Recipes for Coffee: Compositional Construction of JAVA Control Flow Graphs in GROOVE

Eduardo Zambon and Arend Rensink

Abstract The graph transformation tool GROOVE supports so-called *recipes*, which allow the elaboration of composite rules by gluing simple rules via a control language. This paper shows how recipes can be used to provide a complete formalization (construction) of the control flow semantics of JAVA 6. This construction covers not only basic language elements such as branches and loops, but also abrupt termination commands, such as exceptions. By handling the whole JAVA 6 language, it is shown that the method scales and can be used in real-life settings. Our implementation has two major strengths. First, all rule sequencing is handled by recipes, avoiding the need to include extraneous elements in the graphs for this purpose. Second, the approach provides rules modularization: related rules are grouped in recipes, which in turn can be used again to form larger, more elaborated recipes. This gives rise to an elegant, hierarchical rule structure built in a straightforward, compositional way.

1 Introduction

This paper presents two contributions: a fully formalised *control flow specification* for JAVA (language version 6), and an extensive case study demonstrating the concept of *recipes*, which is a mechanism for rule composition in the graph transformation (GT) tool GROOVE [7, 11].

Control Flow Specification Our first contribution addresses the issue of control flow specification for the imperative, object-oriented language JAVA. The step of generating a control flow graph from the source code of a program is a very

E. Zambon (✉)
Federal University of Espírito Santo (UFES), Vitória, ES, Brazil
e-mail: zambon@inf.ufes.br

A. Rensink
University of Twente (UT), Enschede, The Netherlands
e-mail: arend.rensink@utwente.nl

© Springer Nature Switzerland AG 2018
P. Müller, I. Schaefer (eds.), *Principled Software Development*,
https://doi.org/10.1007/978-3-319-98047-8_19

well-known one: it lies at the core of both compiler optimisation and formal program analysis; see, for instance, [1, 9]. The control semantics of basic imperative statement types, such as **`while`**, **`if`**, **`switch`**, **`for`** and the like, is very well-understood, and it is not difficult to come up with an efficient, compositional algorithm for their construction. It is quite a bit more tricky to do the same in the presence of *abrupt termination* (as it is called in JAVA); that is, for **`break`**, **`continue`**, **`throw`** and **`return`** statements occurring anywhere in a block. The presence of abrupt termination firstly requires an extension to the notion of control flow itself—for instance, a flow transition taken because of a thrown exception needs to be treated differently from an ordinary control flow transition—and secondly mandates an overhaul of the construction algorithm.

The challenge involved in control flow generation can be made precise as follows: to devise an algorithm that, given the abstract syntax tree (AST) of an arbitrary (compilation-correct) JAVA program, generates a control flow graph (CFG) that captures all feasible paths of execution, with minimal over-approximation. (Some over-approximation is unavoidable, as the question whether an execution path can actually be taken by some real program run generally involves data analysis and is ultimately undecidable.)

In this paper we aim for a solution to this challenge that satisfies the following criteria:

- It is based on a declarative, rule-based formalism that manipulates (abstract syntax, respectively control) graphs directly; thus, one can alternatively understand our algorithm as a *specification* of the JAVA control flow semantics.
- It covers all of JAVA, rather than just a fragment; thus, we cannot take shortcuts by ignoring the more “dirty” language features.
- It is implemented and executable using a state-of-the-art graph transformation tool.

Recipes as a Rule Composition Mechanism Our second contribution is specifically directed at rule-based specification languages in general, and graph transformation in particular. In rule-based formalisms, it is quite common (as we also point out in the related work discussion below) to offer a way of scheduling rules sequentially or as alternatives; essentially, this comes down to adding imperative control to a declarative formalism. However, this form of composition results in constructs that cannot themselves be regarded as rules: their execution does not show up as a single, atomic step. Thus, one composes *from* rules but not *into* new rules. Instead, in this paper we demonstrate the concept of *recipes*, which are essentially named procedures with atomic behaviour. Recipes can for all intents and purposes be regarded as rules; in particular, they themselves can again be composed and recursively form new recipes.

The grammar for control flow specification presented in this paper very extensively uses the concept of recipes as implemented in the GT tool GROOVE, and hence serves as a demonstrator for their viability.

Related Work On the topic of (explicitly) dealing with abrupt termination in a formal setting, we can point to [8], which extends assertional reasoning to abrupt termination. The notion of control flow is implicitly treated there (as it must be, because the notion of pre- and post-conditions is inextricably bound to control flow) and the paper is rather comprehensive, but does not share our ambition of actually dealing with, and providing tool support for, any complete version of JAVA.

A much earlier version of the approach reported here is given in [14]: there also, graph transformation is used to construct flow graphs for programs with abrupt termination, but the construction is not compositional, relying instead on intricate intermediate structures that cause it to be far less elegant than what one would hope for in a declarative formalism.

On the topic of controlled graph rewriting, i.e., using control constructs for rule composition, quite some work has been done in the field of graph transformation. For instance, many GT tools, such as PROGRES [13], GRGEN [6] and VIATRA [4] include textual control languages, some of which are quite rich. Other tools such as HENSHIN [3] rely on the visual mechanism of *story diagrams* (developed in [5]) for specifying control. In [10] it is investigated what the minimal requirements are for a control language to be, in a strict sense, complete, and [2] generalises the notion of control composition itself. However, none of these approaches explicitly include the notion of atomicity that is essential to be able to regard a control fragment as a rule, as we do in our recipes. In fact, we argue that this is a dimension not covered by Plump [10] that identifies an incompleteness of other control languages.

2 Background

In this section we set the stage by providing some necessary background information.

2.1 Graph Grammars

We use graph transformation as our basic formalism for specifying computations. This means that we rely on *typed graphs* to describe states—in this case, ASTs and flow graphs—and *transformation rules* to describe how the states change, and under what conditions—in this case, how flow graphs are incrementally built on top of ASTs. Without going into full formal detail, we pose the following:

- A problem-specific *type graph* $\mathscr{T}$ that describes all the concepts occurring in the domain being modelled, in terms of node types (which include primitive data types and carry a subtype relation) and edge types. The possible connections between nodes and edges are restricted to those explicitly contained in $\mathscr{T}$.

