A JastAdd-based Solution to the TTC 2018 Social Media Case

René Schöne, Johannes Mey

Software Technology Group, Technische Universität Dresden, Germany
{rene.schoene, johannes.mey}@tu-dresden.de

Abstract

The TTC 2018 live contest case describes a social media network and two queries retrieving elements based on a calculated score. This solution solves the case by employing Reference Attribute Grammars. Thus, we first transform the given EMF model into syntax tree, and secondly compute attributes defined on its grammar. Both grammar and attributes are specified and evaluated using the *JastAdd* system. Utilizing the same source code, we can generate an incrementally evaluated variant, and a non-incremental one.

1 Introduction

The TTC 2018 Social Media Case [3] describes a metamodel comprising users, posts, and comments. Further it defines two queries retrieving the most controversial posts based on the number of comments and the most influential comments based on groups of users all liking the comment. The case provides models in varying sizes along with a set of changes to be applied to the initial model. This setting tends to favour tools providing means for incremental evaluation. Our solution is based on Reference Attribute Grammars [2] and the *JastAdd* system [1]. With the most recent version of *JastAdd*, it is possible to generate two variants from the same specification, one working incrementally and the other requiring to flush all caches instead of only the relevant ones. After a brief description of the used technologies in Section 2, our solution is described in Section 3 and evaluated in Section 4. Section 5 concludes the paper.

2 Background

Attribute Grammars [4] enable the extension of a context free grammar with semantics. Attributes are computations defined for a nonterminal of the grammar, and can access both terminals and attributes of the subtree of the nonterminal they are defined on. Reference Attribute Grammars (RAGs) [2] extend this paradigm such that attributes are allowed to return other nodes of the abstract syntax tree (AST) as a result of their computation. This enables the specification of a graph instead of the basic tree structure by computing the "overlay" edges. An advantage of RAGs is their ability to compute attributes incrementally, i.e., if the AST was changed, then only those attributes affected by the change are recomputed.

In our solution, we use two advanced features of RAGs: nonterminal attributes and circular attributes. *Nonterminal attributes* [7] are, as suggested by their name, both nonterminals and attributes, in the sense, that an attribute is used to dynamically compute a new subtree. This is useful, if a part of the AST is fully determined by the rest of the AST and should therefore not be contained but computed. *Circular attributes* [5] allow recursive definitions of attributes by specifying a fix point computation. This can be used, to find strongly connected components, as needed for the second query.

In: A. Garcia-Dominguez, G. Hinkel, F. Krikava (eds.): Proceedings of the Transformation Tool Contest 2018, Toulouse, France, 29-Jun-2018, published at http://ceur-ws.org

Listing 1: Grammar for a SocialNetwork.

```
abstract ModelElement ::= <Id:Long> ;
SocialNetwork : ModelElement ::= User* Post* ;
User : ModelElement ::= <Name:String> ;
abstract Submission : ModelElement ::= <Timestamp:Long> <Content:String> Comment* ;
Comment : Submission ::= /<Post:Post>/ ;
Post : Submission ::= ;
rel User.friends* -> User ;
rel User.submissions* -> Submission ;
rel User.likes* <-> Comment.likedBy*
```

We use the *JastAdd* RAG system [1] to specify both grammar and attributes for our solution. This system is using aspect-weaving to generate plain Java code from the input specifications. A grammar describes the production rules, which are given as EBNF with extra features such as inheritance and relations. For every nonterminal, a Java class is generated comprising constructors and getters/setters matching their respective production rules. Attributes are defined in aspects containing their definitions, i.e., a Java method to compute an attribute value for a certain nonterminal.

3 Solution

This section describes the used grammar, the attributes to compute both queries, and to apply changes.

