

# Reusing Static Analysis across Different Domain-Specific Languages using Reference Attribute Grammars (Artefact)

Johannes Mey, Thomas Kühn, René Schöne, and Uwe Aßmann

This artefact contains the source code, measurement environment and measurement data of the evaluation of the paper with the same name.

A git repository containing this artefact is available at <https://git-st.inf.tu-dresden.de/jastadd/reusable-analysis/>

## Abstract of the Publication

*Context: Domain-specific languages (DSLs) enable domain experts to specify tasks and problems themselves, while enabling static analysis to elucidate issues in the modelled domain early. Although language workbenches have simplified the design of DSLs and extensions to general purpose languages, static analyses must still be implemented manually.*

*Inquiry: Moreover, static analyses, e.g., complexity metrics, dependency analysis, and declaration-use analysis, are usually domain-dependent and cannot be easily reused. Therefore, transferring existing static analyses to another DSL incurs a huge implementation overhead. However, this overhead is not always intrinsically necessary: in many cases, while the concepts of the DSL on which a static analysis is performed are domain-specific, the underlying algorithm employed in the analysis is actually domain-independent and thus can be reused in principle, depending on how it is specified. While current approaches either implement static analyses internally or with an external Visitor, the implementation is tied to the language's grammar and cannot be reused easily. Thus far, a commonly used approach that achieves reusable static analysis relies on the transformation into an intermediate representation upon which the analysis is performed. This, however, entails a considerable additional implementation effort.*

*Approach: To remedy this, it has been proposed to map the necessary domain-specific concepts to the algorithm's domain-independent data structures, yet without a practical implementation and the demonstration of reuse. Thus, we employ relational Reference Attribute Grammars (RAGs) by creating such a mapping to a domain-independent overlay structure using higher-order attributes.*

*Knowledge: We describe how static analysis can be specified on analysis-specific data structures, how relational RAGs can help with the specification, and how a mapping from the domain-specific language can be performed. Furthermore, we demonstrate how a static analysis for a DSL can be externalized and reused in another general purpose language.*

*Grounding: The approach was evaluated using the RAG system JastAdd. To illustrate reusability, we implemented two analyses with two addressed languages each: (1) a cycle detection analysis used in a small state machine DSL and for detecting circular dependencies between Java types and packages, as well as (2) an analysis of variable shadowing applied to both Java and the Modelica modelling language. Thereby, we demonstrate the reuse of two analysis algorithms in three completely different domains. Additionally, we use the cycle detection analysis to evaluate the efficiency by comparing our external analysis to an internal reference implementation analysing all Java programs in the Qualitas Corpus. Our evaluation indicates that an externalized analysis incurs only minimal overhead.*

*Importance: We make static analysis reusable for both DSLs and general purpose languages, showing the practicality and efficiency of externalizing static analysis using relational RAGs.*

## Running the Evaluation

### Preparation for the Qualitas Corpus

Note, that for all parts involving Java, the Qualitas Corpus is required to be downloaded first. To do that:

1. Visit <http://qualitascorpus.com/docs/faq.html#download> and follow the instructions there to download both parts of the archive.
2. Unpack the archives.
3. Create a symlink to `QualitasCorpus-$VERSION/Systems/` named `qualitas` (alternatively, or if Docker does not follow symbolic links for security reasons, create a directory named `qualitas` and copy all systems into it)
4. Create a directory `docker-results` which serves as a shared directory with the container.

### Preparation for the Docker Image

Load the docker image using `docker load --input reusable-analysis.tar`

### Running the Docker Container

Run the container using

```
docker run --rm -it -v "$PWD"/docker-results:/reusable-analysis/benchmark:Z  
-v "$PWD"/qualitas:/reusable-analysis/qualitas:Z reusable-analysis
```

### Inside the Container

For convenience, there are several scripts to execute parts of the evaluation found in the paper named after the respective part and start with `./run_`. The main evaluation is performed with `./run_scc_java`. Those scripts internally call the correct Gradle tasks.

## Measured Data

The measurements were performed on an Intel i7-8700 workstation with 64GB of memory using Fedora Linux 29 running on kernel 4.18, OpenJDK 1.8 and JastAdd 2.3.

The data obtained using the provided artefact are contained in the file `measured_results.csv`.

This file is comma-separated file with the following columns:

	Title	Explanation
1	Domain	language on which the analysis is performed
2	Analysis	kind of analysis ( <b>type</b> or <b>package</b> )
3	Internal	<b>true</b> if direct, <b>false</b> if reusable
4	JavaFiles	number of files analyzed
5	Nodes	number of nodes in the dependency graph
6	Edges	number of edges in the dependency graph
7	NodesAndEdges	number of elements in the dependency graph
8	SCCs	number of computed SCCs
9	FullTime	total time of the run (parse+generation+analysis)
10	ParseTime	parse time
11	GenerationTime	generation time of the problem-specific structure
12	AnalysisTime	analysis time
13	GenAnaTime	sum of generation and analysis time
14	Run	number of the run (0-100)
15	Scenario	name of the analyzed program
16	TimeTime	wall-clock time as taken by the <b>time</b> command
17	Exit	return value of the benchmark run (always 0)