Specification and Construction of Control Flow Semantics

Ruben Smelik, Arend Rensink and Harmen Kastenberg*
Department of Computer Science
University of Twente, The Netherlands
{smelik,rensink,h.kastenberg}@cs.utwente.nl

Abstract

In this paper we propose a visual language CFSL *for specifying control flow semantics of programming languages. We also present a translation from* CFSL *to graph production systems (*GPS*) for flow graph construction; that is, any* CFSL *specification, say for a language L, gives rise to a* GPS *that constructs from any L-program (represented as an abstract syntax graph) the corresponding flow graph. The specification language is rich enough to capture complex language constructs, including all of Java.*

1. Introduction

There is no commonly agreed language for specifying the semantics of programming languages, on par with EBNF for the syntax (see [7]). Instead, programming language semantics is typically only described in natural language; a good example is Java, see [4]. This is a problem for several reasons: if the semantics is non-trivial, such as for instance in case of the control flow of the Java `try` statement (with `finally` clause), it is hard to give a non-ambiguous natural language explanation; and also, in the absence of a formal specification there is no basis for compiler certification, correctness-preserving transformations (as in MDA, see [9]) or software verification.

On the other hand, a problem with formal specifications is that they are notoriously hard to read, because their language is too far removed from the expertise and experience of the average software engineer. Since in this case we want the programming language designers to be writers of the specification and the programmers to be among its readers, it is crucial to remove this barrier. This entails designing a specification language for programming language semantics that is accessible to these classes of users.

*The author is employed in the GROOVE project funded by the Dutch NWO (project number 612.000.314).

In this paper we restrict ourselves to *control flow* semantics — which is an important part of the semantics of any programming language in the imperative paradigm — and we propose a visual language for its specification, which we simply call CFSL for Control Flow Specification Language. Characteristics of CFSL are:

- It is *modular* in terms of the programming constructs in the language under design; that is, the control flow semantics of each statement type is defined in a separate *control flow specification graph* (CFSG); a CFSL specification is a set of CFSGs.
- It builds upon the *syntax* of the language: each CFSG is created from the right hand side of a Backus-Naur Form (BNF) grammar rule (or in other words, a fragment of an abstract syntax tree) for the programming language in question.
- It is *visual*, in that each CFSG is a graph with the abstract syntax graph (ASG) fragment as its core, and further structure declaring the corresponding control flow.
- Its own semantics (i.e., of CFSL) is defined in terms of *transformations* from ASGs to flow graphs (FGs). These transformations are defined in terms of *graph production systems*.
- The CFSL semantics *definition* is itself also formulated through a graph production system, which transforms every CFSG into a set of graph production rules implementing the effect of that CFSG.

The basis of this paper is the master's thesis [14], in which we give a CFSL specification of Java, thereby validating its usability. Here we discuss the Java `while` and `try` statement (with `finally` clause), for other statements we refer to [14]. The two transformation steps (from CFSGs to their semantics and from ASGs to FGs) have been implemented using the graph transformation tool GROOVE (see [10, 11]).

Figure 1 shows how we envisage the language described in this paper to be used in practice, in terms of models, relations between them and actors involved. We discern three

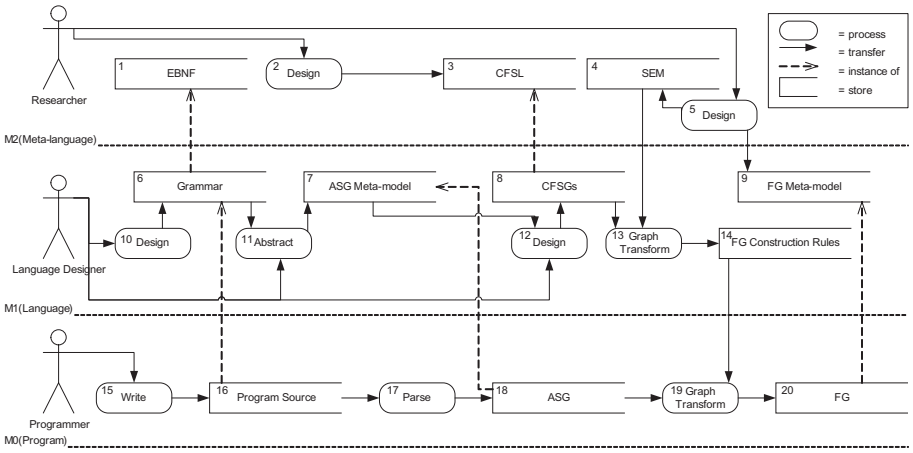

Figure 1: Overview of the elements involved in this research and the levels on which they reside.

