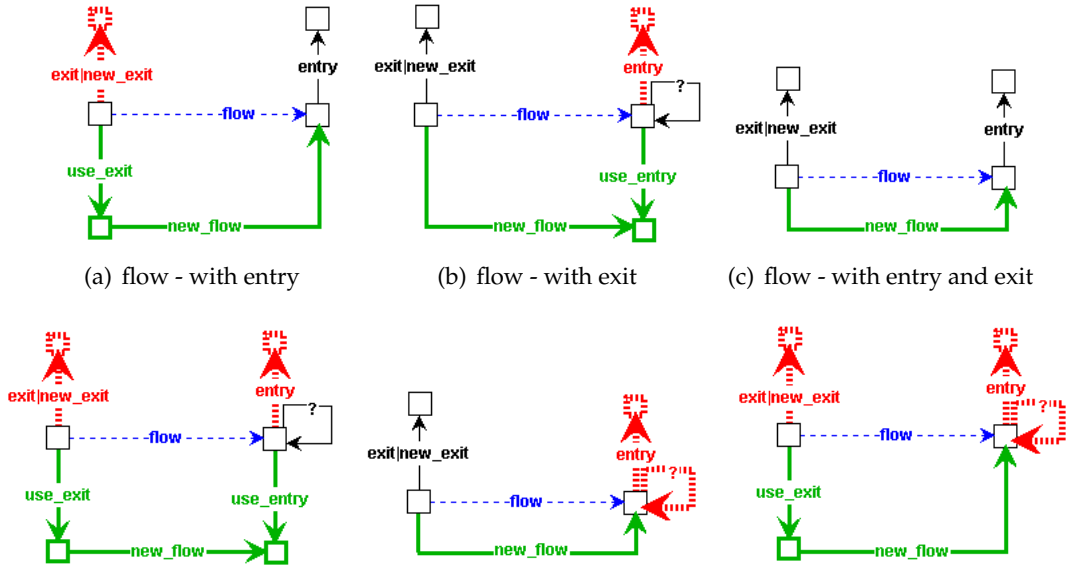
of this node, as this unlabeled node in a specification may map to a FlowConnector or some other FlowElement in a flow graph. In the first case, this is not a problem, because the entry of a FlowConnector is a self-edge of the node. In the second case, the entry may be defined as some sub statement, in which case we connect the flow edge to the wrong flow element. Therefore, we make the unlabeled a special case for these meta-rules (which we check using the regular expression ?, this matches the presence of any label). This results in another two different meta-rules, depending on whether the source of the flow edge has an exit specified or not. The following table gives an overview of the six different meta-rules.

Source exit	Target entry	Exit node	Figure
×	✓	×	6.10(a)
✓	×	×	6.10(b)
✓	✓	×	6.10(c)
×	×	×	6.10(d)
×	✓	✓	6.10(e)
×	×	✓	6.10(f)

Merge flow connectors

In our flow graphs, every flow element has an *entry* and an *exit*. Initially, these are edges pointing to auxiliary FlowConnector nodes. But in many cases, it is possible to define the entry or exit of a flow element as another flow element (for example, a sub statement). In these cases, our flow graph construction rules merge the entry or exit FlowConnector with this flow element. The meta-rules in Figure 6.11 introduce this merging to flow graph construction rules under construction.

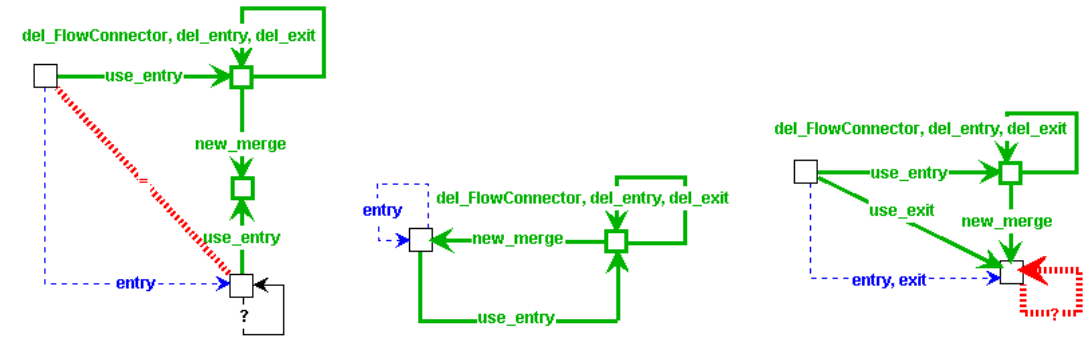
The meta-rule in Figure 6.11(a) merges a flow connector with some statement that is defined as the entry of a statement in a specification. Figure 6.11(b) shows a meta-rule that merges a flow connector with the corresponding flow element itself, if the specification features an entry self-edge. The meta-rule in Figure 6.11(c) merges the entry flow connector with the exit flow connector of a flow element that is to be skipped. Figure 6.11(d) depicts a meta-rule that merges the exit flow connector if a sub statement shares the exit of the KeyElement.



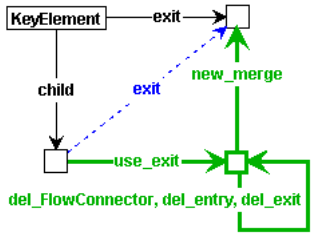
(a) flow - with entry (b) flow - with exit (c) flow - with entry and exit

(d) flow - without entry or exit (e) flow - with exit to exit node (f) flow - without exit to exit node

Figure 6.10: The meta-rules for creating the introduction of flow edges for flow in the specifications.



(a) FlowConnector merge - entry (b) FlowConnector merge - self-edge (c) FlowConnector merge - skip flow element



(d) FlowConnector merge - shared exit

Figure 6.11: The meta-rules for creating the merging of FlowConnectors where appropriate.

Remove abrupt completion resolution

The set of meta-rules for bottom-up abrupt completion resolution rules creates production rules for resolving and, possibly, resuming abrupt completion. Therefore, in the flow graph construction meta-rules set, any nodes or edges related to resolving or resuming abrupt completion present in the specifications should be removed.

First, we remove all reason nodes from the specification, as an abrupt completion (Abort) node in a specification can specify the abrupt completion for several possible reasons (Figure 6.12(a)).

Next, we have two cases: the specification of abrupt completion resolution with or without resumption.

The meta-rule AbortResumeDel in Figure 6.12(b) removes abrupt completion resolution (an Abort node with flow edge) with resumption (another Abort node with the same reason and an abortFrom edge) from the specifications.

