<Programming> 2020

Reusing Static Analysis across Different Domain-Specific Languages using Reference Attribute Grammars

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

* Technische Universität Dresden † Karlsruhe Institute of Technology

The Art, Science, and Engineering of Programming 4.3 (Feb. 17, 2020).
ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2020/4/15.

Problem: Reuse of Domain-Specific Analysis

Goal: Improving Reusability of Static Analysis for DSLs

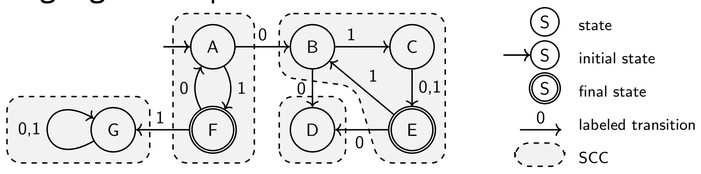
- different languages require same or similar analyses
- analyses not easily reusable since defined on domain-specific concepts

Example: Algorithm for cycle analysis using strongly connected components within state machines, highlighting domain-specific parts
(excerpt from analysis for state machines)

```
void State.traverse(Map<State, Set<State>> visited,
    Deque<State> locked) {
  visited.put(this, null);
  for (Transition t : getOutgoingList()) {
    if (!visited.containsKey(t.getTo())) {
      t.getTo().traverse(visited, locked);
    }
  }
  locked.addFirst(this);
}
```

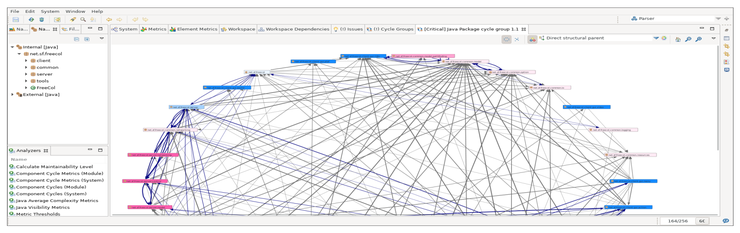
Example Analysis: Cycle Detection

Language 1: Simple State Machine DSL



Language 2: Java (For Packages *and* Classes)

- *same analysis* on *multiple structures* within a language
- standard analysis in analysis tool, e.g., *SonarQube*



Example Analysis: Variable Shadowing

Language 1: Java

```
public class A {
  protected int x; // field
  public A(int x) { // parameter
    this.x = x;
  }
  void m() {
    int x = 3; // local variable
  }
}
```

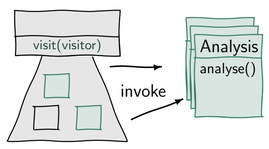
```
public class B extends A {
  int x = 4;              // field
  class C {
    private int x = 5; // field
  }
}
```

Language 2: Modelica

```
model EnclosingClassLookupShadowedConstant
  constant Real x = 4.0; // declare constant variable x1
  model A
    Real x = 3.0;     // declare variable x2, shadowing x1 in line 2
    model B
      Real y = x;     // refers to x2 in line 4 illegally, since
    end B;            // references to enclosing scopes must be constant
    B b;
  end A;
  A a;
end EnclosingClassLookupShadowedConstant;
```

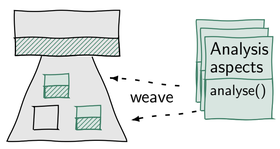
Static Analysis Approaches

External Visitor



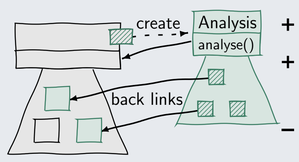
− *domain-specific* analysis
\+ separated analysis and traversal

Attribute Grammar



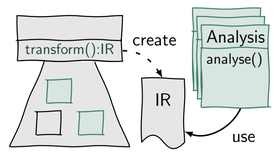
− *domain-specific* analysis
− combined analysis and traversal

