Continuous Model Validation using Reference Attribute Grammars

Mey, Johannes; Schöne, René; Hedin, Görel; Söderberg, Emma; Kühn, Thomas; Fors, Niklas; Öqvist, Jesper; Aßmann, Uwe

Published in:
Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18)

DOI:
10.1145/3276604.3276616

2018

Continuous Model Validation using Reference Attribute Grammars

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

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

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

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

Thomas Kühn
Technische Universität Dresden
Germany
thomas.kuehn3@tu-dresden.de

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

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

Uwe Aßmann
Technische Universität Dresden
Germany
uwe.assmann@tu-dresden.de

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 modelling 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 uni- 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.

CCS Concepts • **Theory of computation → Grammars and context-free languages**; • **Software and its engineering →** *System description languages*; • **Computing methodologies →** *Model verification and validation*;

SLE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276616

Keywords Incremental model evaluation, bidirectional relations, References Attribute Grammars

ACM Reference Format:
Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. 2018. Continuous Model Validation using Reference Attribute Grammars. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276604.3276616

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 [32]. With current software systems increasing in complexity and rate of change [28], these models become more complex and change continuously, too. While maintaining and refining complex models is possible with state-of-the-art tools [22], their continuous evaluation and validation still poses problems for large complex models.

To approach continuous evaluation, researchers recently applied *Reference Attribute Grammars* (RAGs) [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 [35]. Although RAG systems can efficiently rewrite 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.[1] While conceptual models comprise classes with attributes linked by inheritance, containment, and non-containment

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

relations, RAGs feature production rules for abstract syntax trees with nonterminal elements and tokens, as well as attributes defining, amongst other things, cross-tree references.[2] Even though containment relations can be directly mapped to nonterminal productions rules, non-containment relations must be encoded manually by means of reference attributes [7]. Moreover, bidirectional non-containment relations cannot be mapped to RAGs without either requiring manually defined lookup (potentially causing excessive evaluation overhead) or the unmanaged redundancy of two opposing directed relations. Consequently, naive implementations usually incur a performance overhead that reduces the benefits of incremental model evaluation. In short, plain RAGs are not directly usable for continuous validation of conceptual models. To remedy this mismatch, we address the following research questions in this paper:

RQ1 How can RAGs be used to define models?
RQ2 What could improve the suitability of RAGs?
RQ3 Can improvements retain the efficiency of incremental RAG evaluations?

Consequently, we give a detailed account of typical issues of models encoded with RAGs (RQ1). We propose an extension to RAGs introducing a coherent notation for uni- and bidirectional non-containment relations, bridging the semantic gap between conceptual models and RAGs (RQ2). Moreover, we provide a general implementation of bidirectional non-containment relations for RAGs that permits editing and navigating these relations consistently and efficiently (RQ3). Specifically, we provide a prototypical implementation based on *JastAdd* [15], a system supporting RAGs, illustrating the suitability of our approach. Investigating the efficiency of our solution in the presence of incremental evaluation, we employ the *Train Benchmark* [38] comparing the proposed solution with both an idiomatic implementation using simple lookup methods and a manually optimized implementation.

The paper is structured as follows. Section 2 introduces the model of the *Train Benchmark* used throughout the paper as the running example. Section 3 gives background on models and RAGs. Section 4 investigates typical encodings of models with RAGs. Conversely, Section 5 introduces bidirectional relations to RAGs and describes the prototypical implementation with *JastAdd*. Afterwards, Section 6 evaluates the suitability and efficiency of the presented solution both qualitatively and quantitatively. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 Running Example: Train Benchmark

To showcase the mismatch between RAGs and conceptual modeling, we implement a continuous model validation benchmark using plain RAGs and compare this to using

[2]Note that the term *attribute* is used both in the modelling and the attribute grammar community with different meanings (cf. Section 3).

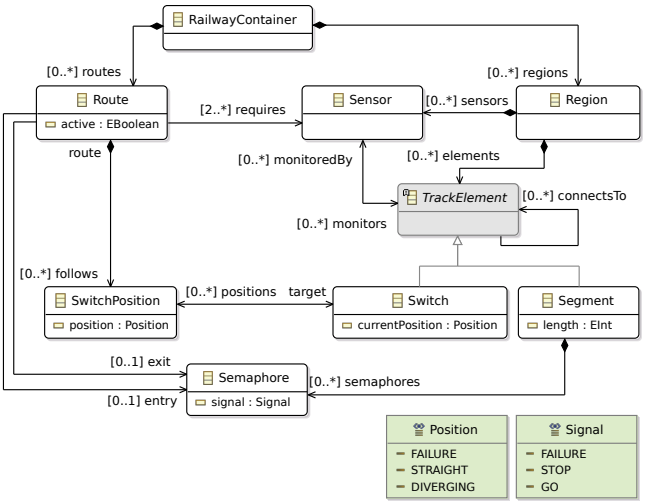


Figure 1. Metamodel, adapted from [38].

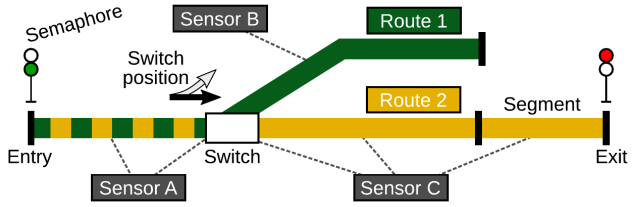


Figure 2. Example of a model instance, from [38].

RAGs extended with non-containment relations. The benchmark used is the *Train Benchmark* [38], in which a railway network is modeled and validated using well-formedness constraints. The hypothetical use case of the benchmark domain is an interactive editor for railway networks, which are alternatingly modified and validated.

Six patterns are described to find matches of inconsistent model states, and transformations are given to either *inject* more faults or *repair* existing ones.

2.1 The Train Metamodel

Figure 1 depicts the metamodel of the *Train Benchmark*. It defines `TrackElements`, either `Switches` or `Segments`, which are monitored by `Sensors`. Furthermore, `Routes` are specified between `Semaphores`, following switches with certain `SwitchPositions`. Additional properties are defined using attributes, which have either built-in data types or enumeration types. Furthermore, and for clarity reasons omitted in the diagram, each model element specializes an abstract superclass `ModelElement` and has an integer-valued `id`.

The model is structured in a containment tree with a `RailwayContainer` root and several non-containment relations between classes, e.g., the `requires` relation between a `Route` and a `Sensor`, or the bidirectional relation `monitors`/`monitoredBy` between `Sensor` and `TrackElement`.

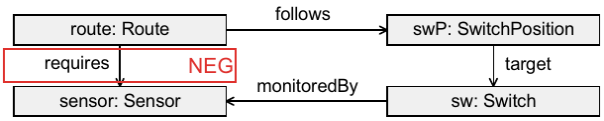


Figure 3. *RouteSensor* query, from [38].

