Parametric Regular Path Queries[*]

Yanhong A. Liu Tom Rothamel Fuxiang Yu Scott D. Stoller
Nanjun Hu

Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794
{liu,rothamel,fuxiang,stoller,nanjun@cs.sunysb.edu}

ABSTRACT

Regular path queries are a way of declaratively expressing queries on graphs as regular-expression-like patterns that are matched against paths in the graph. There are two kinds of queries: existential queries, which specify properties about individual paths, and universal queries, which specify properties about all paths. They provide a simple and convenient framework for expressing program analyses as queries on graph representations of programs, for expressing verification (model-checking) problems as queries on transition systems, for querying semi-structured data, etc. Parametric regular path queries extend the patterns with variables, called parameters, which significantly increase the expressiveness by allowing additional information along single or multiple paths to be captured and related.

This paper shows how a variety of program analysis and model-checking problems can be expressed easily and succinctly using parametric regular path queries. The paper describes the specification, design, analysis, and implementation of algorithms and data structures for efficiently solving existential and universal parametric regular path queries. Major contributions include the first complete algorithm and data structures for directly and efficiently solving universal parametric regular path queries, detailed complexity analysis of algorithms for solving existential and universal queries, detailed analytical and experimental performance comparison of variations of the algorithms and data structures, and investigation of efficiency tradeoffs between different formulations of queries.

1. INTRODUCTION

Many important analysis problems can be expressed as graph queries. This includes program analyses that are essential for improving program performance, safety, security, etc., model-checking problems for verification of sequential and concurrent systems, data mining of semi-structured data, and so on. General frameworks for graph queries, together with tools based on them, allow users to easily specify and efficiently perform a wide variety of analysis tasks, saving enormous effort compared to implementing each analysis separately.

Parametric regular path queries. Regular path queries are a way of declaratively expressing queries on graphs as regular-expression-like patterns that are matched against paths in the graph. They provide a simple and convenient framework for expressing program analyses as queries on graph representations of programs [6] and for expressing model-checking problems as queries on transition systems. Regular path queries are also important in analyzing semi-structured data in database systems [1], particularly data in XML [18], which is increasingly used for representing data, including programs as data.

Regular expression patterns can capture simple but common and important properties easily, even though they are not as powerful as languages in more sophisticated frameworks. The combined power and simplicity of regular expressions contribute to their wide use in computing, from languages and compilers, to database and web information retrieval, to operating systems and security, etc.

Parametric regular path queries extend the patterns with variables, called parameters, which significantly increase the expressiveness by allowing additional information along single or multiple paths to be captured and related, and the amount of such information is not bounded by the size of the pattern. This extension enables analysis of significantly many more important properties about dependencies, concurrency, resource usage, etc. The regular expression patterns used to analyze these properties are simple, easy to write, and succinct.

General algorithms have been studied for solving simpler regular path queries, in particular, queries involving uncorrelated paths [20] and queries containing no variables [12, 6]. A method was also proposed to code parametric regular path queries using logic programs [7]. What have been lacking are complete algorithms and data structures for solving parametric regular path queries directly, efficiently, and with precise complexity analysis.

This paper. This paper studies existential and universal parametric path queries and their use in program analysis and model checking. Existential queries specify properties about individual paths. Universal queries specify properties

[*]This work was supported in part by ONR under grants N00014-01-1-0109 and N00014-02-1-0363 and NSF under grants CCR-0306399, CCR-0204280, CCR-0205376, and CCR-9876058.

about all paths and are much harder to solve. We describe the precise specification, design, analysis, and implementation of complete algorithms and data structures for efficiently solving both kinds of queries.

Major contributions of this paper include the first complete algorithm and data structures for directly and efficiently solving universal parametric regular path queries, detailed complexity analysis of algorithms for solving existential and universal queries, detailed analytical and experimental performance comparison of variations of the algorithms and data structures, and investigation of efficiency tradeoffs between different formulations of queries.

The rest of the paper is organized as follows. Section 2 defines parametric regular path queries and shows how a variety of practical program analysis and model checking problems can be expressed easily and succinctly as such queries. Sections 3 and 4 describe algorithms, data structures, and time and space complexities for solving existential and universal queries, respectively. Section 5 discusses tradeoffs and extensions, as well as improvements in the complexity analysis and performance and in the query language and usage. Section 6 describes implementation and experiments. Section 7 compares with related work and concludes.

2. PARAMETRIC REGULAR PATH QUERIES

When analyzing a program graph or checking a transition graph, two kinds of questions are often asked: "Will something happen along some path of the graph?" and "Will something happen along all paths of the graph?" We refer to these as existential and universal queries, respectively. In parametric regular path queries, "something" is expressed as an extended regular expression that contains parameters, hereafter called a *pattern*. To answer these queries, one needs to examine paths of the graph and check, for an existential query, whether some path matches the pattern and, for a universal query, whether all paths match the pattern.

