Navigating Object Graphs Using Incomplete Meta-Information

Ahmed Abdelmeged Therapon Skotiniotis

Karl Lieberherr

April 13, 2010

1 Introduction

Retrieving information from data blobs is a fundamental operation that is a precursor for other information processing tasks performed by computer programs. Knowledge about the structure of a data blob is essential for retrieving information. The input to an information retrieval operation is a query that specifies the desired information. In this paper we show how to optimize the retrieval operation when two specific approaches are taken to describe both the structure and the queries.

We specify structure as a pair of graphs, the object graph and the class graph, that are in a *conforms* relationship. Informally, an object graph conforms to the class graph if for each node (edge) in the object graph there exists a corresponding node (edge) in the class graph. A path in the class graph characterizes a set of concrete paths in any conforming object graph.

We adopt a purely structural approach for specifying queries, *i.e.*, queries characterize concrete paths where the desired information is located, yet queries are written as predicates over the class graph. Furthermore, we allow queries to contain a form of iterated wild card. We call such queries, *strategies* and we, again, use graphs to define them. Paths defined in the strategy select paths in the class graph and thus characterizing a set of paths in an object graph. The retrieval operation is a walk in an object graph from a specific source node through object graph paths characterized by the strategy.

A naive approach to carry out an information retrieval operation with strategies is to walk the entire object graph. In [3, 2], it was shown that a compilation phase can be used to construct an automaton, though in disguise, called the traversal graph. The traversal graph is simulated at runtime on the local meta information obtained from the object graph being walked, to guide the rest of the walk away from all unnecessary edges. The automaton is constructed, in principle, by gluing together as many copies of the class graph as there are edges in the strategy. The resulting automaton is non deterministic and can become big in size making it expensive to simulate.

One feature of traversal graphs is that every path between any pair of arbitrary nodes must also be labelled with a path in the class graph. Observing that we are only interested in traversing object graphs that conform to the same class graph, we find this property to be not only unnecessary but also wasteful especially in class graphs with high connectivity. By relaxing this property, the language of a traversal automaton can contain paths whose sequence of labels do not describe class graph paths. In effect, it becomes impossible to retrieve any information about the class graph by reading paths off the traversal automaton. Therefore, we say that the traversal automaton contains incomplete meta information.

One contribution of this paper is to formally define the necessary conditions that an automaton must have so that when used to guide a walk, the walk is sound, complete, and optimal. A second contribution of this paper is an algorithm for constructing a traversal automaton that exploits the aforementioned observation which leads to an automaton with fewer states. Furthermore, for an important restricted subset of strategies, called WYSIWYG strategies [1], our algorithm constructs a traversal automaton that is deterministic.

WYSIWYG strategies are a restricted form of strategies whose behavior is more predictable. The key idea is to always exclude all strategy graph nodes from the set of nodes that can match any iterated wild card. The effect of this restriction is that the traversed paths become closer-looking to the strategy paths, hence comes the name.

The paper starts with section 3 that formally presents a simple model for describing class graphs, object graphs and strategies. In section 4.1 we present our characterization of correct traversal automata. In section 5 we present our algorithm for constructing traversal automata and prove it correct. In section 6 we use our characterization of correct traversal automata on a DFS walk of object graphs.

2 Preliminaries

Graphs, paths and automata play a central role in this presentation. In this section we describe our notation for graphs, paths and automata and their operations.

A *graph* is defined as a pair consisting of two finite sets, a set of nodes and a set of edges, $G = (V, E)$. The set of edges E contains pairs of nodes, *e.g.*, (v_1, v_2), that represent the edge $v_1 \to v_2$ in G. For an edge $e = (v_1, v_2)$ we define $e.\mathit{source} = v_1$ and $e.\mathit{target} = v_2$. For a graph G we define the following functions

- $G.\mathit{nodes}$ returns the set of nodes V in G,
- $G.\,\mathit{edges}$ returns the set of edges E in G,
- $G.\,\mathit{outgoing}(v)$ returns a set of edges $O = \{e \mid e \in G \wedge e.\mathit{source} = v\}$.

A node $v \in G$ is said to be a *sink* node if and only if $G.\,outgoing(v) = \emptyset$. A path p is a sequence of nodes $v_1, v_2, \ldots, v_n$ with operations

- *p.first* returns the first node v_1 in p,
- *p.last* returns the last node v_n in p and
- *p.tail* returns $v_2, \ldots, v_n$.

