# A Type Graph Model for Java Programs*

Arend Rensink and Eduardo Zambon

Formal Methods and Tools Group, EWI-INF, University of Twente
PO Box 217, 7500 AE, Enschede, The Netherlands
{rensink,zambon}@cs.utwente.nl

**Abstract.** In this work we present a type graph that models all executable constructs of the Java programming language. Such a model is useful for any graph-based technique that relies on a representation of Java programs as graphs. The model can be regarded as a common representation to which all Java syntax graphs must adhere. We also present the systematic approach that is being taken to generate syntax graphs from Java code. Since the type graph model is comprehensive, i.e., covers the whole language specification, the technique is guaranteed to generate a corresponding graph for any valid Java program. In particular, we want to extract such syntax graphs in order to perform static analysis and model checking of programs written in Java. Although we focus on Java, this same approach could be adapted for other programming languages.

## 1 Introduction

A graph is a flexible structure that is used to represent several different artifacts in computer science. However, the mathematical definition of a graph alone does not allow us to restrict a representation to a certain pattern or form. Such restrictions can be enforced by means of a *type graph*, a model that describes constraints over the sets of nodes and edges of a graph.

A program written in a certain language can be transformed into a syntax tree by a parser. When additional information such as bindings are included in the representation, the syntax tree is extended into a *syntax graph*. One main contribution of our work is to define a type graph model for syntax graphs that represent programs written in Java. The type graph model is complete, i.e., it covers the entire language specification up to version 1.6 [10]. We believe that this model can be of interest to any graph-based technique that relies on a representation of Java programs as graphs. As one example, suppose a visual programming/modeling tool that generates Java code from a graph; this could for instance, be used in the context of graph transformation-based model transformation [1] or code refactoring [3]. By enforcing the graph to be an instance of this type graph model, the tool can generate syntactically correct code.

* The research reported herein was carried out as part of the GRAIL project, funded by NWO (Grant 612.000.632).

In our current research we aim to perform static analysis [7] and model checking [2] of Java programs using GROOVE [8], a tool for state space exploration where states are represented as graphs, and the transitions from one state to another are given by graph transformation rules. A syntax graph is the static representation of a program as a graph, and it is the required initial structure for the subsequent elaboration of the states that constitute the dynamic behavior of the program. Thus, the work here presented is the first, necessary step in our planned method for the verification of code.

In this document we focus on the approach taken for the construction of the type graph model. Due to space limitations, it is not possible to actually present the model, and we refer the interested reader to the accompanying technical report [9], where all the details are given.

## 2 Description of Approach Taken

The task of constructing syntax graphs from given source code consists of two major steps, (i) the building of a type graph model to represent the syntactical elements of the chosen programming language, and (ii) the development of a tool that constructs a valid syntax graph from syntactically correct code. A syntax graph is considered to be valid when it is an instance of the type graph model developed in step (i). Essentially, the work to be done in (ii) boils down to writing a compiler that produces a syntax graph as its target language, instead of machine code.

We decided to adapt an open-source Java compiler for our purposes. In doing so, the implementation effort is kept to a minimum, since we have only to modify the code generation phase of the compiler to construct the syntax graphs. Also, by analysing the source code of the compiler we are able to elaborate the type graph model in a very straightforward way. Thus, with this solution, the definition of the type graph and the construction of the syntax graph generator go hand in hand, and we have the guarantee that a syntax graph generated from code is compliant with the type graph model.

### 2.1 Creating the Type Graph

In order to develop our chosen approach we decided to use the Eclipse Java Compiler [4]. This compiler is also written in Java, and its source code is available for use under the Eclipse Public License. The compiler source is divided in several packages, among which the package `org.eclipse.jdt.internal.compiler.ast`[1] is of particular interest, since it is where the classes that compose the Abstract Syntax Tree (AST) built by the compiler are grouped. By analysing the package contents we are able to construct the type graph model, which is presented in [9].