We define the queries problem precisely using program graphs as examples. We then give examples in program analysis and model checking. These examples show that parametric regular path queries are powerful enough to express a wide variety of useful analysis problems, and that the queries are succinct and easy to write. In contrast, implementing these analyses separately without a framework would require a significant effort. Thus, a tool for efficient evaluation of such queries can greatly reduce the effort needed to implement new analyses. Tools based on more expressive analysis frameworks, such as set constraints or temporal logic, could be used for these analyses, but those frameworks and tools are typically also more complicated and hence accessible only to more sophisticated users.

2.1 Definition of the problem

Graphs. We consider edge-labeled directed graphs. For example, in program graphs, vertices correspond to program points, and labeled edges correspond to operations. These graphs are similar to control-flow graphs, but the roles of vertices and edges are reversed.

We generally use edge labels that reflect only information relevant to the analysis of interest. For example, consider an assignment statement `a:=5` in a program. If we are interested in analyzing reaching definitions, then this statement may be represented by the label **def**(`a`), indicating a definition of (i.e., assignment to) `a`. If we are interested in constant folding, then this statement might be represented by the label **def**(`a`,`5`).

We refer to names, such as **def**, that represent abstract aspects of edge labels, as *constructors*, and we display them in boldface. We refer to names, such as variable name `a` or literal `5`, that represent concrete aspects of edge labels, as *symbols*, and we display them in typewriter font. An *edge label* is a constructor applied to zero or more arguments, where an argument may be a symbol or, recursively, a constructor application.

Parametric regular-expression patterns. We consider patterns that are regular expressions whose alphabet contain parameterized elements. We display parameters in math italic font. For example, **def**(x)***use**(x) represents a consecutive sequence of definitions of a same variable, denoted by x, ended by a use of the same variable. We also allow wildcards and negations, denoted by _ and $\neg$, respectively. For example, _***def**(`a`) represents a sequence of zero or more arbitrary labels followed by a definition of `a`, **def**(_) represents a definition of any variable, $\neg$**def**(`a`) represents anything that is not a definition of `a`, and **def**($\neg$`a`) represents a definition of any variable but `a`.

Precisely, the alphabet of the patterns contains elements, called *transition labels*, that may be a constructor applied to zero or more arguments, its negation, or a wildcard, where an argument may be: (1) a symbol in the edge labels, or its negation; (2) a parameter that can be instantiated to symbols in the edge labels, or its negation; or (3) recursively, a transition label. Note that we only consider matching of parameters with symbols; a generalization to consider matching with constructor applications is possible.

Substitutions and matching. To match patterns against graph paths, we need the notion of substitution. A *substitution* is a map from parameters in the pattern to symbols in the graph. For example, substitution $\{x \mapsto$ `a`$, y \mapsto$ `b`$\}$ maps x to `a` and maps y to `b`. A substitution θ applied to a pattern p, denoted $\theta(p)$, replaces the parameters in p according to the mappings in θ.

We say that an edge label el *matches* a transition label tl under a substitution θ if $\theta(tl)$ contains no parameters and el *matches* $\theta(tl)$, where an edge label el *matches* a transition label tl that contains no parameters if one of the following conditions holds: (1) $el = tl$; (2) $tl = \_$; (3) $tl = \neg tl_1$ and, recursively, el does not match tl_1; (4) $el = f(el_1, ..., el_n)$, $tl = f(tl_1, ..., tl_n)$, and recursively el_i matches tl_i for $i = 1..n$, where f is a constructor. We say that a path g in a graph G *matches* a sentence p accepted by a pattern P under a substitution θ if the sequence of edge labels on g matches the sequence of transition labels on p under θ label by label.

```
a:=1;
if(f(a))
  a:=2;
else
  b:=1;
g(b);
```

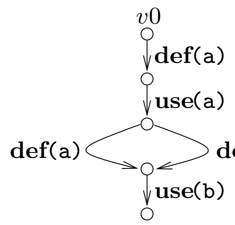

Figure 1: An example program and its program graph.

Figure 1 gives an example program graph for analyses about variable definitions and uses. Given a pattern $(\neg\mathbf{def}(x))^*\mathbf{use}(x)$, the path $\mathbf{def}(\mathtt{a})\mathbf{use}(\mathtt{a})\mathbf{def}(\mathtt{a})\mathbf{use}(\mathtt{b})$ in the graph matches the sentence $(\neg\mathbf{def}(x))^3\mathbf{use}(x)$ accepted by the pattern under the substitution $\{x \mapsto \mathtt{b}\}$.

Parametric regular path queries. We define two kinds of queries:

Existential queries: Given an edge-labeled directed graph G where labels may have parameters, a vertex $v0$ in G, and a parametric regular-expression pattern P, compute all pairs of vertex v in G and a substitution θ for parameters in P such that *there exists* a path from $v0$ to v in G that matches some sentence accepted by P under θ.

Universal queries: Given an edge-labeled directed graph G where labels may have parameters, a vertex $v0$ in G, and a parametric regular-expression pattern P, compute all pairs of vertex v in G and substitution θ for parameters in P such that *every* path from $v0$ to v in G matches some sentence accepted by P under θ.

