Relational Reference Attribute Grammars

Dresden, December 19, 2019
From Model Refactoring to Relational RAGs
Continuously Changing Models

Models:

- **Analyze**
  *Here*: search for refactoring candidates

- **Modify**
  *Here*: apply refactoring
Continuously Changing Models

Models:

— **Analyze**
  
  *Here:* search for refactoring candidates

— **Modify**
  
  *Here:* apply refactoring
Continuously Changing Models

Models:

- Analyze
  *Here*: search for refactoring candidates

- Modify
  *Here*: apply refactoring
Continuously Changing Models

Models:

- **Analyze**
  *Here*: search for refactoring candidates

- **Modify**
  *Here*: apply refactoring
Continuously Changing Models

Models:
- Analyze
  *Here*: search for refactoring candidates
- Modify
  *Here*: apply refactoring

Models at runtime:
- Analyze *incrementally*
- Modify *continuously*
Reference Attribute Grammars as Models

Our approach:
Reference Attribute Grammars (RAGs)
  — Structure: context-free grammar
  — Analysis: attributes
  — Refactoring: tree edits
  — We use: JastAdd

RAGs for modelling offer:
  — Shorthands for navigation and computation on trees
  — Efficiency through memoization
  — Incremental evaluation
Reference Attribute Grammars as Models

Our approach:
Reference Attribute Grammars (RAGs)
  - **Structure**: context-free grammar
  - **Analysis**: attributes
  - **Refactoring**: tree edits
  - We use: JastAdd

RAGs for modelling offer:
  - Shorthands for navigation and computation on trees
  - Efficiency through memoization
  - Incremental evaluation
Reference Attribute Grammars as Models

Our approach:
Reference Attribute Grammars (RAGs)
  — **Structure**: context-free grammar
  — **Analysis**: attributes
  — **Refactoring**: tree edits
  — We use: **JastAdd**

RAGs for modelling offer:
  — Shorthands for navigation and computation on trees
  — Efficiency through memoization
  — Incremental evaluation
Model-Grammar Mismatch

Relations are different:

— In models:
  - Containment relations form *overlay tree*
  - Non-containment relations
  - Bidirectional relations

— In grammars:
  - Containment references: AST
  - Non-containment references
  - Bidirectional references

}\{ Relational RAGs
Models vs Relational RAGs
Metamodels and Grammars

Abstract grammar (JastAdd syntax)

```plaintext
RailwayContainer ::= Route* Region*;
abstract RailwayElement ::= <Id:int>;
Region : RailwayElement ::= TrackElement* Sensor*;
Semaphore : RailwayElement ::= <Signal:Signal>;
Route : RailwayElement ::= <Active:boolean> SwitchPosition*;
SwitchPosition : RailwayElement ::= <Position:Position>;
Sensor : RailwayElement;
abstract TrackElement:RailwayElement;
Segment : TrackElement ::= <Length:int> Semaphore*;
Switch : TrackElement ::= <CurrentPosition:Position>;
```

How to capture non-containment relations?

Reference Attribute Grammars with JastAdd
Johannes Mey, René Schöne, Uwe Aßmann, Niklas Fors, Görel Hedin
Dresden, December 19, 2019
Metamodels and Grammars

Abstract grammar (JastAdd syntax)

RailwayContainer ::= Route* Region*;
abstract RailwayElement ::= <Id:int>;
Region : RailwayElement ::= TrackElement* Sensor*;
Semaphore : RailwayElement ::= <Signal:Signal>;
Route : RailwayElement ::= <Active:boolean> SwitchPosition*;
SwitchPosition : RailwayElement ::= <Position:Position>;
Sensor : RailwayElement;
abstract TrackElement:RailwayElement;
Segment : TrackElement ::= <Length:int> Semaphore*;
Switch : TrackElement ::= <CurrentPosition:Position>;

How to capture non-containment relations?
Handling Non-containment Relations in RAGs