[1] Through the rest of the paper we adhere to the following convention: elements of Java code are shown in `typewriter` font, while elements of the type graph or instance graphs are shown in sans serif font.

```
class IfStmt extends Stmt {
  // fields
  Expr condition;
  Stmt thenStmt;
  Stmt elseStmt;
  ...
  // AST traversal method
  void traverse(ASTVisitor v){
    condition.traverse(v);
    thenStmt.traverse(v);
    if (elseStmt != null)
      elseStmt.traverse(v);
    ...
   }
   ...
}
```

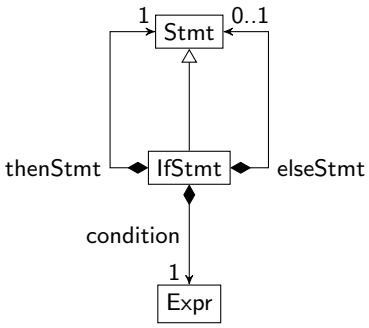


**Fig. 1.** Example of the type graph elaboration from the compiler source code

The `ast` package contains, for example, classes like `Expr` and `Stmt` to represent expressions and statements of the Java language. In fact, every syntactical element of the language has a corresponding class in the `ast` package and those classes are grouped in a certain hierarchy. The top most class is `ASTNode`, which defines a common super type for all elements of the AST. The `ast` package also provides an AST visitor pattern interface [5], which has methods to navigate over the nodes of the AST in a depth-first-like manner.

The way the type graph is elaborated from the elements of the `ast` package can be best explained with an example. Figure 1 shows the relevant code of the class that represents an "if" statement and the corresponding part of the type graph constructed from this code. We start with the class name, `IfStmt`, that gives rise to an homonymous node type in the type graph. Also, since `IfStmt` is a subclass of `Stmt` we create another node type for the super class and we insert an inheritance relation in the type graph, between the corresponding node types. The class fields that are references to other classes of the AST become compositions (in some cases, ordered ones) in the type graph, with labels matching the field names. In this example, the fields named `condition`, `thenStmt`, and `elseStmt` give rise to three compositions in the type graph, with corresponding labels. Additionally, the way the visitor pattern is implemented in the class provides some guidance over the cardinalities of the compositions just created. From the implementation of the `traverse` method we see that fields `condition` and `thenStmt` are always visited. Therefore we can conclude that the IfStmt node type must have mandatory condition and thenStmt compositions, a fact that is illustrated by the cardinality 1 of those compositions in the type graph. On the other hand, the check for non-nullness of the `elseStmt` field indicates that it

may not always exists. Therefore we mark the cardinality of its composition in the type graph as 0..1.

By analysing the classes of interest of the `ast` package in the same way as described in the example above we can construct a large part of the type graph model. However, there are some elements of the type graph that still need to be manually created. As an example we can cite the associations that resolve name and type references, which correspond to the binding edges on syntax graphs. The intuition for identifying where these associations must be created is simple: any reference should have an association with a corresponding declaration. However, the information needed to create these associations is not present in the compiler source code in a uniform way, and therefore manual intervention is necessary. The rationale behind our decisions over what does or does not have to be manually inserted into the type graph comes from our intended purpose for the syntax graphs. Thus, we insert only the elements that we deem necessary for static analysis and simulation.

The resulting type graph obtained from the analysis described in this section is formed by 75 node types, mapped directly from the compiler classes. The complete type graph is presented and explained in our technical report [9].

### 2.2 Constructing Syntax Graphs from Code

To construct syntax graphs from Java code we must change the back end of the Eclipse Java Compiler. By stopping the compiler after parsing and code analysis but before machine code generation we are able to profit from the work done by the compiler until this stage. Specifically, name and type references are already resolved, simplifying the construction of the syntax graph. We developed a syntax graph generator that implements the AST visitor interface provided by the compiler and we plugged it in the compiler back end. To build the syntax graph, our generator visits the AST, performing the following steps.

- For each node in the AST the generator creates a corresponding node in the syntax graph. The types of a syntax graph node are obtained through reflection. By using reflection in Java, one is able to query the virtual machine for run-time information of objects. In our case we obtain the class hierarchy of an AST node via reflection and store this information as a label of the syntax graph node.
- For the construction of edges in the syntax graph we keep an auxiliary mapping of AST nodes into syntax graph nodes. This mapping, along with the bindings produced by the compiler, is sufficient for creating the edges, including the ones that resolve references.

For each node type of the type graph we created a test case input program. With these test cases we can inspect the syntax graphs produced by our tool and check for implementation errors. An example of such test case is given in Fig. 2, along with the syntax graph generated. The complete set of input test cases can be found in the corresponding technical report [9]. The syntax graph