2.2 Program analysis examples

We describe how a range of common and important properties can be expressed easily and succinctly using parametric regular path queries. Some data-flow analysis problems are backward and are expressed using *backward queries*, meaning that all edges in the graph are reversed before the query is evaluated. We assume the program has a unique entry point and a unique exit point, as the default starting vertex for forward queries and backward queries, respectively.

Uninitialized variables. Checking for uses of uninitialized variables is a classic data-flow analysis. Consider a program graph with labels of the forms $\mathbf{def}(\mathtt{a})$ and $\mathbf{use}(\mathtt{a})$, representing definitions and uses, respectively, of some variable a. To find vertices that immediately follow a use of an uninitialized variable, we can use an existential query with pattern $(\neg\mathbf{def}(x))^*\mathbf{use}(x)$. To find only the first use of each uninitialized variable along each path, we can use $(\neg(\mathbf{def}(x)|\mathbf{use}(x)))^*\mathbf{use}(x)$. The substitution for x associated with a vertex in the result set identifies the offending program variable.

Live variables. Finding live variables at each program point is another classic data-flow analysis. A variable is live at a program point if it is used before being defined on some path from that point. We can use a backward existential query with pattern $\_^*\mathbf{use}(x)(\neg\mathbf{def}(x))^*$. It returns all pairs $\langle v,\theta\rangle$ such that at program point v, variable $\theta(x)$ is live. Other classic analyses, such as reaching definitions and dead-code analysis, can also be formulated easily as parametric regular path queries.

Available expressions. An expression a o b for some variables a and b and operation o is available at a program point v if, on every path from the program entry point to v, there is a computation of a o b, represented using label $\mathbf{exp}(\mathtt{a,o,b})$, that is followed by no definitions of a or b on that path. We can use a universal query with pattern $\_^*\mathbf{exp}(x,op,y)(\neg(\mathbf{def}(x)|\mathbf{def}(y)))^*$. This query returns all pairs $\langle v,\theta\rangle$ such that at program point v, expression $\theta(x)\,\theta(op)\,\theta(y)$ is available.

Constant folding. A variable a at a program point v has a constant value k if, on every path from the program entry point to v, there is a definition of a to the constant k, represented using label $\mathbf{def}(\mathtt{a,k})$ instead of $\mathbf{def}(\mathtt{a})$, followed by no definitions of a. Any use of a on outgoing edges of v can be replaced by k. We can use a universal query with pattern $\_^*\mathbf{def}(x,c)(\neg(\mathbf{def}(x)|\mathbf{def}(x,\_)))^*$. It returns all pairs $\langle v,\theta\rangle$ such that at program point v, variable $\theta(x)$ has constant value $\theta(c)$ following any possible path from the entry point.

Files. Operations in typical file I/O libraries, represented using labels of the forms $\mathbf{open}(\mathtt{f})$, $\mathbf{close}(\mathtt{f})$, and $\mathbf{access}(\mathtt{f})$ for file f, must be used following certain sequencing constraints. In particular, a file must be open at the time it is accessed, and open files must be subsequently closed. To detect respective violations, we can use an existential query with pattern $(\epsilon|\_^*\mathbf{close}(f))(\neg\mathbf{open}(f))^*\mathbf{access}(f)$, where ϵ matches the empty string, and a backward existential query with pattern $(\neg\mathbf{close}(f))^*\mathbf{open}(f)$. Other properties, such as that a file must be open at the time close is called on it and that a file is possibly used after it is opened, can also be easily expressed in our framework.

Freed memory. It is an error to free or dereference memory that has already been freed. Consider program graphs with labels of the forms $\mathbf{malloc}(\mathtt{p})$, $\mathbf{free}(\mathtt{p})$ and $\mathbf{deref}(\mathtt{p})$, for allocating, freeing, and dereferencing pointer p, respectively. To find these errors, we can use the existential query with pattern $\_^*\mathbf{free}(p)(\neg\mathbf{malloc}(p))^*(\mathbf{free}(p)|\mathbf{deref}(p))$. It returns all pairs $\langle v,\theta\rangle$ such that at program point v, pointer $\theta(p)$ has just been freed or dereferenced and, earlier along some path to v, it was freed without subsequently being assigned fresh memory.

Security. In UNIX, to ensure proper access control, a program executed with superuser privilege should close all open files before changing its effective user id to that of a non-superuser (by calling seteuid with a non-zero argument, represented as $\mathbf{seteuid}(\neg 0)$). To detect violations of this rule, we can use an existential query with pattern $\_^*\mathbf{open}(f)(\neg\mathbf{close}(f))^*\mathbf{seteuid}(\neg 0)$. This returns all pairs $\langle v,\theta\rangle$ such that at v, file $\theta(f)$ is still open on some path and seteuid has just been called with a non-zero argument.

