Graph-Based Specification and Simulation of Featherweight Java with Around Advice

Tom Staijen
Software Engineering Group
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands
staijen@cs.utwente.nl

Arend Rensink
Formal Methods and Tools Group
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands
rensink@cs.utwente.nl

ABSTRACT

In this paper we specify an operational run-time semantics of Assignment Featherweight Java — a minimal subset of Java with assignments — with around advice, using graph transformations. We introduce a notion of correctness of our specification with respect to an existing semantics and claim a number of advantages over traditional mathematical notations, that come forth from the executable nature of graph-transformation-based semantics.
Using test programs as graphs during specification of the semantics, simulation can help in verifying the correctness of the rules simply by testing, increasing the rigorousness of the specification process. Also, execution of the semantics results in a state space that can be used for analysis and verification, giving rise to an effective method for aspect program verification.
As a criterion for correctness, we use a structural operational semantics of this language from the so-called Common Aspect Semantics Base.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—Operational semantics

General Terms

Languages, Theory

1. INTRODUCTION

Aspect-oriented programming (AOP) [6] is a popular paradigm that allows for the modular specification of cross-cutting concerns. However, aspect-oriented programs are not easy to get right and even harder to test or debug. For this reason, it is attractive to investigate formal verification for aspectual programs. For the purpose of formal verification of AOP languages and programs, it is essential to specify the semantics of such languages formally and unambiguously.

FOAL'09, March 2, 2009, Charlottesville, Virginia, USA.

In this paper we propose a formal specification approach for the run-time semantics of Assignment Featherweight Java (AFJ). We illustrate an instruction based representation of an AFJ program and define the semantics of a "machine" running these instructions. This language is an extension to Featherweight Java (FJ), a minimal subset of Java. This simple language - although not suitable for industrial implementations - is typically suitable for studying language extensions. We study an extension of this language with *around advice*, which can also be used to represent before and after advice.
The specification method proposed in this paper is graph transformations. Graph transformation is a formal specification technique that supports rule based specification as well as an intuitive visual representation of states and rules. As a matter of fact, we use Groove [8] for the definition of the rules and graphs. Using the GROOVE tool, the graph transformation-based operational semantics directly provides an executable model; given a start graph representing a program and the graph transformation based operational semantics of the language, this system can be simulated resulting in a state space, represented as a labelled transition system (LTS). We claim that graph transformation has the following main advantages over traditional, more mathematical notations of operational semantics.

- Specifying a semantics can be a complicated task; mistakes are easily made. The directly executable nature increases ease and confidence of specification of a semantics by giving the user a way to test the semantics without having to write a interpreter first, which may contain errors either copied from the semantics or made during implementation.
- By giving the semantics in this way, the road is opened towards applying existing verification methods, such as the work we have presented in [1]. Also, the LTS lends itself directly for model checking (see [5]).

In addition, we believe that the visual nature of the graph transformation rules will appeal to many readers that are not experts in mathematics.
To increase confidence in the correctness of our definitions, we show that they coincide with a formal specification of the AFJ language in a work called the Common Aspect Semantics Base (CASB) [2]. The CASB is presented as a reference model for the run-time semantics of aspect-oriented programming languages. It presents a structural operational

semantics (SOS) for the language at hand (AFJ with an aspectual extension).
In the next section we will explain details of Assignment Featherweight Java with around advice and give an impression of the used execution mechanism. Section 3 provides a background on graph transformations and the visual notation used in this paper, followed by brief intuition in Section 4 of the actual graphs and rules used in the graph transformation based semantics. In Section 5 we briefly discuss the reference semantics and formulate our notion of correctness. Finally, in Section 6 we briefly mention essential related work, followed by our conclusions in Section 7.

2. ASSIGNMENT FEATHERWEIGHT JAVA WITH AROUND ADVICE

In this paper, we specify an operational semantics of Assignment Featherweight Java extended with the possibility of declaring *around* advice. In this section, we describe the required background.