Approach 1: Name analysis
- Unique identifier Id for each object
- Non-containment relations as Id uses
- Resolve with name analysis attributes

Approach 2: Explicit intrinsic reference attributes
- Store references as (Java) object references
- Resolve during model loading

Problem: Bidirectional relations:
- Either use collection attributes to reverse references (slow!)
- Or two unidirectional relations (risk of inconsistency!)
Non-Containment Relations in Detail

Non-Containment References

Cardinality
- $1 : 1$
- $1 : \{0..1\}$
- $1 : N$

Bidirectional References
Non-Containment Relations in Detail

Non-Containment References:
- **In RAGs**: typed reference nodes: \[ R \]

Cardinality:
- \( 1 : 1 \)
- \( 1 : \{0..1\} \)
- \( 1 : N \)
- **In RAGs**: optional \[ O \] and list nodes: \[ L \]

Bidirectional References
Non-Containment Relations in Detail

Non-Containment References:

— **In RAGs**: typed reference nodes: \( R \)

Cardinality:

— \( 1 : 1 \)
— \( 1 : \{0..1\} \)
— \( 1 : N \)

— **In RAGs**: *optional* \( O \) and *list nodes*: \( L \)

Bidirectional References:

— **In RAGs**:
  
  One direction in grammar,
  
  the other in grammar or attribute
Solution: Relational RAGs

Abstract grammar
RailwayContainer ::= Route* Region*;
abstract RailwayElement ::= Id:int;
Region : RailwayElement ::= TrackElement* Sensor*;
Semaphore : RailwayElement ::= Signal:Signal;
Route : RailwayElement ::= Active:boolean SwitchPosition*;
SwitchPosition : RailwayElement ::= Position:Position;
Sensor : RailwayElement;
abstract TrackElement:RailwayElement;
Segment : TrackElement ::= Length:int Semaphore*;
Switch : TrackElement ::= CurrentPosition:Position;
Solution: Relational RAGs

Abstract grammar
RailwayContainer ::= Route* Region*;
abstract RailwayElement ::= <Id:int>;
Region : RailwayElement ::= TrackElement* Sensor*;
Semaphore : RailwayElement ::= <Signal:Signal>;
Route : RailwayElement ::= <Active:boolean>
    SwitchPosition*;
SwitchPosition : RailwayElement ::= <Position:Position>;
Sensor : RailwayElement;
abstract TrackElement:RailwayElement;
Segment : TrackElement ::= <Length:int> Semaphore*;
Switch : TrackElement ::= <CurrentPosition:Position>;

Extending RAGs with relations
rel Route.requires* -> Sensor;
rel Route.entry? -> Semaphore;
rel Route.exit? -> Semaphore;
rel SwitchPosition.target <-> Switch.positions*;
rel Sensor.monitors* <-> TrackElement.monitoredBy*;
rel TrackElement.connectsTo* -> TrackElement;
Solution: Relational RAGs

Modifying relations

```java
public java.util.List<TrackElement> Sensor.getMonitors() {
    return Collections.unmodifiableList(get_impl_monitors());
}
```

```java
public void Sensor.addMonitors(TrackElement o) {
    ArrayList<TrackElement> list = get_impl_monitors();
    ArrayList<Sensor> list2 = o.get_impl_monitoredBy();
    list.add(o);
    list2.add(this);
    set_impl_monitors(list);
    o.set_impl_monitoredBy(list2);
}
```

```java
public void Sensor.removeMonitors(TrackElement o) {
    ArrayList<TrackElement> list = get_impl_monitors();
    if (list.remove(o)) {
        ArrayList<Sensor> list2 = o.get_impl_monitoredBy();
        list2.remove(this);
        set_impl_monitors(list);
        o.set_impl_monitoredBy(list2);
    }
}
```
Solution: The RelAST Preprocessor

Automatically generates
- Grammar with non-containment relations
- Accessor attributes
- Setter attributes
  - Ensuring consistency for bidirectional relations
- Optionally
  - Serialization and deserialization methods
  - Parsing support for non-containment relations
Extension: Serialization and Deserialization