- A universe of graphs $\mathscr{G}$, consisting of nodes and edges that have an associated (node or edge) type, subject to the restrictions imposed by $\mathscr{T}$.
- A universe of rules $\mathscr{R}$, with each rule r associated with a *signature* $sig(r)$, consisting of possibly empty sequences of input and output parameter (node) types.
- An application relation, consisting of tuples of the form

$$G \xRightarrow{r(\mathbf{v},\mathbf{w})} H$$

where G, H are graphs, r is a rule and $\mathbf{v}, \mathbf{w}$ are sequences of nodes from G and H, respectively, typed by $sig(r)$: these represent the actual input and output parameters and have to satisfy the rule signature.

An application instance such as the one above is called a *transformation step*, with G as *source graph* and H as *target graph*. A rule is called *applicable* to a given graph G if there exists an application step with G as source.

As a toy example, consider the rule fib depicted in Fig. 1a, with two example applications shown next to the rule in Fig. 1b. This specifies (in GROOVE syntax):

- A third **Cell** is created and appended to two existing next-linked **Cell**-nodes; the new **Cell** gets a val-attribute that is the sum of the val-attributes of the existing **Cell**s. In GROOVE syntax, element creation is indicated by fat, green outlines.
- The first of the two existing **Cell**s is deleted, together with its outgoing edges. In GROOVE syntax, this is indicated by blue, dashed outlines.

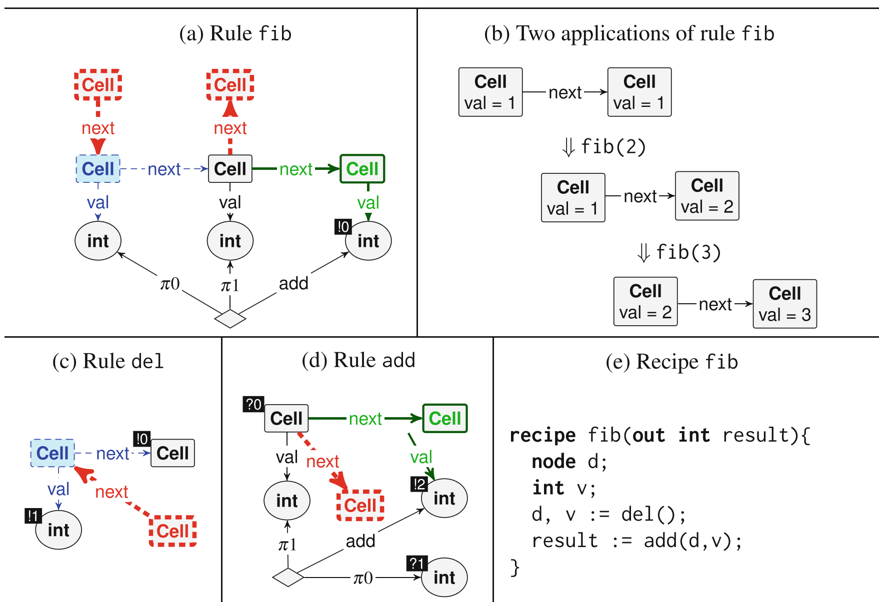

Fig. 1 From simple GT rules to recipes. (**a**) Complete simple GT rule `fib`. (**b**) Two applications of rule `fib`. (**c**) Ingredient rule `del`. (**d**) Ingredient rule `add`. (**e**) Recipe `fib`

- However, this only occurs if there is not a third **Cell** already. In GROOVE syntax, this is indicated by the red, dotted **Cell** nodes on top, which are so-called *negative application conditions*.
- The rule has a single output parameter, and no input parameters. In GROOVE syntax, this is indicated by the black adornment with the inscribed parameter number preceded by the exclamation mark **!**. An input parameter has a **?**-prefix instead.

A graph *grammar* is essentially a set of graph transformation rules, together with a start graph. We say that a graph grammar defines the "language" of graphs that can be derived through a sequence of rule applications from the start graph. One of the ways in which this can be used is to define transformations from one graph to another, namely by setting one graph as start graph and then considering all reachable graphs in which no more rules are applicable.

2.2 Recipes for Rule Composition

One of the attractive aspects of graph transformation (which it shares with other rule-based formalisms) is its declarative nature: each rule describes a particular change combined with conditions under which it can be applied, but there is typically no a priori prescription on how that change is effected or how different changes are combined. However, in practice it does occur quite frequently that an algorithm encoded in a set of transformation rules requires some dependencies between rules, resulting in a certain built-in order for their application. Moreover, it is also quite common that different rules contain similar parts, in which case it is desirable to be able to share those as sub-rules. Both are scenarios which can benefit from a notion of *rule composition*, meaning that a notion of control is imposed on top of the declarative rules, restricting the order in which rules may be applied and allowing the same rule to be applied in different contexts. Rule parameters can then be used to pass information between rule applications, circumventing the need to artificially put control information into the graphs themselves.

The GT tool GROOVE supports such rule composition in the form of *recipes*, which are procedure-like constructs with:

- A signature, consisting (like for rules) of a name and sequences of input and output parameters;
- A body, which specifies rule sequencing, repetition and choice.

A recipe is guaranteed to have atomic (transaction-like) behavior: it either finishes completely, in which case its effect, which may consist of many consecutive rule applications, is considered to be a single step; or it is aborted if at some point during its execution the next scheduled rule is inapplicable, in which case the recipe is considered to not have occurred at all.

As an example, Fig. 1c–e show how the rule from Fig. 1a can be specified instead as a recipe, by composing two ingredient rules (`del` and `add`) and using rule

parameters to ensure that the second rule is applied using the **Cell**- and **int**-nodes obtained from the first. The atomic nature of recipe execution implies that, if rule `add` is not applicable because the **Cell** already has a next, the execution is aborted and the graph is "rolled back" to the one before the `del`-transformation.

As stated in the introduction, for all intents and purposes a recipe itself behaves like a rule. The concept of a grammar is therefore extended to consist of rules, recipes and a start graph.

2.3 Constructing Abstract Syntax Trees

As explained in the introduction, the first contribution of this paper is to specify the construction of JAVA CFGs on top of ASTs; but in order to do so, first we have to obtain the ASTs themselves. The overall picture of how this is done can be seen in Fig. 2. For a more detailed discussion on the topics of this section we refer the interested reader to [15].

To bridge the world between JAVA source code and GROOVE graphs we built a specialized *graph compiler* that receives as input one or more `.java` files and produces a corresponding AST in GROOVE format for each compilation unit. The graph compiler was created by replacing the back-end of the Eclipse JAVA compiler with one that outputs the AST in the GXL format used in GROOVE. Our restriction to JAVA version 6 in this work stems from the Eclipse compiler used, which is limited to this language version. As future work, we plan to update the graph compiler to the latest JAVA version, namely version 10 at the time of writing.

