

A Dedicated Language for Object-Oriented Design Analyses

Cristina Marinescu*

Radu Marinescu*

Tudor Gîrba**

* LOOSE Research Group, Research Institute e-Austria Timișoara

** Software Composition Group, University of Berne, Switzerland

{cristina,radum}cs.utt.ro

girba@iam.unibe.ch

Abstract

A lot of static analyses techniques that address the quality of object-oriented design appeared in the last decade. These analyses were implemented using a large variety of programming languages. Due to this heterogeneity of expression the understanding and potential interconnection of these analyses is severely hampered. In this paper we introduce SAIL¹, a language dedicated to the aforementioned type of static analyses. By its properties the language allows a simple and understandable expression of different analyses.

Motivation When we have in front of us the source of a program, besides its functional accuracy we are also interested in the quality of the design or of the implementation as it is reflected in the source code. This is what code analysis is all about. If the source code is as small as a few or hundreds of lines of code, we can examine the code manually; but as the source code becomes larger we want to find out a lot of additional things like the impact of change a class functionality on the other classes and all of these things cannot be done manually.

Static source code analysis [3] is the process by which software developers check their code for problems and inconsistencies before compiling. Development organizations have two basic alternatives for conducting static source code analysis: manual code check or automatic static source code analysis. Automatic source code static analysis addresses many difficult problems faced by the software industry today.

As we pointed out in the abstract, a lot of static analyses that address the quality of object-oriented design appeared in the last decade. Thus, we encounter in the literature metrics [4] that are implemented in SQL or JAVA, detection strategies, defined in [5], using a dedicated language called SOD or problem detection techniques built on the analyses of structure graphs, implemented using PROLOG clauses.

Analysing all these implementation choices we concluded that the main reason for all this diversity of expression resides in the fact that none of the used languages provide simultaneously a powerful *query mechanism* (like SQL or PROLOG) along with the main mechanisms found in structured (or object-oriented) programming languages. This diversity makes it oftentimes impossible to understand and, moreover, to reuse and correlate these analyses.

Nonfunctional requirements As a result of this study we decided to define a new programming language dedicated to those static analyses that address the quality of object-oriented design, based on the following set of criteria:

- *Simplicity of expression.* The language should provide mechanisms that would facilitate a *concise* and *natural* expression of the desired analyses. This way we want to increase the *understandability* and *changeability* of the analyses' description (not like SQL) and in the same

¹Static Analysis Interogative Language

time to *reduce the verbosity* (often encountered when structured languages are used, due to the lack of a mechanism for concise expression of queries).

- *Modularity*. We heavily need mechanisms that allow us to reuse analyses and to compound them into more abstract and complex types of code inspections.

Key Language Mechanisms In the following we will describe the key language mechanisms of SAIL that support the aforementioned set of criteria.

As we will see next, SAIL addresses the *simplicity of expression* by introducing, on the one hand, simple definition of and powerful operation with structures and collections; and on the other hand by defining a query mechanism, inspired from the SQL `select` statement, yet adapted to our needs. In order to address the *modularity* criterion, we use in SAIL functions (even in the `select` statement!) and a mechanism that allows us to include and reuse analyses defined in other files.

Mechanisms addressing simplicity

- **Powerful operation with structures and collections.** By studying the different code inspection techniques we found out that they can all be roughly reduced to simple (or more complex) manipulations of collections. What collections? The primary collections encountered in these static analyses store all the information about the inspected system (e.g. information about all the classes, methods, packages). These collections build together the *model of the system*, while the structure of the entities stored in these collections form the *meta-model*.

That's why in SAIL *structures* and *collections* play a central role and operating on these must be extremely intuitive and concise. Thus, defining structures and collections in SAIL looks very similar to C or JAVA. In the same time SAIL provides all the operations on collections one would expect (e.g. intersection, reunion and difference operators, iterators etc.). These will be illustrated in the example provided later in this paper.

- **Query mechanism (the `select` statement).** A lot of analyses need sometime from the *model* only those entities that satisfy some properties and it has to be easy to obtain them. We introduce the `select` instruction, similar to the well-known `select` from SQL. But what makes it different?
 - We dropped a part of the syntax that we felt as an overhead for our purposes.
 - We increased its power by allowing to call SAIL functions. Instead of having obfuscated compound expression in the `from` and `where` clauses we can *encapsulate* that complexity in a SAIL function. This increases the *readability* of the `select` statement, and in the same time it serves the *modularity* criteria by letting those functions be reused in the context of another analysis.
 - We integrated it with the rest of the language (which is rather a *structured* one) by allowing the result of a `select` statement to be assigned to a SAIL collection.