The meta-rule AbortResolveDel in Figure 6.12(c) removes abrupt completion resolution without resumption.

(a) Abort deletion - removing reasons

(b) Abort deletion - resuming abrupt completion

(c) Abort deletion - abrupt completion resolution

Figure 6.12: The meta-rules for deleting Abort nodes in the specifications related to abrupt completion resolution.

Remove KeyElement edge

After we have transformed a specification in a top-down flow graph construction rule, we remove the label KeyElement from the main node (Figure 6.13), as this label does not have any meaning in a flow graph construction rule. This rule has the lowest application priority.

Figure 6.13: The meta-rule that removes the KeyElement edge from the specifications.

6.1.3 Meta-rules for bottom-up abrupt completion resolution rules

We again first list the main tasks for the set of production rules that construct bottom-up abrupt completion resolution rules:

- If the specification does not feature any instance of abrupt completion resolution, all elements should be erased;

- Else, replace one Abort node that is an instance of abrupt completion resolution with a corresponding Abort;
- Introduce the creation of resuming bottom-up abrupt completion, if necessary;
- Remove all information relevant to top-down flow graph construction;
- Remove all other Abort nodes.

We again have a set of meta-rules, in which each meta-rule has a certain priority and is related to a one of the mentioned tasks. Table 6.2 enumerates the meta-rules and their properties. The most notable rule is the ResolveAbortIntro rule. All applications of this meta-rule lead to different production rules, as explained before. The other rules are confluent with respect to each other. We treat these meta-rules in the order in which they are presented in the table.

Name	Task	Priority	Figure
AllNoAbortResolutionDel	1	3	6.14
ResolveAbortIntro	2	3	6.15
AbortFlowToEntryCreateIntro	2	3	6.16(a)
AbortFlowNotToEntryCreateIntro	2	3	6.16(b)
AbortFlowToExitNodeCreateIntro	2	3	6.16(c)
ResumeAbortWithExitCreateIntro	3	2	6.17(a)
ResumeAbortWithoutExitCreateIntro	3	2	6.17(b)
AbortResumeCreateIntro	3	3	6.18
AllExceptAbortDel	4, 5	1	6.19
FlowDel	4	1	6.20
EntryDel	4	1	6.21(a)
ExitDel	4	1	6.21(b)
KeyElementDel	4	0	6.22

Table 6.2: Priorities of meta-rules for bottom-up abrupt completion resolution rules.

Specifications without abrupt completion resolution

Specifications that do not feature abrupt completion resolution should be cleared, because this set of meta-rules should not create rules for these kind of specifications. The meta-rule in Figure 6.14 simply removes all nodes from these kind of specifications. Remember that in our specifications we represent abrupt completion resolution as an Abort node with outgoing flow edge.

Introduce abrupt completion resolution

For each abrupt completion resolution rule that is generated from a specification, one instance of abrupt completion resolution in the specification is selected. On this instance, the

Figure 6.14: The meta-rule that removes everything from specifications that do not feature abrupt completion resolution.

ResolveAbortIntro meta-rule is applied (Figure 6.15). From thereon, the other rules in this set can be applied in a linear fashion.

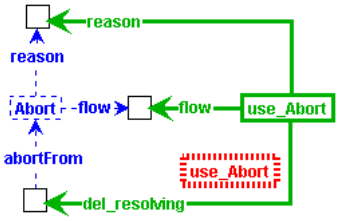
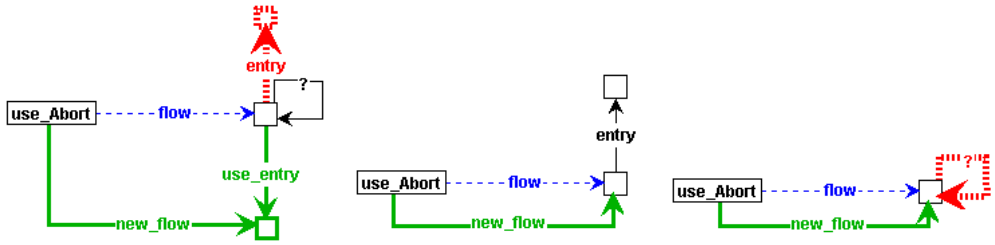


Figure 6.15: The meta-rule that introduces abrupt completion resolution.

The meta-rules in Figure 6.16 handle the resolution of an Abort, by introducing the creation of a flow edge to the element to which the Abort node in the specification featured a flow edge.

The three meta-rules in Figure 6.16 treat the case in which the target of the flow edge has no entry in the specification, does have some entry or is an exit node.



(a) Abort - flow to entry (b) Abort - flow not to entry (c) Abort - flow to exit node

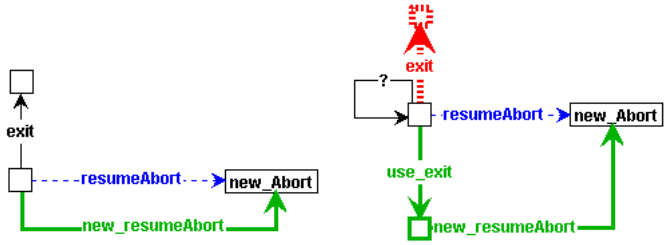
Figure 6.16: The meta-rules for introducing the creation of flow edges for resolving abrupt completion.

Introduce abrupt completion resumption

We have abrupt completion resumption when abrupt completion, because of some reason, is resolved and, next, abrupt completion because of the same reason is introduced (in specification, we denote this resuming using a resumeAbort edge).

This set of meta-rules also handles the case that abrupt completion resumes after resolution. The meta-rule in Figure 6.18 introduces the creation of the new abrupt completion for the same old reason. The meta-rules in Figure 6.17 treat the resumeAbort edge (coming

from a syntax node or the exit of the syntax node) that are present in the specification and are introduced to the flow graph by the resulting production rule.



(a) resumeAbort - exit (b) resumeAbort - no exit

Figure 6.17: The meta-rules for introducing the creation of abort edges for resuming abrupt completion.

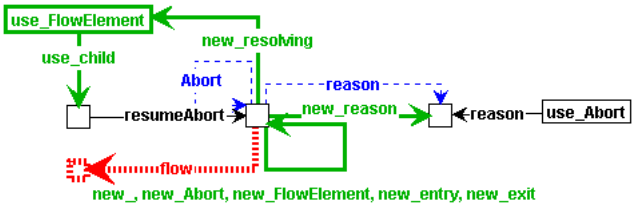


Figure 6.18: The meta-rule that introduces the resuming of abrupt completion.