Locking discipline. Many concurrent programs follow a synchronization discipline in which shared variables are protected by non-reentrant locks. A variable a is protected by a lock l if a is accessed (represented as $\mathbf{access}(\mathtt{a})$) only when l is held, i.e., a is not accessed outside a duration where l is acquired (represented as $\mathbf{acq}(\mathtt{l})$) and not subsequently released (represented as $\mathbf{rel}(\mathtt{l})$). We can use a universal query with pattern $((\neg\mathbf{access}(x))^*\mathbf{acq}(l)(\neg\mathbf{rel}(l))^*)^*$. It returns all pairs $\langle v,\theta\rangle$ such that variable $\theta(x)$ is protected by lock $\theta(l)$ on all paths to v. Each substitution associated with the program exit point gives a variable that is globally protected by a lock. Correct use of locks is also subject to sequencing constraints, for example, only a held lock can be released; these constraints can easily be expressed in our framework.

Deadlock avoidance. A classic strategy for deadlock avoidance is to introduce a partial order $\prec$ on locks, and acquire locks in that order, i.e., a process or thread that holds a lock $\mathtt{l}_1$ can only acquire a lock $\mathtt{l}_2$ if $\mathtt{l}_1 \prec \mathtt{l}_2$. Assuming locks are non-reentrant, to find pairs of locks such that one is held while the other is acquired on some path, we can use an existential query with pattern $\_^*\mathbf{acq}(l_1)(\neg\mathbf{rel}(l_1))^*\mathbf{acq}(l_2)\_^*$.

It returns all pairs $\langle v, \theta\rangle$ such that, on some path to v, $\theta(l_2)$ is acquired while $\theta(l_1)$ is held. Examination of all substitutions associated with the program exit point easily reveals whether locks are acquired consistent with some partial order $\prec$.

Some of these properties, such as file I/O and freed memory, are affected by equalities between program variables, such as aliasing of file pointers. Our current implementation takes into account equalities caused by passing parameters and return values. We plan to extend it to track additional equalities, as discussed in Section 5.

2.3 Model checking examples

Many model-checking problems can be expressed as parametric regular path queries. We consider a few representative model-checking problems for labeled transition systems (LTSs). An LTS is a finite graph, together with a distinguished starting vertex $v0$, in which each edge is labeled with **act**(a) for some action a, and each vertex corresponds to a unique state. We transform LTSs to make the vertices (i.e., states) explicit in the edge labels. The transformation to produce graphs suitable for existential queries augments each vertex v with an edge from v to v labeled **state**(v). The transformation to produce a graph suitable for universal queries replaces each vertex v with two vertices v_{in} and v_{out} connected by an edge with label **state**(v), and incoming and outgoing edges of v are re-directed to v_{in} and v_{out}, respectively. In the rest of this subsection, we consider the resulting graphs.

Deadlock. An LTS has a deadlock if it contains a vertex (i.e., a state) that is reachable from $v0$ and has no outgoing edges. To find vertices that have outgoing edges, we use an existential query with pattern $\_^*$**state**(s)**act**(_). It returns all pairs $\langle v, \theta\rangle$ such that state $\theta(s)$, i.e., vertex $\theta(s)$, is reachable from $v0$ and is followed by some action just before v. If every vertex appears in some substitution in the result, then every vertex has an outgoing edge, and the LTS contains no deadlock.

Livelock. An LTS has a livelock if it has a reachable cycle that contains only the invisible action i. To find paths that end with such cycles, we can use an existential query with pattern $\_^*$**state**(s)**act**(i)$^+$**state**(s). It returns all pairs $\langle v, \theta\rangle$, where v is also $\theta(s)$, such that vertex v is reachable from $v0$ and is on a cycle of invisible actions. The LTS contains a livelock iff the result of the query is non-empty.

2.4 Notation

We consider a graph to be a set G of labeled edges of the form $\langle v_1, el, v_2\rangle$, with source and target vertices v_1 and v_2 respectively and edge label el. A specific vertex $v0$ in G is given as the starting vertex. We will convert patterns to finite automata. An automaton is a set P of labeled transitions of the form $\langle s_1, tl, s_2\rangle$, with source and target states s_1 and s_2 respectively and transition label tl; a start state $s0$; and a set F of final states.

Given an edge label el and a transition label tl, let $match(tl, el)$, which takes a set of symbols as an implicit argument, be the set of minimal substitutions θ such that el matches tl under θ. The resulting set has at most one element when tl contains no negations but can be very large otherwise. For example, $match$(**use**(a),$\neg$**use**(x)) is the set of substitutions of the form $\{x \mapsto \mathtt{b}\}$, where b is any symbol other than a. Given a set S of substitutions, $merge(S)$ is (1) *undefined* if any two substitutions in S disagree on the mapping of any variable in the intersection of their domains and (2) the union of the substitutions in S otherwise. We sometimes write $merge(\{\theta_1, \theta_2\})$ as $merge(\theta_1, \theta_2)$.

Our complexity analysis uses the notation in Figure 2.