2.1 The Featherweight AspectJ Language

Featherweight Java (FJ) [3] is a subset of Java that contains only five forms of expression: object creation, method invocation, field access, casting, and variables. Assignment Featherweight Java (AFJ) [7] has extended this language with mutable field variables to bring it closer to the way Java programs are usually written. The minimal syntax and operational semantics make it a handy language for conceptual studies on the implications of language extensions. This makes the language useful for trying AOP language features. We actually study an extension of the AFJ language with around advice. For the duration of this paper, we will refer to the extended language as Featherweight AspectJ (FAJ). The grammar of FAJ is as folows:

$$
\begin{aligned}
\mathit{Prog} &::= \overline{L}; e; \overline{A} \\
L &::= \textbf{class}\ T\ \textbf{extends}\ T\ \{\overline{T\ f};\ \overline{M}\} \\
M &::= T\ m(\overline{T\ x})\{e;\} \\
e &::= x \mid e.f \mid e.m(\overline{e}) \mid \textbf{new}\ T(\overline{e}) \\
&\quad \mid e.f = e \mid (T)e \\
A &::= T\ \textbf{around}(\overline{T x})\ :\ P\ \{e'\} \\
P &::= \textbf{call}(T^*.m^*(\overline{T^*})) \\
e' &::= e \mid e.\textbf{proceed}(\overline{e}) \\
T^* &::= T \mid \textbf{*} \mid T+ \\
m^* &::= m \mid \textbf{*}
\end{aligned}
$$

Throughout the paper we use the overbar notation for lists. A program consists of a set of classes, a main expression, and a set of advice declarations. Classes contain a list of field names and types, and a list of methods. A method consists of a return type, an identifier, a list of arguments and a method body, which is an expression. Expressions can be (from left to right) variables (e.g. a method parameters), fields accesses, method invocations with a sequence of expressions as arguments, object creations with a sequence of expressions as parameter, castings, assignments, and proceed expressions (see below). Object creation is not handled by an explicit constructor. Instead, the ordered list of arguments is assigned to the ordered list of fields.
Aspects are represented as (global) declarations of an around advice and a point-cut. Advices, depicted in the grammar above by the letter A, are methods that can optionally contain a *proceed* expression. As usual, we can use this also to mimic *before* and *after* advice, by adding a `proceed` instruction after or before the instructions of the advice, respectively. An advice declaration is combined with a point-cut declaration (depicted by P in the grammar) that selects certain expressions. In this language we have limited the point-cut language to the selection of method calls. Such a point-cut is specified as a call to a certain receiver type T^*, which can be a concrete type, a wildcard "*" standing for an arbitrary type, or $T+$, selecting a type and all its subtypes. The same is used for the parameters of the call. The method identifier can be either a concrete method identifier, or a wildcard "*" selecting an arbitrary identifier.

$$
\begin{aligned}
\mathit{mcode}(id, r) &= S(\mathit{mbody}(id, r)); \textsc{return} \\
\\
S(x) &= \textsc{var}_x \\
S(e.f) &= S(e); \textsc{get}_f \\
S(e_0.f = e) &= S(e); S(e_0); \textsc{set}_f \\
S(\textsc{new}\ T(e_0, \ldots, e_n)) &= S(e_0); \ldots; S(e_n); \textsc{new}_T \\
S(e.m(e_0, \ldots, e_n)) &= S(e_0); \ldots; S(e_n); S(e); \textsc{call}_m \\
S(e.\textsc{proceed}(e_0, \ldots, e_n)) &= S(e_0); \ldots; S(e_n); S(e); \textsc{proceed}
\end{aligned}
$$

Figure 1: Sequentialisation

2.2 Run-time Semantics

The run-time semantics of this language is specified in terms of sequences of instructions. That is, every expression in the grammar can be sequentialised into a sequence of stack-based instructions of the types: CALL, RETURN, NEW, VAR, GET, SET, and PROCEED. In this paper, we assume that expressions are pre-evaluated into such sequences; whenever we represent an FAJ program, method-bodies consist of a sequence of instructions instead of an expression. We define this sequentialisation as a function $\mathit{mcode} : Id \times T \rightarrow \overline{\mathit{Instr}}$ that returns a sequence of instructions given a method identifier. Given a function $\mathit{mbody} : Id \times T \rightarrow \mathit{Expr}$ that finds the body expression for a method identifier and a receiver type. This is defined as show in Figure 1.
Thus, the *mcode* function will use the given identifier and type to the *mbody* function, which looks up the method and returns the body expression. This expression is broken down into a sequence of instructions by function S. Finally, a RETURN instruction is added.
Run-time information is stored in both a heap and a number of global stacks:

- A so-called *continuation stack* contains the currently scheduled instructions, the top instruction being the first to be executed. Execution terminates when the continuation stack is empty.
- The results of evaluating an expression are placed on a so-called *value stack*.