Remove flow graph construction elements

In the abrupt completion resolution rules, nothing related to the top-down flow graph construction process may persist.

The meta-rule in Figure 6.19 deletes all nodes it can safely remove from a specification that features abrupt completion resolution. Care must be taken that this rule does not remove elements relevant to the resolution or resumption of abrupt completion.

The meta-rule in Figure 6.20 removes all sequential flow from the specifications. Figure 6.21 features meta-rules for taking care of the, in this case, unwanted entry and exit edges. Finally, Figure 6.22 features a rule for removing the specification-related KeyElement edge.

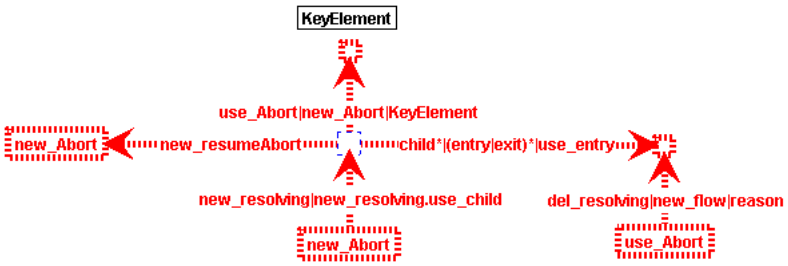


Figure 6.19: The meta-rule that removes everything that is not needed for abrupt completion resolution from the specifications.

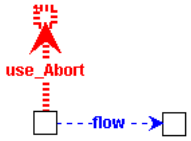
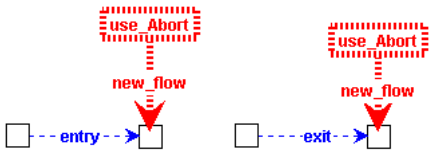


Figure 6.20: The meta-rule that removes flow edges from the specifications.



(a) Remove entry (b) Remove exit

Figure 6.21: The meta-rules for deleting entry and exit edges from the specifications.

Figure 6.22: The meta-rule that removes the KeyElement edge from the specifications.

Chapter 7

Evaluation of the Framework

We evaluate our control flow semantics framework in two ways. First we evaluate the control flow semantics framework on its applicability to other programming languages than our example language: Java. We identify different and unconventional constructs that are featured in existing programming languages and describe ways of specifying them in CFSL along with any encountered problems. Next we list limitations of the research itself. Due to the time-constraints that come with a master thesis not all possible and relevant research topics have been explored. We list what research could have been beneficial to our framework when more time had been available. These suggestions are also part of the relevant further work we indicate in Chapter 8.

7.1 Applicability of the framework

Although we in Chapter 1 claim to introduce a *generic*, i.e. programming language independent, control flow semantics framework, all examples provided in this thesis come from the Java programming language. Therefore we think it is useful to evaluate the applicability of our framework to other programming languages. We of course cannot *prove* that our framework can be applied to any programming language based on imperative constructs currently existing or yet to come, but we can at least show that it can be applied to most constructs that are featured in current programming languages.

Instead of examining a number of selected programming constructs, we could have worked out a set of control flow specifications in CFSL for another programming language (e.g. Pascal). But as most programming languages feature the same or very similar set of programming constructs, most control flow specifications would be very or completely similar to the specifications presented for Java. We therefore focus on constructs that really differ on their control flow semantics.

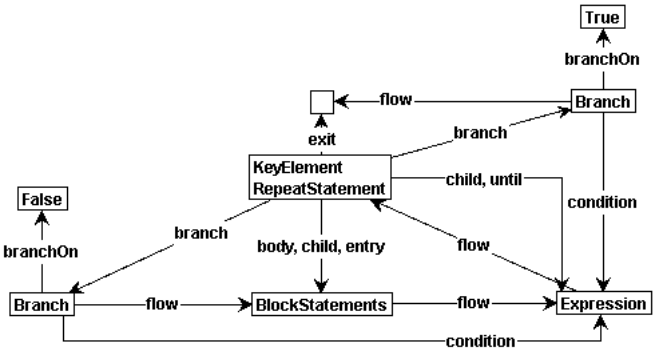
7.1.1 Conventional constructs

We first examine some conventional programming constructs that are featured in popular programming languages and are actually used to some degree. The control flow of these constructs turns out to be quite simple to specify in CFSL.

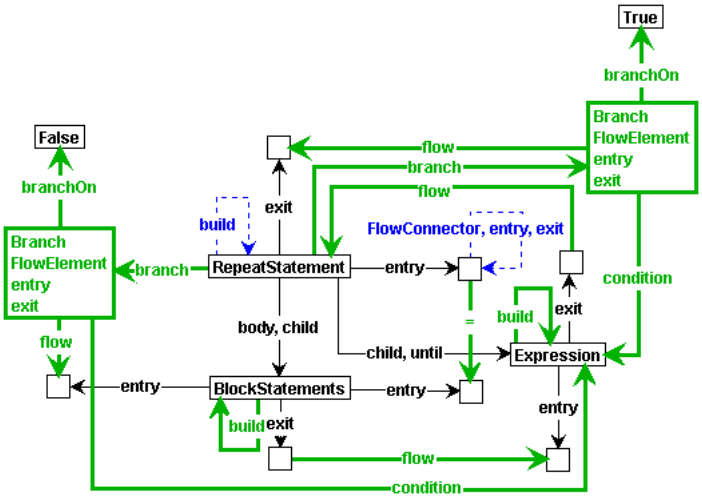
Repeat-statement

Pascal features a loop-statement that iterates until its condition evaluates to `true` (where in Java loops are iterated as long as their condition holds): the repeat-statement. The

repeat-statement is actually a do-statement with an inverted condition. Figure 7.1.1 shows a specification in CFSL and the associated flow graph construction rule. Note that it features a conditional branch path leading to the exit of the repeat-statement, when the condition evaluates to `true`.