Name	Meaning
verts	number of vertices in G
states	number of states in P
symbs	number of symbols in G that parameters in P can be instantiated to
pars	number of parameters in P
substs	1 (for *undefined*) plus the number of substitutions that match some path from $v0$ in G with some path from $s0$ in P; this is $O(symbs^{pars})$
labelsize	maximum size of a single edge label in G or transition label in P
edgelabels	number of distinct edge labels in G
translabels	number of distinct transition labels in P
labelpars	maximum number of parameters in a transition label of P

Figure 2: Notation for complexity analysis.

3. SOLVING EXISTENTIAL QUERIES

For existential queries, we convert a parametric regular-expression pattern straightforwardly to an NFA (nondeterministic finite automata) of the same size. The existential query problem is: given G, $v0$, P, $s0$, and F, compute all pairs $\langle v, \theta\rangle$ such that there is some path from $v0$ to vertex v that matches some path from $s0$ to some state in F under substitution θ. Explicit computation of matching information for individual graph paths would be too expensive, so we instead compute matching information for reachable vertices.

Let $reach(G, P, v0, s0)$, called the *reach set*, be the set of triples $\langle v, s, \theta\rangle$ such that some path from $v0$ to v in G matches some path from $s0$ to s in P under substitution θ. Then the existential query is to compute

$$\{\langle v,\ \theta\rangle \mid \exists s \in F : \langle v, \theta, s\rangle \in reach(G, P, v0, s0)\} \qquad (1)$$

Basic algorithm. It is easy to see that the following rules hold for computing $reach(G, P, v0, s0)$:

(i) if $\langle v0, el, v\rangle \in G$ and $\langle s0, tl, s\rangle \in P$ and $\theta \in match(tl, el)$, then $\langle v, s, \theta\rangle \in reach(G, P, v0, s0)$.

(ii) if $\langle v, s, \theta\rangle \in reach(G, P, v0, s0)$ and $\langle v, el, v_1\rangle \in G$ and $\langle s, tl, s_1\rangle \in P$ and $\theta_1 \in match(tl, el)$ and $\theta_2 = merge(\theta, \theta_1) \neq undefined$, then $\langle v_1, s_1, \theta_2\rangle \in reach(G, P, v0, s0)$.

We can compute $reach(G, P, v0, s0)$ by repeatedly adding triples according to the two rules, and compute the query result according to (1). This leads to the following basic algorithm, where R is the set of triples already considered for the reach set, W is the worklist of triples yet to consider, and E is the result of the existential query:

```
$R := \{\}$;                                  //initialize reach set
$W := \{\}$;                                  //initialize worklist
for $\langle v0, el, v \rangle$ in $G$                    // based on rule (i)
    for $\langle s0, tl, s \rangle$ in $P$
        for $\theta$ in $match(tl, el)$
            $W := W \cup \{\langle v, s, \theta \rangle\}$;
$E := \{\}$;                                  //initialize query result
while exists $\langle v, s, \theta \rangle$ in $W$        //take from worklist
    $R := R \cup \{\langle v, s, \theta \rangle\}$;            //add to reach set
    $W := W - \{\langle v, s, \theta \rangle\}$;            //update worklist          (2)
    for $\langle v, el, v_1 \rangle$ in $G$                // based on rule (ii)
        for $\langle s, tl, s_1 \rangle$ in $P$
            for $\theta_1$ in $match(tl, el)$
                if $\theta_2 = merge(\theta, \theta_1) \neq undefined$
                    if $\langle v_1, s_1, \theta_2 \rangle \notin R$
                        $W := W \cup \{\langle v_1, s_1, \theta_2 \rangle\}$;
    if $s \in F$                              //update query result
        $E := E \cup \{\langle v, \theta \rangle\}$;
```

When transition labels contain no negations or wildcards, there is at most one element in the result of *match*, and *match* and *merge* can be computed easily by scanning the argument labels and substitutions, respectively; how to handle negations and wildcards are described below.

We use adjacency list representations for G and P. We can use nested arrays, hash tables, or combinations of them for R and W, as well as for E. We use natural numbers, called *keys*, to represent vertices, states, symbols, parameters, and substitutions. Substitutions are represented as arrays that map parameters to symbols. We maintain an array of substitutions, indexed by the key for substitutions. Substitutions are created dynamically, so we use the standard technique of doubling the size of the array when the array is full; this does not increase the worse-case asymptotic time and space complexities. When a substitution is constructed, we add it to this array if it is not already present; to efficiently check whether it is present, we also maintain a nested array structure representing all previously constructed substitutions.

This algorithm has worst-case running time $O(|G| \times |P| \times substs \times (labelsize + pars))$, since it considers each triple $\langle v, s, \theta \rangle$ in W and R, iterates over all outgoing edges of v and outgoing transitions of s, and computes a *match* and possibly a *merge* taking time $O(labelsize)$ and $O(pars)$, respectively, in each iteration. The factor *substs* instead of $symbs^{pars}$ is used because only substitutions that are the third component of a triple in W and R, i.e., that match some path from $v0$ in G with some path from $s0$ in P, are considered. The algorithm takes $O(|G|+|P|+verts \times states \times symbs^{pars} + pars \times symbs^{pars})$ space, when nested arrays are used for R and W, and $O(|G|+|P|+verts \times states \times substs + pars \times symbs^{pars})$ space if hashing is used for at least the last component of R and W; the four summands are for P, G, R and W, and *substs*, respectively.