Higher-Order Attribute and Relations



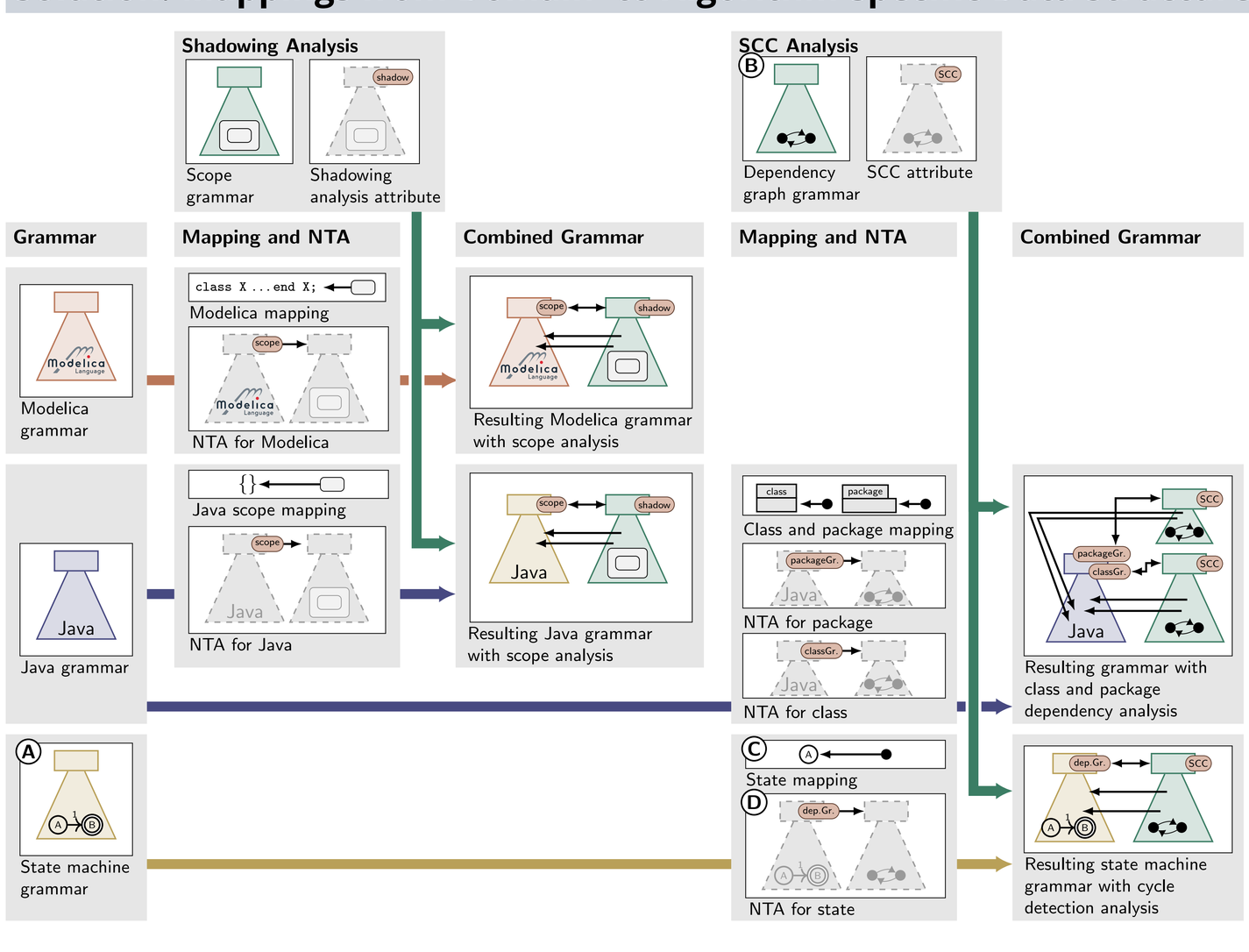
\+ *problem-specific* analysis
\+ separated analysis and traversal
− problem-specific mapping/view required

Intermediate Representation



\+ *domain-independent* analysis
− requires construction of IR

Solution: Mappings from Domain- to Algorithm-specific Data Structure using Relational RAGs



Source Code for State Machine Dependency Analysis

State machine grammar

A
```
StateMachine ::= Element*;
abstract Element ::= <Label:String>;
State : Element;
Transition : Element;
rel Transition.From <-> State.Outgoing*;
rel Transition.To <-> State.Incoming*;
rel StateMachine.Initial -> State;
rel StateMachine.Final* -> State;
```

Dependency graph grammar

B
```
DependencyGraph ::= Component*;
Component;
rel Component.From* <-> Component.To*;
```

Mapping from problem-specific to domain-specific grammar

C
```
rel Component.State -> State;
rel DependencyGraph.StateMachine -> StateMachine;
```