For executing around advice, the following concepts are required:

- a *proceed stack* is used for postponing an action that triggers an advice; PROCEED instructions pop the top of the proceed stack onto the continuation stack.

Furthermore, a number of auxiliary instructions will be used:

- a DO instruction is used for invoking advices;
- a PUSHP instruction pushes the top of the continuation stack on top of the proceed stack;
- a POPP instruction pops the top of the proceed stack;

When a CALL instruction is matched by any aspects, these aspects are first scheduled (in a certain order) by placing a DO instruction on either the continuation stack (for the first advice), or the proceed stack (for any other advices). The PUSHP and POPP instructions are added to achieve a uniform handling of multiple around advice, all of which may contain a PROCEED instruction. To prevent instructions from being intercepted more then once, they are *tagged* the first time.

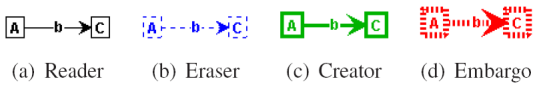
3. GRAPH TRANSFORMATIONS

Graph transformation is a systematic, rule-based transformation technique. It has a solid research foundation [9] and applications in many areas of computer science.
A graph is a type (N, E) where N is a set of nodes and $E \subseteq N \times Lab \times N$ a set of labelled edges. Nodes are graphically represented as black bordered boxes and edges as black arrows. 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 consists of two partially overlapping graphs, a *left hand side* and a *right hand side*, and a set of *negative application conditions*, which are also (connected) graphs partially overlapping with the left hand side.
Graph transformations provide an attractive visual representation. In our visual representation of a rule used in this paper (which is taken from the GROOVE tool [8]) we combine all elements together in one graph, made up of four types of elements:

- *Readers*: elements that are used for matching; are depicted as with black borders and arrows (see Figure 2.a);
- *Erasers*: elements that will be erased during the transformation are depicted with thin dashed borders and arrows (see Figure 2.b);
- *Creators*: elements that will be created during transformation are depicted with thick light gray borders and arrows (see Figure 2.c);
- *Embargoes*: elements that are not allowed to be present in the graph when the rule is matched are depicted with dashed, dark gray edges (see Figure 2.d)

4. SEMANTICS

We specify the run-time semantics of FAJ using graph transformations. Due to the limited amount of space, we are merely able to give an intuition of the workings: the encoding of the graphs, and the actual rules that form the semantics.



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

Figure 2: The graph production rule elements

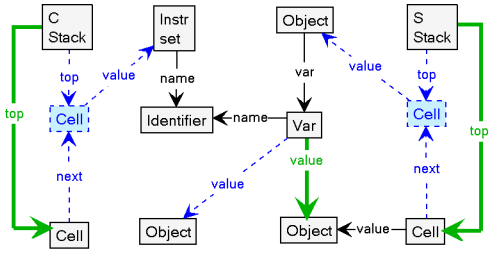


Figure 3: Rule for the set instruction

Graphs consists of a graph-based representation of an FAJ specification, and the run-time state. This run-time state consists of the stacks that we have introduced in Section 2, and nodes representing Objects. In fact, no distinction is made between memory locations and the actual instances. Expressions (method-bodies and the main expression) have already been process to sequences of instructions; the main expression is placed on the continuation stack.
There are rules for each of the instructions discussed in Section 2. To get a feeling of the proposed semantics, we describe the SET instruction. The complete set of rules can be found at *http://www.cs.utwente.nl/~staijen/faj/*.
Figure 3 shows the rule for the SET instruction, which originates from an expression of the kind $e.f = e_0$. The rule applies when a SET instruction is popped from the continuation stack, represented by a node labelled `Stack, C`. An outgoing edge of this stack points to the top `Cell` of the stack. A `next` edge points to the `Cell` underneath. The receiver object e and the new value e_0 are on the value stack, labelled with `Stack, S`. The variable to be updated is selected, that has the same name as the name argument of the SET instruction. The value of the variable is replaced, and the receiver is popped from the value stack. The new value remains on the value stack, since it is also result of $e.f = e_0$.