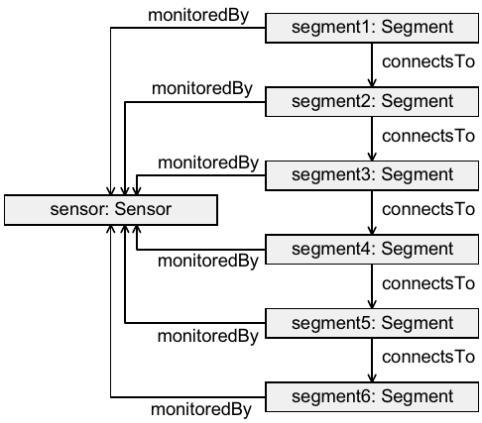


Figure 4. *ConnectedSegments* query, from [38].

Figure 2 shows an instance of the metamodel defining two routes sharing a common segment.

2.2 Investigated Queries and Transformations

The *Train Benchmark* defines six queries to cover a range of different query patterns. Each query defines an undesired pattern of objects and relations. While we implemented all of them, in this work, we focus on two of the queries, *RouteSensor* and *ConnectedSegments*. Both make use of bidirectional references, the former features a negative match and few relations, while the latter is more complex. Thus, two very different exemplary queries have been selected, showing different performance properties, as is shown in Section 6.

RouteSensor. The query *RouteSensor* identifies a missing relation between a `Route` and a `Sensor`: A `Route` follows a number of switches that are monitored by sensors. Each such sensor is said to be required by the `Route`, and if that *requires* relation is missing, there is an error in the model. Note that using RAGs, the required sensors of a `Route` could be modelled by a derived attribute rather than by an explicit relation, making this query obsolete. We have chosen this query to show the effectiveness and efficiency of the bidirectional relation `monitoredBy`. The *inject* transformation randomly deletes one `requires` relation, whereas to repair, a match is found using the query of Figure 3 and a `requires` relation is added between the route and the sensor.

ConnectedSegments. Figure 4 shows the query *ConnectedSegments*. It matches any sensor monitoring more than five `Segments` connected in a row. The motivation for choosing this query was its unusual structure matching a certain number of elements of the same type, and its inherent complexity.

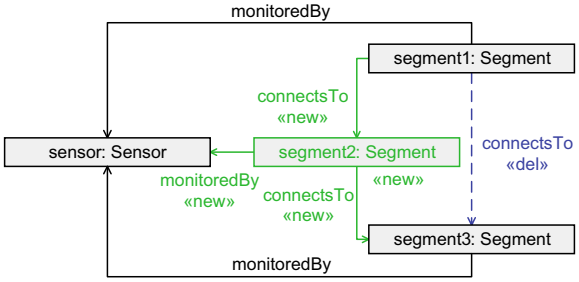


Figure 5. *ConnectedSegments* inject transformation, from [38].

To inject a potential fault, a new segment is inserted and connected between two existing segments monitored by the same sensor as shown in Figure 5. Repairing deletes the second segment of a match found by the query in Figure 4.

3 Background on Models and RAGs

This section presents an overview of models and Reference Attribute Grammars (RAGs).

3.1 Components of Structural Models

Models are considered in the commonly used format *Ecore*, the reference implementation of the Essential Meta-Object Facility (EMOF) standard by the Object Management Group (OMG) [30, 37]. Ecore, which acts as the core metamodel of the Eclipse Modeling Framework EMF, has been selected because of its wide usage in the modelling community. The basic required features of a conceptual model are classes with single inheritance, named and directed relations with cardinality, and attributes of classes. Models can have arbitrary relations between elements, resulting in object graphs. In contrast, trees (as described by abstract grammars) have only a hierarchical parent-child relation, and can thus be seen as special cases of models. The Ecore metamodel is particularly suited for mapping to a grammar because it requires a tree-shaped containment relation for metamodel elements:

Containment. Elements in the model are required to have at most one container (parent) reference. This means that the elements of an Ecore model form a set of trees with additional non-containment model relations.

Spanning Tree. For reasons of simplicity, it is assumed that each element has exactly one container reference, except for one singular element which we call the *root element*. This requirement can be achieved easily for models adhering to the containment requirement by adding an artificial root node containing all elements with no existing container. It can be noted that many models already have a spanning tree to support serialization into a tree-structured format like XML, JSON, or YAML.

3.2 Components of Reference Attribute Grammars

Reference Attribute Grammars (RAGs) [14] are abstract grammars extended with derived attributes. They generalize the original attribute grammars by Knuth [23] by supporting that attributes are references to tree nodes.

Abstract Grammar. An abstract grammar defines valid trees of objects (nodes), similar to the spanning trees defined by an Ecore model. For containment relations, multiplicities of `1`, `0..1`, and `0..*` are supported. The abstract grammar furthermore defines *intrinsic* attributes of objects, i.e., attributes that are given values when the object is created. When implementing programming languages, intrinsic attributes are typically used for representing tokens, with primitive types like `String` or `int`. They can also reference other objects, thereby giving certain support for non-containment relations. Listing 1 shows a simple example grammar with six classes (classes are also known as *nonterminals*).

Listing 1. Example grammar.

```
A ::= b:B ;      // class A with normal child b of type B
abstract B ::= <t:int> ; // abstract class with token t
C:B;             // class C extends B
D:B ::= [E] ;    // optional child (0..1 multiplicity)
E:B ::= F* ;     // list child (0..* multiplicity)
F   ::= <c:C> ; // intrinsic reference c of type C
```

Derived Attributes. Derived attributes in a RAG are defined by directed equations placed in classes, and whose right-hand sides may use any other attributes (intrinsic or derived) accessible from that class. The attributes may have primitive or reference values. In implementation of programming languages, the derived attributes are typically used for representing name bindings, types, validation errors (static-semantic errors), generated code, etc.

There are many kinds of derived attributes: *synthesized* and *inherited* like in Knuth's attribute grammars, *parameterized* [14], *higher-order* [41], *collection* [26], *circular* [27], etc., with slightly different rules for how equations are written and in what class they are placed. Listing 2 shows the definition of an example synthesized attribute.

Listing 2. Example synthesized attribute.

```
syn int F.d();       // F declares synthesized attribute d
eq F.d() {           // F has equation for d
  return getC().t(); // right-hand side given as method body
}
```

We use the notation of the RAG tool *JastAdd* [15] for the derived attributes, but the ideas presented in this paper are general and would apply to any RAG tool.

Attribute evaluation. The value of a derived attribute is computed automatically, on demand, when it is accessed the first time. The value can be cached to speed up subsequent accesses, thus enabling incremental evaluation. If a part of the tree is changed (called an "edit"), caches of attributes that depend on those parts are cleared. These attributes are then re-evaluated the next time they are accessed. Technically, the containment relation is represented by implicit intrinsic parent and child attributes, and a dependency graph over attribute instances (both intrinsic and cached derived) is computed dynamically, during attribute evaluation. Each edit corresponds to a number of changes to intrinsic attributes and the dependent derived attributes are found by traversing the dependency graph starting from the changed intrinsic attributes [35]. For the purpose of this paper, we are primarily taking advantage of simple synthesized and inherited attributes as well as their incremental evaluation to provide automatic support for continuous model validation.

Listing 3. Basic *Train Benchmark* grammar without non-containment relations.

```
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>;
```

4 Encoding Models in Plain RAGs

To encode a conceptual model as a RAG, each of the basic features of models need to be encoded: classes, single inheritance, attributes of classes, containment relations, and non-containment relations. Of these, the first four are straightforward, whereas the non-containment relations can be encoded in different ways. In this section, we will present different approaches to encode those relations, thus tackling RQ1.

