

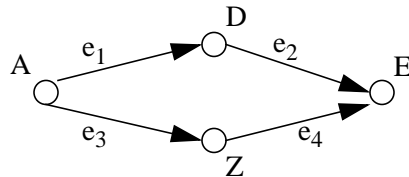
Section 1 contains general suggestions. Section 2 talks about extending the strategy and predicate model to add expressiveness.

1 General Suggestions

1.1 Asymmetry in the Model

The definition of a strategy seems asymmetrical. In particular a strategy seems to allow *conjunction* in the form of universal quantification (“for each node the predicate must be true”) but no *disjunction* (“there exists a node for which the predicate is true”). Consider **Definition 4.6** (page 10). The 2nd part states that *all* the interior elements of a class graph sub-path must satisfy the predicate. It looks like the only way to specify disjunction (“either this or that”) is to have separate paths in the strategy graph.

For instance, the strategy graph:



is basically a complex way to specify disjunction. We want to go from A to E but the specified paths are these that **either** go through D **or** through Z. Specifying disjunction using predicates, however, will make the model more expressive.

What does disjunction mean in practical terms? I’m certainly not an expert in Demeter/Java, but it seems to me that right now it too is asymmetric. We can say

```
bypassing -> * 1 *
```

but we cannot say

```
through * 1 *
```

because the second would require disjunctive path predicates (i.e., predicates that are true of the whole path if they are true of any one node). In other words, the system currently seems to have much better capabilities for excluding paths (predicate based — abstract) than for including paths (strategy based — all included paths have to be specified explicitly).

Disjunction in the form of existential quantification (“there exists a node in the path ...”) can easily be added to your model. The efficient compilation strategy will be preserved (the time complexity will still be polynomial). More on this in Section 2.

1.2 Simple Class Graphs

In the algorithms implementing the traversal strategy (Algorithm 1 and Algorithm 2) you emphasize that the input class graph must be *simple* (top of page 12). Nevertheless, it seems to me that this fact is only used in step 1 of Algorithm 2. There you form T' from all classes reachable from T through at most one subclass edge (edge labelled by diamond). The property of simple class graphs that is used is that there can be no sequence of more than one subclass edges (and the construction of Algorithm 1 produces a simple traversal graph from a simple class graph). If I am correct and this is indeed the only place where you use the fact that the input class graph is simple, then step 1 of Algorithm 2 can be re-written so that T' is the set

of all vertices reachable from T following only subclass edges. The benefit is that instead of converting a class graph into a simple class graph we can operate directly on the original class graph. Then we can trade some time for space (although according to my next suggestion we don't even have to). In other words, instead of running the `simplify` transformation on a graph (which could square the number of edges) we can perform a reachability computation at run-time of Algorithm 2.

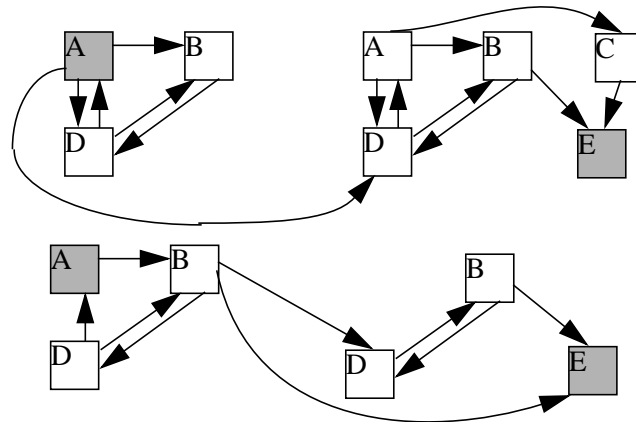
1.3 Dropping Subclass Edges — Simplifying Algorithm 2

The traversal implementation algorithm (consisting of Algorithm 1 and Algorithm 2) is complicated by the fact that there are two kinds of edges in class (and traversal) graphs: subclass edges and reference edges. Subclass edges need to be maintained up to a certain point but in the final traversal graph they seem extraneous. There will never be an instance of an abstract class (hexagon in the figures) in an object graph, so why keep the abstract classes around? We can drop them by adding steps 4d and 4e to Algorithm 1:

- (d) “Collapse” all subclass edges and virtual classes in G' . Formally, for each virtual class u and foreach subclass edge $u \leftrightarrow v$ (my word-processor isn't too good with formulas) add edges $u' \rightarrow v$ for all u' such that $u' \rightarrow u$
- (e) Remove from the graph all virtual classes and their adjacent edges.

After step 4e all subclass edges have been dropped from the graph (since our original graph was simple and the simplicity of the traversal graph is guaranteed from the previous steps of Algorithm 1). In general, this would work even without the assumption that the graph is simple (only we would have to drop all subclass edges explicitly in step 4e above).

Following this approach, the example of Figure 3 (7) would become:



It is easy to see that Algorithm 2 would work exactly the same (e.g., take the example of Figure 4).