5. CORRECTNESS

We now give an intuition of how we can show that our operational semantics is *correct* in the sense that it corresponds to the prior semantics defined in SOS (Structural Operational Semantics) style in [2]. For this purpose, we define a mapping from the configurations in the SOS semantics to graphs, such that there is a one-to-one correspondence between SOS derivations and (sequences of) graph derivations. (It should be noted that our language is a slight adaptation of that in [2]; the most important difference is that we only allow call point-cuts, whereas they can define point-cuts for arbitrary instructions; on the other hand, we include parameters into the advice, which they do not. To establish correctness, we use an accordingly modified version of the SOS semantics.)

The static structure of a given FAJ program is captured by three partial functions:

- $FDecl : T \to (Ident \times T)^*$, yielding for each class type the sequence of field declarations (where *Ident* is the universe of identifiers);
- $MDecl : (T \times Ident) \rightharpoonup (Ident \times T)^* \times T \times Expr$, yielding for each class type the corresponding method declarations. A method declaration consists of a list of parameters (pairs of identifiers and types), a return type and a method body.
- $ADecl : (T \times Ident) \to \mathcal{P}((Ident \times T)^* \times Expr)$, yielding for each pair of class and method identifier the set of aspects that statically match calls of that method.

For every program, this triple of functions together with the initial expression plays exactly the same role as the initial graph. , except that there all expressions (i.e., the initial expression and the bodies of all methods and advices) have been sequentialised as discussed in Sect. 2.2. In fact, there is a straightforward translation from each valid combination $(FDecl, MDecl, ADecl, Expr)$ to an aspect program graph. For lack of space we omit the definition here, but below we use `Instr`, `Ident` and `Class` to denote the sets of nodes corresponding to *Instr*, *Ident* and T in the program.
The *dynamic* structure, i.e., the states of the program, are encoded in the SOS semantics as *configurations* (S, C, Σ, P) consisting of the same stacks and store as in our graph-based semantics:

- $C \in (Instr \times Bool)^*$ is the continuation stack, containing the instructions to be executed, combined with booleans indicating whether the instruction has already been adviced.;
- $S \in Object^*$ is the value stack, containing intermediate results; *Object* means location or heap address here;
- $\Sigma : Object \to T \times (Ident \rightharpoonup Object)$ is the heap, containing the run-time type and field values of all objects;
- $P \subseteq (Instr \times Bool)^*$ is the proceed stack, containing the (tagged or untagged) advices scheduled to be executed.

On the basis of these configurations, the SOS semantics consists of two types of rules: first, rules to sequentialise expressions to their corresponding instructions; and second, rules modelling the the execution of the instructions. In our semantics we have chosen to do sequentialisation as part of the pre-processing in Section 2.2; for the purpose of showing correctness in this section, we assume the same has happened on the SOS semantics side; that is, we assume that all expressions are already transformed into sequences of instructions.
The execution rule of instruction INSTR always has the form

$$\frac{\textit{side conditions}}{(\text{INSTR} : C, S, \Sigma, P) \to (C', S', \Sigma', P')}$$

meaning that, if the side conditions are fulfilled, a configuration in which the first instruction on the continuation stack is INSTR can perform a step, changing into the configuration on the right hand side. For instance, the rules for SET, CALL and RETURN are:

$$\text{SET}\quad \frac{\Sigma(v_0) = (T, F)}{(\text{SET}_f : C, v_0 : v : S, \Sigma, P) \to_b (C, v : S, \Sigma[v_0 \mapsto (T, F[f \mapsto v])], P)}$$

$$\text{CALL}\quad \frac{\Sigma(v_0) = (T, F) \quad MDecl(m, T) = ((x_1, \ldots, x_n), e)}{(\text{CALL}_m : C, v_0 : v_1 : \ldots : v_n : S, \Sigma, P) \to_b (e[x_1/v_1, \ldots, x_n/v_n], this/v_0 : C, S, \Sigma, P)}$$

$$\text{RETURN}\quad \frac{}{(\text{RETURN} : C, S, \Sigma, P) \to_b (C, S, \Sigma, P)}$$