(a) Specification

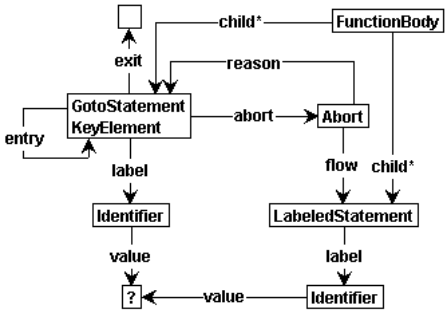


(b) Rule

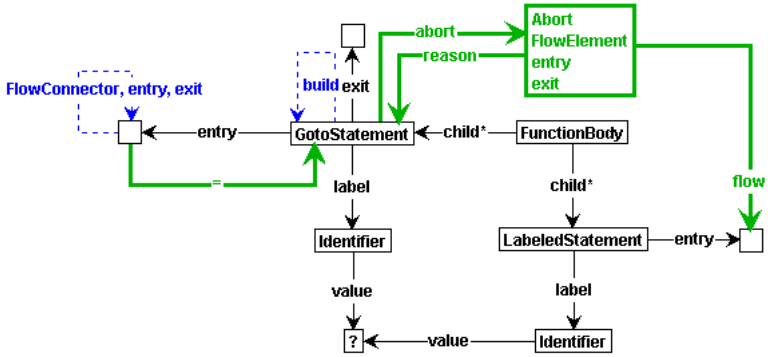
Figure 7.1: The repeat-statement in Pascal.

Goto-statement

Many programming languages (e.g. Pascal, C, BASIC) feature the primitive, unconditional jump-statement known as `goto`. A goto-statement immediately transfers control to a labeled statement that can be anywhere in the same method (or function) body. This control transfer may thereby abruptly complete any number of enclosing statements that were currently executing (e.g. loop-statements). It is therefore logical to represent the control flow jump of a goto-statement with abrupt completion that features immediate resolution (as the abrupt completion target is known due to the label). Figure 7.1.1 shows both the specification and the rule for a goto-statement (in C).



(a) Specification



(b) Rule

Figure 7.2: The goto-statement in C.

7.1.2 Exotic constructs

Now that we have treated some commonly used statements in popular programming languages we examine some more exotic and, as a result, far less often used statements, to see what the limitations of CFSL are. These statements are mostly more obscure variants on the well known unconditional jump (`goto`). As the goto-statement is considered to increase the complexity of code and therefore could be considered "harmful" [4], these statements are even more harmful.

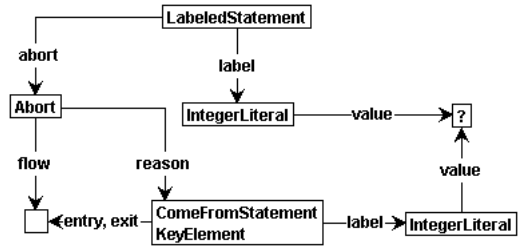
Come-from-statement

The come-from-statement [2] is roughly the opposite of a goto-statement. It features a label that is identical to the label of some other statement within the operation. But instead of introducing an abrupt jump to that identically labeled statement, it introduces a jump at the moment the labeled statement *completes* to the statement sequentially ordered to execute after the come-from-statement. When sequential execution reaches the come-from-statement itself, the come-from-statement is simply skipped.

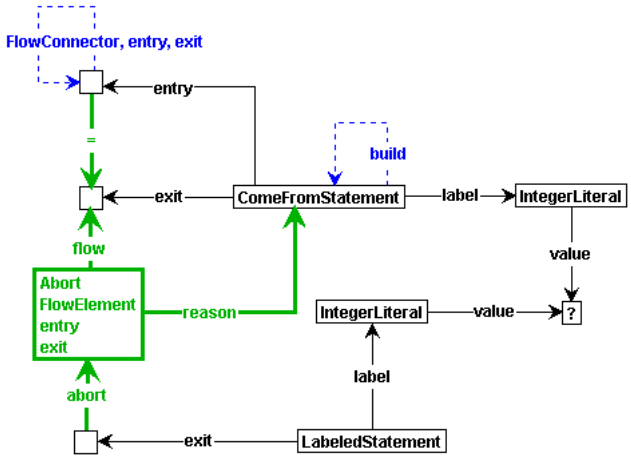
Although this statement was introduced by the author of [2] as a joke, it has been implemented in C-INTERCAL and some version of FORTRAN.

In Figure 7.1.2 we show a CFSL specification and corresponding flow graph construction rule for the `COMEFROM`. Note the (counter-intuitive) jump they specify/introduce from the exit

of the labeled statement to the exit of the come-from-statement.



(a) Specification



(b) Rule

Figure 7.3: The comefrom-statement, roughly the opposite of the infamous goto-statement.

Conditional come-from-statement

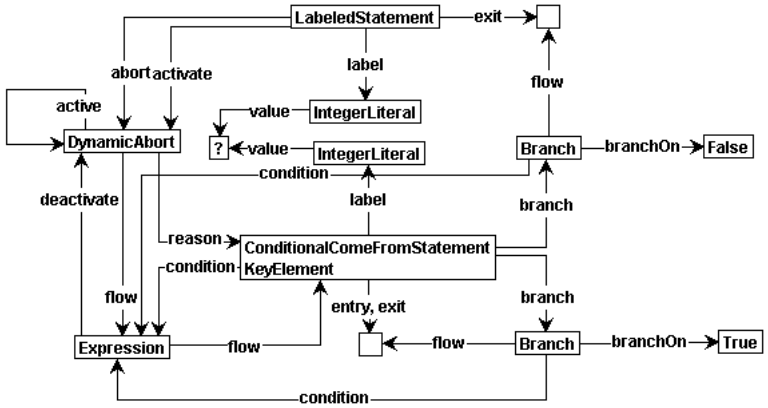
An even worse variant of the come-from-statement introduced above is the conditional come-from-statement. This statement introduces an abrupt jump after completion of the identically labeled statement if the associated condition holds, else control is transferred back to the exit of the labeled statement.

The conditional come-from-statement introduces a problem: from the exit of the conditional statement we jump to the Expression of the ComeFromStatement, but after evaluating this expression, one possible Branch leads back to the exit of the labeled statement to resume sequential execution. Because of the fact that abrupt completion control flow overrules sequential control flow, the jump to the Expression will be made again and again.