levels. The lowest level (M0) is the level of programs written in a particular programming language (note that in another context programs equate to models); the middle level (M1) is the level of programming languages, and the uppermost level (M2) is the level of meta-languages in which (parts of) programming languages are defined. Each level is dedicated to a certain role: the *researcher* who has defined the meta-languages, the *language designer* who defines programming language syntax and semantics in those meta-languages, and a *programmer* who writes programs in those specific programming languages.

The language designer specifies the syntax of a new programming language (6 in Figure 1) in BNF (1). This grammar is abstracted to a meta-model (7) for ASGs. Next, he specifies the control flow semantics of the language in CFSL (3). By applying (13) the set language-independent FG meta-rules (which we have named SEM) (4), he obtains the CFSL semantics in the form of a set of language-specific FG construction rules (14).

The programmer writes (15) some program (16) in this new language, conforming to the grammar (6). The program is parsed (17), resulting in an ASG (18) that conforms to the ASG meta-model (7). The ASG is transformed by the previously generated set of FG construction rules (14) into a FG (20) that conforms to the FG meta-model (9).

The status of our research is that we have elaborated the necessary concepts and transformations, but the concrete graph syntax of CFSL has not yet been optimized for readability.

In the remainder we present the highlights of the approach; for a complete, detailed discussion see the thesis [14]. Section 2 describes the CFSL (3), and Section 3 the transformations that define the CFSL semantics (14). Section 4 briefly discusses the construction of the CFSL semantics (4). Finally, Section 5 gives conclusions and discusses related work.

2. Control Flow Specification Language

In the introduction we mentioned several important characteristics of CFSL; CFSL is a modular, visual, graph-based specification language that builds upon programming language syntax.

In this paper, a *graph* consists of a finite set of *nodes* and a finite set of labelled, directed, binary *edges*. Nodes are not labelled by definition, but can have outgoing edges pointing to itself. These edges are called *self-edges* of a node. 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.

Each CFSG conforms to the CFSL meta-model, i.e. when representing the CFSL meta-model as a graph (called a *type graph*) there is a unique mapping from the elements of each CFSG to those of the type graph. This meta-model consists of *programming language independent* graph elements (i.e. nodes and edges) used to denote control flow. We have chosen graphs as the basis of our specification language for a number of reasons. We prefer a structure that is more expressive than the tree-structure, as is present in a parsed syntax tree or, less explicitly, present in a (E)BNF grammar rule. Also, by using graphs we are able to apply *graph transformations* to CFSGs, as we will see in Section 4.

2.1. Abstract syntax

As mentioned above, we base CFSGs on the language syntax, as encoded in its (context-free) grammar. To be precise, we take BNF rules enriched with names for the nonterminals; for example,

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

From this, we generate an abstract syntax graph (ASG) in which we leave out pure syntactic details. These structures are graphs instead of trees, since in some specific cases (e.g., in Java, a labelled `break`-statement) elements need to be shared. In ASGs, the tree relation between parent and child elements is preserved, for uniformity, by edges labelled child. The ASG corresponding to the BNF-rule just given is shown in Figure 2.

Figure 2: ASG *of the Java* `while` *statement.*

As a rule of thumb, we can say that for every non-terminal in the abstracted grammar of a programming language we have to design a CFSG.

2.2. CFSL meta-model

Control flow information describes the order in which the individual, atomic instructions of a program are executed. In this paper we distinguish three types of control flow: *sequential* flow, *conditional* flow, and *disruptive* flow.

We will discuss each type of control flow by means of the elements from the CFSL meta-model (Figure 4) that are involved. First, we will describe some elements that form the basis for the meta-model.