Mechanisms addressing modularity

- **Functions.** Functions, by definition, provide *modularity* and *reusability* but the possibility of parameterizing the `select` instruction with a function increases also *simplicity*. The structure of this mechanism is the same with the one from C or JAVA.
- **Imports.** In order to compound analyses into more abstract and complex ones we need to have a mechanism for reusing other analyses defined in the past. The `import` instruction, similar to the existing one in JAVA, provides it. We want to build a powerful library of analysis which can be properly packaged, for example based on the analysis purpose and on its level of abstraction.

An example After the brief introduction of the key mechanisms of SAIL, we would like to discuss an implementation example that should illustrate some of the aforementioned mechanisms. In order to do this we will show how a non-trivial metrics like Tight Class Cohesion (**TCC**) metric [1] can be implemented in SAIL. **TCC** is defined as the fraction of the number methods pairs in a class, that are connected through an access to a common instance variable. Please note that the numbers in the below TCC metric's implementation represent the line numbers of the code.

In 1 we define a function `tcc` that returns for the parameter `class` the TCC's value. From 3 to 7 a lot of variables are defined. In 3, for example, we have a collection and it's type is `Method`. Being a part of the *meta-model*, the structure `Method` is a predefined one. It models a method as defined in an object-oriented language. In the used *meta-model* a class has methods, a method has a method-body (predefined type `MethodBody`) and among others a method-body contains a collection of attributes (predefined type `Attribute`) accessed in that method.

Class constructors aren't counted and that's the reason of selecting in 8 only the methods whose attribute `isConstructor` is false. The `select` instruction, as you see, is in fact a mechanism for *shaping* new collections from existing ones, and *filling* them with information primarily taken from the initial collections.

If a class doesn't have methods, we aren't allowed to perform the division from 17 and that's why we have to find out the number of methods the `class` has. The number of elements in a collection (in this case the methods of a class) is easily obtained using the `#` operator for collections.

Because we need to count pairs of methods having common accessed attributes, in 11 and 12 we iterate through the methods and for each pair we retain in 14 the common attributes. Due to the used *meta-model* the expression `m1.body.attributeAccesses` returns a collection of the method's `m1` accessed attributes. The common attributes are got by the intersection `m1.body.attributeAccesses * m2.body.attributeAccesses` from 14. We count in 16 if both methods of the pair use the same attributes.

Finally we have the `tcc`'s value in 17 and the last thing to do is to return it. The operation is performed in 20.

```
1 float tcc(Class class)
2 {
3   Method[] methodsWithoutConstructors;
4   float tcc;
5   int count;
6   Method m1,m2;
7   Attribute[] commonAttributes;

8   methodsWithoutConstructors = select (*) from class.methods
9                               where isConstructor == false;

10  if (methodsWithoutConstructors.# > 1) {
11    iterate(methodsWithoutConstructors, m1) {
12      iterate(methodsWithoutConstructors, m2) {
13        if (m1 != m2) {
14          commonAttributes = m1.body.attributeAccesses *
15                            m2.body.attributeAccesses;

16          if (commonAttributes.# != 0) { count++; }}}

17    tcc = (100 * count) / (methodsWithoutConstructors.# *
18                          (methodsWithoutConstructors.# - 1));
19  }
20  return tcc;
21 }
```

Conclusions and Future Works We started this paper by emphasizing the importance of a homogenous manner of expression for analyses that address the quality of object-oriented design and by pointing out the state-of-the-art problems. We identified the criteria of SAIL and we briefly presented how the key language mechanisms support them and, in order to demonstrate its applicability and functionality, we provide an example.

We need to implement more different analysis in order to validate the expressivity and efficiency of SAIL. At last, we intend to incorporate SAIL into an specialized environment for object-oriented systems' understanding and enhancement.

References

- [1] J.M. Bieman, B.K. Kang. Cohesion and Reuse in an Object-Oriented System. *Proc. ACM Symposium on Software Reusability*, April 1995.
- [2] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. *Technology of Object-Oriented Languages and Systems - TOOLS 30*, pages 18-32, Santa Barbara, CA, August 1999.
- [3] Daniel Jackson, Martin Rinard. Software Analysis: A Roadmap. *International Conference on Software Engineering*, pages 133-145, June 2000.
- [4] Radu Marinescu. A Survey on Object-Oriented Measurement in the Context of Reengineering. *PhD Presentation*, Timișoara, February 2000.
- [5] Radu Marinescu. Design Flaws and Detection Strategies. *PhD Presentation*, Timișoara, June 2001.