The SDMLib solution to the Class Responsibility Assignment Case for TTC2016

Christoph Eickhoff, Lennert Raesch, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany
raesch—christoph—zuendorf@uni-kassel.de

Abstract

This paper describes the SDMLib solution to the Class Responsibility Assignment Case for TTC2016. SDMLib provides reachability graph computation ala Groove. Thus, the simple idea was to provide rules for possible clustering operations and then use the reachability graph computation to generate all possible clusterings. Then, we apply the CRAIndex computation to each generated clustering and identify the best clustering. Of course, this runs into scalability problems, very soon. Thus, we extended our reachability graph computation to do an A* based search space exploration. Therefore, we passed the CRAIndex computation as a metric to our reachability graph computation and in each step, we consider the set of not yet expanded graphs and choose the one, that has the best metric value for expansion. The paper reports about the results we achieved with this approach.

1 Introduction

This paper describes the SDMLib solution to the Class Responsibility Assignment Case for TTC2016 [1]. SDMLib provides reachability graph computation ala Groove [2]. For a given start graph and a given set of rules, the reachability graph computation generates all graphs that may be derived from the start graph by applying all rules at all possible matches as often as possible in all possible orders. Each time a new graph is computed, we search through the set of already computed graphs for an already known isomorphic graph. As proposed by [2], SDMLib computes node and graph certificates which are then used as hash keys to access potentially isomorphic graphs efficiently. The node certificates then also help to do the actual isomorphism test. If a new graph has been generated, we create a so-called reachable state node and we connect the reachable state node of the predecessor graph with the reachable state node for the new graph via a rule application edge labeled with the name of the rule used. In addition, a root node of the graph is attached to the reachable state node. Altogether, the generated reachability graph has a top layer consisting of reachable state nodes connected via rule application edges and each reachable state node refers to the corresponding application graph via a graphRoot link. In SDMLib, this whole structure is again a graph, and graph rules may be applied to it in order to find e.g. reachable states with a maximal metric value for the attached application graph or to find states where all successor states have lower metric values or to find the shortest path leading to the best state. Actually, any graph related algorithm may be deployed.

The Class Responsibility Assignment Case challenges the rule orchestration mechanisms provided by the different model transformation approaches. Thus, our solution uses the SDMLib reachability graph computation for rule orchestration. This is a very simple way to apply all rules in all possible ways and
in addition we are able to investigate all intermediate results in order to identify which paths through the search space are the most interesting ones. The drawback of this approach is that it wastes a lot of runtime and memory space for copying the whole class model graph each time a rule is applied and for the search of already known isomorphic copies of the generated graphs. As shown in the case description, the number of possible clusterings grows with the Bell number, i.e. for larger examples a complete enumeration of all possible clustering is not possible in a meaningful time. As only a small fraction of the search space can be explored, it might be helpful to be able to investigate all intermediate states to identify the most promising spots for further expansion. Thus, we hope that the flexibility provided by the SDMLib reachability graphs to investigate different intermediate states pays off, in the end.

As it is usually not possible to generate the whole reachability graph for a given example, our reachability graph computation may be restricted to a maximum number of reachable states to be generated. Next, we have extended our reachability graph computation with an A* like search space exploration that takes a metric as parameter and at each step chooses the state with the best metric value for expansion. We have developed two variants of this A* algorithm which will be discussed below.

The next section introduces the rules we use to solve the Class Responsibility Assignment Case and then Section 3 shows the different search strategies we use in this example. Finally, Section 4 shows our performance measurements.

2 The Model Transformation Rules

Our feature clustering approach uses two SDMLib model transformation rules. In the preparation phase we use the rule shown in Figure 1 to create one class for each feature in our class model.

![Rule adding initial classes](image)

Figure 1: Rule adding initial classes

This rule starts by matching the pattern object c1 to the ClassModel object passed as parameter. Then, f2 is matched to a Feature object attached to this ClassModel object. The \{allMatches\} constraint causes the rule to be applied to all possible matches. Thus, for each Feature object in our current ClassModel, the \<< create>> stereotype on pattern object c3 causes the creation of a new Class object. In addition the new Class object is attached to the ClassModel via a classes link and to the Feature object via an encapsulates link. Finally, the new Class object’s name attribute gets assigned the concatenation of the prefix “Class4” and the name of the current Feature. Thus, after the execution of this rule, each feature has its own class containing just this feature. This class model is then used as starting point for the repetitive application of our clustering rule.