Of course, if there are no subclass edges, Algorithm 2 can be rewritten so that step 1 is:
 $T' \leftarrow \{v \mid v \text{ belongs to } T \text{ and } \text{Class}(v) = \text{Class}(\text{this})\}$

Now we can see what Algorithm 2 really is: it computes the intersection of two sets of paths (alternatively, it tests a set of strings for acceptance by an automaton). One set is potentially infinite and represented by the traversal graph. The other set is finite and is represented by the object graph. [Computing intersections of families of paths is what Algorithm 1 is also about, as I am going to argue in Section 2]. In any case, the interesting part is that Algorithm 2 is fairly standard and what we really want is a way to get a representation of a family of paths as the traversal graph. This is the role of Algorithm 1. In Section 2 I will propose

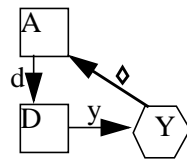
different variations of Algorithm 1 (more expressive encodings of sets of paths) but Algorithm 2 will still be valid.

2 Extending the Model

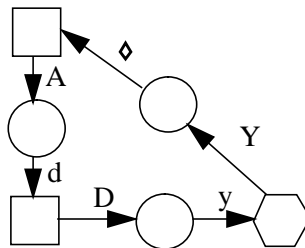
2.1 Automata

All graphs in the paper (class, strategy, traversal, object) are compact representations of sets of paths. In particular, all graphs correspond to finite automata. Automata are used to represent sets of strings (paths in the automaton graph). Algorithm 1 is really a variation on an algorithm computing the cross product of two automata (intersection of the regular expressions that the automata define). I'm not going into a lot of detail since your message indicated that you are already considering an automata-based approach.

Anyway, class graphs are really automata with a minor modification. So if we have a class graph

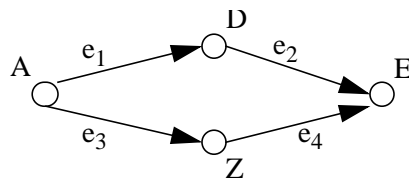


strictly speaking, the corresponding automaton is:

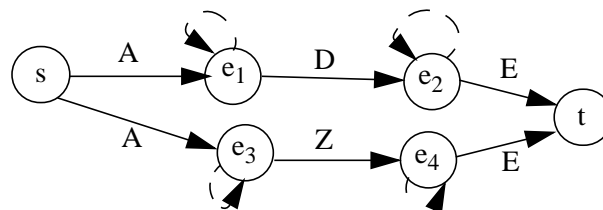


(the circles represent final, or accepting, states). In other words we just introduce extra states so that the language that the automaton accepts is exactly the same as the collection of paths represented by the class graph. The symbols in the language of the automaton are labels and class names.

Similarly, strategy graphs are not exactly automata (in the way they are represented in the paper). For instance, strategy graph



corresponds to a finite automaton with the edges of the strategy graph mapping into states (vertices):



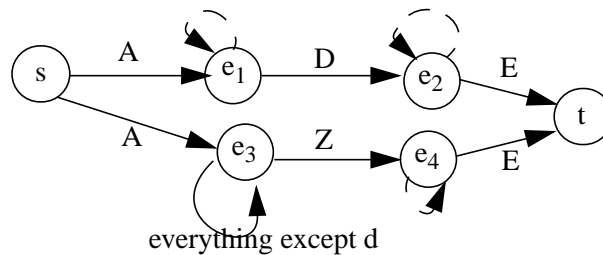
where s is the source (starting state) and t is the target (finish state) of the automaton. For now, let's leave the labels of the self-loop edges empty (meaning that they match any label).

When a class graph and a strategy graph are converted to automata in the above fashion, Algorithm 1 cor-

responds (with the exception of a couple of details) to an algorithm for computing the cross product of two automata (i.e., the automaton representing the intersection of the regular expressions represented by the two automata) and simplifying it by removing empty edges. Standard automata algorithms like automaton minimization can then be applied (the simplification of Section 1.3 is a special case of minimization).

In the paper, strategies and constraint maps are treated differently. If strategies are represented as automata, however, constraint maps can be parts of them. In general, finite automata can represent all regular expressions. It looks like the traversal language used in Demeter/Java (`through`, `bypassing`, etc.) is perfectly expressible in terms of regular expressions. Since there is an algorithm to transform any regular expression into the corresponding finite automaton, all usual constraints should be expressible as part of a strategy. In fact, it may be a good idea to add language primitives to the traversal language so that it corresponds exactly to regular expressions. This way the language will become more expressive.

For instance, we can model both a strategy and a constraint map as the automaton:



This represents the strategy of the example in Figure 3. In general, automata seem to be a better representation for strategies than the combination of constraint maps and strategy graphs.

2.2 Class Graph Predicates

The paper discusses predicates in a constraint map that are defined on the vertices and edges of the class graph. This is slightly different than predicates on the class names and labels. For instance, saying “not k” means that the predicate is false for any edge labelled k. Saying “not A -k-> B” means the predicate is false for the particular k edge between A and B in the class graph. This can be simulated with automata, but the transformation of a class graph to an automaton will be slightly different (need to be able to distinguish edges even if they have the same label so a renaming is needed). It may turn out however that this is not even needed. Since class names are unique in a class graph, we can identify edges by their adjacent classes using regular expressions.