Memoization and precomputation.

It is easy to see that for a pair of *el* and *tl* that come out of vertex v and state s, respectively, $match(tl, el)$ may be computed multiple times. We may save and reuse the results of *match* in an auxiliary map, M_s, called *substitution map*, that maps a pair $\langle el, tl \rangle$ to the resulting substitution. We initialize M_s to $\{\}$ immediately after R is initialized to $\{\}$, and replace each call to *match* with a lookup in M_s, returning the result if found and otherwise computing *match*, adding the result to M_s, and returning it. To support efficient operations on M_s, we map edge labels and transition labels, respectively, into natural-number keys. We represent the map as an array indexed by the keys for edge labels, where each array element is an array indexed by the keys for the transition labels and whose elements are keys of substitutions.

This memoization reduces the total cost of computing all *match*'s to $O(|G| \times |P| \times labelsize)$, and reduces the time complexity of the algorithm to $O(|G| \times |P| \times labelsize + |G| \times |P| \times substs \times pars)$. The space usage is increased by $O(edgelabels \times translabels)$ for the substitution map.

One may further notice that for each pair of vertex v and state s, the same *set* of substitutions that match outgoing edges of v and outgoing transitions of s is computed repeatedly for different substitutions θ such that $\langle v, s, \theta \rangle$ is taken from W. Indeed, we may precompute an auxiliary map, M_{ts}, called *target-and-substitution map*, that maps a pair $\langle v, s \rangle$ to the set of triples $\langle v_1, s_1, \theta_1 \rangle$ such that there is $\langle v, el, v_1 \rangle$ in G and $\langle s, tl, s_1 \rangle$ in P and $match(tl, el) = \theta_1$. This yields the following pseudo-code, in place of initialization of worklist in (2), that computes M_{ts} as well, where R_{ts} and W_{ts} are needed for computing M_{ts},

```
$W := \{\}$;                               //initialize $W$ as in
$R_{ts} := \{\}$;                             // (2), except that
$W_{ts} := \{\}$;                             // $R_{ts}$, $W_{ts}$, and $M_{ts}$
$M_{ts} := \{\}$;                             // are initialized
for $\langle v0, el, v \rangle$ in $G$                 // together with $W$
    for $\langle s0, tl, s \rangle$ in $P$
        for $\theta$ in $match(tl, el)$
            $W := W \cup \{\langle v, s, \theta \rangle\}$;
            $W_{ts} := W_{ts} \cup \{\langle v, s \rangle\}$;
            $M_{ts} := M_{ts} \cup \{\langle v0, s0, v, s, \theta \rangle\}$;                 (3)
while exists $\langle v, s \rangle$ in $W_{ts}$ //compute $M_{ts}$, with
    $R_{ts} := R_{ts} \cup \{\langle v, s \rangle\}$;     // help of $R_{ts}$ and $W_{ts}$,
    $W_{ts} := W_{ts} - \{\langle v, s \rangle\}$;     // as for computing $W$
    for $\langle v, el, v_1 \rangle$ in $G$            // but w/o 3rd compo
        for $\langle s, tl, s_1 \rangle$ in $P$        // -nent or call to $merge$
            for $\theta$ in $match(tl, el)$
                if $\langle v_1, s_1 \rangle \notin R_{ts}$
                    $W_{ts} := W_{ts} \cup \{\langle v_1, s_1 \rangle\}$;
                    $M_{ts} := M_{ts} \cup \{\langle v, s, v_1, s_1, \theta \rangle\}$;
```

and the following simplified update, in place of update of worklist in (2), that uses the precomputed map M_{ts}:

```
$W := W - \{\langle v, s, \theta \rangle\}$;                 //update $W$, as
for $\langle v, s, v_1, s_1, \theta_1 \rangle$ in $M_{ts}$            // in (2), except
    if $\theta_2 = merge(\theta, \theta_1) \neq undefined$// that only $M_{ts}$   (4)
        if $\langle v_1, s_1, \theta_2 \rangle \notin R$            // is needed
            $W := W \cup \{\langle v_1, s_1, \theta_2 \rangle\}$;
```

This precomputation avoids enumerating all outgoing edges of v and outgoing transitions of s repeatedly, but enumerates only target vertices and states led to by successful substitutions in M_{ts}. This may improve the running time over memoizing only results of *match*, though not asymptotically, to $O(|G| \times |P| \times labelsize + |M_{ts}| \times substs \times pars)$, where M_{ts} is $O(|G| \times |P|)$ but could be much smaller. It takes an additional asymptotic space of $O(|G| \times |P|)$.

Negations and wildcards.