The node labelled AbstractSyntaxElement can be seen as a generic node representing all nodes from the ASG to which control can be transferred during execution. When specifying the CFSG for a particular language construct, the KeyElement-edge is used as a self-edge for that AbstractSyntaxElement-node. The edges labelled entry and exit identify the point at which the actual execution of this AbstractSyntaxElement starts or ends, respectively. The exit-node can either be one of the related AbstractSyntaxElements or a node to be freshly introduced (the left-most, non-labelled node in Figure 4).

Sequential flow refers to the type of control flow where statements are executed in the order they appear in the program. Statements that are executed subsequently are connected by edges labelled flow.

Conditional flow exists when the execution order is based on the value of some Expression. For each value the Expression can evaluate to we introduce a Branch-node (connected to the KeyElement by a branch-edge) referring to the original Expression with a condition-edge, and to the corresponding value with a branchOn-edge. The branchDefault-edge represents the branch that is taken when no other branches apply.

Finally, we say that a statement introduces *disruptive flow* if it is the cause of an abrupt termination. This is modelled by an Abort-node to which the control then flows via an abort-edge. This Abort-node has a reason-edge pointing to the AbstractSyntaxElement that caused the disruption. In some cases of disruptive flow it is not immediately clear at which statement to continue the program. This has been modelled by introducing the abortFrom and resumeAbort-edge. For details on how this is specified we refer to [14].

The meta-model is accompanied by a number of constraints on the combination of different elements. For example, a CFSG can have at most one exit-edge while the meta-model allows for at most two. A complete list of additional constraints can be found in [14].

2.3. Example CFSG design

As an example, we show how to design the CFSG for the Java `while`-statement (Figure 3).

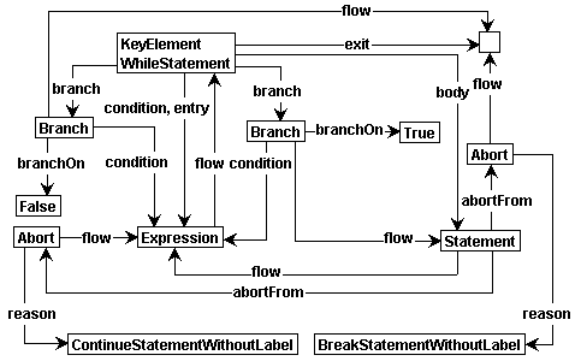

Figure 3: CFSG *of the Java* `while` *statement.*

We start with the ASG shown in Figure 2. We specify the WhileStatement to be the KeyElement, its conditional Expression to be the entry-point (execution of the WhileStatement starts by evaluating the condition) and introduce an unlabelled node to be its exit. We specify *sequential flow*: 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 WhileStatement, where the decision whether to continue iterating the body is made. The second flow-edge indicates that after execution of the body, the condition will be reevaluated.

The *conditional flow* is specified by two outgoing branch-edges from the WhileStatement-node: on `false`, the WhileStatement is completed (normally) by transferring con-

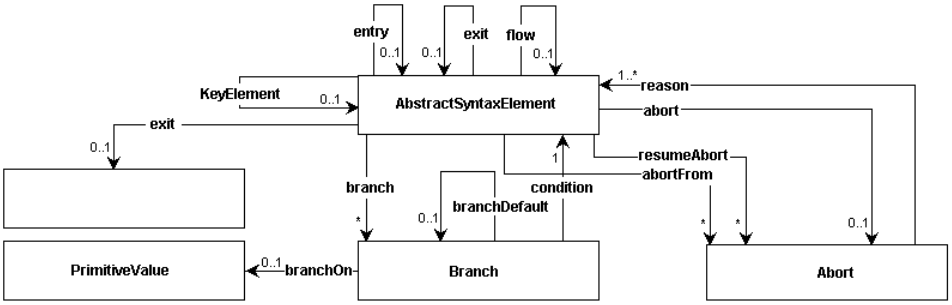

Figure 4: CFSL *meta-model.*

trol to the exit-node, on `true`, the body is entered for another iteration.

The execution of the body of a WhileStatement can be *disrupted* when the program reaches for example a `break` or `continue`-statement. This is specified by two Abort-nodes having incoming abortFrom-edges originating from that body. Each Abort-node has an outgoing reason-edge by which it keeps track of the AbstractSyntaxElement that caused the disruptive flow. Figure 3 specifies that when reaching a `break`-statement, control flows to the exit of the WhileStatement. Instead, when a `continue`-statement is executed, the condition of the WhileStatement will be reevaluated. The labelled variants of `break` and `continue` are not discussed here.