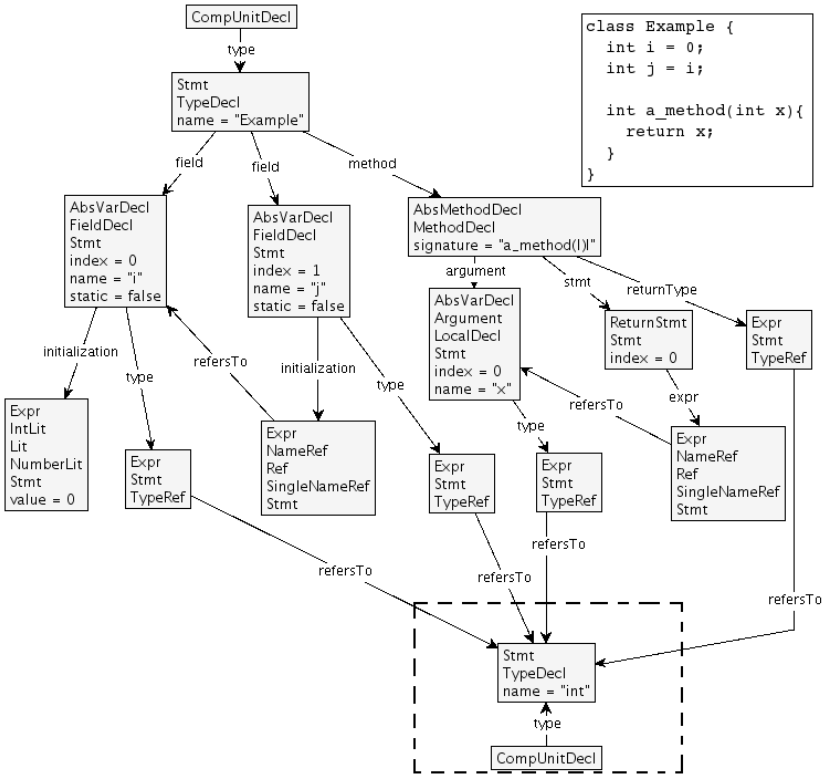


**Fig. 2.** Example of a syntax graph built from code. The dashed box represents a system wide compilation unit where primitive types of the language are declared.

in Fig. 2 has a node labeled TypeDecl, which represents a class. The name of the declared class is stored as an attribute of the node, which also has three outgoing edges that correspond to the field and method compositions. It is important to note that name and type reference nodes have an outgoing edge that binds the reference to its corresponding declaration. We consider the existence of a "system" compilation unit, where the primitive types of the language, and also the classes of `java.lang.*`, are declared. Part of this "system" compilation unit is shown in the bottom of Fig. 2 (within the dashed box), with the declaration of the primitive type `int`.

## 3 Conclusion and Future Work

To sum up, the contributions of our work are threefold.

– We have created a comprehensive type graph that covers all executable elements of the Java programming language. Such type graph can be of interest as a model for tools that represent Java programs as graphs.

– We have shown a straightforward and systematic approach for the construction of the type graph model by analysing a compiler source code. Although our described method focused on Java, we believe that it can be adapted (with varying degrees of difficulty) to other programming languages as well.
– We explained how the back end of a compiler can be adapted in order to automatically construct a syntax graph representation from source code.

The work described in this paper is the first step in our planned approach for the verification of Java programs. Now that we are able to generate syntax graphs from code the next step is the construction of *flow graphs*, structures that model the sequential execution relation between elements of the syntax graph. We plan to define graph transformations rules over syntax graphs for flow graph construction, as described in [6]. Together, a syntax graph and a flow graph form a *program graph*. The subsequent step is then use the GROOVE tool to simulate the execution of program graphs. Another important aspect of this step is that we want to apply abstract interpretation techniques to simplify the program graphs and thus improve the performance of the simulation.

## References


1. Alanen, M., Lundkvist, T., Porres, I.: Creating and reconciling diagrams after executing model transformations. Sci. Comput. Program. **68**(3) (2007) 155–178
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, New York (May 2008)
3. Baumer, D., Gamma, E., Kiezun, A.: Integrating refactoring support into a Java development tool. In: OOPSLA 2001 Companion. (2001)
4. Eclipse Foundation: JDT core component development resources. `http://www.eclipse.org/jdt/core/dev.php`
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY (1995)
6. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In Gorrieri, R., Wehrheim, H., eds.: Formal Methods for Open Object-Based Distributed Systems (FMOODS). Volume 4037 of Lecture Notes in Computer Science., Springer (2006) 186–201
7. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
8. Rensink, A.: The GROOVE simulator: A tool for state space generation. In Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: Applications of Graph Transformations with Industrial Relevance (AGTIVE). Volume 3062 of Lecture Notes in Computer Science., Springer (2003) 479–485
9. Rensink, A., Zambon, E.: A type graph model for Java programs. Technical Report TR-CTIT-09-01, University of Twente, Enschede (February 2009)
10. Sun Microsystems: The Java language specification. `http://java.sun.com/docs/books/jls/`