4.1 Models without Non-containment Relations

We first look at how to encode models without non-containment relations. Model classes and their single inheritance translates directly to RAG classes and their single inheritance. Attributes of model classes translate to intrinsic RAG attributes. The containment relation translates to normal, optional, and list children in RAG classes. Listing 3 shows how the *Train Benchmark* model described in Section 2 can be written as a RAG (ignoring the non-containment relations). Here, `Signal` and `Position` are `enum` types with the possible values shown in Figure 1.

Containment relations. RAGs give immediate support for containment relations in the child direction, by generated accessor methods. For example, from a `Region` object `r`, the `TrackElements` can be accessed as follows:

```
r.getTrackElementLisr() // access all children
r.getTrackElement(i)    // access a specific child
```

The inverse relation, from child to parent, is supported directly by an intrinsic attribute `parent`.

4.2 Encoding Non-containment Relations

For non-containment relations, there is no obvious corresponding feature in plain RAGs, so they need to be encoded. This can be done either using names that must be resolved or using intrinsic attributes. To support all cardinalities and provide a common interface for both variants, we add an artificial generic wrapper class, `Ref`, to the grammar. The `Ref` node encapsulates one outgoing reference in a relation. Using these new `Ref` wrapper objects, we can extend the basic abstract grammar from Listing 3 as shown below.

Listing 4. Extending the *Train Benchmark* grammar with `Ref` nodes for non-containment relations.

```
Route ... ::= ... requires:Ref* [entry:Ref] [exit:Ref];
SwitchPosition ... ::= target:Ref;
Sensor ... ::= monitors:Ref*;
abstract TrackElement ... ::= connectsTo:Ref*;
```

Compared to the train model of Figure 1, the bidirectional relations (`monitors/monitoredBy` and `target/positions`), only encode one of the directions. We will discuss later how to handle bidirectional relations.

We have not yet shown the implementation of the class `Ref`. We will discuss two different strategies: one is based on name analysis, and the other on intrinsic reference attributes.

4.2.1 Reference Resolution and Name Analysis

Reference resolution is a common problem in both the grammar and the model world, since for both models and languages, serialization is commonly used, and references are represented as some kind of names or identifiers in the serialized form. For programming languages, resolving names can be a rather complex task, involving scoping and precedence rules. In models, on the other hand, name resolution is typically much easier. In Ecore, the most commonly used serialization is the XML Metadata Interchange (XMI) format. XMI supports both path expressions and global identifiers to describe references. However, we focus on global identifier resolution only, in order to simplify the problem. This name resolution is much simpler than in programming languages since it only involves a global identifier namespace, and no scoping or other complex mechanisms. All `RailwayElement` objects in the Train model have a unique identifier, represented by the intrinsic attribute `Id` in Listing 3.

Name Analysis Strategy. To use the name analysis strategy for resolving references, we simply let the `Ref` wrappers contain the target `Id` of the reference, implementing `Ref` as:

```
Ref ::= <Id:int>
```

The actual references making up the relation can then be computed using a simple name analysis, specified using derived RAG attributes. This implementation has the advantage that it makes it very simple to deserialize a model: simply create the containment tree of objects, setting their local `Id` attributes. The name analysis RAG attributes take care of the reference resolution completely automatically. A straightforward way to efficiently implement the name analysis using RAG attributes is to define one derived attribute in the root that computes a map from `Id` values to objects, and one derived attribute in the `Ref` object that looks up the appropriate `RailwayElement`, using the `Id` value of the `Ref`. Due to attribute caching, the map is computed only once.

Intrinsic Reference Strategy. As we will see in Section 6, however, the name analysis solution has some performance problems when using incremental evaluation. Therefore, we also consider an alternative strategy, namely to use intrinsic reference attributes, using the following implementation of the Ref wrapper:

```
Ref ::= <Ref:ASTNode>
```

For this strategy, the deserializer will be slightly more complex as it needs to explicitly set the intrinsic references. This can be done in a second pass after building the containment tree, i.e., in the same way as in an ordinary model tool. Serializing a model is very simple for both strategies.

Listing 5. RAG-based computation of inverse relation.

```
coll Set<Sensor> TrackElement.monitoredBy() [new HashSet()];
Sensor contributes this
  to TrackElement.monitoredBy()
  for each monitors();
```

4.2.2 Bidirectional References

So far, we have only discussed how to handle unidirectional non-containment relations. For bidirectional relations, there are two major options: double unidirectional references and derived inverse references.

Double Unidirectional References. A bidirectional relation can be modelled simply as two unidirectional relations. However, consistency is an issue when editing the model. When adding or removing a reference in one direction, the opposite direction has to be added or removed as well. One possibility is to add special edit operations keeping the invariant of bidirectionality maintained, rather than to directly use the primitive edit operations for intrinsic attributes.

Derived Inverse References. An alternative option is to define one of the directions as primary and model it as a unidirectional reference. Then, a derived attribute can be used to compute the opposite direction in order to automatically maintain consistency. A disadvantage of this solution is the different treatment of the two directions, and that only one of the directions can be edited directly. One solution here is to add special edit operations that always edit the forward direction of the relation.

Listing 5 shows an example of how to implement the inverse relation `monitoredBy` from the forward relation `monitors` using RAG attributes and equations. For each `TrackElement`, a set `monitoredBy` is defined as a *collection* attribute, and each `Sensor` contributes to the collections of all

its monitored `TrackElements`. A collection attribute, indicated by the `coll` keyword, is defined by a set comprehension of so called *contributions* that can be anywhere in the model [26].

4.2.3 Discussion

While the above solutions for implementing non-containment references are reasonably straightforward, they do call for some boilerplate RAG code. Furthermore, while both the name analysis and the inverse relation computation are efficient for normal (non-incremental) evaluation, they do not have very good incremental performance, as indicated by our measurements in Section 6. The reason is that, at least with globally valid names, the maps and sets involved depend on essentially the whole model, so that most changes will lead to re-computation of the complete name analysis, the inverse relations, and all of their dependent attributes.

To improve the performance and avoid boilerplate code, we therefore suggest specific support for relations in RAGs, as will be discussed in the next section.

5 Extending RAGs to Support Models

Section 4 demonstrated how (bidirectional) non-containment relations can be implemented in attribute grammars. However, both proposed implementation variants have disadvantages concerning the need for boilerplate code and efficiency. Those disadvantages are the main concern of RQ2 and RQ3, respectively. Hence, bidirectionality cannot be expressed directly, even though it is a structural property of the model. As a third and better solution, we extend RAGs with explicit high-level support for non-containment relations.

5.1 Extending RAGs with Non-containment Relations

Adding non-containment relations to abstract grammar production rules would be difficult since bidirectional relations belong to both sides of the relation equally. Therefore, we specify the relations as separate clauses in the abstract grammar. For each relation, information is supplied on source and target classes, role names, direction, and cardinalities.

Assume two abstract grammar classes named `Class1` and `Class2`. There can be several kinds of relations between these classes which differ in directionality and cardinality.