3.1 Grammar

Listing 1 shows the grammar used to represent the problem model. As in the given metamodel, there are nonterminals for `User`, `Submission`, `Comment`, and `Post`. We use the abstract class `ModelElement` to make those types identifiable using a numerical identifier, also including social network, to later insert new users or posts. Further, the relation of a comment to its containing post was replaced by a nonterminal attribute, that is recursively going up the containment hierarchy until a parent, which is a post, is found. All non-containment relations are specified using a special syntax described in one of our previous works [6]. With this, the bidirectional relation describing what comments a user likes can be described as an n-to-m relation like in last line of Listing 1.

3.2 Parsing the Initial Model

Because the initial model is given as an EMF model, we have to transform it into an AST. To achieve this, we have followed two different approaches. The first one uses an off-the-shelf XML-parser, thus creating an intermediate representation directly from the parsed XML. In a second step, every node is transformed into its respective nonterminal of the grammar shown in Listing 1 while resolving found references. This approach resulted in various challenges like resolving the different formats of references other nodes, or having forward references to nodes in a `Changeset` which are not created yet. A second approach to get an AST was to use the EMF parser, and transform the resulting EMF model with a simple transformation into our AST representation. However, the EMF parser is slower than the manual implementation as shown in Section 4.

3.3 First Query: Most Controversial Posts

The first task was to find the top three of the most influential posts, where the score of a post is 10 times the number of comments referring to this post plus the total likes for all of those comments. Our implementation splits the query into two parts. First, all comments referring to a post are computed in the attribute `commentsForPost`. With this, the score for a post can be computed by iterating over all those comments and according to the definition in the task description [3]:

```
syn int Post.score() {
  int result = 0;
  for (Comment comment : commentsForPost()) {
    result += 10 + comment.likedBy().size();
  }
  return result;
}
```

If a change affects any part of the AST accessed by the attributes involved in the computation of the query, a recomputation is needed. Because of the demand-driven nature of *JastAdd*, the attributes are not recomputed immediately, only their cache is marked as invalid, causing a computation if it is called the next time. This way, all attributes are computed at most once for one change set. Possible changes for the first query include the insertion of new posts or comments, or a new like. In this case, only the score of the affected post is recalculated. However, the computation of the query itself is always recomputed if something relevant has changed, since then at least one score is different.

3.4 Second Query: Most Influential Comments

The second task was to find the three most influential comments, where the score of a comment is determined by groups of users, which are both friends with each other and like the comment. Again, this query is split into two parts. First the group of friends is calculated with the circular attribute shown below. It starts with a result set containing the starting user, continues to add friends liking the given comment, and recursively calls itself for each of those friends. The attribute is guaranteed to terminate as the set can at most contain all users.

```
syn java.util.Set<User> User.getCommentLikerFriends(Comment comment) circular [new java.util.HashSet<User>()];
eq User.getCommentLikerFriends(Comment comment) {
  java.util.Set<User> s = new java.util.HashSet<>();
  s.add(this);
  for (User f : friends()) {
    for (Comment otherComment : f.likes()) {
      if (otherComment == comment) {
        s.add(f);
        for (User commentLikerFriend : f.getCommentLikerFriends(comment)) {
          s.add(commentLikerFriend);
        }
      }
    }
  }
  return s;
}
```

To compute the score of a comment, for all users liking it, the set of users is calculated by the previous attribute and stored in another set to avoid counting the same group of users twice. For all remaining sets of users, their squared size is summed up:

```
syn int Comment.score() {
  int score = 0;
  java.util.Set<java.util.Set<User>> commentLikerGroups = new java.util.HashSet();
  for (User user : likedBy()) {
    commentLikerGroups.add(user.getCommentLikerFriends(this));
  }
  for (java.util.Set<User> userSet : commentLikerGroups) {
    int usize = userSet.size();
    score += usize * usize;
  }
  return score;
}
```

3.5 Application of Changes

To apply the changes to the initial model, we first have to transform them into an AST. The grammar used for a change set mainly resembles the structure prescribed in the task description, but only the four types of elementary changes that are actually used. It is shown in Listing B of the Appendix. All elements pointing to the model use their most specific type, e.g., `ModelElement` in case of the element to be added in `CompositionListInsertion`.