Our clustering rule merges classes along functional or data dependencies, cf. Figure 2. The matching of this rule starts with the ClassModel object which is bound to c1 at rule invocation. Then we follow a classes edge to find a match for c2, i.e. a Class object in our ClassModel. Next, we follow an encapsulates edge to match a Method object m4 contained in c2. The object matched by m4 must have a dataDependency edge or a functionalDependency edge to a Feature object matched by the pattern object f5. This Feature object in turn must be contained in a Class matched by c6. By default, SDMLib allows homomorphic matches, thus c2 and c6 would be allowed to match the same Class object. Via the \{matchOtherThen c2\} clause, we enforce isomorphic matching, i.e. c2 and c6 must match two different Class objects. Finally, the Class matched by c6 must belong to our ClassModel c1. When such a match is found, the subpattern containing the FeaturePO pattern object f7 is executed on all possible matches. Pattern object f7 matches for all features contained in the Class matched by c6. (Note, f7 exploits homomorphic matching and will
Our clustering rule merges classes only if there is a dependency between them. This already utilizes application specific knowledge about our CRAIndex metric. Our search space expansion will start with classes containing only one feature each. Merging classes without a dependency between them is not going to improve the CRAIndex of the resulting class model. Merging classes has the potential to improve the CRAIndex only if the classes contain features that depend on each other. Thus, our clustering rule is already optimized for the optimization of the CRAIndex. Using a different metric would perhaps require a more general clustering strategy. As the metric is evaluated during the search space exploration, it would be easy to simply merge any two classes, as any graphs resulting from applying a non metric improving rule would immediately be dismissed anyways.

### 3 The Search Space Exploration Mechanisms

At the beginning, a ReachbilityGraph object is initialized with a start graph or startState and with a set of rules that shall be applied to the different reachable states. Our standard A* search space exploration algorithm is shown in Listing 1. For the A* search space exploration, we call e.g.

```
rg.explore(25000, g -> CRAIndexCalculator.calculateCRAIndex((ClassModel) g));
```

where the first parameter is the maximal number of states to be generated and the second parameter is the metric function that guides our search space expansion algorithm. First, our expansion algorithm initializes its todo list with the startState and adds the startState to a hash table of reachable states where a graph certificate is used as key, as proposed by [2]. Then, line 5 loops through the todo list until it drains or the maximal depth of the search space is reached. Line 6 of our explore method just sorts the todo list before choosing a new element in line 7. Thereby, each exploration step considers the state with the best CRAIndex for further expansion resulting in a depth first like expansion strategy. Then line 8 and line 9 iterate through all rules and all matches. For each match, we clone() the current state and
apply() the rule changes to that clone, resulting in a newState. As the newState may have been created by
other rule applications already, line 11 tries to find an isoOldState, i.e. the find operation computes the
certificate of the newState and tries to look it up in the states hash table. This involves an isomorphism
check to exclude accidentally matching certificates. If no isoOldState is found, line 13 adds the newState
to the hash table of reachable states, line 14 adds an edge labeled with the applied rule from the current
state to the newState, and line 15 adds the newState to our todo list. If there is an isoOldState line 17
just adds an edge from the current state to the isoOldState.

```
1 ReachabilityGraph::explore(depth, metric) {
  todo = new ArrayList();
  todo.add(this.startState);
  states.put(certificate(this.startState), startState);
  while (!todo.isEmpty() && states.size() <= depth) {
    sort(todo, metric);
    current = todo.get(0); todo.remove(0);
    for (Rule r : this.rules) {
      while (r.findMatch()) {
        newState = current.clone().apply(r);
        isoOldState = find(states, newState);
        if (isoOldState == null) {
          states.put(certificate(newState), newState);
          addEdge(current, r, newState);
        } else {
          addEdge(current, r, isoOldState);
        }
      }
    }
  }
}
```
Listing 1: Default A* based Search Space Expansion

Within each step, our default A* search space exploration strategy generates all successors of the current
state. For case E there are about 400 dependencies, thus, the initial state has about 400 successor states.
While this number decreases by one with each rule application, the first 100 rule applications have 350
successors on average resulting in 35000 states, which already exceeds our memory space. To improve this,
we added a variant of our algorithm called Ignore Decline mode. The Ignore Decline mode improves our
exploration algorithm by comparing the metric value of the newState with the current bestMetric. If
the metric of the newState is lower then the bestMetric we ignore the newState, i.e. we do not add it
to our reachability graph nor to our todo list. This may exclude some important candidates from later
consideration but it reduces the number of states to be added to our reachability graph considerably thus
reducing memory space consumption.

The second variant of our search space exploration algorithm is called Promote Improvements mode. The
Promote Improvement mode computes the metric for each new state and in case of an improvement compared
to the current state, we stop the expansion of the current state (putting it back into the todo list). Then we
jump back to line 6 and start with a new iteration, i.e. we sort the todo list (bringing the new best state to
the front) and continue the search space exploration with this new state. The Promote Improvement mode
reaches local optima of the reachability graph very fast. Note, when the search is exhausted for some state
and we go back to earlier states, rule application on those earlier states will first produce the same matches
as in earlier runs. These same old matches will be identified by line 11 as isoOldStates and thus ignored.
However, this requires the computation of a certificate and an isomorphism check. To avoid this effort, our
real implementation of the Promote Improvement mode stores the number of already created successors for
each state and on reconsideration, this number of rule applications is directly ignored.
4 Performance Results

Figure 3 shows the CRA Index we achieve for the different input models and the time our best algorithm needed to compute this.

<table>
<thead>
<tr>
<th>CRA Index</th>
<th>Input A</th>
<th>Input B</th>
<th>Input C</th>
<th>Input D</th>
<th>Input E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>3</td>
<td>3,75</td>
<td>2,94</td>
<td>0,49</td>
<td>0,43</td>
</tr>
</tbody>
</table>

| Performance | 00:00:412 | 00:00:565 | 04:14:479 | 32:11:173 | 04:56:309 |

Figure 3: Performance Summary

References