Unidirectional Relations. If there is a directed relation from `Class1` to `Class2` called `role1` and the cardinality of this relation is 1, i.e., there is exactly one element of type `Class2` related to each element of type `Class1`:

```
rel Class1.role1 -> Class2;
```

Unlike in UML or Ecore diagrams, the role name is positioned next to the source of the relation rather than the target. This is because the role can be seen as an attribute of the source.

The cardinality can also be *zero-or-one*, i.e., the relation is optional, denoted as "?", or *zero-to-many*, denoted as "*". The cardinality is positioned next to the role name and, therefore, also placed next to the source object of the relation.

```
rel Class1.role1*  -> Class2;
rel Class1.role1?  -> Class2;
```

Bidirectional Relations. For bidirectional relations, the notation is extended to include a second role name `role2` and cardinality on the right hand side of the rule and a bidirectional arrow. In the bidirectional case, any combination of cardinalities is supported. Examples:

```
rel Class1.role1  <-> Class2.role2;
rel Class1.role1* <-> Class2.role2?;
rel Class1.role1* <-> Class2.role2*;
```

5.2 A Grammar for the Train Model

To specify the model of Figure 1, we extend the basic abstract grammar from Listing 3 with the following definitions of non-containment relations:

Listing 6. RAG non-containment relations for the *Train Benchmark* model.

```
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;
```

5.3 API for Non-containment Relations

For each role in a relation, an API is generated that allows the relations to be accessed and edited. The API is slightly different depending on the cardinality of the role. All provided methods contain the name of the role to ensure uniqueness.

***Cardinality* 1.** Roles with a cardinality of one use an interface similar to intrinsic attributes. The setter uses a `set` prefix, while the getter is simply the name of the role. For a role name `role` and classes `Source` and `Target`, the generated interface looks as follows:[3]

```
void Source.setRole(Target t);
Target Source.role();
```

***Cardinality* ?.** If the role is optional, two additional methods are generated: one to check if the relation is present, and another to clear it.

```
boolean Source.hasRole();
void Source.clearRole();
```

[3] We use AspectJ's inter-type declaration syntax here. So to illustrate that class `A` has a method `void m(){...}`, we write `void A.m(){...}`.

***Cardinality* *.** List roles require different accessors. Methods prefixed with `addTo` and `removeFrom` are used to modify the list. To access the list, the name of the role can be used as with the other cardinalities. However, the returned list is immutable to ensure consistency of bidirectional references and validity of incremental evaluation, so only the provided interface in the class `Source` is able to modify the list.

```
void Source.addToRole(Target t);
void Source.removeFromRole(Target t);
void Source.clearRole();
java.util.List<Target> Source.role();
```

5.4 Ensuring Consistency

The generated API ensures consistency of bidirectionality and upper bounds of cardinalities. For example, consider the relation `monitors`/`monitoredBy`. Suppose we have a sensor `s` and track element `t`. If `s.addToMonitors(t)` is called, the bidirectional relation *s* <-> *t* will be added, and this call is equivalent to `t.addToMonitoredBy(s)`.

For relations with cardinality 1 or ? on one or both roles, setting or adding a role might mean that other relations need to be removed to ensure consistency. For example, suppose we call `s.addToPositions(sp)` where `s` is a switch and `sp` is a switch position. If `sp` has a `target`/`postitions` relation to some other switch, that relation must be removed.

Harkes and Visser [13] present all 16 combinations of situations when adding a relation, considering endpoints of cardinality 1 and *, and how they need to be handled to ensure bidirectionality and upper bounds. Our API handles all these situations. (The cardinalities for ? are similar to the ones for cardinality *1* and are also handled.)

Like Harkes and Visser, we do not ensure lower bounds. For many operations, ensuring lower bounds for individual operations is not possible. For example, when deserializing, both source and target must be created before a relation between them can be added. Similarly, a *one-to-one* relation cannot be edited with simple add/remove relation operations without temporarily violating the lower bound. Instead, our API provides methods to check if lower bounds are violated. These methods can be called after deserialization and after edit operations, which is useful for debugging.

5.5 Implementation

We have implemented support for non-containment relations as a preprocessor to *JastAdd*. The preprocessor takes a file in the extended abstract grammar format (`.relast`), and generates an abstract grammar in the old format (`.ast`) together with a *JastAdd* aspect file (`.jadd`) with Java methods implementing the relation API. The generated code uses the standard *JastAdd* API.

To represent the relations, the generated abstract grammar includes an intrinsic attribute (token) for each role, so bidirectional relations are represented by two intrinsic attributes,

Listing 7. Extended grammar with three relations.

```
Root ::= A* B*;
A ::= <Name:String>;
B ::= <Name:String>;
rel A.r1   -> B;
rel A.r2?  -> B;
rel A.r3* <-> B.r4*;
```

Listing 8. *JastAdd* grammar generated from Listing 7.

```
Root ::= A* B*;
A ::= <Name:String> <_impl_r1:B> <_impl_r2:B>
        <_impl_r3:ArrayList<B>>;
B ::= <Name:String> <_impl_r4:ArrayList<B>>;
```

Listing 9. (Simplified) interface for `r3` defined in Listing 7.

```
public java.util.List<B> A.r3() {
  return Collections.unmodifiableList(get_impl_r3());
}
public void A.addToR3(B o) {
  ArrayList<B> list = get_impl_r3();
  ArrayList<A> list2 = o.get_impl_r4();
  list.add(o);
  list2.add(this);
  set_impl_r3(list);
  o.set_impl_r4(list2);
}
public void A.removeFromR3(B o) {
  ArrayList<B> list = get_impl_r3();
  if (list.remove(o)) {
    ArrayList<A> list2 = o.get_impl_r4();
    list2.remove(this);
    set_impl_r3(list);
    o.set_impl_r4(list2);
  }
}
```

one on each end of the relation. For roles of cardinality ? and 1, a simple intrinsic attribute is used. For list roles, Java's `ArrayLists` are employed.

For each role, one intrinsic attribute is used, resulting in *JastAdd* seeing a role as one atomic entity, even though it might be a list. This not only entails that the attributes may not be modified, only reset, but also defines the granularity of an attribute's dependency tracking, which works on whole lists rather than list element access. As an example, consider the extended RAG grammar in Listing 7. The preprocessor creates two files, the plain RAG grammar shown in Listing 8 and an aspect file, parts of which are shown in Listing 9, i.e., the accessors for role `r3` on class `A`.

Note that consistency is ensured by setting and removing both sides of the relation at once. The generated API is implemented using the standard *JastAdd* API for manipulating intrinsic attributes. This ensures that incremental evaluation of *JastAdd* is used correctly when relations are added or removed, i.e., affected attribute caches are invalidated.

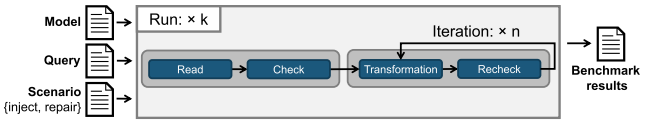


Figure 6. Benchmark process, adapted from [38].

6 Evaluation

In this section, we will compare the described approaches of Section 4 and Section 5 based on quantitative and qualitative criteria in order to answer the three research questions.