Wildcards in transition labels are easy to handle: we only need to extend *match* to make any edge label match a wildcard under the empty substitution. Negations, if surrounding only parameters that are already bound earlier in the pattern and thus earlier on the paths, can also be handled easily in the main loop of the basic algorithm (2): we simply modify $match(tl, el)$ to check whether el matches tl under θ by definition, and to return $\{\{\}\}$ if so and $\{\}$ otherwise. This is actually a degenerate case of our general method for handling negations, described below.

One might think of treating negations as alternatives when converting a pattern to an automata, and then using the above algorithms; for example, replace $\neg$**def**(a) with an alternation containing all other edge labels in G. However, this method is typically very inefficient, and it is not applicable when there is a parameter to a negated constructor, as in $\neg$**def**(x).

Another inefficient approach is to naively compute $match$ and consider all resulting substitutions. This increases the time to compute $match$ from $O(labelsize)$ to $O(symbs^{labelpars} \times labelsize)$ and results in $O(symbs^{labelpars})$ substitutions. This significantly increases the overall complexity.

Our method is to modify $match$ together with $merge$ so that, overall, we consider only substitutions that actually match some path in G with some path in P, not enumerating other substitutions. Observe that, in the basic algorithm (2), the result of $match(tl, el)$ is used as an argument to $merge$. So instead of computing and considering all substitutions from the result of $match(tl, el)$ and then computing $merge$ for each, we consider only substitutions that are extensions of the other argument θ to $merge$ that cover the parameters in tl, denoted $extensions(\theta, tl)$, and we simply check whether el matches tl under each of the extensions. That is, we change the two lines in (2) consisting of the last **for**-clause and the first **if**-clause to

```
for $\theta_2$ in $extensions(\theta, tl)$
    if $match(\theta_2(tl), el) \neq \{\{\}\}$
```

In general, the worst-case time complexity is $O(|G| \times |P| \times \min(substs \times symbs^{labelpars}, symbs^{pars} \times (pars-1)!) \times labelsize)$, where $labelsize$ is for checking a match; in the left argument of min, $substs$ is the number of all θ's considered, and $symbs^{labelpars}$ bounds the number of extensions of a θ; in the right argument of min, $symbs^{pars}$ bounds the number of all substitutions of all parameters, and each substitution is considered at most $(pars - 1)!$ times. The factorial factor is because each substitution for n parameters may be considered 2^n times as an extension of a substitution for a subset of the n parameters, and $2^{pars} + (pars - 1)2^{pars-1} + (pars - 1)(pars - 2)2^{pars-2} + ... = O((pars - 1)!)$. The space complexity is the same as for the basic algorithm.

Two drawbacks of this algorithm are its higher complexity and that it prevents memoization and precomputation. We remedy this for the usual case where each transition label contains at most one negation: we proceed as before for the basic algorithm and for memoization and precomputation except for two changes. First, when matching an edge label with a transition label that contains negation, we record successful bindings for parameters outside and inside the negation separately, and call them $agree$ and $disagree$, respectively, if the bindings are consistent and where redundant bindings in $disagree$ are removed. For example, $match($**def**(x,$\neg c$), **def**(a,5)) returns a set containing a single element that is a pair of $\{x \mapsto \texttt{a}\}$ as $agree$ and $\{c \mapsto \texttt{5}\}$ as $disagree$. Second, we change the **if**-clause that contains $merge(\theta, \theta_1)$, where θ_1 is now a pair of $agree$ and $disagree$, to iterate over extensions of $merge(\theta, agree)$ that cover parameters in $disagree$ and check if the extension disagrees with some bindings in $disagree$. That is, we change the single line **if** $\theta_2 = merge(\theta, \theta_1) \neq undefined$ in (2) and (4) to

```
if $\theta_1 = merge(\theta, agree) \neq undefined$
    for $\theta_2$ in $extensions(\theta_1, disagree)$
        if $merge(\theta_2, disagree) = undefined$
```

This reduces the time complexity to the same as that for the basic algorithm, for memoization, and for precomputation, respectively, except with the factor $substs$ replaced by $substs \times 2^{labelpars}$. To see reasons for this factor, let sa and sd denote the sets of parameters in $agree$ and $disagree$, respectively. We have (1) the test $merge(\theta_2, disagree) = undefined$ succeeds for all θ_2's enumerated except the one that agrees with all in $disagree$, and thus all θ_2's considered except for one will become the third component of a triple put into W and R, which contributes to the factor $substs$, (2) fix a θ_2, there are $O(2^{|sd|})$ possible θ_1's for which θ_2 extends θ_1 and covers sd, because in θ_1 each parameter in sd has two choices—it maps to either nothing or what θ_2 maps to, (3) similarly, fix a θ_1, there are $O(2^{|sa|})$ possible θ's for which θ_1 extends θ and covers sa, and (4) $|sa| + |sd| = O(labelpars)$. Thus, compared to the algorithms for without negations, the overall complexity is increased by at most a factor of $O(2^{labelpars})$. Since $labelpars$ is a small constant in our applications, the asymptotic time complexity remains the same. The space complexity is the same as that for the basic algorithm, for memoization, and for precomputation, respectively.