A path p is said to be *in* graph G, denoted as $path(p, G)$, if for every two consecutive nodes v_i and v_{i+1}, $(v_i, v_{i+1}) \in G.\,edges$. A path p is said to be a *prefix* of another path q, denoted as $p \sqsubseteq q$ if p can be written as $v_1, \ldots, v_n$ and q can be written as $v_1, \ldots, v_n, v_{n+1}, \ldots, v_m$. We also define concatenation of two paths $p = p_1, \ldots, p_n$ and $p' = p'_1, \ldots, p'_m$ as $p \bullet p' = p_1, \ldots, p_n, p'_2, \ldots, p'_m$ if and only if $p_n = p'_1$.

We define the predicate $expansion(p, q, R)$ (read p is an expansion of q, modulo R) to hold between two paths p and q and a set of nodes R iff

1. $p.first = q.first$ and $p.last = q.last$.
2. p is obtainable from q by inserting zero or more nodes and each node is not in the set R.

For a given graph G we also define the predicate $reaches(G, v_1, v_n, allowed)$ to hold iff $allowed \subseteq G.nodes$ and $\exists p : path(p, G)$ and $p.first = v_1$ and $p.last = v_n$ and $\forall i : 1 < i < n$, $v_i \in allowed$.

A walk starting at node s in graph G traverses a set of paths P where $\forall p \in P, path(p, G) \wedge p.first = s$. A walk is called unrestricted if P is maximal.

A *deterministic finite state automaton* (DFA) is a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite set of states
- Σ is a finite set of symbols called the *alphabet*
- δ is a total transition function $\delta : Q \times \Sigma \longmapsto Q$
- an initial state $q_0 \in Q$
- and a set of final states $F \subseteq Q$.

We use the dot notation to refer to elements of the quintuple of a DFA, *e.g.*, $N.\Sigma$ refers to the alphabet Σ of automaton N. Also, we define $stuck(q)$ to be a predicate over the states of a DFA N that holds for a state q iff

1. $q \notin N.F$, and
2. $\forall \alpha : \alpha \in N.\Sigma \Rightarrow (q, \alpha, q) \in N.\delta$.

Let N be a DFA, p be a path such that $p.nodes \in N.\Sigma$, and $q \in N.Q$, we define the function We also define $simulate(N, p) = simulate(N, N.q_0, p)$.

3 The Model

In this section we formally present a simple model of class graphs and strategies. In the interest of simplifying our algorithm and its correctness proof we restrict class graphs to simple unlabeled graphs. We also restrict strategies to be embedded WYSIWYG strategies with one target node that must be a sink node.

A *class graph* ($\mathcal{CG}$) is a graph and a *strategy graph* embedded in a specific class graph ($\mathcal{SG}\langle\mathcal{CG}\rangle$) is also a graph with a distinguished source node *source* and a distinguished target node *target* such that

1. *target* is a sink node.

2. $\mathcal{SG}.nodes \subseteq \mathcal{CG}.nodes$ [*Embedded*].

3. $\forall n \in \mathcal{SG}.nodes$, $reaches(\mathcal{SG}, \mathcal{SG}.source, n, \mathcal{SG}.nodes)$ and $reaches(\mathcal{SG}, n, \mathcal{SG}.target)$. [*No dead nodes*]

4. $\forall e \in \mathcal{SG}.\,edges : \exists p : path(p, \mathcal{CG})$ where $expand(p, (e.source, e.target), \mathcal{SG}.nodes)$ [*Compatibility*]

A path q in a class graph $\mathcal{CG}$ is said to satisfy an embedded WYSIWYG strategy graph ($\mathcal{SG}\langle\mathcal{CG}\rangle$), denoted as $satisfies(q, \mathcal{SG})$, iff there exists a path s in $\mathcal{SG}$ such that $s.first = \mathcal{SG}.source$, $s.last = \mathcal{SG}.target$, and $expansion(q, s, \mathcal{SG}.nodes)$ holds.