6.1 Evaluation Setup

Figure 6 describes the process for evaluating the *Train Benchmark* [38]. First, the model is read and an initial set of matches is computed in the *Check* phase. Then, the model is changed and matches are recomputed. Modification and recomputing matches is iterated n times. To specifically test the performance of incremental model validation, only a small fraction of the model is changed every time.

There are two scenarios: *inject* and *repair*. In accordance with [38], *inject* creates a fixed number of 10 new faults with $n = 12$ iterations, whereas *repair* removes 5 percent of the faults with $n = 8$ iterations. Thus, both a constant and a proportional amount of changes is investigated. We split the benchmark into individual runs for each query as opposed to the *Train Benchmark* paper, where querying, injecting, and repairing was executed together for all queries. Thus, we have a detailed view on the performance of specific queries. The benchmark can be scaled to evaluate the complexity of queries w.r.t. increasing model sizes. For the smallest size 1, approximately 5000 elements are generated. Other element counts are shown in the measurements, e.g., in Figure 7.

As described in the previous sections, we investigated three different approaches to compute the given queries:

Name Lookup Naive name resolution based on symbolic references and collection attributes described in the first part of Section 4.2.1.

Intrinsic References Intrinsic references and collection attributes for inverse part of bidirectional relations described later in Section 4.2.1.

Grammar Extension Intrinsic references and bidirectional relations shown in the previous Section 5.

Each approach can either use incremental evaluation or recompute all RAG attributes after every transformation. While there is no implementation difference in using incremental evaluation, we distinguish incremental and non-incremental evaluation since they have different runtime characteristics. We compare these RAG-based approaches to *Tinkergraph* (an implementation of Apache Tinkerpop [9]), the fastest non-incremental tool, and to *VIATRA* [3], the fastest incremental tool, as of the published status in [38].

Listing 10. *RouteSensor* query.

```
syn Collection<Match> Route.routeSensorMatches();
eq Route.routeSensorMatches() {
  List<Match> matches = new ArrayList<>();
  Collection<Sensor> requiredSensors = requiredSensors();
  for (SwitchPosition sp : getSwitchPositionList()) {
    Switch sw = sp.target();
    for (Sensor sensor : sw.monitoredBy()) {
      boolean validSensor = false;
      for (Sensor sensor2 : requiredSensors) {
        if (sensor2 == sensor) {
          validSensor = true; break;
        }
      }
      if(!validSensor) {
        matches.add(new Match(this, sensor, sp, sw));
      }
    }
  }
  return matches;
}
```

6.2 Feasibility and Suitability (RQ1 and RQ2)

The feasibility of RAGs for defining and analysing models is indicated by the presented implementation of the complete *Train Benchmark*. To investigate the degree of suitability of RAGs in general and the proposed extension in particular, the three implementation variants[4] are compared with respect to conciseness and implementation effort we highlight some of their differences. In general, two classes of differences can be observed.

Attribute/Accessor. In the *name lookup* and *intrinsic references* variants, navigations are not performed uniformly: some are performed using attributes and others by AST accessors. In contrast, in the *grammar extension* variant, all references are uniformly navigated by AST accessors.

Edit Directions. Transformations require edits of the relations. However, when relations are encoded by computed attributes, like for the *name lookup* and *intrinsic references* variants, these cannot be edited directly.

In the following, we discuss the implementations of the two selected queries and the respective transformations described in Section 2.

RouteSensor: Query. The attribute for the first query, *RouteSensor*, is shown in Listing 10. It remains identical for all three approaches, with one exception, namely the call to `monitoredBy()` (highlighted), which gets all sensors monitoring a switch. In the *name lookup* and *intrinsic references* variants, it is an attribute specified in Listing 11, wherein a handmade lookup map is used to retrieve the monitoring sensors based on the id of a track element. The attribute computing the map iterates over all sensors and builds up

[4] Available at https://git-st.inf.tu-dresden.de/stgroup/trainbenchmark.

the resulting map by inserting it into the list for the track elements it monitors. In the *grammar extension* approach, the attribute `monitoredBy` is replaced by an accessor for the bidirectional relation of the same name.

RouteSensor: Transformations. Transformations are implemented by iterating over matches and inserting or removing associations between route and sensor pairs. The *repair* scenario is the same for all variants, with the only difference being the use of bidirectional relations in the *grammar extension* approach and references in the other two. In the *inject* scenario, the difference is shown in Listing 12. Both for *name lookup* and *intrinsic references*, a manual iteration over the reference to sensors is required for their removal. However, the *grammar extension* approach requires only a single call to remove a sensor from the required association. This is an example of an *edit direction* difference.

ConnectedSegments: Query. The query attribute for *ConnectedSegments* is nearly identical for all approaches and shown in Listing 13. The only difference is, again, the use of references in the first two approaches. The attribute `transConnectedSegments` computes all transitively connected segments and allows a better caching behaviour.

ConnectedSegments: Transformations. For the transformations of *ConnectedSegments*, we can observe similar effects as for *RouteSensor*, resulting in shorter code for the *grammar extension* variant.

6.3 Code Complexity Metrics Comparison

We use the following source code metrics to measure and compare implementation size and complexity of queries, transformations, and utility attributes:

- **LOC** Lines of code, as a measure of total programming effort
- **CFC** Control flow complexity (number of control flow constructs `if`, `for` and `return`)
- **AC** Number of attributes, as a coarse grain indicator for the number of additional attributes necessary to achieve acceptable performance

The control flow complexity metric (CFC) was introduced by Vinju and Godfrey to avoid misleading estimates in raw code complexity [40]. Table 1 reports the results of applying the selected metrics to the three approaches. For each implementation, we separately measured utility attributes, queries, and transformations. Utility attributes are used to, e.g., compute reverse relations and for resolving references.

In general, complexity scores decrease from *name lookup* to *intrinsic references*, and from *intrinsic references* to *grammar extension*. This holds especially for the utility attributes. The *intrinsic references* approach removes the need for reference resolving attributes, and the *grammar extension* approach additionally has no need of manual reverse relation accessors. One exception is a query not described in detail in

Listing 11. Attribute `monitoredBy` used by *name lookup* and *intrinsic references* for *RouteSensor* query.

```
syn Collection<Sensor> TrackElement.monitoredBy() {
  return getRoot().monitoredByMap().get(this.id());
}
```

Listing 12. *Inject* transformation in *RouteSensor* for *name lookup* and *intrinsic references* (-/red) compared to *grammar extension* (+/green) approach.

```
   public void activate(final Collection<Match> matches) {
-    List<SensorRef> refsToBeRemoved = new ArrayList<>();
     for (final Match match : matches) {
-      for (SensorRef ref : match.getRoute()
-           .getRequiredSensors()) {
-        if (ref.getSensor() == match.getSensor()) {
-          refsToBeRemoved.add(ref);
-        }
-      }
-    }
-    for(SensorRef ref : refsToBeRemoved) {
-      ref.removeSelf();
+      match.getRoute().removeRequiredSensor(
+        match.getSensor());
     }
     driver.flushCache();
   }
```

Listing 13. *ConnectedSegments* query.

```
syn Collection<Match> Sensor.connectedSegmentsMatches();
eq Sensor.connectedSegmentsMatches() {
List<Match> matches = new ArrayList<>();
  for (Segment segment: monitoredSegments()) {
    sequenceLoop: for (List<Segment> segmentSequence :
              segment.transConnectedSegments()) {
      if (segmentSequence.size() < 5) continue;
      for (int index = 0; index < 5; index++) {
        if (!this.monitors(segmentSequence.get(index)))
                continue sequenceLoop;
      }
      matches.add(new Match(this, segment,
                segmentSequence));
    }
  }
  return matches;
}
```

Table 1. Code Complexity in the *Train Benchmark*.