Higher-order (nonterminal) attribute to derive the dependency analysis structure

D
```
syn lazy DependencyGraph StateMachine.dependencyGraph() {
  DependencyGraph dg = new DependencyGraph();
  dg.setStateMachine(this);
  Map<State,Component> componentMap = new HashMap<>();
  for (State s: states()) {
    Component n = new Component();
    n.setState(s);
    dg.addComponent(n);
    componentMap.put(s, n);
  }
  for (Transition t: transitions())
    componentMap.get(t.getFrom()).addTo(componentMap.get(t.getTo()));
  return dg;
}
```

Relational Reference Attribute Grammar System

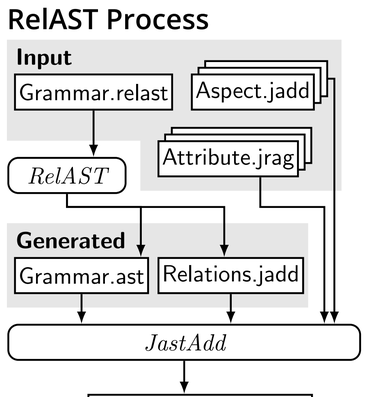
JastAdd RAG System [1]

- Java-based reference attribute grammars
- definition of derived attributes in *aspects*
- attributes: synthesized, inherited, reference, higher-order, collection, circular, …

Relational RAGs Preprocessor *RelAST* [2]

- preprocessor producing additional aspects
- API for navigating and editing bidirectional noncontainment relations

RelAST Process



RelAST Syntax

```
// grammar

Root ::= A* B*;
A ::= <Name:String>;
B ::= <Name:String>;


// relations

rel A.r1  -> B;
rel A.r2? -> B;
rel A.r3* <-> B.r4*;

//          direction
//   cardinality
```

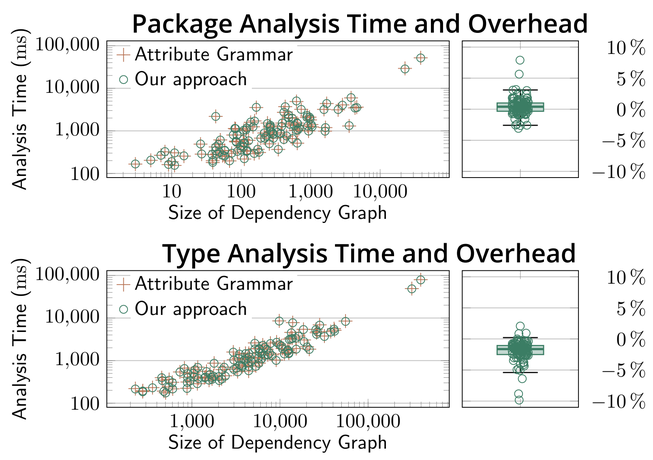
Evaluation Results for Cycle Analysis for Java

Setup

- 112 well-known Java programs taken from the Qualitas Corpus (`www.qualitascorpus.com`)
- analysis time after parsing all source files in each program
- baseline: attribute grammar-based approach working directly on the domain-specific structure

Results

- negligible performance penalty or even speed-up observed



References

[1] Görel Hedin and Eva Magnusson. "JastAdd: an aspect-oriented compiler construction system". In: *Science of Computer Programming* 47.1 (2003), pp. 37–58. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(02)00109-0.

[2] Johannes Mey et al. "Relational Reference Attribute Grammars: Improving Continuous Model Validation". In: *Journal of Computer Languages* 57 (Jan. 20, 2020). ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.100940.

Contact:
Johannes Mey johannes.mey@tu-dresden.de
Thomas Kühn thomas.kuehn@kit.edu
René Schöne rene.schoene@tu-dresden.de
Uwe Aßmann uwe.assmann@tu-dresden.de

Artefact:
https://git-st.inf.tu-dresden.de/jastadd/reusable-analysis/
https://zenodo.org/record/3659198
DOI 10.5281/zenodo.3659198

Acknowledgments:
This work is partly supported by the German Research Foundation (DFG) in the project "RISCOS" and as part of Germany's Excellence Strategy – EXC 2050/1 (CeTI), and by the German Federal Ministry of Education and Research within the projects "OpenLicht" and "KASTEL".