3. CFSL semantics

The semantics of CFSL is defined through a mapping from abstract syntax graphs to corresponding *flow graphs*. In this section we formalize that mapping through a *graph production system* that turns any abstract syntax graph into a flow graph.

3.1. Flow graphs

A flow graph (FG) is a widely known way of visualizing the flow of control in a program (e.g. [3]). In a flow graph, statements are shown as nodes, and arrows or graph edges indicate how the control is transferred between statements. There are many possible levels of modelling, ranging from detailed to abstract: for instance, one may abstract from the actual values in a branch, and just represent the fact that there exist multiple outcomes. We use a very detailed model, essentially consisting of the elements already shown in Figure 4, in which all aspects of the control flow are represented faithfully, so that one could actually do an accurate simulation of the program based on our FG representation.

To construct FGs from ASGs we use graph transformations that essentially *enrich* the existing structure; in other words, for us a flow graph is an abstract syntax graph decorated with control flow information.

3.2. Graph transformations

Graph transformation is a systematic, rule-based transformation technique. It has a solid research foundation [12] and applications in many areas in computer science.

A graph production system (GPS) is a set of *graph production rules*, each of which can transform a *source graph* into a new graph called the *target graph*. The rule specifies both the conditions under which it applies and the changes it makes to the source graph. Technically, a graph production rule[1] consists of two partially overlapping graphs, a *left hand side* L and a *right hand side* R, and a set of *negative application conditions* $\mathcal{N}$, which are also (connected) graphs partially overlapping with L. In order to apply the rule, the left hand side L is *matched* to (a part of) the source graph G, after which the image of L in G is replaced by a copy of R; but a matching is only valid if it cannot be extended to any of the graphs in $\mathcal{N}$ — in other words, the structure in the negative application conditions is forbidden in the source graph. In our visual presentation of a rule used in this paper (which is taken from the GROOVE tools) we combine all these elements together into one graph, made up of four types of elements:

- *Readers*: elements present in both L and R. They have to be present in the source graph for L to match and are preserved in the target graph;
- *Erasers*: elements present in L but not in R. They are matched in the source graph but are not preserved in the target graph, i.e. they are removed.
- *Creators*: elements absent in L but present in R. They are introduced to the target graph.

[1] For the purpose of this paper we do not use an explicit notion of graph morphism — formally, the morphisms are all (partial) embeddings; the setup corresponds to the Single Pushout approach (see [2]) with Negative Application Conditions (see [6]).

- *Embargoes*: elements absent in L but present in one of the negative application conditions in $\mathcal{N}$.

To distinguish these four types visually, each element has a distinct colour and form, as shown in Figure 5: *readers* are black, *erasers* are dashed blue (darker gray in black and white presentations) *creators* are bold green (light gray in black and white presentations) and *embargoes* are bold, dashed red (dark gray in black and white presentations).

(a) Reader (b) Eraser (c) Creator (d) Embargo

Figure 5: The graph production rule elements.

The effect of applying a GPS (rather than a single rule) is obtained by applying individual rules consequently, as long as there is an applicable rule; when no more rule can be applied, the transformation has terminated and returns the resulting graph. Two important aspects in this process are:

- At any moment during the transformation process, several rules may be applicable to the intermediate graph, and their application does not immediately give rise to the same result. However, the GPSs generated as semantics for CFSL are designed to be *confluent*, which implies that the application order of rules does not influence the resulting final graph. (We have not formally proved confluence; however, we believe it can be shown to follow from the fact that our rules essentially only add structure and do not remove it combined with the fact that the rules do not use embargoes.)
- The transformation process may fail to terminate. Again, we set up the CFSL semantics to be terminating; in this case it follows from the fact that we essentially implement a two-pass traversal over the ASGs, which are acyclic. (Again, we have not formally proved this.)

To execute the graph transformations, we use GROOVE.