Metric	Aspect	Name lookup	Intrinsic references	Grammar extension
LOC	Utility	304	254	165
	Queries	258	258	263
	Transformation	295	295	262
CFC	Utility	86	64	42
	Queries	77	77	79
AC	Utility	37	27	15
	Queries	26	26	27

this paper (*SwitchMonitored*), which is more complex in the *grammar extension* implementation due to a different way to write the query using a bidirectional relation.

As an example, Listing 12 shows the *inject* transformation for *RouteSensor* for all approaches. Using the *grammar extension* approach, this method has 7 LOC with a CFC of 1, whereas using the other two approaches results in 15 lines and a CFC of 3.

Concluding the observations, using the *grammar extension* approach, we can reduce the complexity when modelling bidirectional relations used for matching query patterns and transformations. While the queries themselves are almost identical in all variants, most utility attributes are not required with the proposed *grammar extension* and transformations become much more concise. Thus, it is easier to implement those, increasing the suitability of RAGs, which addresses RQ2 posed in Section 1.

6.4 Quantitative Evaluation (RQ3)

Finally, the benchmark was run for the presented variants using the configuration presented in Section 6.1. The measurements were performed on a Intel E5-2643 server with 64 gigabytes of memory using Ubuntu 16.04. All tools were run with Oracle Java, version 1.8.0_171, with a maximum heap size of 32G. We measured all phases as shown in Figure 6 and show the combined times for reading and checking the initial model in Figure 7, as well as the median of the combined times for transforming and rechecking the transformed model in Figures 8 and 9. Included as diagrams are both *inject* and *repair* scenario for the *ConnectedSegments* and *RouteSensor* queries. In all diagrams, the median execution time of 10 runs for each model size is depicted. In case of an overall time out, no measurements of the run are taken.

Read and Check Phase. For the *Read and Check* phase, we only included a diagram for the *repair* scenario, as the times for the *inject* scenario are very similar, because model loading times by far exceed query processing times. In both cases, a linear growth of the execution time with model size for all tools and for both queries can be observed. However, the incremental and non-incremental *grammar extension* approach are fastest, followed by *TinkerGraph* and *VIATRA*.

Transformation and Recheck Phase. Regarding transformation and recheck, there are subtle differences depending on both the query and the scenario.

For *RouteSensor* in the repair case and for both queries in the *inject* scenario, similar observations can be made. All incremental tools perform better than their non-incremental counterparts. For *RouteSensor*, this is because only a non-containment reference is removed or added in the *inject* or *repair* scenario, respectively. Thus, nearly all the analysis can be read from cache.

For the sizes 1 and 2 in *ConnectedSegments* (and size 1 in *RouteSensor*), there are less than 20 matches, resulting in no repairs (because of the set 5% to be fixed) and, thus, almost incremental tools do not require any re-evaluation. In larger sizes, a linear growth can be seen for all tools, because either the query is run again in case of a non-incremental tool, or a large part of the analysis has to be recomputed in case of incremental tools. The repair operation for *ConnectedSegments* removes a complete segment, invalidating the analysis for all segments transitively connected to it. This results in a scenario where the incremental bookkeeping overhead exceed the benefits of partial re-evaluation.

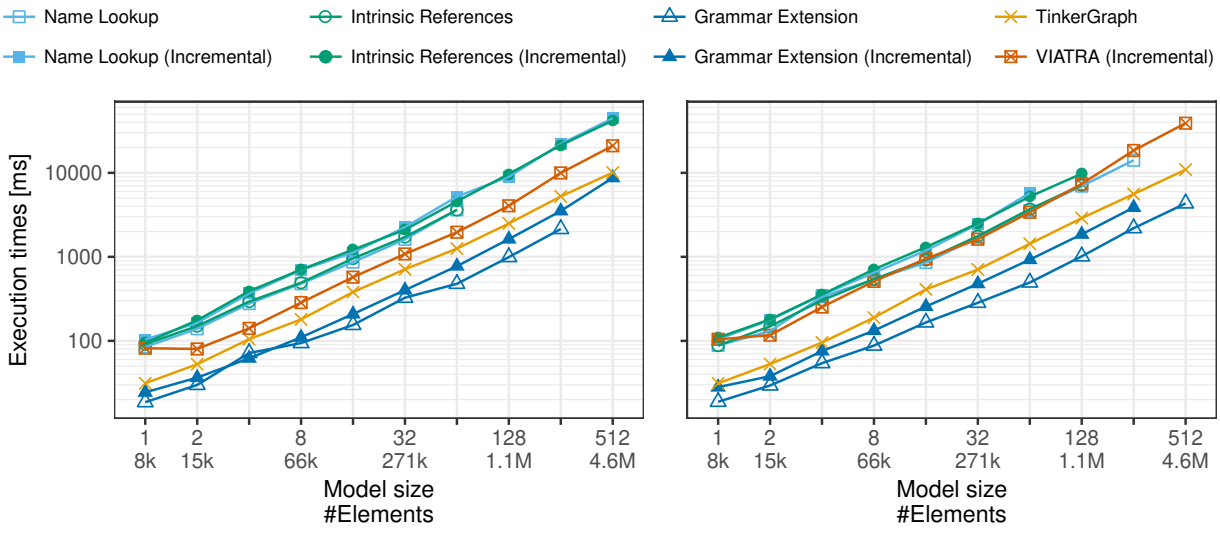
For both queries in the *inject* case, only a fixed number of changes is made to the model, thus a near constant time is needed for transformation and recheck.

Summarizing the benchmark results, we were able to show that in most cases the presented *JastAdd* variants, especially the *grammar extension* variant, perform similarly to other incremental tools while providing a concise, yet flexible notation for complex model queries and analysis.