As explained above, our graphs and rules are *typed* by a problem-specific type graph $\mathscr{T}$ that describes the allowed graph elements and their connections. In particular, program ASTs generated by the graph compiler conform to a JAVA AST type graph (see also Fig. 2), the node types of which are shown in Fig. 3. This type graph was manually constructed based on the AST structure produced by the Eclipse compiler. Additional typing structure is present in the form of edge types, which are omitted from Fig. 3 but presented in detail in [12].

Flow graph construction adds extra elements on top of the AST, which are also typed, this time by a JAVA CFG type graph (see Fig. 4), which extends the AST type

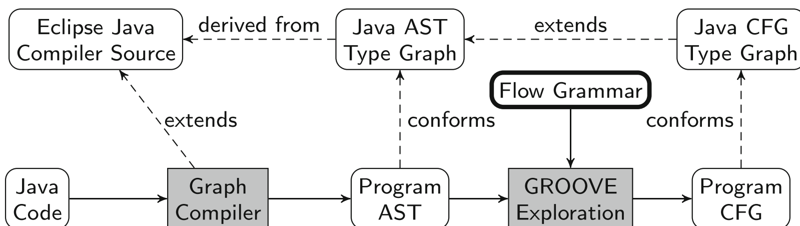

Fig. 2 Overview of the conversion from JAVA sources to GROOVE graphs

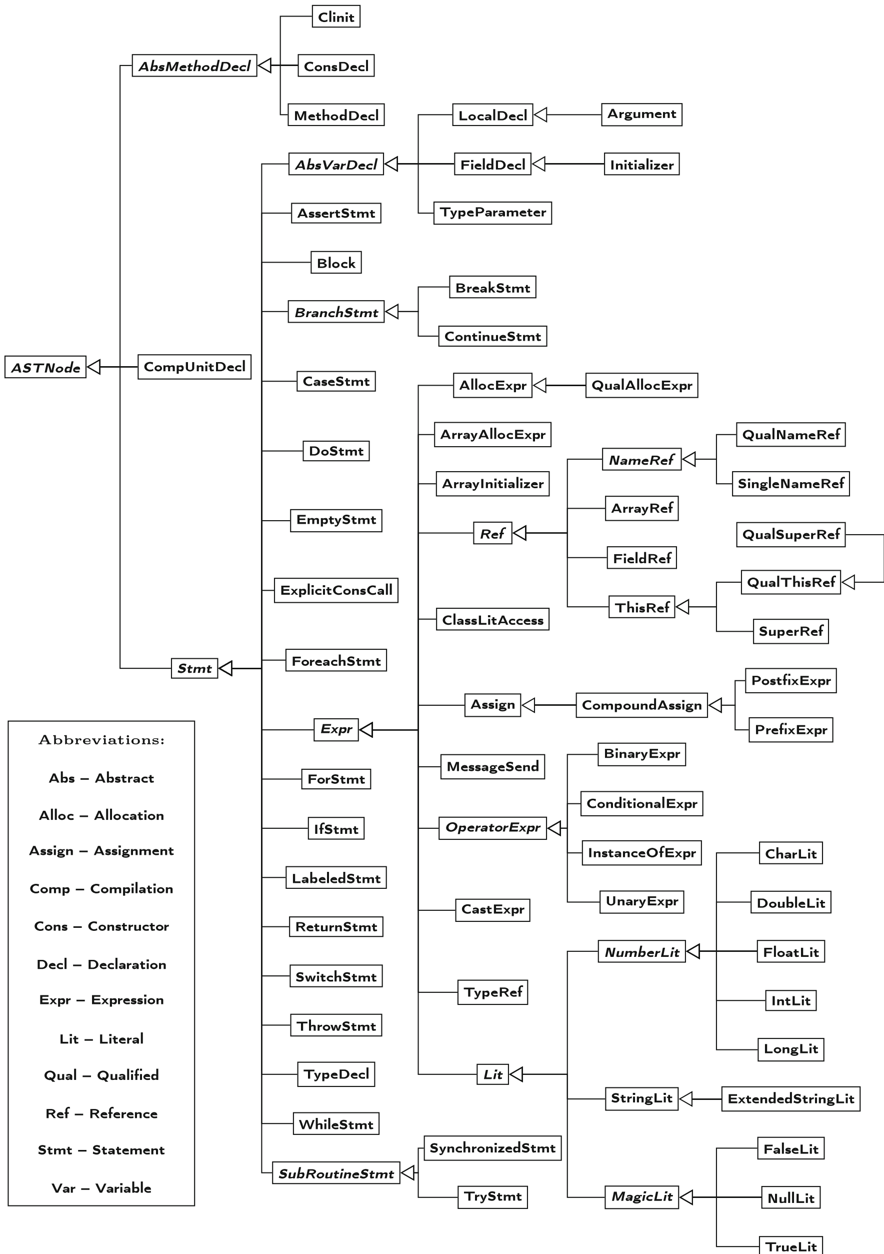

Fig. 3 JAVA AST type graph

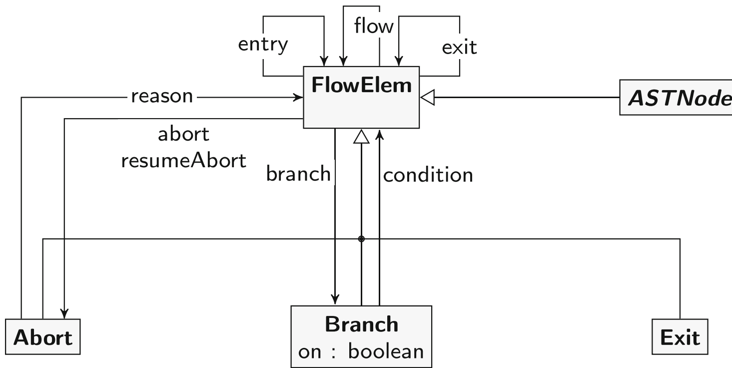

Fig. 4 JAVA CFG type graph

graph. Node type **FlowElem** is the top super-type of all possible elements that can occur in a CFG. Every executable block of code, such as a method body, has an entry point marked by an entry-edge. Execution then follows flow-edges until the **Exit** node is reached. Additional elements in Fig. 4 are **Branch** nodes, which are used in the over-approximation of loops and conditional statements, and **Abort** nodes, which point to commands that cause abrupt termination (see Sect. 3.2).