An *object graph* that conforms to a class graph ($O\langle\mathcal{CG}\rangle$) is a graph with a total function $meta : \ O.nodes \longmapsto \mathcal{CG}.nodes$ such that: $\forall e \in O.\,edges, \exists e' \in \mathcal{CG}.\,edges$ such that $e.source = meta(e.source)$ and $e'.target = meta(e.target)$. We also say that the type of object o is $meta(o)$. The function $meta$ is lifted to paths in O, *e.g.*, for a path $p = p_1, \ldots, p_n$, $meta(p) = meta(p_1), \ldots, meta(p_n)$. We also say that a path p in O satisfies a strategy graph $\mathcal{SG}$ embedded in $\mathcal{CG}$ if and only if $satisfies(meta(p), \mathcal{SG})$.

4 Characterizing Walks

Given an object graph $O\langle\mathcal{CG}\rangle$, a node $o \in O.nodes$, and a strategy graph $\mathcal{SG}\langle\mathcal{CG}\rangle$, $\mathcal{SG}$ characterizes a set of paths in O. We call these paths valid under $\mathcal{SG}$. We say that a path p in O is valid under $\mathcal{SG}$, denoted $valid(p, \mathcal{SG})$, iff

$$\exists q : path(q, \mathcal{CG}) \wedge meta(p) \sqsubseteq q \wedge satisfies(q, \mathcal{SG})$$

A walk of O, starting at o, is said to be sound and complete under $\mathcal{SG}$ iff the set of paths P traversed by the walk contains all paths that are valid under $\mathcal{SG}$. Such a walk can be used to retrieve the set of paths characterized by $\mathcal{SG}$ in O starting at o.

To obtain a sound and complete walk of an object graph O under $\mathcal{SG}$, we use a traversal automaton to guide (Definition 1) an unrestricted walk.

4.1 Characterizing Traversal Automata

A traversal automaton N for class graph $\mathcal{CG}$ and a strategy graph $\mathcal{SG}$ (Definition 1), is an automaton with $\Sigma = \mathcal{CG}.nodes$ that is used to guide a walk of an object graph O conforming to $\mathcal{CG}$ only through paths that satisfy the given strategy. Intuitively, for a traversal automaton N we have that:

- all words w in the language $L(N)$ satisfy the strategy.
- all paths in the class graph that satisfy the strategy are in $L(N)$.
- Simulating N on any path that is not a prefix of a path satisfying the strategy, leaves N in a stuck state.

Definition 1. ***(Traversal Automaton)***
An automaton, $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ *is a* traversal automaton *for a class graph* $\mathcal{CG}$ *and a strategy* $\mathcal{SG}$ *if and only if*

1. $\Sigma = \mathcal{CG}.nodes$,
2. $\forall q \in L(N)$, $satisfies(q, \mathcal{SG})$,
3. $\forall q : satisfies(q, \mathcal{SG}) \iff path(q, \mathcal{CG}) \wedge q \in L(N)$,
4. $\forall p : path(p, \mathcal{CG}) \wedge (\exists q : path(q, \mathcal{CG}) \wedge satisfies(q, \mathcal{SG}) \wedge p \sqsubseteq q) \Leftrightarrow \neg stuck(simulate(N, p))$.

5 Constructing A Traversal Automaton

We construct a traversal automaton using the operation `buildTA`$(\mathcal{CG}, \mathcal{SG}\langle\mathcal{CG}\rangle)$ which takes as inputs, a class graph $\mathcal{CG}$, and, a strategy graph $\mathcal{SG}$ embedded in $\mathcal{CG}$, $(\mathcal{SG}\langle\mathcal{CG}\rangle)$. The operation is defined in Algorithm 1.

We also prove that `buildTA` returns a traversal automaton according to Definition 1 (Theorem 2). Furthermore, `buildTA` returns a DFA (Theorem 1).

Theorem 1. ***(*`buildTA` *constructs a DFA)***
For all class graphs $\mathcal{CG}$ *and for all embedded strategies* $\mathcal{SG}\langle\mathcal{CG}\rangle$, `buildTA`$(\mathcal{CG},\mathcal{SG}\langle\mathcal{CG}\rangle)$ *returns an automaton* N *that is a* DFA.

Proof.