7 Related Work

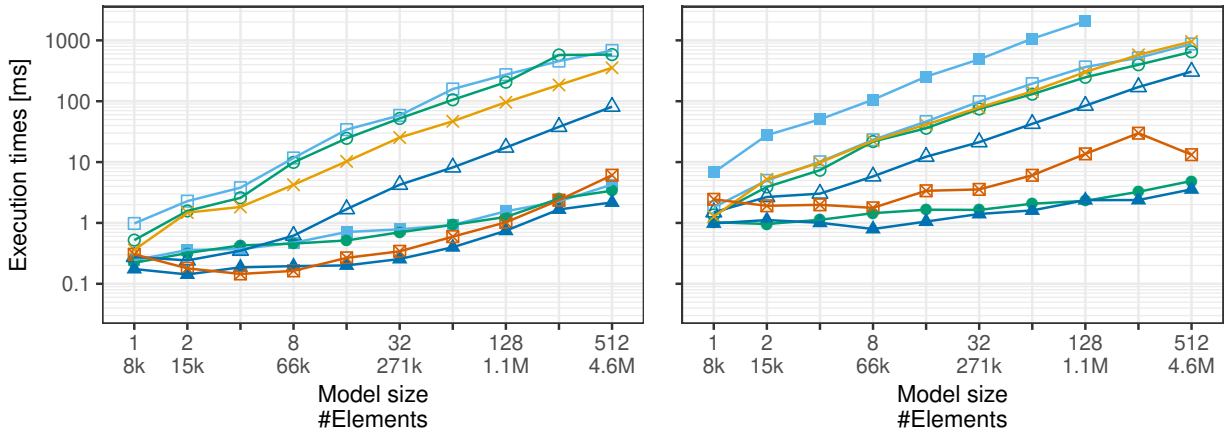
As discussed in Section 3, the general mismatch between conceptual models and RAGs shares significant similarities with the object-relational impedance mismatch [18]. In both cases, the lack of bidirectional relationships in one domain leads to manual implementations prone to errors and poor performance. Rumbaugh [33] already identified this issue in 1987. Various researchers followed his example and introduced first-class relationships to query and programming languages. For instance, [34] introduced relationships to object models and [1] to database programming languages. At the same time, Bock and Odell [5] first argued for relationships in object-oriented programming languages. Afterwards, multiple other approaches emerged that introduced first-class relationships, such as [4, 29]. More recently, relationships were employed to continuously validate [2] and evaluate [12] object-relational models. Closing the loop, Jäkel et al. [20] extended SQL adding relationships to perform queries upon compartment role object models [24]. For a detailed survey the reader is referred to [25, 36].

In contrast to these, there are a plethora of dedicated tools for generating, querying, and transforming models. Discussing all of them is beyond the scope of this paper. The interested reader is referred to [22] for a thorough comparison. Henceforth, we only discuss approaches for incremental model queries and validation. An early approach for incremental evaluation of queries was presented in [31] and later on used for materialized views in [10]. Later, Jackson [19] designed *Alloy*, a lightweight modelling notation with a formal underpinning also supporting incremental simulation of model changes. Since then, tools evolved to deal efficiently



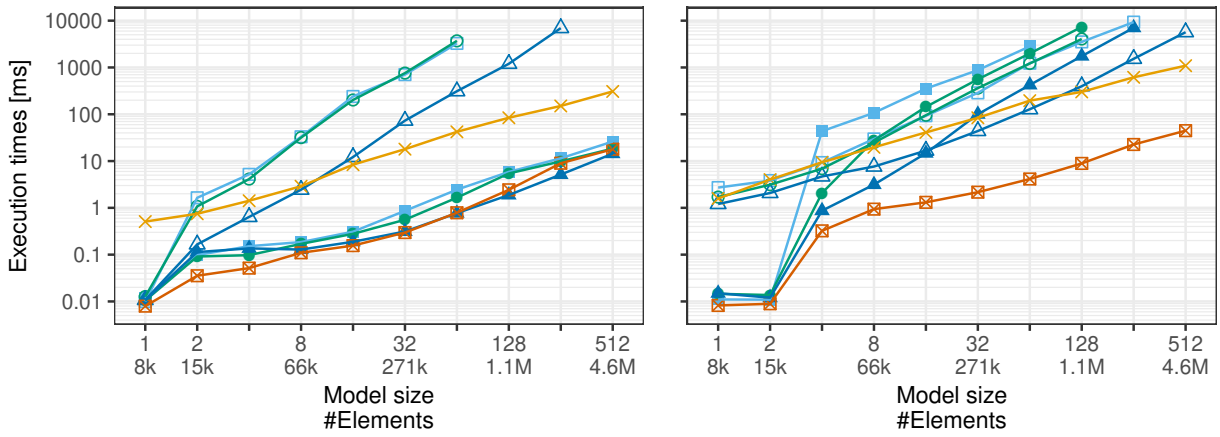
(a) *RouteSensor.* **(b)** *ConnectedSegments.*

Figure 7. *Read* and *Check* phases in the *repair* case.



(a) *RouteSensor.* **(b)** *ConnectedSegments.*

Figure 8. *Transformation and Recheck* phases in the *inject* case.



(a) *RouteSensor.* **(b)** *ConnectedSegments.*

Figure 9. *Transformation and Recheck* phases in the *repair* case.

with large and complex models, e.g., *Adapton* [11], *EMF-incquery* [39] now known as *VIATRA* [3], *Active Operation Framework* [21], and recently *NMF* [16]. In general, these approaches support incremental queries of large, complex models, yet usually lack an underlying formal foundation.

Conversely, our approach is based on the well-studied concept of attribute grammars. To our knowledge, the earliest work combining attributes and relations was done in the *Cactis* [17] system. Similar to our approach, relations were expressed explicitly and attributes were evaluated on demand and incrementally. However, *Cactis* supported only synthesized attributes without reference values, and had no specific support for containment relations. *IceDust 2* [13] is a data modeling language that supports derived attributes and relations, and implements several strategies for incremental evaluation. It differs from our approach in many ways. For example, it does not have specific support for the containment relation, and no subtyping, so it does not have the equivalent of an abstract grammar. Furthermore, its derived attributes are limited from an attribute grammar viewpoint, for example not supporting inherited, higher-order, nor circular attributes. Interestingly, the derived bidirectional relations of *IceDust 2* have some similarities with the expressiveness of collection attributes, e.g., reversing edges in a relation as was shown in Listing 5. A more related approach combining RAGs and metamodels is *JastEMF* [7]. Compared to our approach, *JastEMF* makes use of RAGs to specify semantics but does not try to express the full metamodel in RAGs. The non-containment relations are handled by the EMF framework and not by RAGs. In addition, *JastEMF* does not have support for incremental evaluation. *JavaRAG* [8] is another related approach allowing RAGs to be added on top of any spanning tree, as long as a traversal API can be implemented. *JavaRAG* has been used to add a RAG to an EMF metamodel. However, like with *JastEMF*, non-containment relations are handled by the underlying EMF framework, and there is no support for incremental evaluation. *RACR* [6] is an extension of RAGs that supports incremental evaluation and graph rewriting, but does not contain support for explicit relations. Our suggested approach for extension with relations could similarly be translated to *RACR*.

8 Conclusion

Dealing with more complex and continuously changing models is one of the major challenges for current software systems. Approaching this challenge, we propose to employ RAGs to benefit both from their concise means to express structure and from their efficient incremental evaluation of defined computation. This first step affirms our first research question. However, there is still a mismatch between models and RAGs, which leads to a manual specification and resolution of non-containment relations resulting in possible errors and potentially inefficiency. To remedy this mismatch and answer RQ2, we introduced non-containment relations to RAG specifications, which especially support bidirectionality, and presented a generic implementation using the *JastAdd* system. We employed the *Train Benchmark* to show the efficiency of this extension compared to other approaches, answering RQ3. This paper illustrates the suitability and efficiency of RAGs to handle large and complex conceptual models with bidirectional relations.