3 Building JAVA Control Flow Graphs with Recipes

In this section we give an overview of the GROOVE grammar for JAVA CFG construction. The grammar is composed of over 50 recipes and more than 250 simple GT rules, and therefore, a complete discussion in this paper is unfeasible. Instead, we focus on key aspects of grammar design and on some representative recipe cases. In Sect. 3.1, we give a simple example that shows the general idea of CFG construction for sequential execution of statements. Subsequently, Sect. 3.2 discusses the more elaborate case of dealing with abrupt termination commands. Finally, Sect. 3.3 presents the basic guidelines followed during grammar construction.

3.1 Basic Construction of CFGs from ASTs

Given an AST representing a valid JAVA program, we construct the CFG by adding flow-edges between AST nodes. The fundamental aspect of this construction is similar to the operation of a recursive descent parser. The AST is visited in a top-down manner, starting at the AST root node and traversing down the children nodes

by recursive calls to appropriate recipes. Recursion stops at the AST leaf nodes, where recipes terminate and return to their caller. When the entire calling sequence is finished, i.e., when the recipe for the root node completes, the entire AST was visited and the whole CFG was constructed.

Suppose a simple JAVA assignment expression, such as `x = 2+3*4`. Assuming that `x` is a simple local `int` variable, language semantics states that the right-hand side expression must be evaluated left-to-right[1] and the resulting value should be stored in `x`. Considering the usual operator precedence, the expression in post-fix notation corresponds to `(2 (3 4 *) +)`.

In JAVA, a *method* is the usual unit for grouping statements. Hence, the control flow grammar builds one CFG for each method in the AST. Commands inside a method are normally executed first-to-last, left-to-right. However, our recursive descent on the AST visits its nodes from right-to-left. In other words, the construction of the CFG is done in reverse order. This choice was made to limit the need for additional elements in the program graph.

The construction of a method CFG starts at a **MethodDecl** node, the root of the method in the AST. Upon visiting such node, a special **Exit** node is created, to indicate the method exit point (both for normal and abrupt termination). Then, method statements are visited last-to-first.

Every recipe for CFG construction has the following signature:

```
recipe SomeCommand(node root, node exit, out node entry) { ... }
```

Thus, recipe `SomeCommand` receives as input the `root` node of the sub-tree being visited and the `exit` node where the execution should flow after `SomeCommand` finishes. The recipe is responsible for constructing the CFG of `SomeCommand` and of all its composing sub-commands, by recursively descending in the AST starting at `root`. Finally, once this part of the CFG is constructed, the entry point of the entire sub-tree is known. This node is assigned to the `entry` parameter, and is returned to the caller.

To illustrate the process explained above, we present a step-by-step construction of the CFG for the assignment `x = 2+3*4`. The AST for this command is depicted in Fig. 5a. The `Assign` recipe (not shown) starts by calling `BinaryExpr(n2, n0)`, meaning that the CFG construction traverses down the AST, to the root node of the right-hand side expression (node `n2`), whose exit point is the **Assign** node (`n0`). Execution then continues inside the `BinaryExpr` recipe, given in Listing 1.

[1] For the argument's sake we assume the compiler does not perform optimizations such as *constant folding*, which would simplify the expression during compile time.

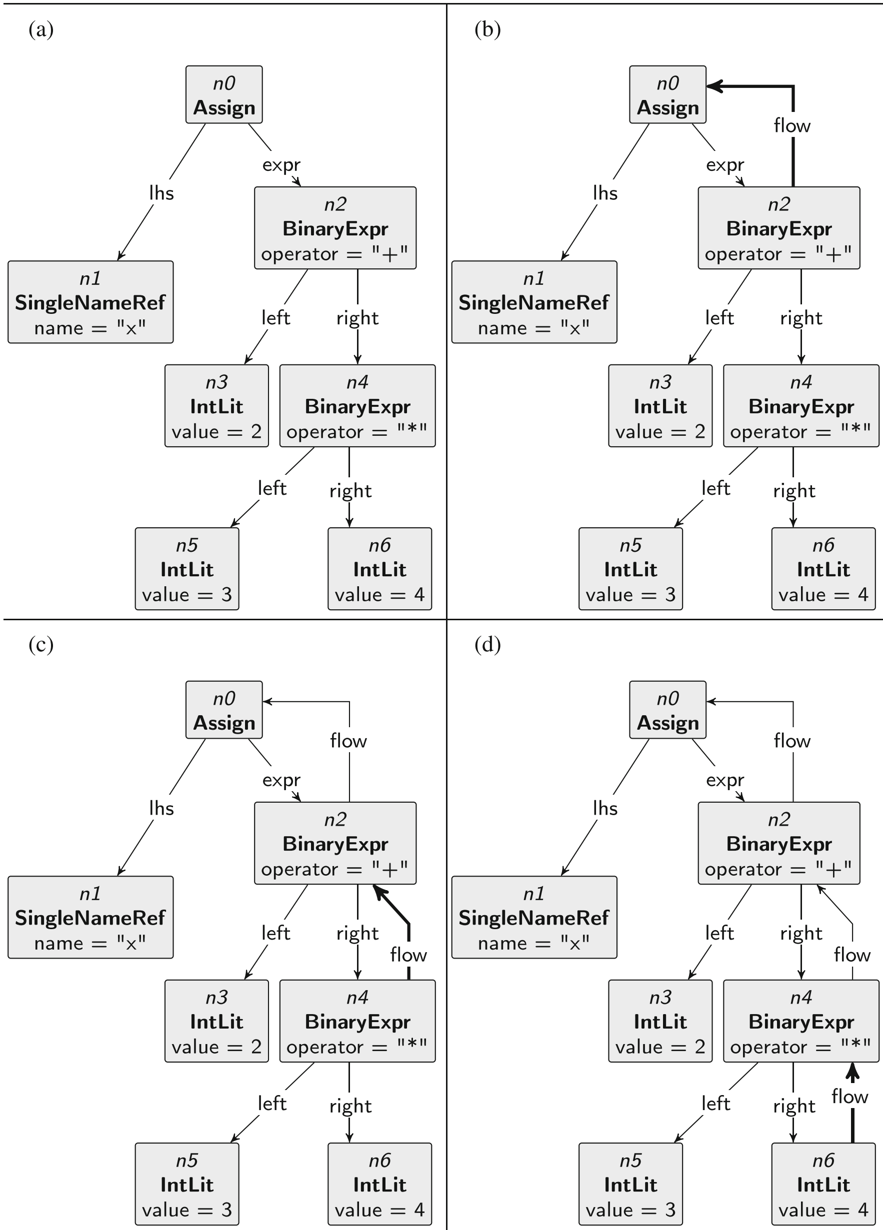

Fig. 5 Step-by-step construction of the CFG for `x = 2+3*4`. **(a)** `BinaryExpr(n2, n0)`, **(b)** `create-FlowEdge(n2, n0)`, **(c)** `BinaryExpr(n4, n2)`, **(d)** `Lit(n6, n4)`