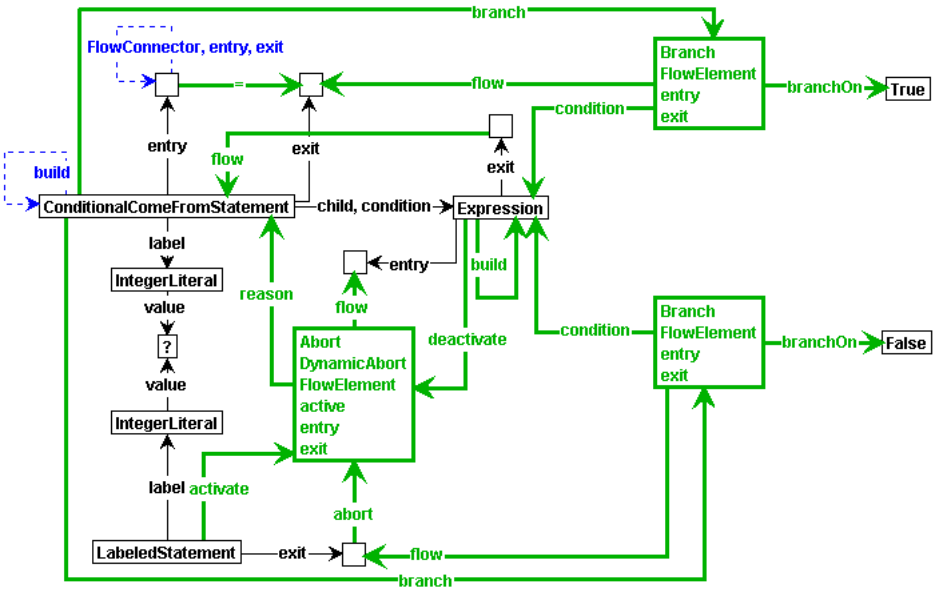
A solution is to specialize the auxiliary Abort as a DynamicAbort. The DynamicAbort is an Abort node that features a *status*: it is either active or notActive. If the DynamicAbort is notActive, the sequential control flow is followed, else the abrupt completion occurs. With two additional edges labeled activate and deactive we can specify that when the flow of control reaches a certain flow element, the status of the DynamicAbort is changed.

Figure 7.1.2 shows a CFSL specification and corresponding flow graph construction rule

of the conditional come-from-statement that use this solution.



(a) Specification



(b) Rule

Figure 7.4: The conditional comefrom-statement solved with a dynamic abort node.

This auxiliary DynamicAbort node is interesting and could perhaps be included in the framework. Example usage scenarios include catch-clauses, were we could model the possible resumption of abrupt completion when the thrown exception is not caught at run-time by any of the clauses with a DynamicAbort that activated by default and is deactivated when entering one of the clauses's bodies. Another possible use is to introduce DynamicAborts to every method invocation, deactivated by default, to indicate were control is transferred in case the invocation is terminated because of an uncaught exception. Unfortunately, as we have not fully explored run-time behavior and its simulation (see Section 7.2), we cannot (yet) safely make the choice whether to include this DynamicAbort.

Arithmetic If

The arithmetic if-statement that features FORTRAN consists of an integer expression and three labels. Depending on whether the expression evaluates to a value greater than, equal or less than zero, one of the labels is chosen to `goto`.

This statement should be handled by three Branches with more advanced and complex branchOns (possible by applying attributed graph grammars [11]), each leading to one of three goto-like constructions.

Computed goto-statement

The computed goto-statement that features FORTRAN can be seen as some kind of primitive switch-statement. It consists of an integer expression that results in an index to a list of labels. The selected label is used to `goto`. If the index exceeds the length of the label list, the statement sequentially ordered to be executed after the computed goto-statement is executed.

To handle this statement we should introduce a Branch for each label, with a branchOn the corresponding index literal value.

Gosub-statement

Old versions of the programming language BASIC feature the `GOSUB` / `RETURN` statement, some sort of primitive method invocation mechanism.

The `GOSUB` part behaves exactly like an ordinary goto-statement, introducing an abrupt jump to some identically labeled statement. The `RETURN` part makes a control flow jump back to the most recently executed `GOSUB` statement. Which statement this is can of course only be determined at runtime, therefore our framework does not (yet) support this statement.

Longjump-statement

The `longjmp` statement in C introduces a non-local jump (possible through several nested method calls) to the corresponding `setjmp` statement. This `setjmp` stores information on the execution environment in a buffer when it is reached by a sequential transfer of control and restores the environment when it is reached through the non-local (long) jump.

This statement is used in C as an (primitive) exception handling mechanism, but is not supported by our framework.

7.2 Limitations of the research

The framework presented in this thesis lacks elaboration on some fronts. Due to time constraints some topics could not be explored and in some cases the scope of the research was narrowed to fit in the tight schedule that is inherent to a master thesis. We list research options that would have been beneficial to the framework had time allowed for them.

During the design of our flow graph construction approach and our control flow specification language we have focused on programming constructs that occur inside method or function bodies. Method invocation, object creation and initialization have only been explored to a limited extent. Concurrency has not been explored at all. Given more time, these topics could have been explored in more detail, making the research more complete.

As a proof of concept, we then would have been able to present a complete specification of the control flow semantics of all Java constructs.

To the extent to which it is possible, we solve control flow statically in flow graphs or specifications. Some constructs however have partially or completely runtime control flow behavior. Examples include method invocation and exception handling. In these cases, we indicate possible control flow paths that can be followed at run-time. Given more time, we could have created a simulation system for run-time control flow behavior, as is done in [10]. Such a simulation system is (again) a graph transformation system consisting of a set of control flow simulation rules and a start flow graph. Each simulation rule specifies the execution semantics of the corresponding statement and performs a run-time transfer of control (sequential, branching or abrupt completion) by moving a "program counter" to the construct to be executed next. These simulations could have provided more insight in what kind of information (related to control flow determined at run-time) we have to provide in control flow specifications or control flow graphs in order to accommodate for control flow simulations.

On a more detailed level, it is useful to review one of the design choices of our flow graph construction approach. This choice involves the uniform introduction of flow connectors before the actual flow graph construction process commences. This choice has had several advantages, making flow graph construction and flow graph construction rule generation (through the use of meta-rules) more straightforward.

It is not without its disadvantages, however. Many flow graph construction rules contain flow connector merging operations and, due to the fact that we have chosen to limit the possible merging scenarios to four (see Section 4.1.3), a constructed flow graph contains quite a few flow connectors.

An alternative option would be to introduce flow connectors on the fly (i.e. during construction, only were deemed necessary). Our reason for choosing the uniform approach is that on the fly introduction seemed to complicate the flow graph construction process and would introduce the need to discern too many different cases (e.g. introduce a flow connector for a particular programming construct only when it is placed in a particular context). Still, it remains worthwhile to explore whether there is a more elegant alternative to the uniform introduction of flow connectors.