1. To show that N is indeed a DFA, we show that Algorithm 1 never adds two tuples to $N.\delta$ with the same first two components (current state and symbol). There are five locations (lines 9, 11, 16, 21, 24) in the algorithm where we update δ.
   (a) By construction, the symbols mentioned in the line 9 (q_m, r, q_m) are always different from the set added in line 11 (q_m, n, q_n). The symbol s cannot be a strategy graph node while n is always a strategy graph node.

```
Input:   cg : a class graph
         sg : a strategy graph embedded in cg
Output:  A traversal automaton $N = \langle Q, \Sigma, \delta, q_0, F \rangle$.
$Q = \{q_0\} \cup \{q_v \mid v \in$ sg $.nodes\}$;
$\Sigma =$ cg$.nodes$;
let $A =$ cg$.nodes \setminus$sg$.nodes$;
foreach $e \in$ sg$.edges$ do
    $m = e.source$;
    $n = e.target$;
    let $R = \{v \in$ cg$.nodes \mid reaches($cg$, m, v, A) \wedge$
                                  $reaches($cg$, v, n, A) \wedge$
                                  $v \notin \{m, n\}\}$;
    foreach $r \in R$ do
        $\delta = \delta \cup \{(q_{\mathrm{m}}, r, q_{\mathrm{m}})\}$;
    end
    $\delta = \delta \cup \{(q_{\mathrm{m}}, n, q_{\mathrm{n}})\}$;
end
$s =$ sg$.source$;
$t =$ sg$.target$;
$F = \{q_{\mathrm{t}}\}$;
$\delta = \delta \cup \{(q_0, \mathrm{s}, q_{\mathrm{s}})\}$;
$Q' = Q \cup \{q_\perp\}$ ;
foreach $\alpha \in \Sigma$ do
    foreach $q \in Q$ do
        if $\nexists q' : (q, \alpha, q') \in \delta$ then
            $\delta = \delta \cup \{(q, \alpha, q_\perp)\}$;
        end
    end
    $\delta = \delta \cup \{(q_\perp, \alpha, q_\perp)\}$
end
return $\langle Q', \Sigma, \delta, q_0, F \rangle$
```

Algorithm 1: `buildTA`(cg,sg)

(b) By construction, two tuples added by the line (q_m, n, q_n) can only have the same first two components if they are identical. Which cannot happen because $\mathcal{SG}$ is not a multi-graph.

(c) At line 16 (q_0, s, q_s) where s is $\mathcal{SG}.source$. The state q_0 does not correspond to any class graph node and thus no transition was defined for q_0 because of lines 9 and 11

(d) At line 21 the inner loop only adds a transition to $q_\perp$ if there is no transition already defined for $q \in Q$ and $\alpha \in \Sigma$. Therefore δ does not contain 3-tuples whose first two elements are the same.

(e) Line 24 adds transitions (loops) on $q_\perp$ for each element in Σ and these are the only transitions that start in $q_\perp$ and thus there are not two transitions that start from $q_\perp$, have the same label.

$\square$

Theorem 2. ***(Correctness of*** `buildTA`***)***
Given a class graph $\mathcal{CG}$ *and a strategy graph* $\mathcal{SG}$*,* `buildTA`$(\mathcal{CG}, \mathcal{SG}\langle\mathcal{CG}\rangle)$ *is a traversal automaton for* $\mathcal{CG}$ *and* $\mathcal{SG}$*.*

Proof. let $N =$ `buildTA`$(\mathcal{CG}, \mathcal{SG}\langle\mathcal{CG}\rangle)$, we show that N satisfies all the properties of a traversal automaton.

1. Immediate.

2. We start by showing that every word r in $L(N)$ must start with $\mathcal{SG}.source$ and end with $\mathcal{SG}.target$. By definition we know that r is in $L(N)$ iff we can trace a path through N starting by $N.q_0$ and ending with a state in $N.F$ whose sequence of labels are the same as r. By construction, there is one edge going out of q_0 whose label is $\mathcal{SG}.source$. Therefore, every word in $L(N)$ starts with $\mathcal{SG}.source$. Also, by construction, we have a single final state labeled $q_{\mathcal{SG}.target}$. By the definition of $\mathcal{SG}$, $\mathcal{SG}.target$ is a sink node. Therefore, all the edges leading into $q_{\mathcal{SG}.target}$ are created by line 11 (q_m, n, q_n), *i.e.*, all the edges leading into the final state of N are labelled $\mathcal{SG}.target$.