3.3. Flow graph construction approach

The flow graph construction rules construct a FG by transforming an ASG. In this transformation process, control flow information elements are introduced to the ASG. For consistency, we use the same elements in FG's as we used to specify control flow in CFSGs (see Figure 4). Our FG construction approach consist of a number of design choices we have made:

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. The 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 uniformly provide auxiliary flow connectors for these entries and exits; these are merged during the construction.
4. We resolve disruptive flow using a bottom-up resolution process.

We discuss these choices below.

Ad 2. Flow graph construction is, in our case, a *top-down* process, meaning that we start at an ASG node that is defined to be the *root* of the FG under construction and continue along its ASG children. More concretely, we start by marking the root node as eligible for FG construction, using a special self-edge labelled build, and pass on these markers top-down. FG construction rules match on build edges.

Ad 3. All ASG elements with control flow semantics are considered to be FlowElements, and to have an entry and exit. The control flow of a given flow element has to take into account, among others, the execution of its children in the ASG. As result of the top-down construction, however, when a parent flow element is under construction, the control flow of its children has not yet been determined. This introduces a problem: 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 (FlowConnector) as the (initial) targets for the entry and exit edges of each flow element. Parent flow elements can connect their flow edges to these connector nodes, and children can introduce their internal control flow starting and ending at their entry and exit flow connectors. Our strategy is to uniformly introduce entry and exit flow connectors, and afterwards remove superfluous flow connectors by merging the nodes with other flow elements.

An example flow graph construction rule for the Java `while` statement is shown in Figure 6. In this rule we see the use of the build-marker, the merging of a FlowConnector and the introduction of both sequential (flow) and conditional branching (Branch) control flow.

Ad 4. The construction of disruptive flow is somewhat more involved. The main issue here is that the rules have to examine the FG context of an abrupt completion statement to determine the target statement of the resulting (disruptive) flow. Which types of flow elements are eligible as targets depends on the type of abrupt completion statement. We refer to the process of finding the correct target flow element and introducing the disruptive flow to this statement as *abrupt completion resolution*. We use a bottom-up resolution process. First, when an abrupt completion

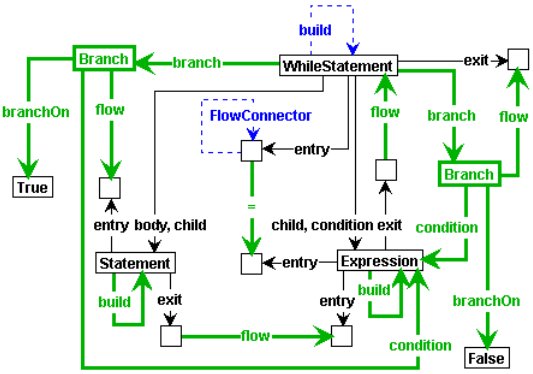

Figure 6: FG *construction rule for the Java* `while`*-statement (generated by* SEM*).*

statement is marked for construction, a corresponding flow graph construction rule introduces an abort-edge to an Abort-node, with a resolving-marker indicating that there is a situation that should be resolved. Next, the resolving-marker is propagated upward in the syntax tree until a resolving flow element is reached. Finally, another construction rule, matching both the abrupt completion statement and the resolving target, removes the marker and completes the disruptive flow. Figure 7 shows how a `break`-statement in a `while`-statement's body is resolved (compare to Figure 3).

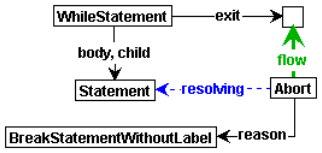

Figure 7: FG *construction rule for aborting a* `while`*-statement due to a* `break`*-statement (generated by* SEM*).*

In some cases, after abrupt completion has been resolved, it is reintroduced (*resumed*). In particular, this happens in the `finally`-clause of the `try`-statement, as we will see below.

3.4. Java flow graph construction example

We illustrate the FG construction process on the Java snippet shown in Listing 1. The listing has a `while`-statement which contains a `try`-statement with `finally`-part. The body of the `try`-statement includes a `break`-statement. We will see that upon executing the `break`-statement, control is first transferred to the `finally`-statement and, after its execution, abrupt completion is resumed, thereby terminating the execution of the enclosing `while`-statement.

```
while (true)
  try {
    ...
    break;
    ...
  } finally {
    ...
  }
```

Listing 1: Example Java code snippet.