Fig. 6 Rule `get-Children`, used by recipe `BinaryExpr`

Listing 1 Recipe for a binary expression

```
1  recipe BinaryExpr(node root, node exit, out node entry) {
2      create-FlowEdge(root, exit);
3      // Match the left and right components of the expression.
4      node lroot, rroot := get-Children(root);
5      // Build the flow for the right sub−expression.
6      node rentry := Expr(rroot, root);
7      // Build the flow for the left sub−expression.
8      entry := Expr(lroot, rentry);
9  }
```

In line 2 of Listing 1, the simple GT rule `create-FlowEdge` is applied to create a flow-edge from node `n2` to node `n0`, resulting in the graph presented in Fig. 5b. The recipe then continues in line 4, where rule `get-Children`, shown in Fig. 6, is invoked.

Rule `get-Children` is a common kind of simple GT rule used in our solution. Its sole role is to traverse along the AST, according to the given parameters, as can be seen from its signature in Fig. 6. (Recall from Sect. 2.1 that adornments with symbol **?** indicate an input parameter node, whereas **!** marks an output node.) Thus, given the root for the binary expression, the rule in Fig. 6 matches and returns the two roots for the left and right sub-expressions.

Execution of Listing 1 then continues on line 6, with recipe `Expr` being called on the right sub-expression (`rroot` node). This recipe (given in Listing 5 in an abbreviated form) just dispatches the call to the appropriate recipe, depending on the type of root node given as input. This dispatch leads to the recursive call `BinaryExpr(n4, n2)`, which starts by creating the flow-edge between `n4` and `n2`, leading to the graph shown in Fig. 5c.

The second call of `BinaryExpr` continues in the same vein as previously explained, until leaf node `n6` is reached with a call to recipe `Lit`, presented in Listing 2.

Listing 2 Recipe for all types of Literals

```
1  recipe Lit(node root, node exit, out node entry) {
2      // A literal is always a leaf in the AST, so we close the recursion here.
3      create-FlowEdge(root, exit);
4      entry := root;
5  }
```

Since literals do not have children in the AST, the recursion stops at the `Lit` recipe, which just flows to the exit and returns the literal node itself as the entry point, yielding the graph in Fig. 5d.

The current `BinaryExpr` reaches line 8, where the left sub-expression is constructed. Note that the entry of the right sub-expression (`rentry`) is now used as the exit point. This leads to call `Lit(n5, n6)`, and the resulting graph from Fig. 7e. The second `BinaryExpr` call returns, leading to the traversal of node `n3` with call `Lit(n3, n5)`, and the corresponding graph in Fig. 7f. Finally, the right-hand side expression of the assignment was traversed and node `n3` was identified as the entry point, as can be seen in Fig. 7g. By following the flow-edges starting from the entry, we obtain exactly the post-fix expression `(2 (3 4 *) +)` from the beginning of this example.

Construction of a CFG for sequential statements is as straightforward as that of expressions. When dealing with branching commands and loops, however, it is necessary to generate **Branch** nodes in the CFG, to mark the possible paths of execution. The overall method for handling branching is quite similar to the idea discussed above, and thus it will be skipped for brevity's sake. Instead, in the following we show the more complex case of abruptly terminating commands.

3.2 Handling Abrupt Termination

In JAVA, four statements cause abrupt termination, viz., **`break`**, **`continue`**, **`throw`** and **`return`**. These are so named because they "break" the normal execution flow of a program, causing a block to terminate early. The semantics of these commands can get quite convoluted due to nesting and the possibility of abrupt termination inside **`try-finally`** blocks. The JAVA method `m()` given in Listing 3 illustrates this complex case.

Listing 3 Method with **return** in **try-finally** block

```
1 void m() {
2       ; // C1
3       try {
4             ; // C2
5             return;
6       } finally {
7             ; // C3
8       }
9       ; // Unreachable
10 }
```

We use empty statements (`;`) in method `m()` to simplify the example, but these can be seen as place-holders for any block of commands. JAVA semantics states that a **`finally`** block must always be executed, even in the case of an abrupt termination. Thus, execution of `m()` starts at command `C1`, enters the **`try`** block and executes