Problem:
- Intrinsic relations must be resolved during parsing before the computed attributes are evaluated

Solution: (De-)Serialization Component
- Generate (de-)serialization components that handle this automatically
- Result: problem-specific JSON notation

```json
{  
  "type": "Switch",  
  "id": "15",  
  "children": {  
    "Id": 52,  
    "CurrentPosition": "STRAIGHT"  
  },  
  "relations": {  
    "connectsTo": ["201"],  
    "monitoredBy": ["18", "19", "20", "21",  
                    "22", "23", "24", "25"],  
    "positions": ["14"]  
  }  
}
```
Extension: Ecore to Relational RAGs
Automatically!

JastAdd Grammar

RailwayContainer ::= Route* Region*;
abstract RailwayElement ::= <Id:int>;
Region : RailwayElement ::= TrackElement* Sensor*;
Semaphore : RailwayElement ::= <Signal:Signal>;
Route : RailwayElement ::= <Active:boolean>
  SwitchPosition*;
SwitchPosition : RailwayElement ::= <Position:Position>;
Sensor : RailwayElement;
abstract TrackElement:RailwayElement;
Segment : TrackElement ::= <Length:int> Semaphore*;
Switch : TrackElement ::= <CurrentPosition:Position>;

JastAdd Relations
rel Route.requires* -> Sensor;
rel Route.entry? -> Semaphore;
rel Route.exit? -> Semaphore;
rel SwitchPosition.target <-> Switch.positions*;
rel Sensor.monitors* <-> TrackElement.monitoredBy*;
rel TrackElement.connectsTo* -> TrackElement;
Extension: Ecore to Relational RAGs

Relations in Ecore:

**EClass**
- becomes nonterminal
- multiple inheritance

**EAttribute**
- become terminal

**EReference**
- becomes child
  - if containment is true
- becomes relation
  - if containment is true
- various properties

Reference Attribute Grammars with JastAdd
Johannes Mey, René Schöne, Uwe Aßmann, Niklas Fors, Görel Hedin
Dresden, December 19, 2019

https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html
Continuous Model Validation using Reference Attribute Grammars

Johannes Mey
Technische Universität Dresden
Germany
johannes.mey@tu-dresden.de

Emma Söderberg
Lund University
Sweden
emma.soderberg@cs.lth.se

Thomas Kühn
Technische Universität Dresden
Germany
thomas.kuehn@tu-dresden.de

René Schöne
Technische Universität Dresden
Germany
rene.schoene@tu-dresden.de

Jesper Öqvist
Lund University
Sweden
jasper.oqvist@cs.lth.se

Niklas Fors
Lund University
Sweden
niklas.fors@cs.lth.se

Görel Hedin
Lund University
Sweden
gorel.hedin@cs.lth.se

Emma Söderberg
Lund University
Sweden
emma.soderberg@cs.lth.se

Johannes Mey, René Schöne, Uwe Aßmann, Niklas Fors, Görel Hedin

1 Introduction

More and more software systems rely on models to easily reference, refine, and validate aspects of a business domain in a cost-effective way [12]. With current software systems increasing in complexity and rate of change [13], these models become more complex and change continuously, too. While maintaining and refining complex models is possible with state-of-the-art tools [12], their continuous evaluation and validation still poses problems for large complex models. To approach continuous evaluation, researchers recently applied Reference Attribute Grammars (RAG) [14] to encode and validate models, e.g., [6-8], because RAG systems offer mechanisms to perform an incremental analysis efficiently using dynamic dependency tracking [15]. Although RAG systems can efficiently iterate and re-evaluate complex, large tree structures with derived information, including references, there exists a fundamental semantic mismatch between models, generally represented as graphs, and RAG trees. While conceptual models comprise classes with attributes inherited by inheritance, containment, and non-containment