One change in the grammar was needed in contrast to the original meta model: A list of pending new elements. This is needed, because elements to be added can either be new AST nodes or existing ones. Therefore, they can not be direct children of the change node as existing elements already have a parent. Thus, all new AST nodes introduced by changes are added to this list, and are later be moved to the model.

```java
public void AssociationCollectionInsertion.apply() {
  if (getAffectedElement().isUser()) {
    User user = getAffectedElement().asUser();
    switch (getFeature()) {
      case "submissions": user.addToSubmissions(getAddedElement().asSubmission()); return;
      case "friends": user.addToFriends(getAddedElement().asUser()); return;
      case "likes": user.addToLikes(getAddedElement().asComment()); return;
    }
  } else if (getFeature().equals("likedBy")) {
    // AssociationCollectionInsertion for likedBy will be handled by attributes
  }
}
```

To apply a change, the type of affected element and the feature stored as a String determine the action to take as shown above in an example for `AssociationCollection`. Using `switch` for the feature is needed, as there are no first-class relations in *JastAdd* as opposed to EMF. After applying a change, all affected attributes are invalidated, thus ensuring correct re-evaluation of the query value.

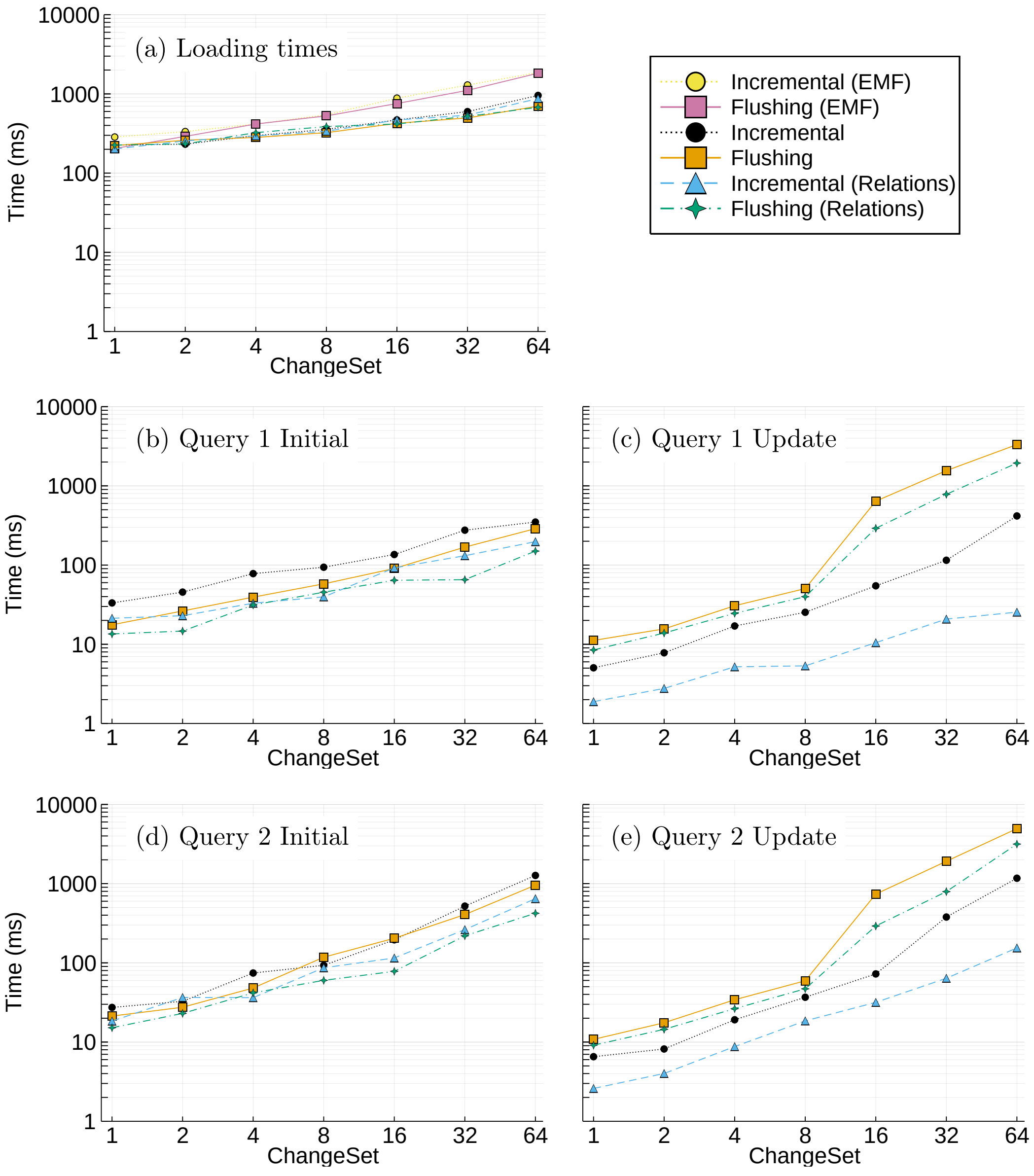

Figure 1: Execution times of the variants of our solution.

4 Evaluation

To evaluate, we used the provided benchmark infrastructure to execute it on a Intel Xeon E5-2643 machine with 3.3GHz, 16 cores and 64 gigabytes of memory using Debian 8 and Oracle Java version 1.8. Because of the declarative specification of attributes, we are able to produce variants of our solution showing the impact of features like incremental evaluation or bidirectional relations by using different build parameters.

Figure 1 shows the execution times of all different variants of our solution, where the times in case of *Update* are the mean over all iterations. "Flushing" refers to the non-incremental variant, and "Relations" states, that bidirectional relations are used. For the alternative EMF-based model loading mechanism, only loading times are shown, because the queries are always computed using *JastAdd*. The price of reusing EMF modelling code is a much slower loading speed compared to the hand-written model loader. Since the performance of the queries is independent of the loading mechanism, the EMF variant is omitted in the query measurements.