Note that in these rules, the continuation stack elements are given as plain instructions rather than pairs of instructions and booleans; this is to indicate that we do not care about the instruction tags here.
The P-stack is only used for advice execution. Two example rules are given below: the AROUND-rule to schedule advice execution, and the rule for executing PROCEED. Notice that in these rules, we do care about tags.

$$\text{AROUND}\quad \frac{\Sigma(\mathbf{this}) = (T, Fd) \quad \Psi(T, m) = \{a_q \ldots a_n\}}{((\text{CALL}_m, \mathbf{ff}) : C, S, \Sigma, P) \to (\text{DO}_{a_1} : \text{POPP} : C, S, \Sigma, \text{DO}_{a_2} : \ldots : \text{DO}_{a_n} : (\text{CALL}_m, \mathbf{tt}) : P)}$$

$$\text{PROCEED}\quad \frac{}{(\text{PROCEED} : C, S, \Sigma, i : P) \to (i : \text{PUSHPP}_i : C, S, \Sigma, P)}$$

5.1 From Configurations to Graphs

The translation of SOS configurations to graphs is defined by

$$Tra : (C, S, \Sigma, P) \mapsto [\![C]\!] \cup [\![S]\!] \cup [\![\Sigma]\!] \cup [\![P]\!]$$

where $[\![C]\!]$, $[\![P]\!]$ etc. are the graphs corresponding to the individual data structures; the combined graph is the union of these. The individual graphs in turn are defined as follows:

- For each of the stacks, we introduce a single special `Stack`-node that stands for the stack as a whole, and `Cell`-nodes that stand for the stack positions. As representatives we can use integer numbers:

  $$\begin{aligned}\texttt{Stack} &= \{-1\}\\ \texttt{Cell} &= \{0, \ldots, n\} \quad \text{where } n \text{ is the stack size}\end{aligned}$$

  The nodes are linked with `top`-, `next`- and `value`-edges in accordance with the generated graphs. Using $|C|$ to denote the size of C and C^i to denote the value at position i (where the first position is numbered 0 and the last $|C| - 1$), the formal definition is:

  $$\begin{aligned}[\![C]\!] &= (\texttt{Stack} \cup \texttt{Cell} \cup \texttt{Instr}, E_C) \quad \text{where}\\ E_C &= \{(-1, \texttt{top}, 0)\} \cup\\ &\quad \{(i, \texttt{next}, i+1) \mid 0 \le i < |C|\} \cup\\ &\quad \{(i, \texttt{value}, x) \mid 0 \le i < |C|, C^i = (x, b)\} \cup\\ &\quad \{(i, \texttt{tag}, i) \mid 0 \le i < |C|, C^i = (x, \mathbf{tt})\}\end{aligned}$$

  Note that stacks always contain a spurious `Cell`-node for the sake of uniformity, so that even the empty stack has a `top`-edge.

  The P-stack is encoded in the same way; so is the S-stack, except that the `value`-edges point to `Objects`, and no `tag`-edges are required.

- For the store, we assume a set of nodes `Object` corresponding to the objects in $dom(\Sigma)$, that is, those