   Now we show that any word $r = r_1, r_2, \ldots r_n$ in $L(N)$ satisfies the strategy. Let $s = s_1, s_2, \ldots, s_k$ be the longest subsequence of r such that $s.nodes \in \mathcal{SG}.nodes$. Therefore $expansion(r, s, \mathcal{SG}.nodes)$ holds. As we have shown above $r_1 = \mathcal{SG}.source$, therefore, $s_1 = r_1 = \mathcal{SG}.source$ because otherwise, we can prepend $\mathcal{SG}.source$ to s and get a longer subsequence of r whose nodes are in $\mathcal{SG}.nodes$. Likewise, $s_k = r_n = \mathcal{SG}.target$. By the definition of *satisfies*, if we can show that s is in $\mathcal{SG}$, then we can conclude that $satisfies(r, \mathcal{SG})$.

First, we show that s is in $L(N)$ and then show that $s \in \mathcal{SG}$. By construction, the nodes in r but not in s are added as self loop labels by line 9 (q_m, r, q_m). Since we can trace a path through N whose labels are the same as r, we can also trace a path through N whose labels are the same as s by avoiding self loops.

Finally, we show that s is in $\mathcal{SG}$. Since s is a word in $L(N)$, by construction we know that each symbol s_i (in $\mathcal{SG}.nodes$) labels a transition that leads to state q_{s_i} (added by line 11 (q_m, n, q_n)). Furthermore, every two consecutive symbol s_i and s_{i+1} show as labels on two transitions, one leading into state q_{s_i} and the other leading out of q_{s_i} and into $q_{s_{i+1}}$. The second transition can only exist if there is an edge (s_i, s_{i+1}) in $\mathcal{SG}.\,edges$. Therefore, s is a path in $\mathcal{SG}$.

3. We first show the forward direction, given $\mathit{satisfies}(p, \mathcal{SG})$ we show that $\mathit{path}(p, \mathcal{CG})$ and $p \in L(N)$. By definition of *satisfies* we know that p is a path in $\mathcal{CG}$, thus $\mathit{path}(p, \mathcal{CG})$ holds.

   Let $p = p_0, p_1, \ldots, p_n$ be a path in $\mathcal{CG}$ and $\mathit{satisfies}(p, \mathcal{SG})$. We show that p can only moves N from the initial state to the final state.

   We note that, by construction, only symbols in $\mathcal{SG}.nodes$ can move N through one of the transitions added by line 11 (q_m, n, q_n) to a different state other than $q_\perp$. Symbols in $\mathcal{CG}.nodes \setminus \mathcal{SG}.nodes$ can either move N through one of the self-loops added by line 9 (q_m, r, q_m) to the same state or to the stuck state $q_\perp$.

   We first show that nodes in p but not $\mathcal{SG}.nodes$ can not move N to the stuck state $q_\perp$ and therefore can not change the state of N. Suppose that $p_i \notin \mathcal{SG}.nodes$ is the first node in p to move N from a non stuck state into the stuck state $q_\perp$. Let p_h be its closest predecessor node that is in $\mathcal{SG}.nodes$ and p_j be its closest successor node that is in $\mathcal{SG}.nodes$. p_h and p_j always exist and are uniquely identifiable by the definition of *satisfies*. Then p_i must not be mentioned on any self-loop at q_{p_h}, otherwise, p_i will not move N to $q_\perp$. For that to happen, p_i must be unreachable in $\mathcal{CG}$ from p_h through nodes not in $\mathcal{SG}.nodes$ or cannot reach p_j in $\mathcal{CG}$ through nodes not in $\mathcal{SG}.nodes$ or both. The first condition, contradicts our assumption that p_h can reach p_i through a path in $\mathcal{CG}$ (a subpath of p) that does not contain any node in $\mathcal{SG}.nodes$. The second condition, contradicts our assumption that p_j is reachable from p_i through a path in $\mathcal{CG}$ that does not contain any node in $\mathcal{SG}.nodes$.