For the initial query computation, two observations can be made for both queries. First, the approaches using bidirectional relations are strictly better than their counterparts. Second, incremental evaluation needs more time for this first computation, as the dependency graph has to be built up. When the model is updated, the incremental evaluation pays off, since both incremental approaches are strictly faster than their counterparts. Further, as for the initial case, using bidirectional relations results in shorter executions times. As shown in previous work [6], the inverse relation is not computed, but instead it is set after each change application, thus avoid unnecessary computation. Overall, all approaches show a linear growth after an update. This is worse than the constant time, which an ideal incremental solution could achieve. The reason for not achieving it lies mainly in the needed complete recomputation of the support attributes, especially the circular attribute.

5 Conclusion

We have shown an unconventional solution to the TTC 2018 live contest, most other solutions of which were EMF-based. Our approach is based on Reference Attribute Grammars, enabling a declarative definition of structure and computation, as well as incremental evaluation. However, an initial transformation from the given EMF model into an AST is necessary. Our solution is implemented using the *JastAdd* system. We were able to show, that our approach scales well with the problem size, but did not achieve an ideal *constant* update time.

Acknowledgements

This work has been funded by the German Research Foundation within the Collaborative Research Center 912 Highly Adaptive Energy-Efficient Computing and the research project "Rule-Based Invasive Software Composition with Strategic Port-Graph Rewriting" (RISCOS) and by the German Federal Ministry of Education and Research within the project "OpenLicht".

References

[1] G Hedin and E Magnusson. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 2003.

[2] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[3] Georg Hinkel. The TTC 2018 Social Media Case. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018)*, CEUR Workshop Proceedings. CEUR-WS.org, June 2018.

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

[5] Eva Magnusson and Görel Hedin. Circular reference attributed grammars—their evaluation and applications. *Science of Computer Programming*, 68(1), 2007.

[6] Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. Continuous Model Validation Using Reference Attribute Grammars. In *Proc. of the 11th International Conference on Software Language Engineering*, 2018.

[7] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *PLDI '89*, New York, NY, USA, 1989. ACM.