The ASG corresponding to this code snippet is shown in Figure 8. The dots in the graph correspond to the dots in the listing, i.e., context that was omitted. When we apply our Java FG construction rules to this ASG, the construction process is executed. First, the entry and exit FlowConnectors are created; then the top-down flow constructions starts, beginning at the `while`-statement. After reaching the `break`-statement, abrupt completion is propagated bottom-up. It is resolved to lead to the `finally`-statement and then resumed to terminate the `while`-statement. The resulting FG is shown in Figure 9.

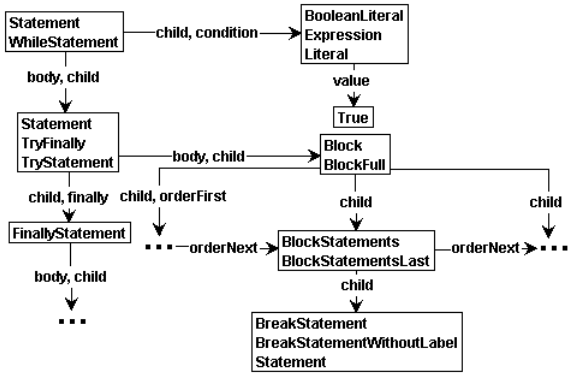

Figure 8: ASG *of Listing 1.*

4. Construction of the CFSL semantics

We now have, on the one hand, CFSGs specifying control flow semantics for statement types, and on the other, FG construction rules actually generating flow graphs for ASGs. The latter are actually intended as semantics for the former. For the purpose of defining the semantics we again use graph transformation; that is, we describe here (briefly) a GPS that can turn any CFSL specification (i.e., set of CFSGs) into the corresponding set of FG construction rules. We use the name SEM to refer to this GPS.

The rules in SEM can be thought of as *meta-rules* since they are graph production rules that create graph production rules. Obviously SEM is programming language independent: it can be applied to arbitrary CFSL specifications. The SEM-rules rules only match elements in the CFSL metamodel (Figure 4) and only introduce FG elements (including auxiliary elements needed for FG construction).

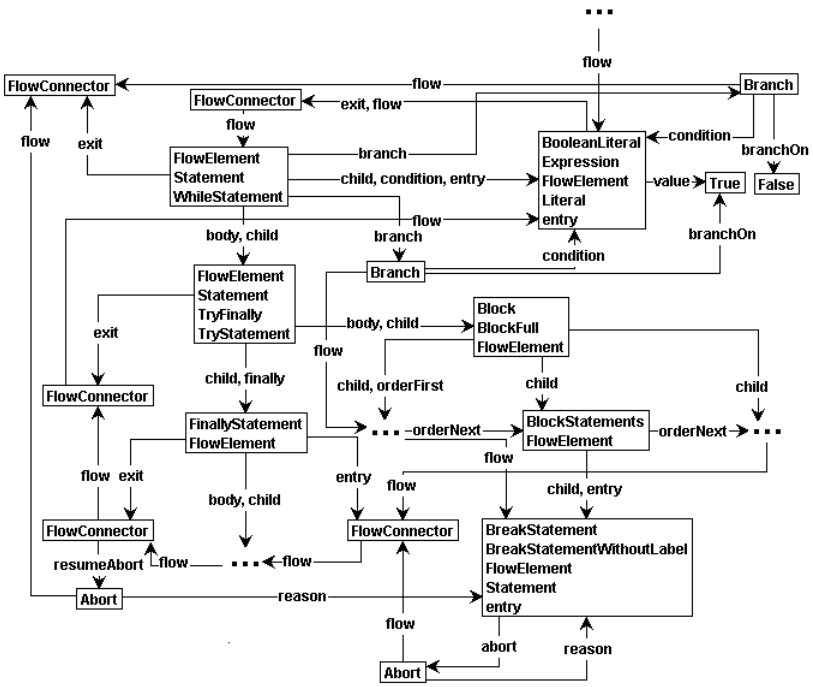

Figure 9: Constructed FG *of Listing 1.*

In contrast to the case for FG construction, discussed in the previous section, SEM does *not* always map CFSGs to single rules: instead, a single CFSG may give rise to multiple rules for FG construction. We have already seen an example of this: the CFSG of Figure 3 yields the rules in Figure 6 and Figure 7, among others. In other words, SEM is *not* confluent: instead, every sequence of rule applications is considered, and the outcome of the transformation process is the *set* of target graphs that cannot be transformed further.