   We now show that nodes in p that are in $\mathcal{SG}.nodes$ can only move N to its only final state $q_{\mathcal{SG}.target}$. By definition of *satisfies*, there exists a path $s = s_0, \ldots, s_k$ in $\mathcal{SG}$ that contains all nodes in p that are in $\mathcal{SG}.nodes$

and $p_0 = s_0 = \mathcal{SG}.source$, which by construction moves N from the initial state q_0 to $q_{\mathcal{SG}.source}$. For any following symbol s_i where $i > 0$, there must be an edge (s_{i-1}, s_i) in $\mathcal{SG}.edges$ because, by the definition of *satisfies*, s is in $\mathcal{SG}$. Therefore, by construction, there is a transition $(q_{s_{i-1}}, s_i, q_{s_i})$ (added by line 11 $(q_m,\ n,\ q_n)$) in N. Furthermore, by the definition of *satisfies*, $p_n = s_k = \mathcal{SG}.target$, which moves N to $q_{\mathcal{SG}.target}$ which is the only final state of N by construction.

For the reverse direction we are given that $path(p, \mathcal{CG})$ and that $p \in L(N)$ and we need to show that $satisfies(p, \mathcal{SG})$. Immediate by clause 2 of Definition 1 proven in the preceding step.

4. ($\Rightarrow$ direction) From the preceding property, $q \in L(N)$. Therefore, every prefix of q must not move N from the initial state to any stuck state. Otherwise, $q \notin L(N)$.

   ($\Leftarrow$ direction)

   Given N at state q_0 consider a simulation of N on input $p = p_1, p_2, \ldots, p_n$ that moves N to a state q_m we need to show that $\exists q : path(q, \mathcal{CG})$ and $satisfies(q, \mathcal{SG})$ and $p \sqsubseteq q$. We proceed by cases on $p.last$.

   - $p.last \in N.F$ then select $p = q$, and we know that $q \in L(N)$ thus $satisfies(q, \mathcal{SG})$ and $p \sqsubseteq q$.
   - $p.last = q_0$. By Lemma 1 we know that there exists a path p in $\mathcal{CG}$ and $p.first = \mathcal{SG}.source$ such that given N at state $q_{\mathcal{SG}.source}$ a simulation of N on $p.tail$ moves N to $q_{\mathcal{SG}.target}$. By the construction of N we know that q_0 has one and only one transition $(q_0, \mathcal{SG}.source, q_{\mathcal{SG}.source})$ that moves N to a non-stuck state. Thus given N at state q_0 a simulation of p will move N to $q_{\mathcal{SG}.target}$. Thus, $p = q$, and we know that $q \in L(N)$, therefore $satisfies(q, \mathcal{SG})$ and $p \sqsubseteq q$.
   - $p.last \in \mathcal{SG}.nodes \backslash N.F$. By construction of N we know that N just completed a transition between two distinct states q_1, q_2. We have already shown (Lemma 1) that from any state $q' \in N.Q \backslash \{q_0, q_\perp, \}$ we can always find a path r in $\mathcal{CG}$ such that a simulation of N at state q' on input $r.tail$ moves N to $\mathcal{SG}.target$. We also know that p moves N from q_0 to q_m where q_m is not a stuck state or an accepting state. We also know that $r.first = p.last$ thus we can construct $q = p \bullet r$ and $path(q, \mathcal{CG})$ and $p \sqsubseteq q$.

     We need to show that $satisfies(q, \mathcal{SG})$. But we have already shown that for all $w \in L(N)$ then $satisfies(w, \mathcal{SG})$ and we know that $q \in L(N)$.
   - $p.last \notin \mathcal{SG}.nodes$. We know that a simulation of N at state q_0 on input p moves N in state q_m. By construction of N, $(q, v, q) \in N.\delta$

iff $v \in \mathcal{CG}.nodes$ and $v \notin \mathcal{SG}.nodes$. Thus the last transition taken by N while simulating p was a self loop on state q_m. We also know by construction of N that for all $v \in \mathcal{CG}.nodes$, $(q_m, v, q_m) \in N.\delta$ iff there exists $q_k \in N.Q$ and path $r = r_1, r_2, \ldots, r_n$ in $\mathcal{CG}$ such that

$$\begin{array}{ll} r_1 = m & \wedge \\ r_n = k & \wedge \\ \forall j : 1 < j < n : r_j \notin \mathcal{SG}.nodes & \wedge \\ \exists i : 1 < i < n : r_i = v & \end{array}$$

We can thus select transitions

$$(q_m, r_i + 1, q_m), (q_m, r_i + 2, q_m), \ldots, (q_m, r_{n-1}, q_k), (q_m, r_n, q_k)$$