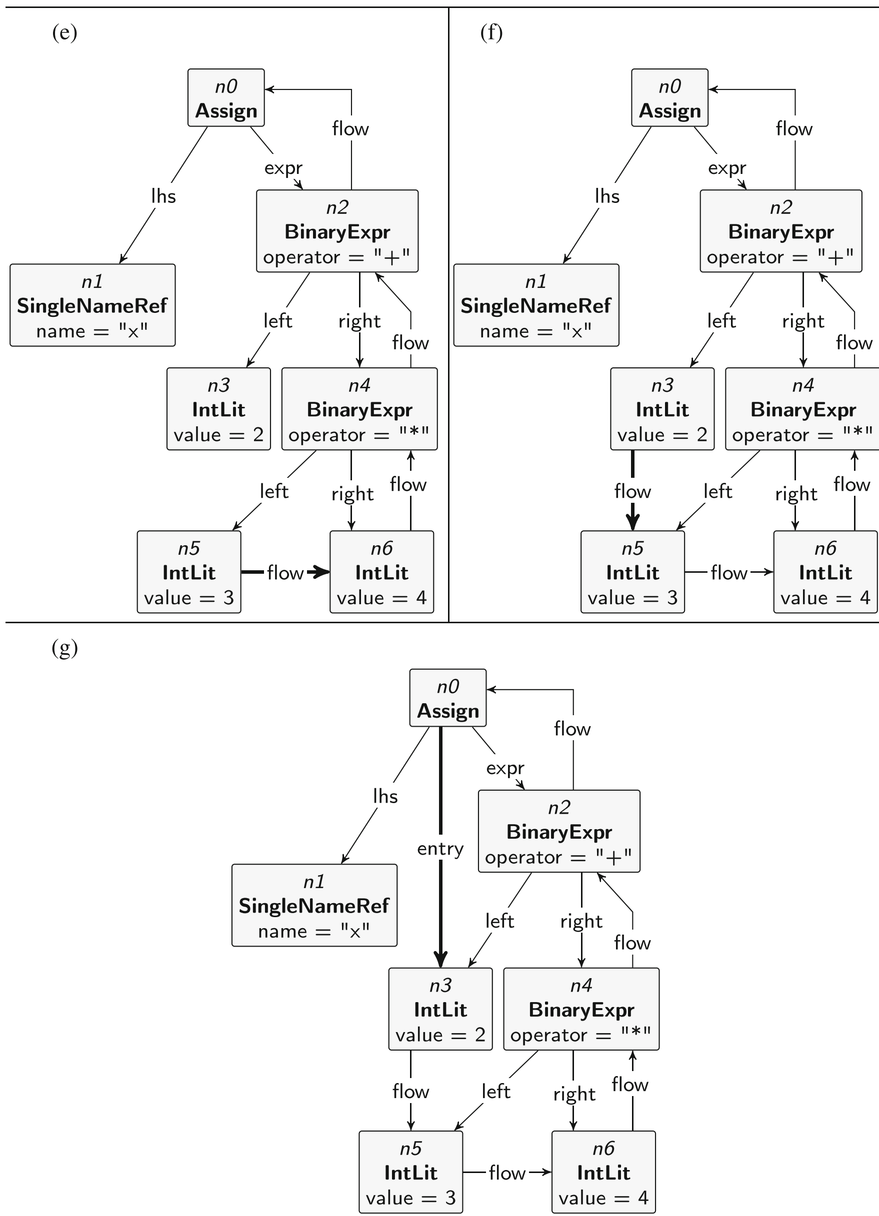

Fig. 7 Step-by-step construction of the CFG for `x = 2+3*4` (cont'd). (**e**) `Lit(n5, n6)`, (**f**) `Lit(n3, n5)`, (**g**) `create-EntryEdge(n0, n3)`

C2, and is *aborted* by the **return** statement. Control then flows to the **finally** block, causing the execution of C3. Lastly, the abortion is *resumed* on the enclosing scope, where execution flows immediately to the method exit point, bypassing the command indicated as unreachable. The AST of method m() is shown in Fig. 8a, where **EmptyStmt** nodes are marked with the same comments as in Listing 3. Integer attribute index is used in the AST to record the ordering of statements within a method body or block.

Control flow for abrupt termination requires information about flow in normal execution. Thus, during the AST traversal, we construct the CFG for non-abrupt statements as usual, and create **Abort** nodes in the CFG to mark abruptly terminating commands. An intermediate CFG in our running example can be seen in Fig. 8b, where node n10 marks the abort caused by the **ReturnStmt** node. The resolving-edge between these two nodes indicates that the abortion still needs to be handled. Control flow is now extended with abort-edges, which can be traversed as, but have priority over flow-edges, leading to **Abort** nodes and their associated reason for abrupt termination. On a complete CFG, **Abort** nodes flow to some point of the AST, so that execution can continue.

After the basic CFG construction finishes, we enter a phase called *abort resolution*, where all the aborted commands are properly analyzed and the CFG is finalized. As long as possible, **Abort** nodes are matched and recipes are called to resolve them. All these recipes are neatly contained in a control package called Abort, and Listing 4 shows the code for one of such recipes.

Listing 4 Recipe for resolving pending Return commands

```
1 package Abort;
2 recipe ResolveReturn(node root, node aroot) {
3   do {
4     choice root := propagate(root, aroot);
5     or root,aroot := resolve-ReturnStmt-In-TryBlock(root, aroot);
6     or root,aroot := resolve-CatchBlock(root, aroot);
7     or root,aroot := resolve-FinallyBlock(root, aroot);
8   } until (resolve-AbsMethodDecl(root, aroot))
9 }
```

Recipe ResolveReturn receives as input a **ReturnStmt** node (parameter root) and its associated **Abort** node (aroot). The recipe enters a loop, where abort resolution traverses up the AST, by means of rule propagate, shown in Fig. 9a. This rule receives as input an **Abort** node and the current **FlowElem** being resolved, and returns the parent statement in the AST. This rule is used to propagate abortions along nested blocks, for example, in Fig. 8b, from node n7 to node n4.

Abort propagation continues until it hits the method root node, when the abort is finally resolved by rule resolve-AbsMethodDecl, shown in Fig. 9b, which immediately flows to the method exit point. Lines 5–7 in Listing 4 handle the cases when abort propagation cannot go up, due to an enclosing **try** statement. Rule resolve-ReturnStmt-In-TryBlock is given in Fig. 9c. When reaching the outer