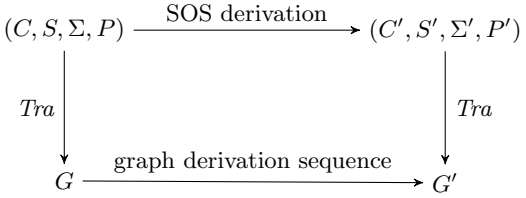
objects that are actually allocated on the heap. We also need auxiliary nodes to represent the object fields; these will be encoded as pairs (o, f) where $o \in \texttt{Object}$ and f is a field declared for o's type:

$$\texttt{Var} = \{(o, id) \mid \Sigma(o) = (t, Fd), id \in dom(Fd)\}$$

Using this set of nodes, the graph for Σ is defined by

$$\begin{aligned}[\![\Sigma]\!] = (\texttt{Class} \cup \texttt{Object} \cup \texttt{Ident} \cup \texttt{Var}, E_\Sigma) \quad \text{where} \\ E_\Sigma = \{(v, \texttt{name}, id) \mid v = (o, id)\} \cup \\ \{(o, \texttt{var}, v) \mid v = (o, id)\} \cup \\ \{(v, \texttt{value}, o) \mid v = (o', id), Fd(o') = o\} \cup \\ \{(o, \texttt{type}, t) \mid \Sigma(o) = (t, Fd)\}\end{aligned}$$

On the basis of these definitions, the correctness criterion is that the following correspondence must hold between the graph semantics and the SOS semantics:



This picture can be read top-down or bottom-up: for all single SOS derivations, there is a corresponding sequence of graph derivations, and for all sequences of graph derivations *between non-intermediate graphs* there is a corresponding SOS derivation — where a graph is intermediate if it is in between of a number of graph derivations that together correspond to the SOS derivation, i.e. when a SOS rule is specified using more then one graph transformation rule. The corresponding derivations are defined as a bisimulation. We are currently working on this final step.

6. RELATED WORK

The idea of the work reported here arose from [4], where a full graph transformation-based semantics is given for a custom defined object-oriented language. Also based on that work is the graph transformation-based semantics of a the Composition Filters language mentioned in [1], which however does not include a base language semantics and can therefore merely execute subsequent advices at a single join point. Both models use a different run-time state representation that is more suitable for object-oriented "machines". As mentioned before as our correctness criterion, Douence et al. [2] give an operational semantics of two base languages — a simple functional language and AFJ — and a large number of features of aspect-oriented language. The notation used is semi-formal yet mathematical and it does not provide means for execution.

7. CONCLUSION

In this paper we have propose a graph transformation-based semantics for a simple object-oriented language with around advice. The specified language, Assignment Featherweight Java, leans itself very well for studying language extensions. We have extended this language with around advice bound to point-cuts that select certain instructions.
We have illustrated that a graph transformation based operational semantics is a formal specification technique and can be complete with respect to a certain reference semantics. We have introduced a notion of correctness and illustrated how to prove this correctness. A graph tranformation based semantics is directly executable. This can help in finding bugs and testing the semantics. Due to its executable nature, the graph transformation-based specification has led to the discovery of oversights in the specification that is used as a correctness criterion; we see these errors as an unfortunate consequence of a purely formal notation.
The executable semantics allows simulation of program written in the specified language, if this program is represented as a graph as described in this paper. This gives a simple view on the execution of the program, and opens the road towards applying existing verification methods such as analysis based on model checking.

8. REFERENCES

[1] Mehmet Aksit, Arend Rensink, and Tom Staijen. A Graph-Transformation-Based Simulation Approach for Analysing Aspect Interference on Shared Join Points. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-Oriented Software Development, Charlotteville, Virginia, USA*, Charlottesville, Virginia, USA, March 2009.

[2] Rémi Douence, Simplice Djoko Djoko, Pascal Fradet, and Didier Le Botlan. Towards a common aspect semantic base (casb). In *Deliverable 54, AOSD-Europe, EU Network of Excellence in AOSD*, August 2006.

[3] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[4] H. Kastenberg, A. G. Kleppe, and A. Rensink. Engineering object-oriented semantics using graph transformations. Technical Report TR-CTIT-06-12, University of Twente, Enschede, March 2006.

[5] H. Kastenberg and A. Rensink. Model checking dynamic states in groove. In A. Valmari, editor, *Model Checking Software (SPIN), Vienna, Austria*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305, Berlin, 2006. Springer-Verlag.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[7] Thomas Mølhave and Lars H. Peterseny. Assignment featherweight java: Bringing mutable state to featherweight java. Master's thesis, University of Aarhus, 2005.

[8] Arend Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

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