Appendix

Listing A shows all attributes to perform the actual computation of both queries. The algorithm to compute the top three matches can be shared between both queries, because they operate on different types, thus depending on the query to compute, the correct `score` and `hasBetterScoreThan` attributes are called. To compute the first query, one needs to call

```
SocialNetwork socialNetwork = loadSocialNetwork();
String result = socialNetwork.query(1);
```

Analogously, to compute the second query, `socialNetwork.query(2)` has to be called.

Listing A: Attributes to compute both queries

```
aspect Queries {
  syn String SocialNetwork.query(int queryId) {
    Iterable<? extends Submission> l;
    switch (queryId) {
      case 1: l = getPostList(); break;
      case 2: l = comments(); break;
      default: return null;
    }
    Submission[] elements = new Submission[3];
    for (Submission elem : l) {
      if (elem.hasBetterScoreThan(elements[2])) { // at least better than #3
        if (elem.hasBetterScoreThan(elements[1])) {
          elements[2] = elements[1];
          if (elem.hasBetterScoreThan(elements[0])) { // new highscore
            elements[1] = elements[0];
            elements[0] = elem;
          } else { // better than second
            elements[1] = elem;
          }
        } else {
          elements[2] = elem;
        }
      }
    }
    return elements[0].getId() + "|" + elements[1].getId() + "|" + elements[2].getId();
  }

  syn int ModelElement.score() = 0;
  eq Comment.score() {
    int score = 0;
    java.util.Set<java.util.Set<User>> commentLikerGroups = new java.util.HashSet();
    for (User user : getLikedByList()) {
      commentLikerGroups.add(user.getCommentLikerFriends(this));
    }
    for (java.util.Set<User> userSet : commentLikerGroups) {
      int usize = userSet.size();
      score += usize * usize;
    }
    return score;
  }
  eq Post.score() {
    int result = 0;
    for (Comment comment : commentsForPost()) {
      result += 10 + comment.getLikedByList().size();
    }
    return result;
  }

  syn java.util.List<Comment> Post.commentsForPost() {
    java.util.List<Comment> result = new java.util.ArrayList<>();
    addToComments(result);
    return result;
  }

  syn java.util.Set<User> User.getCommentLikerFriends(Comment comment) circular [new java.util.HashSet<User>()];
  eq User.getCommentLikerFriends(Comment comment) {
    java.util.Set<User> s = new java.util.HashSet<>();
    s.add(this);
    for (User f : getFriends()) {
      for (Comment otherComment : f.getLikes()) {
        if (otherComment == comment) {
          s.add(f);
          for (User commentLikerFriend : f.getCommentLikerFriends(comment)) {
            s.add(commentLikerFriend);
          }
        }
      }
    }
    return s;
  }

  syn boolean Submission.hasBetterScoreThan(Submission other) {
    return other == null || this.score() > other.score() ||
      (this.score() == other.score()  this.getTimestamp() > other.getTimestamp());
  }
}
```

Listing B shows the used grammar for the changes being applied to the model.

Listing B: Grammar for a ChangeSet

```
ModelChangeSet ::= ModelChange* PendingNewElement:ModelElement* <SocialNetwork:SocialNetwork> ;

abstract ModelChange ::= ;

ChangeTransaction:ModelChange ::= SourceChange:ModelChange NestedChange:ModelChange* ;

abstract ElementaryChange : ModelChange ::= <AffectedElement:ModelElement> <Feature:String> ;

abstract AssociationChange : ElementaryChange ::= ;
AssociationCollectionInsertion : AssociationChange ::= <AddedElement:ModelElement> ;
AssociationPropertyChange : AssociationChange ::= <NewValue:ASTNode> ;

abstract AttributeChange : ElementaryChange ::= ;
AttributionPropertyChange : AttributeChange ::= <NewValue:String> ;

abstract CompositionChange : ElementaryChange ::= ;
CompositionListInsertion:CompositionChange ::= <Index:int> <AddedElement:ModelElement> ;
```