Chapter 8

Conclusion

In this thesis we aimed at designing a generic specification language for the control flow semantics of programming languages. We presented the graph-based specification language CFSL. A language designer can use CFSL to specify the control flow semantics of the programming language he or she is designing. This results in a set of control flow specifications, one for each programming construct that is featured in the programming language. These control flow specification graphs conform to the CFSL meta-model we presented (and can be checked on conformance with respect to the additional constraints we impose).

We also aimed at developing a structured approach for constructing flow graphs for programs. Our presented approach consists of introducing control flow information to an abstract syntax graph representation of a program's source code. We presented meta-models for both our abstract syntax graphs and our flow graphs. For each construct in a programming language, our flow graph construction system features one (or several) flow graph construction rule(s). Our flow graph construction process operates top-down in most cases. We made an exception for abrupt completion control flow: this type of control flow we solve in a bottom-up way.

Last, we completed our control flow semantics framework by introducing flow graph meta-rules, with which we can generate from specifications in CFSL a set of corresponding flow graph construction rules.

With this we believe to have introduced a valuable control flow specification and construction framework that is *generic* to the extent we think is desirable (see Chapter 7), although more (case) studies on other programming languages will have to be performed to further consolidate this belief.

The next section compares our research to related work. The last section summarizes further research that can be performed to extend our framework.

8.1 Related work

In this section, we give a brief overview of related research. We first discuss research that is (almost) directly comparable to our work. Next we treat the concept of flow graphs and how they are typically used for complexity analysis. Last we discuss graph transformations as a transformation technique with a broad range of application areas.

8.1.1 Research with comparable content

The research that is closely related to our work is that of [10]. In this report, Rensink et al. present the complete path from parsing a program to simulating its execution for a

custom developed programming language called TAAL. The TAAL language is an object-oriented language which features the basic imperative constructs and inheritance, but when compared to, for example, Java, lacks abrupt completion and exception handling constructs. The authors apply graph transformation for flow graph construction and program execution simulation. For flow graph construction, they have designed a set of flow graph construction rules by hand, one for each TAAL construct. This work has been the starting point for the current research project.

Another example of closely related work is [3]. In [3] the authors introduce a translation mechanism from Java programs to graph transformation system for execution simulation of these programs. Among others, their research differs from this research in that it is specific to Java and that, for a portion of Java, they introduce translation schemas that convert Java source code *directly* into a simulation system.

Their simulation systems consists of basic simulation rules applicable to all Java programs (constructs that can occur inside a method body) and program-specific rules (for each method / constructor in a specific Java program they create a rule that replaces a call to a method with the graph representation of that method's body). The data and control information during simulation are represented as hyperedges with labeled tentacles (connectors of the hyperedge). They enforce sequential execution by propagating a program counter (a global GO hyperedge) between the in and out (compare to our entry and exit).

Currently, their model does not support simulation of loop-statements (or abrupt completion statements), because a control hyperedge is deleted on execution by its rule (each statement can therefore be executed once). However, using recursion loops can be introduced (as each method invocation rule introduces a copy of the method body graph).

In [8] Gurevich and Huggins use a form of control flow graphs to describe the transfer of control (to which they refer as changing the current "task") that C statements cause, as part of a complete specification of the semantics of the C programming language.

Although not directly comparable to our work, the work on Montages is also worth noting. In [1] an extensive framework is introduced for aiding a language designer when specifying the syntax and the (static and dynamic) semantics of a programming language. The authors share many design principles and choices with us. They too compose a complete programming language specification from a set of specifications, one for each individual programming construct, and, as with ours, their specifications have a very close relation to the EBNF grammar. Their control flow graph are also based on abstract syntax trees to which control flow information is introduced per statement.

The main difference is that they do not use graphs and graph transformations for flow graph construction but local finite state machines that decorate an abstract syntax tree and are connected to each other later. A specification of a programming language construct in [1] is called a *Montage* and consist of three parts: a plain BNF syntax rule, a local finite state machine (FSM) specified in the Montage Visual Language (MVL) and several action rules associated to nodes in the local state machine. Figure 8.1 shows an example Montage for a referenced variable (for the toy language presented in [1]).

A local FSM of a programming construct in MVL consists of two types of nodes: nodes that refer to AST child nodes of the construct and nodes that represent a state corresponding to the construct (to which an action rule may be associated). By decorating each node in an AST with the corresponding local FSM a global, hierarchical, FSM is obtained. Each local FSM features a *initial* (I, compare to entry) node and a *terminal* (T, compare to exit) node. By merging these nodes between local FSM's, the hierarchical global FSM can be flattened.

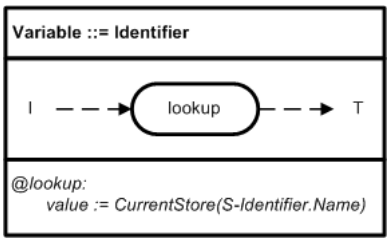


Figure 8.1: An example Montages for a referenced variable.

8.1.2 The concept of flow graphs

Flow graphs have been widely used as an abstracted representation of code for measuring (cyclomatic) complexity. Examples include the work of Fenton and Pfleeger on Software Metrics [5]. In [20], Tang, Dogru, Kurfess and Tanik introduce Cubic Flow Graphs (graphs in which each node has 3 edges incident to the node) and show how the properties of these graphs relate to the cyclomatic complexity of the represented program. They prove that the cyclomatic complexity of a program is equal to the number of decision nodes in its cubic flow graph plus one and equal to the number of prime flow graphs into which the cubic flow graph can be decomposed.

8.1.3 The graph transformation technique

Graph transformations are a transformation technique with many possible application areas within Computer Science [17, 15]. Many extensions have been proposed to the basic graph transformation technique (e.g. the negative application conditions we use) in order to increase its expressiveness and thereby the applicability of the technique to areas as modeling, model transformation, simulation, verification, constraint checking, etc.

In [11], Kastenberg gives an overview on how to introduce attributed graphs in the GROOVE tool (the graph transformation tool we use). This will likely further increase the expressiveness of graphs as a way of modeling, among others, control flow semantics.

In [18], Schürr argues that the common approach of having one-to-one relationships between elements in the source graph (e.g. an abstract syntax graph) and the target graph (e.g. a control flow graph) has it shortcomings. He introduces *correspondence graphs* to accomodate for *m-to-n* relationships between elements in source and target graphs. His Triple Graph Grammars thus consist of tuples of a left hand side graph, a correspondence graph and a right hand side graph.