As an example, one of the rules in SEM is shown in Figure 10. This rule represents a step in the construction of a FG construction rule for conditional flow, described in Section 2. Note the use of the prefix "new:" in the creators: this reflects the role indications of the nodes and edges in the target rule.

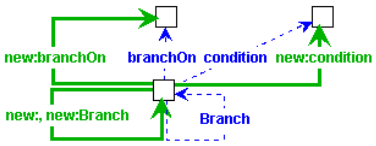

Figure 10: Example SEM *meta-rule.*

5. Conclusion

We consider CFSL to be successful in achieving our aims, set out in the introduction: it is a language for specifying control flow semantics, in a modular and visual way, as an extension of the syntax definition. We have validated the language by providing a full specification of the Java control flow semantics, including exception handling and other types of abrupt termination; see [14]. In doing so, we have also demonstrated the usefulness of graph transformation in this context, as a vehicle for defining the CFSL semantics as well as the construction of actual flow graphs.

That said, there are a number of points to be improved before we can claim to have a language that we can really expect a language designer to use:

- We should prove confluence of any GPS that is constructed from CFSGs.
- Instead of BNF, we should be able to start with an arbitrary EBNF grammar.
- As mentioned in the introduction, the concrete syntax of CFSL leaves much to be desired: a specification graph such as Figure 3, although it contains no redundant information, is not in a very readable format. By introducing an attribute-like notation in CFSL a lot can be gained; for instance, the branchOn- and reason-edges can be turned into attributes. Another good

possibility is to use hyperedges instead of the Branch and Abort-nodes. For instance, Figure 11 would represent the same CFSG as the one in Figure 3, but using a richer (*ad hoc*) visual syntax.

- As an alternative to the visual language, it might be a good idea to introduce a textual notation, so as to make the connection to the underlying BNF grammar more obvious.

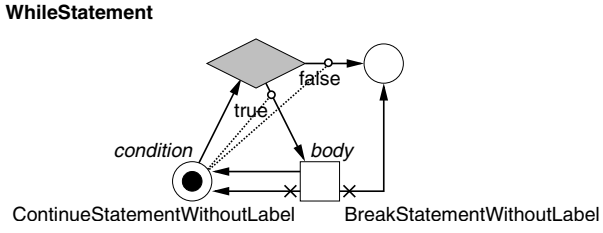

Figure 11: `while` control flow specification in alternative concrete syntax.

5.1. Related work

The work reported here originally arose from [8], where we present a full graph transformation-based semantics for a (custom) object-oriented language. Flow graph construction is part of that semantics; in that paper, we designed a special-purpose GPS for FG construction. Using CFSL we could specify that GPS much faster, more compact and more intuitively.

Another setting in which the generation of flow graphs has been studied is that of *triple graph grammars* (TGGs) [13]. Used especially for model transformation (e.g., [5]), TGGs are uniquely suited to reason not just about models, such as (in our case) ASGs and FGs, but also about their relation. In fact, triple graph grammars could well be considered as an alternative mechanism for the FG construction process. It is less clear, and indeed has not been the subject of study, whether an analogue to CFSL can also be formulated so as to benefit from the capabilities of TGGs.

Closely related to our work is the Montages project, described in, e.g., [1]. This provides a framework for aiding a language designer in specifying the syntax and the (static and dynamic) semantics of a programming language. The authors share many design principles with us; they too compose a complete programming language specification from a set of specifications and, as in our case, their specifications build upon the language grammar. Montages contain local finite state machines that decorate an abstract syntax tree (these can later be connected to form a flow graph). An important difference is that dynamic semantics in Montages are defined in text-based action rules. This can result in complex, non-visual action rules, e.g. for disruptive flow, whereas in CFSL this is structured more intuitively.

References

[1] M. Anlauff, P. W. Kutter, and A. Pierantonio. Enhanced control flow graphs in montages. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI'99)*, volume 1755 of *LNCS*, pages 40–53. Springer, 2000.

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

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

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

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

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

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

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

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

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

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

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

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

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