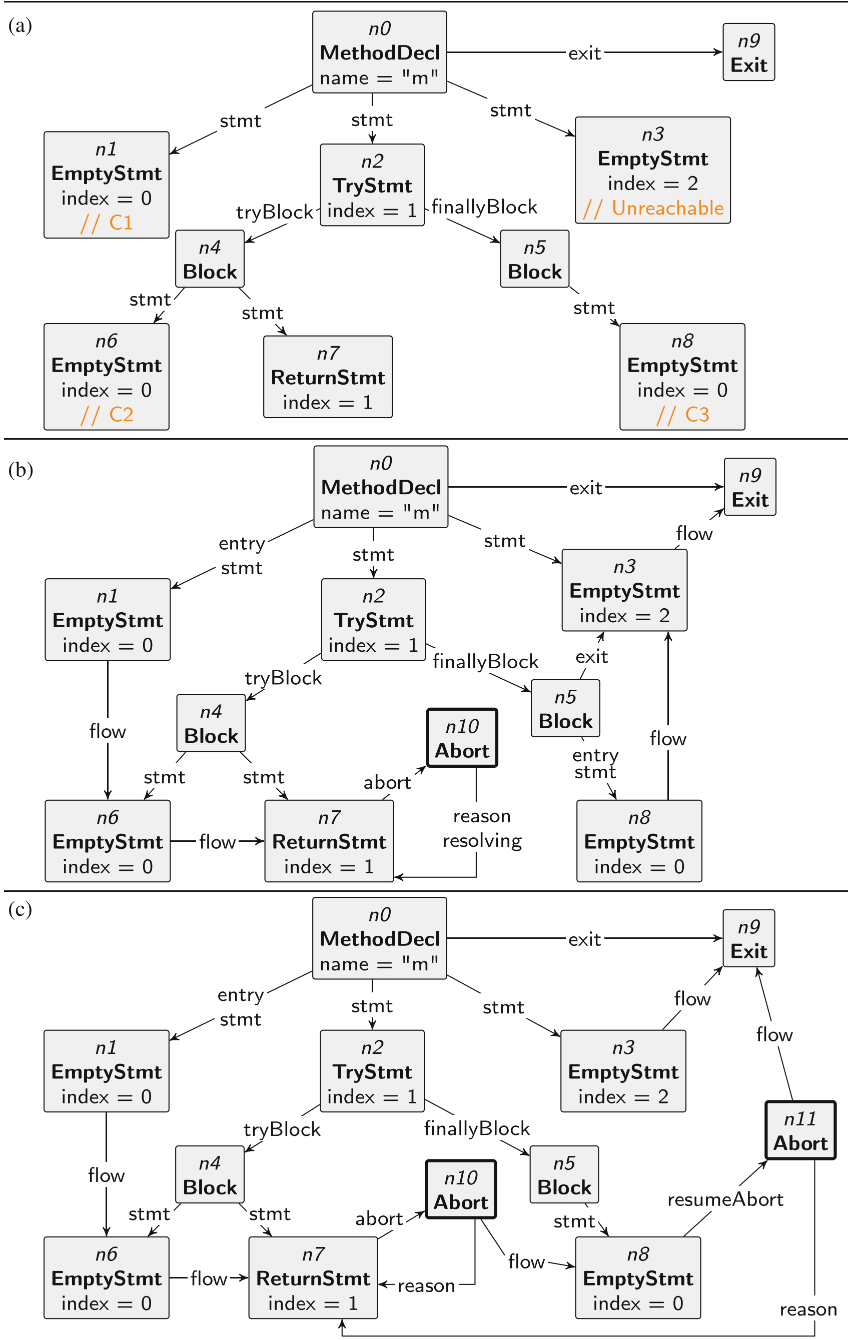

Fig. 8 Handling abruptly terminating commands. (**a**) Start state. (**b**) CFG after traversal with pending **Abort** node. (**c**) Final CFG after resolution

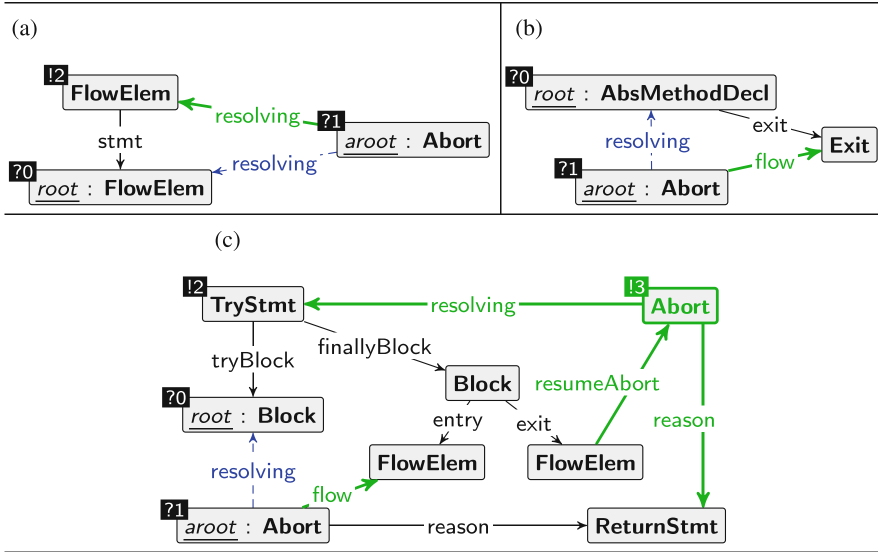

Fig. 9 Three rules used in recipe `Abort.ResolveReturn`. **(a)** `Abort.propagate`, **(b)** `Abort.resolve-AbsMethodDecl`, **(c)** `Abort.resolve-ReturnStmt-In-TryBlock`

scope of a **`try`** block that has an associated **`finally`** block, the given **Abort** node flows to the entry point. However, abort propagation does not end there, otherwise execution would continue normally after the **`finally`** block. Therefore, the rule also creates a new **Abort** node in the enclosing scope and returns it, so that propagation can proceed.

The rules used in lines 6–7 of Listing 4 are similar to the one in Fig. 9a and thus are not shown. The final CFG for this running example is given in Fig. 8c, with an additional **Abort** node (`n11`) created. Edge resumeAbort has the same semantics and priority as an abort-edge. It is interesting to note that some code analysis can now be performed on the finished CFG. For instance, node `n3` has no incoming flow edge and is therefore unreachable (as expected).

3.3 Rationale for Recipe Elaboration

Given that the control flow grammar is quite large, some guidelines were defined to rationalize its construction. We present these guidelines here with the hope that they will be useful in similar case studies.

The entire grammar construction was based on the JAVA AST type graph (Fig. 3) that was previously published in [12] and later updated in [15]. By looking at type nodes, the following guidelines were applied:

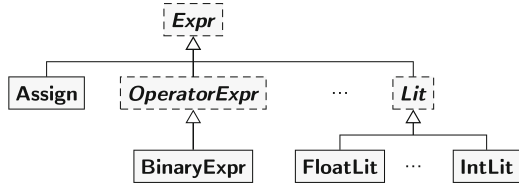

Fig. 10 Part of the AST type graph showing the abstract type **Expr** and some of its subtypes

1. Types that are not composed by other elements, i.e., that cannot have children in the AST, are the base case for the recursive descent of recipes on the AST. An example of such type is **Lit**, with its recipe given in Listing 2. Other types that also fit into this category are **EmptyStmt**, **NameRef**, and **ThisRef**, which follow the exact same recipe structure from `Lit`.
2. Abstract types lead to recipes that only dispatch the call to the correct recipe, based on the type of the given root node. For example, Fig. 10 shows part of the type graph from Fig. 3 zoomed in the hierarchy of abstract type **Expr**. The associated recipe is given in Listing 5.

Listing 5 Part of the recipe for an Expression

```
1  recipe Expr(node root, node exit, out node entry) {
2      choice entry := Assign(root, exit);
3      or entry := OperatorExpr(root, exit);
4      or entry := Lit(root, exit);
5      [...]
6  }
```

This recipe sequentially tries to call each of the recipes associated with the direct subtypes of **Expr**. If node `root` passed to `Expr` has type **Assign**, then the call to recipe `Assign` succeeds and `Expr` returns. If the node has a different type, the call to `Assign` fails and the `Expr` recipe tries the next line, until a successful call is made. It is interesting to note that recipe `Lit` does not follow this structure, despite **Lit** being an abstract type, because it falls in case 1.
3. Types that have one or more sub-trees in the AST give rise to the standard type of recipe, of which the `BinaryExpr` recipe in Listing 1 is a prime representative. Elaborating this type of recipe was the core of this work, and basically amounted to constructing the CFG for each sub-tree in the reverse order, following the JAVA semantics.