In the context of MDA [13], graph transformations are being proposed as a suitable formal technique for model transformations by several authors. Examples include [7], in which the authors make this point by presenting a case study on model transformations using graph transformations for UML Statecharts.

8.2 Future work

Although we think this thesis introduces an interesting framework for the specification of control flow semantics of programming languages, there is still much work left that can be

done to improve it. We have already described in detail some limitations of the framework in Chapter 7. Summarizing, the three main research topics that we think should be explored in order to improve the framework are:

1. A structured approach for simulating program execution using graphs and graph transformations;
2. A structured approach for abstracting the syntax of programming languages;
3. Extensions of the control flow semantics framework for the specification of other kinds of semantics of programming languages.

As mentioned in Chapter 7, we have not explored the simulation of the dynamic control flow behaviour of programs. We believe that the development of a structured approach for program simulation will provide additional insights in the dynamic control flow semantics of programming languages and that the resulting simulation approach will be a valuable addition to the framework.

Both specifications in CFSL and flow graphs are based on an abstract syntax graph representation. Although we have provided a meta-model and several guidelines and constraints for abstracting programming language syntax, we have left some to (ad hoc) choices to be made by a language designer. Examples include the choice between inheriting and composing right hand side non-terminals in our graphs. A completely structured approach for abstracting the concrete EBNF syntax of programming languages in graphs (see Section 3.1) is in our view desirable. An additional benefit of structuring the syntax abstraction is that the grammar conversion can then be automated.

As mentioned in Chapter 1, we have focused on the *control flow* semantics of programming languages and presented the control flow specification language CFSL. Again related to MDA, a specification language for all semantics of programming languages is required for correct model transformations. With some extensions to our graph models (for example attributed graph grammars [11]) we believe graphs and graph transformations should be powerful enough to serve as the basis of such a semantics specification language. Related research here is the GRASLAND project [16], which strives to develop a Language Definition Language for specification of the semantics of all software languages, including software specification languages (for example as are part of UML).

Bibliography

[1] Anlauff, M., Kutter, P.W., Pierantonio, A., *Enhanced Control Flow Graphs in Montages*, Lecture Notes In Computer Science, Vol. 1755, 1999.

[2] Clark, R.L., *A Linguistic Contribution to GOTO-less Programming*, Communications of ACM, 27, pp. 349-350, 1984

[3] Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L., *Translating Java Code to Graph Transformation Dystems*, Second International Conference on Graph Transformation, Lecture Notes in Computer Science, vol. 3256, Springer-Verlag, 2004.

[4] Dijkstra, E.W., *Go To Statement Considered Harmful*, `http://www.acm.org/classics/oct95/`, 1968

[5] Fenton, N.E., Pfleeger, S.L., *Software Metrics: a rigorous & practical approach*, 2nd edition, International Thomson Computer Press, London, United Kingdom, 1997.

[6] Gosling, J., Joy, B., Steele, G., Bracha, G., *The Java Language Specification*, 3rd edition, Addison-Wesley, Boston, 2005.

[7] Grunske, L., Geiger, L., Zündorf, A., Van Eetvelde, N., Van Gorp, P., Varró, D., *Using Graph Transformation for Practical Model Driven Software Engineering*, Model-driven Software Development - Volume II of Research and Practice in Software Engineering, July, 2005.

[8] Gurevich, Y., Huggins, J.K., *The Semantics of the C Programming Language*, Lecture Notes in Computer Science, vol. 702, Springer, 1993.

[9] International Organisation for Standardization, *ISO/IEC 14977: The EBNF Standard*, 1996.

[10] Kastenberg, H., Kleppe, A., Rensink, A., *Engineering Object-oriented Semantics using Graph Transformations*, Universiteit Twente, Enschede, 2005.

[11] Kastenberg, H., *Towards Attributed Graphs in Groove*, Workshop on Graph Transformation for Verification and Concurrency, CTIT Technical Report 05-34, Department of Computer Science, University of Twente, 2005

[12] Kleppe, A., Warmer, J., Bast, W., *MDA Explained, the model driven architecture: practice and promise*, Pearson Education, Boston, 2004.

[13] Object Management Group, *Model Driven Architecture*, `http://www.omg.org/mda`.

[14] Rensink, A., Kastenberg, H., *GRaphs for Object-Oriented VErification (GROOVE)*, `http://groove.sf.net`, 2005

[15] Rensink, A., *The joys of Graph Transformations*, Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica, 9, 2005.

[16] Rensink, A., *Graphs for Software Language Definitions*, `http://trese.cs.utwente.nl/grasland`, 2004.

[17] Rozenberg, G., *Handbook of Graph Grammars and Computing by Graph Transformation, volume A: Foundations*, World Scientific, Singapore, 1997.

[18] Schürr, A., *Specification of Graph Translators with Triple Graph Grammars*, IEEE Transaction on Software Engineering, p. 151-163, 1994.

[19] Sun Microsystems, *Java Technology*, `http://java.sun.com`, 2005

[20] Tang, Y., Dogru, A.H., Kurfess, F.J., Tanik, M.M., *Computing Cyclomatic Complexity with Cubic Flowgraphs*, Journal of Systems Integration, 10, p. 395-409, Kluwer Academic Publishers, The Netherlands, 2001.

[21] Watt, D.A., Brown, D.F., *Programming Language Processors in Java: compilers and interpreters*, Pearson Education, Harlow, United Kingdom, 2000.

Appendix A

Abstract Java Grammar

This appendix presents our abstracted BNF grammar for a large portion of the Java programming language [19]. This grammar is based on the EBNF syntax grammar presented in [6]. Chapter 3 describes the abstractions we made to the original grammar and the relation between this abstract grammar and the abstract syntax graph representations in this thesis.