By construction we know that the transitions labeled $r_{i+1}, r_{i+2}, \ldots, r_{n-1}$ are defined in $N.\delta$. We also know by constrution that $(q_m, r_n, q_k) \in N.\delta$. We have shown (Lemma 1) that for any state $q_i \in N.Q$ such that $q_i \notin \{q_0, q_\perp, q_{\mathcal{SG}.target}\}$ we can construct a path p' in $\mathcal{CG}$ such that given an N at state q_i simulating N on input $p'.tail$ moves N from state q_i to $q_{\mathcal{SG}.target}$. We can thus construct the path $q = p \bullet (r_i, \ldots r_n \bullet p')$ such that $path(q, \mathcal{CG})$ and $p \sqsubseteq q$. Since $q \in L(N)$ we can also conclude that $satisfies(q, \mathcal{SG})$.

□

Lemma 1. *For every state q_v in $N.Q \backslash \{q_\perp, q_0, q_{\mathcal{SG}.target}\}$, $\exists p' : path(p', \mathcal{CG}) \wedge p'.first = v$ such that having N at state q_v and simulating $p'.tail$ moves N from q_v to $q_{\mathcal{SG}.target}$.*

Proof. By the definition of $\mathcal{SG}$ (no dead states condition),

$$\exists r : path(r, \mathcal{SG}) \wedge r.first = v \wedge r.last = \mathcal{SG}.target$$

By the definition of $\mathcal{SG}$ (compatibility condition) we know that

$$\begin{aligned} \forall e \in \mathcal{SG}.\,edges : \exists r' : path(r', \mathcal{CG}) \wedge \\ expansion(r', (e.source, e.target), \mathcal{SG}.nodes) \end{aligned}$$

Let $r' = r'_1, \ldots, r'_m$. By the defnition of *expansion* and by construction of N, we know that $r'_1 = e.source$ and $r'_m = e.target$ and

$$\begin{array}{lll} \forall i : & 1 < i < m : & r'_i \notin \mathcal{SG}.nodes \quad \wedge \\ & & (q_{r'_1}, r'_i, q_{r'_1}) \in N.\delta \end{array}$$

Thus r'_i cannot move N to a stuck state. Observe that $q_{r'_1} = q_{e.source}$ and $q_{r'_n} = q_{e.target}$ for $e \in \mathcal{SG}.\,edges$. Thus

$$(q_{r'_1}, r'_n, q_{r'_n}) = (q_{e.source}, r'_n, q_{e.target})$$

and by construction of N we know that $(q_{e.source}, r'_n, q_{e.target}) \in N.\delta$. We can thus select

$$(q_{r'_1}, r'_2, q_{r'_1})(q_{r'_1}, r'_3, q_{r'_1}) \ldots (q_{r'_1}, r'_n, q_{r'_n})$$

and move N from $q_{r'_1}$ to $q_{r'_n}$.

We can repeat the same process for each strategy edge e_j connecting two consecutinve nodes in r and obtain a set of paths π_j such that each $\pi_j.tail$ moves N from state $q_{e_j.source}$ to $q_{e_j.target}$. Concatenating all π_j returns a path π in $\mathcal{CG}$ such that given N at state q_v, a simulating of N on $\pi.tail$ moves N from q_v to $q_{\mathcal{SG}.target}$. □

6 Walking an Object Graph

In this section, we define the operation `walk`$(O\langle\mathcal{CG}\rangle, N, o_s)$ which takes an object graph $O\langle\mathcal{CG}\rangle$ that conforms to a class graph $\mathcal{CG}$, and a traversal automaton N, and an object (o_s) in $O.nodes$ and walks O guided by N. We also prove that `walk` returns a set P that is the largest set that contains all paths p in O such that p is a prefix of a path q in O and q satisfies $\mathcal{SG}$.

```
Input:  og : an object graph that conforms to a class graph cg
        ta : a traversal automaton
        os : an object in the object graph.
Output: a walk of og. guided by ta
1 return walkHelper (og,ta,ta.$q_0$,{os},$\emptyset$,{os})
```

Algorithm 2: `walk`(og,ta,os)

Theorem 3. *For all $\mathcal{CG}$ and for all traversal automata N, and all object graphs $O\langle\mathcal{CG}\rangle$ and o_s such that $o_s \in O.nodes$ and $meta(o_s) = \mathcal{SG}.source$, let $P =$* `walk`(O, N, o_s) *then for all paths p in P $valid(p, \mathcal{SG})$ holds.*

Proof.