Abstract

Just like current software systems, models are characterised by increasing complexity and rate of change. Yet, these models only become useful if they can be continuously evaluated and validated. To achieve sufficiently low response times for large models, incremental analysis is required. Reference Attribute Grammars (RAGs) offer mechanisms to perform an incremental analysis efficiently using dynamic dependency tracking. However, not all features used in conceptual modeling are directly available in RAGs. In particular, support for non-containment model relations is only available through manual implementation. We present an approach to directly model such and bidirectional non-containment relations in RAGs and provide efficient means for navigating and editing them. This approach is evaluated using a scalable benchmark for incremental model editing and the JastAdd RAG system. Our work demonstrates the suitability of RAGs for validating complex and continuously changing models of current software systems.

CSE Concepts: Theory of computation → Grammars and context-free languages; Software and its engineering → System description languages; Computing methodologies → Static verification and validation.

1 There is a striking similarity to the object-relational impedance mismatch [18].

70

---

Reference Attribute Grammars with JastAdd
Johannes Mey, René Schöne, Uwe Aßmann, Niklas Fors, Görel Hedin
Dresden, December 19, 2019

---

Continuous Model Validation using Reference Attribute Grammars

1 Introduction

More and more software systems rely on models to easily reference, refine, and validate aspects of a business domain in a cost-effective way [12]. With current software systems increasing in complexity and rate of change [13], these models become more complex and change continuously, too. While maintaining and refining complex models is possible with state-of-the-art tools [12], their continuous evaluation and validation still poses problems for large complex models. To approach continuous evaluation, researchers recently applied Reference Attribute Grammars (RAG) [14] to encode and validate models, e.g., [6-8], because RAG systems offer mechanisms to perform an incremental analysis efficiently using dynamic dependency tracking [15]. Although RAG systems can efficiently iterate and re-evaluate complex, large tree structures with derived information, including references, there exists a fundamental semantic mismatch between models, generally represented as graphs, and RAG trees. While conceptual models comprise classes with attributes inherited by inheritance, containment, and non-containment

Abstract

Just like current software systems, models are characterised by increasing complexity and rate of change. Yet, these models only become useful if they can be continuously evaluated and validated. To achieve sufficiently low response times for large models, incremental analysis is required. Reference Attribute Grammars (RAGs) offer mechanisms to perform an incremental analysis efficiently using dynamic dependency tracking. However, not all features used in conceptual modeling are directly available in RAGs. In particular, support for non-containment model relations is only available through manual implementation. We present an approach to directly model such and bidirectional non-containment relations in RAGs and provide efficient means for navigating and editing them. This approach is evaluated using a scalable benchmark for incremental model editing and the JastAdd RAG system. Our work demonstrates the suitability of RAGs for validating complex and continuously changing models of current software systems.

CSE Concepts: Theory of computation → Grammars and context-free languages; Software and its engineering → System description languages; Computing methodologies → Static verification and validation.

1 There is a striking similarity to the object-relational impedance mismatch [18].
Relational RAGs for (Runtime) Models: An Example Use Case

Running example: Modeling train tracks and routes.

Example model, from [Szárnyas et al., 2017]

Use Case:

— Modeling editor for rail networks
— Continuously find and repair faults

Relational RAGs for (Runtime) Models: An Example Use Case

Running example: Modeling train tracks and routes.

Use Case:

— Modeling editor for rail networks
— Continuously find and repair faults

Example model, from [Szárnyas et al., 2017]

Relational RAGs for (Runtime) Models: An Example Use Case

Running example: Modeling train tracks and routes.

Use Case:

— Modeling editor for rail networks
— Continuously find and repair faults

Reference:

Relational RAGs: An Evaluation

We investigated:

1. **Usability and Conciseness**
   - Measure complexity reduction

2. **Performance**
   - Compare the three approaches with model- and graph-based solutions

Use Case:
- Iterative model analysis and transformation with *Train Benchmark* [Szárnyas et al., 2017]
- Six model queries
- Fault injection and repair transformations for each

Questions?

event-place: Boston, MA, USA.