Listing A.1: Abstract Java BNF grammar

```
MethodBody ::=
        Block

Block ::=
        BlockFull | BlockEmpty

BlockFull ::=
        <LCUR> orderFirst:BlockStatements <RCUR>

BlockEmpty ::=
        <LCUR> <RCUR>

BlockStatements ::=
        BlockStatementsNext | BlockStatementsLast

BlockStatementsNext ::=
        Statement orderNext:BlockStatements

BlockStatementLast ::=
        Statement

Statement ::= Block | IfStatement | ForStatement | WhileStatement | DoStatement | TryStatement | SwitchStatement |
      BreakStatement | ContinueStatement | ThrowStatement | ReturnStatement | LabelledStatement | EmptyStatement |
      StatementExpression | LocalVariableDeclarationStatement

IfStatement ::=
        IfThen | IfThenElse

IfThen ::=
        <IF> <LPAR> condition:Expression <RPAR> thenPart:Statement

IfThenElse ::=
        <IF> <LPAR> condition:Expression <RPAR> thenPart:Statement <ELSE> elsePart:Statement

ForStatement ::= ForEver | ForWithInit | ForWithInitCondition | ForWithInitUpdate | ForWithInitConditionUpdate |
      ForWithCondition | ForWithConditionUpdate | ForWithUpdate

ForEver ::=
        <FOR> <LPAR> <SC> <SC> <RPAR> body=Statement

ForWithInit ::=
        <FOR> <LPAR> init:ForInit <SC> <SC> <RPAR> body:Statement

ForWithInitCondition ::=
        <FOR> <LPAR> init:ForInit <SC> condition:Expression <SC> <LPAR> body:Statement

ForWithInitConditionUpdate ::=
        <FOR> <LPAR> init:ForInit <SC> condition:Expression <SC> update:ForUpdate <RPAR> body:Statement

ForWithCondition ::=
        <FOR> <LPAR> <SC> condition:Expression <SC> <RPAR> body:Statement

ForWithConditionUpdate ::=
        <FOR> <LPAR> <SC> condition:Expression <SC> update:ForUpdate <RPAR> body:Statement

ForWithUpdate ::=
        <FOR> <LPAR> <SC> <SC> update:ForUpdate <RPAR> body:Statement

ForInit ::=
```

```
        Expression Expressions

ForUpdate ::=
        StatementExpressionList

LabelledStatement ::=
        label:Identifier <C> Statement

EmptyStatement ::=
        <SC>

WhileStatement ::=
        <WHILE> <LPAR> condition:Expression <RPAR> body:Statement

DoStatement ::=
        <DO> body:Statement <WHILE> <LPAR> condition:Expression <RPAR> <SC>

TryStatement ::=
        TryCatch | TryFinally | TryCatchFinally

TryCatch ::=
        <TRY> body:Block catches:Catches

TryFinally ::=
        <TRY> body:Block finally:FinallyStatement

TryCatchFinally ::=
        <TRY> body:Block catches:Catches finally:FinallyStatement

FinallyStatement ::=
        <FINALLY> body:Block

BreakStatement ::=
        BreakStatementWithoutLabel | BreakStatementWithLabel

BreakStatementWithoutLabel ::=
        <BREAK> <SC>

BreakStatementWithLabel ::=
        <BREAK> label:Identifier <SC>

ContinueStatement ::=
        ContinueStatementWithoutLabel | ContinueStatementWithLabel

ContinueStatementWithoutLabel ::=
        <CONTINUE> <SC>

ContinueStatementWithLabel ::=
        <CONTINUE> label:Identifier <SC>

ReturnStatement ::=
        ReturnStatementWithoutValue | ReturnStatementWithValue

ReturnStatementWithoutValue ::=
        <RETURN> <SC>

ReturnStatementWithValue ::=
        <RETURN> returnValue:Expression <SC>

ThrowStatement ::=
        <THROW> exception:Expression <SC>

Catches ::=
        orderFirst:CatchClauses

CatchClauses ::=
        CatchClausesNext | CatchClausesLast

CatchClausesNext ::=
        CatchClause orderNext:CatchClauses

CatchClausesLast ::=
        CatchClause

CatchClause ::=
        <CATCH> <LPAR> type:FormalParameter <RPAR> body:Block

SwitchStatement ::=
        <SWITCH> <LPAR> condition:Expression <RPAR> <LCUR> block:SwitchBlock <RCUR>

SwitchBlock ::=
        orderFirst:SwitchBlockStatementGroups

SwitchBlockStatementGroups ::=
        SwitchBlockStatementGroupsNext | SwitchBlockStatementGroupsLast

SwitchBlockStatementGroupsNext ::=
        SwitchBlockStatementGroup orderNext:SwitchBlockStatementGroups

SwitchBlockStatementGroupsLast ::=
```

```
        SwitchBlockStatementGroup

SwitchBlockStatementGroup ::=
        SwitchLabels block:BlockStatements

SwitchLabels ::=
        SwitchLabel SwitchLabels | SwitchLabel

SwitchLabel ::=
        SwitchLabelNotDefault | SwitchLabelDefault

Assignment ::=
        left:ExpressionName operator:AssignmentOperator right:AssignmentExpression

AssignmentExpression ::=
        Expression

LocalVariableDeclarationStatement ::=
        mods:VariableModifiers type:Type orderFirst:VariableDeclarators

VariableDeclarators ::=
        VariableDeclaratorsNext | VariableDeclaratorsLast

VariableDeclaratorsNext ::=
        VariableDeclarator orderNext:VariableDeclarators

VariableDeclaratorsLast ::=
        VariableDeclarator

VariableDeclarator ::=
        VariableDeclaratorWithInit | VariableDeclaratorWithoutInit

VariableDeclaratorWithInit ::=
        name:Identifier <BASSIGN> init:Expression

VariableDeclaratorWithoutInit ::=
        name:Identifier

MethodInvocation ::=
        MethodInvocationWithArguments | MethodInvocationWithoutArguments

MethodInvocationWithArguments ::=
        name:MethodName args:ArgumentList

MethodInvocationWithoutArguments ::=
        name:MethodName

MethodName ::=
        ExpressionName | StaticMethodName | PrimaryExpressionMethodName | SuperMethodName | ClassNameSuperMethodName

ArgumentList ::=
        orderFirst:Arguments

Arguments ::=
        ArgumentsNext | ArgumentsLast

ArgumentsNext ::=
        Expression orderNext:Arguments

ArgumentsLast ::=
        Expression

Expression ::=
        Literal | ExpressionName | AdditiveExpression | RelationalExpression | Assignment | MethodInvocation

ExpressionName ::=
        Identifier

AdditiveExpression ::=
        left:Expression operator:AdditiveOperator right:Expression

RelationalExpression ::=
        left:Expression operator:RelationalOperator right:Expression

RelationalOperator ::=
        <LESSTHAN> | <LESSTHANEQUAL> | <GREATERTHANEQUAL> | <GREATERTHAN>

AdditiveOperator ::=
        <ADD> | <SUBSTRACT>

StatementExpression ::=
        Assignment | MethodInvocation | ..

Literal ::=
        IntegerLiteral | BooleanLiteral | StringLiteral | FloatingPointLiteral | CharacterLiteral | NullLiteral
```