Future Work Further exploration of using RAGs for models could look into making use of more advanced derived attributes. For instance, higher-order attributes allow complete derived models to be computed and further attributed, and circular attributes can be used for fix-point properties.

As a second path, we will look into refurbishing *JastEMF* to the latest versions of both, EMF and *JastAdd*. This will enable tight integration of RAGs and EMF, while still have an incremental evaluation.

Comparing our current implementation of the *Train Benchmark* use case to others like VIATRA focusing on pattern matching, another possible extension would be a declarative specification of tree (or graph) patterns. Out of such a declaration, attributes could automatically be generated to match those patterns still profiting from both bidirectional relations and incremental evaluation.

Acknowledgments

This work is partly supported by the German Research Foundation (DFG) in the SFB 912 "Highly Adaptive Energy-Efficient Computing", the project "RISCOS" and within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907), and by the German Federal Ministry of Education and Research within the project "OpenLicht". This work is also partly supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA) in the PIIA project 2017-02371 and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation (KAW).

References

[1] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. 1991. A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language.. In *VLDB*, Vol. 91. 565–575.

[2] Stephanie Balzer and Thomas R. Gross. 2011. Verifying Multi-object Invariants with Relationships. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer, Berlin, Heidelberg, 358–382. https://doi.org/10.1007/978-3-642-22655-7_17

[3] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. 2015. Viatra 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*, Dimitris Kolovos and Manuel Wimmer (Eds.). Springer International Publishing, Cham, 101–110.

[4] Gavin Bierman and Alisdair Wren. 2005. First-Class Relationships in an Object-Oriented Language. In *ECOOP 2005 - Object-Oriented Programming*. Springer, 262–286.

[5] Conrad Bock and James Odell. 1998. A more complete model of relations and their implementation: Roles. *Journal of Object-Oriented Programming* 11, 2 (1998).

[6] Christoff Bürger. 2015. Reference Attribute Grammar Controlled Graph Rewriting: Motivation and Overview. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015)*. ACM, New York, NY, USA, 89–100.

[7] Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann. 2011. Reference Attribute Grammars for Metamodel Semantics. In *Software Language Engineering*, Brian Malloy, Steffen Staab, and Mark van den Brand (Eds.). Springer, Berlin, Heidelberg, 22–41.

[8] Niklas Fors, Gustav Cedersjö, and Görel Hedin. 2015. JavaRAG: A Java Library for Reference Attribute Grammars. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2724525.2724572

[9] The Apache Software Foundation. 2018. Apache TinkerPop. http://tinkerpop.apache.org/

[10] Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. 1997. Incremental updates for materialized OQL views. In *Deductive and Object-Oriented Databases*, François Bry, Raghu Ramakrishnan, and Kotagiri Ramamohanarao (Eds.). Springer, Berlin, Heidelberg, 52–66. https://doi.org/10.1007/3-540-63792-3_8

[11] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-driven Incremental Computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 156–166. https://doi.org/10.1145/2594291.2594324

[12] Daco Harkes and Eelco Visser. 2014. Unifying and Generalizing Relations in Role-Based Data Modeling and Navigation. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, Cham, 241–260.

[13] Daco C. Harkes and Eelco Visser. 2017. IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 14:1–14:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.14

[14] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[15] Görel Hedin and Eva Magnusson. 2003. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0

[16] Georg Hinkel. 2018. NMF: A Multi-platform Modeling Framework. In *Theory and Practice of Model Transformation (Lecture Notes in Computer Science)*. Springer, Cham, 184–194.

[17] Scott E. Hudson and Roger King. 1989. Cactis: A Self-adaptive, Concurrent Implementation of an Object-oriented Database Management System. *ACM Trans. Database Syst.* 14, 3 (Sept. 1989), 291–321. https://doi.org/10.1145/68012.68013

[18] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. 2009. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*. IEEE, 36–43.

[19] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290. https://doi.org/10.1145/505145.505149

[20] Tobias Jäkel, Thomas Kühn, Stefan Hinkel, Hannes Voigt, and Wolfgang Lehner. 2015. Relationships for Dynamic Data Types in RSQL. In *Datenbanksysteme für Business, Technologie und Web (BTW)*. 157–176.

[21] Frédéric Jouault and Olivier Beaudoux. 2016. Efficient OCL-based Incremental Transformations.. In *OCL@ MoDELS*. 121–136.

[22] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varró. 2018. Survey and classification of model transformation tools. *Software & Systems Modeling* (12 March 2018).

[23] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[24] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. 2015. A Combined Formal Model for Relational Context-Dependent Roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 113–124.

[25] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*. Lecture Notes in Computer Science, Vol. 8706. Springer, 141–160.

[26] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. 2009. Demand-driven evaluation of collection attributes. *Automated Software Engineering* 16, 2 (2009), 291–322.

[27] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars-their evaluation and applications. *Science of Computer Programming* 68, 1 (2007), 21–37.

[28] Stephan Murer, Carl Worms, and Frank J Furrer. 2008. Managed evolution. *Informatik-Spektrum* 31, 6 (2008), 537–547.

[29] Stephen Nelson, David J. Pearce, and James Noble. 2008. First Class Relationships for OO Languages. In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008)*. http://hdl.handle.net/2142/11788

[30] Object Management Group (OMG). 2016. Meta-Object Facility (MOF) Specification, Version 2.5.1. OMG Document Number formal/2016-11. http://www.omg.org/spec/MOF/2.5.1

[31] Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1994. Efficient Incremental Evaluation of Queries with Aggregation. In *Proceedings of the 1994 International Symposium on Logic Programming (ILPS '94)*. MIT Press, Cambridge, MA, USA, 204–218.

[32] Jeff Rothenberg, Lawrence E Widman, Kenneth A Loparo, and Norman R Nielsen. 1989. *The nature of modeling*. Vol. 3027. RAND Corporation, Santa Monica, CA. https://www.rand.org/pubs/notes/N3027.html

[33] James E. Rumbaugh. 1987. Relations as Semantic Constructs in an Object-Oriented Language. In *OOPSLA*. 466–481.

[34] Marc H. Scholl and Hans-Jörg Schek. 1990. A relational object model. In *ICDT '90*, Serge Abiteboul and Paris C. Kanellakis (Eds.). Springer, Berlin, Heidelberg, 89–105. https://doi.org/10.1007/3-540-53507-1_72

[35] Emma Söderberg and Görel Hedin. 2012. *Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking*. Technical Report 98. Lund University. LU-CS-TR:2012-249, ISSN 1404-1200.

[36] Friedrich Steimann. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering* 35, 1 (2000), 83 – 106. https://doi.org/10.1016/S0169-023X(00)00023-9

[37] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: Eclipse Modeling Framework*. Pearson Education.

[38] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. 2017. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling* (Jan. 2017), 1–29. https://doi.org/10.1007/s10270-016-0571-8

[39] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. 2015. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming* 98 (2015), 80–99.

[40] J. J. Vinju and M. W. Godfrey. 2012. What Does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 154–163. https://doi.org/10.1109/SCAM.2012.17

[41] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*. ACM, New York, NY, USA, 131–145. https://doi.org/10.1145/73141.74830