As a passing note, we should point that one of the recipe features highlighted in Sect. 2.2, namely the possibility to "rollback" a failing recipe is not needed in the CFG constructing grammar. Since we work with syntax-correct ASTs, the only place where a recipe can fail is on "dynamic recipe dispatching", i.e., on recipes for abstract types such as **Expr**, where we know we have an expression but we have yet to determine its concrete type. The **`choice-or`** construction sequentially calls recipes until one succeeds, but even in this case there is no real need of rollback,

because a recipe can only fail when inspecting the type of its given root node, and we are sure that exactly one of the recipes in the `choice-or` construct will succeed. (Since we are at an abstract node and it can only have one of the subtypes prescribed by the type graph, which are all potentially tested by the corresponding recipe.)

4 Conclusions and Future Work

This paper presented a GROOVE recipe-based solution for the problem of generating control flow graphs over abstract syntax trees of JAVA programs. Recapping the contributions stated in the introduction:

- The CFG generating grammar can be seen as an alternate, formal, executable specification of the control flow semantics of JAVA, which is presented in the language manual in plain English.
- The use of recipes provides an elegant, hierarchical rule composition mechanism. Although we tried in the text to make this point across, it can only be fully grasped when handling the real grammar in GROOVE. A previously attempted, non-recipe-based solution quickly became unwieldy, as one was forced to work in an unstructured set of several hundred rules, which could interact in complex ways and "polluted" the CFG with extraneous elements to ensure rule composition. In contrast, in this solution, rule composition and sequencing lend themselves neatly to a recipe-based implementation. This, in turn, simplifies the rules used in the grammar, making it easier to understand and maintain. We refer the interested reader to http://groove.cs.utwente.nl/downloads/grammars/, where the complete grammar here described is available for download.

As future work, the most pressing task is to update the graph compiler to JAVA version 10. No major obstacles are foreseen in lifting this approach to the latest JAVA version, given that new language constructs (such as lambdas) are defined in terms of existing ones to ensure backwards language compatibility. After a new compiler is available, the grammar here presented must be updated, but again no major hurdles are expected.

Having the ability to import real-life JAVA code to a GT tool such as GROOVE opens up a myriad of possibilities. One possible extension would be the creation of an optimizing grammar that does dead code elimination and other types of code analysis. Another would be the creation of a simulating grammar, that can follow the constructed CFG and "execute" the program, effectively rendering GROOVE a GT-based JAVA Virtual Machine. Since the major functionality of GROOVE is state space exploration, this could in turn allow for the model-checking of JAVA code. We plan to follow this line of investigation in the future.

References

1. F.E. Allen. “Control Flow Analysis”. In: *ACM SIGPLAN Notices* 5 (7 1970), pp. 1–19.
2. M. Andries et al. “Graph Transformation for Specification and Programming”. In: *Sci. Comput. Program.* 34.1 (1999), pp. 1–54. https://doi.org/10.1016/S01676423(98)000239
3. T. Arendt et al. “HENSHIN: Advanced Concepts and Tools for In-Place EMF Model Transformations”. In: *Model Driven Engineering Languages and Systems (MODELS), Part I*. Vol. 6394. Lecture Notes in Computer Science. Springer, 2010, pp. 121–135.
4. A. Balogh and D. Varró. “Advanced model transformation language constructs in the VIATRA2 framework”. In: *Proceedings of the 2006 ACM Symposium on Applied Comput- ing (SAC)*. ACM, 2006, pp. 1280–1287. ISBN: 1-59593-108-2. https://doi.org/10.1145/1141277.1141575.
5. T. Fischer et al. “Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and JAVA”. In: *Theory and Application of Graph Transformations (TAGT)*. Vol. 1764. Lecture Notes in Computer Science. Springer, 2000, pp. 296–309. ISBN: 3-540-67203-6. https://doi.org/10.1007/9783540464648_21.
6. R. Geiß and M. Kroll. “GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool”. In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Vol. 5088. Lecture Notes in Computer Science. Springer, 2008, pp. 568–569. ISBN: 978-3-540-89019-5. https://doi.org/10.1007/9783540890201_38.
7. A. Ghamarian et al. “Modelling and analysis using GROOVE”. In: *STTT* 14.1 (2012), pp. 15–40.
8. M. Huisman and B. Jacobs. “JAVA Program Verification via a Hoare Logic with Abrupt Termination”. In: *FASE*. LNCS 1783. 2000, pp. 284–303. https://doi.org/10.1007/3-540-46428-X_20.
9. C. Ouyang et al. “Formal semantics and analysis of control flow in WS-BPEL”. In: *Science of Computer Programming*. 67.2 (2007), pp. 162–198.
10. D. Plump. “The Graph Programming Language GP”. In: *Third International Conference on Algebraic Informatics (CAI)*. Vol. 5725. Lecture Notes in Computer Science. Springer, 2009, pp. 99–122. ISBN: 978-3-642-03563-0. https://doi.org/10.1007/9783642035647_6.
11. A. Rensink. “The GROOVE Simulator: A tool for state space generation”. In: *AGTIVE*. LNCS 3062. 2003, pp. 479–485.
12. A. Rensink and E. Zambon. “A Type Graph Model for JAVA Programs”. In: *FMOODS/-FORTE* LNCS 5522. Full technical report: Centre for Telematics and Information Technology TR-CTIT-09-01, University of Twente. Springer, 2009, pp. 237–242.
13. A. Schürr. “Internal Conceptual Modeling: Graph Grammar Specification”. In: *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Ed. by M. Nagl. Vol. 1170. Lecture Notes in Computer Science. Springer, 1996. Chap. 3, pp. 247–377. ISBN: 3-540-61985-2.
14. R. Smelik, A. Rensink, and H. Kastenberg. “Specification and Construction of Control Flow Semantics”. In: *2006 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, 2006, pp. 65–72.
15. E. Zambon. “Abstract Graph Transformation – Theory and Practice”. PhD thesis. Centre for Telematics and Information Technology University of Twente, 2013.