By induction on the depth of recursive calls to `walkHelper`. Consider the first call to `walkHelper`. $P = \{o_s\}$ and we know that $meta(o_s) = \mathcal{SG}.source$. We also have that N is in state q_0 and by construction of N we know that there is a transition $(q_0, meta(o_s), q_{\mathcal{SG}.source})$. By Lemma 1 we know that there exists a path q such that $path(q, \mathcal{CG})$ and a simulation of N on $\mathcal{SG}.source, q$ moves N to $\mathcal{SG}.target$. Thus, $\mathcal{SG}.source, p \in L(N)$ and by Definition 1 we can deduce that $satisfies(\mathcal{SG}.source, p, \mathcal{SG})$

Assume that after k recursive calls to `walkHelper` the theorem holds and consider the $k+1$ recursive call. By the definition of `walkHelper` there are two cases that recursively call the function, lines 9 and 17.

```
Input:  og      : an object graph that conforms to a class graph cg
        ta      : a traversal automaton whose alphabet is cg.$nodes$
        state   : the current state of ta after simulating $meta$(list)
        reach   : set of nodes to process
        visited : set of nodes to already processed
        P       : set of paths in og
Output: set of paths P where for each path $p \in$ P there exists a path
        $q \in$ og such that $meta(q) \in L$(ta) and $p \sqsubseteq q$
if reach = $\emptyset$ then
    return P;
end
oc = reach.randomElement ();
reach' = reach.remove (oc);
visited' = visited $\cup$ {oc };
state' = ta.$\delta$(state, meta (oc));
if state' = $q_\perp$ then
    walkHelper (og,ta,state, reach', visited', P);
end
reach' = reach' $\cup$ {$o'$ | $e \in$ og. $outgoing$(oc) $\wedge$ $e.target = o'$};
foreach $p \in$ P do
    if $p.last$ = oc' $\wedge$ (oc', oc) $\in$ og then
        P = P $\cup$ {$p$.oc};
    end
end
walkHelper (og,ta,state', reach', visited', P)
```

Algorithm 3: `walkHelper`(og,ta,state,reach, visited, P)

1. Line 9. The set P remains unchanged. By the induction hypothesis $\forall p \in P : valid(p, \mathcal{SG})$

2. Line 17. We know that $\texttt{ta}.\delta(\texttt{state},\ meta(\texttt{oc})) \neq q_\perp$. For each $p \in P$ we update P to include $p, \texttt{oc}$ if there exists an edge $(p.last, \texttt{oc}) \in O$. By the induction hypothesis we know that $valid(p, \mathcal{SG})$. Let q' be the automaton's new state after reading in $meta(\texttt{oc})$. Since $q' \neq q_\perp$ we know that by Lemma 1 there exists an input r such that simulating N at state q' on r moves N to its target node. Let $h = meta(p), meta(\texttt{oc}), r$, then $h \in L(N)$ and by Definition 1 we can deduce $satisfies(h, \mathcal{SG})$

□

Theorem 4. *For all $\mathcal{CG}$ and for all traversal automata N, and all object graphs $O\langle\mathcal{CG}\rangle$ and o_s such that $o_s \in O.nodes$ and $meta(o_s) = \mathcal{SG}.source$, let $P =$ `walk`(O, N, o_s) then P is maximal.*

Proof. The definition of `walkHelper` exhaustively processes all reachable nodes (line 11) and terminates after all reacheable nodes have been processed. For each node `oc` such that $meta$ (`oc`) moves N to a non stuck state, the algorithm adds a new path p' in P if $(p.last, \texttt{oc}) \in O$ for each $p \in P$. P starts with the start node o_s and creates new paths from each neighbor explored thus covering all possible paths starting from o_s in O. □

7 Related Work

8 Conclusion

References

[1] Karl Lieberherr and Boaz Patt-Shamir. The refinement relation of graph-based generic programs. In M. Jazayeri, R. Loos, and David Musser, editors, *1998 Schloss Dagstuhl Workshop on Generic Programming*, pages 40–52. Springer, 2000. LNCS 1766.

[2] Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[3] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.