4. SOLVING UNIVERSAL QUERIES

For universal queries, our basic algorithm requires a determinism condition, described below, so we first convert the parametric regular-expression pattern to a DFA (deterministic finite automata), also denoted by P, which can be exponentially larger in the worst case, but the size of the pattern is small in practice. However, an automaton that appears deterministic might become nondeterministic under some substitutions. For example, if a state has outgoing transitions labeled **use**(x) and **use**(y), then **use**(a) matches both labels under substitution $\{x \mapsto \texttt{a}, y \mapsto \texttt{a}\}$. Our algorithms handle this.

The universal query problem is: given G, $v0$, P, $s0$, and F, compute all pairs $\langle v, \theta \rangle$ such that every path from $v0$ to vertex v matches some path from $s0$ to some state in F under substitution θ. Again, for efficiency, we compute matching information for reachable vertices, not individual paths. To maintain only matches for reachable vertices v and be able to conclude about matches for all paths to v, we require that for each path from $v0$ to v in G, all paths in P that match it (under any substitutions) pass through the same set of states. We call this the *determinism condition*. Then we check that for all paths from $v0$ to v in G, the matching paths in P all end in states in F and that the substitutions for these matchings all agree with each other. Our basic algorithm conservatively checks the determinism condition while it proceeds; we describe separately below how to handle the case when the determinism check fails.

We extend the definition of $reach(G, P, v0, s0)$ to include triples $\langle v, undefined, undefined \rangle$ when there is a path from

$v0$ to v in G that does not match any path from $s0$ to any state in P under any substitution. Then the universal query is to compute all pairs $\langle v, \theta\rangle$ such that for all $\langle v, s, \theta_1\rangle$ in the extended $reach(G, P, v0, s0)$, s is in F, and θ is the successful merge of all θ_1's, i.e.,

$$\begin{aligned}\{\langle v,\theta\rangle \mid\ & (\forall\langle v,s,\theta_1\rangle \in reach(G,P,v0,s0) : s \in F)\ \wedge \\ & \theta = merge(\{\theta_1 : \langle v,s,\theta_1\rangle \in reach(G,P,v0,s0)\}) \\ & \neq undefined\}\end{aligned} \quad (5)$$

Basic algorithm. It is easy to see that rules (i) and (ii) and the following rule hold for computing the extended $reach(G, P, v0, s0)$:

(iii) if $\langle v, s, \theta\rangle \in reach(G, P, v0, s0)$ and $\langle v, el, v_1\rangle \in G$ and $\forall\langle s, tl, s_1\rangle \in P : \forall\theta_1 \in match(tl, el) : merge(\theta, \theta_1) = undefined$, then $\langle v_1, undefined, undefined\rangle \in reach(G, P, v0, s0)$.

We can compute the extended $reach(G, P, v0, s0)$ by repeatedly adding triples of vertex, matching state, and matching substitution according to all three rules. Then we do two things to compute the query result according to (5): maintain a bit for each vertex indicating whether all matching states belong to F, and merge all matching substitutions. We obtain the following basic algorithm, where R and W are as for existential queries, *matched* tracks whether some transition has matched the edge being processed, T maps each vertex to the bit indicating whether all matching states are final, and U is the result of the universal query, represented as a map from each vertex to the corresponding substitution or *undefined*:

```
$R := \{\}$;                                   //initialize reach set
$W := \{\}$;                                   //initialize worklist
for $\langle v0, el, v\rangle$ in $G$                         // based on rule (i),
    for $\langle s0, tl, s\rangle$ in $P$                     // as in (2)
        for $\theta$ in $match(tl, el)$
            $W := W \cup \{\langle v, s, \theta\rangle\}$;
$T := \{\}$;                                   //init. test for match
$U := \{\}$;                                   //initialize query result
while exists $\langle v, s, \theta\rangle$ in $W$              //take from worklist
    $R := R \cup \{\langle v, s, \theta\rangle\}$;              //add to reach set
    $W := W - \{\langle v, s, \theta\rangle\}$;                //update worklist
    for $\langle v, el, v_1\rangle$ in $G$                    // based on rules (ii)
        $matched := false$;                    // and (iii), and check
        for $\langle s, tl, s_1\rangle$ in $P$                // determinism
            for $\theta_1$ in $match(tl, el)$                                    (6)
                if $\theta_2 = merge(\theta, \theta_1) \neq undefined$
                    if $matched$           //check determinism
                        exit("determinism check failed");
                    $matched := true$;
                    if $\langle v_1, s_1, \theta_2\rangle \notin R$
                        $W := W \cup \{\langle v_1, s_1, \theta_2\rangle\}$;
            if $\neg matched$                  //use rule (iii)
                if $\langle v_1, undefined, undefined\rangle \notin R$
                    $W := W \cup \{\langle v_1, undefined, undefined\rangle\}$;
    if ($T(v)$ is not defined) $\vee$ $T(v)$ //update test for match
        $T(v) := (s \in F)$;
    if $T(v)$                                  //update query result
        $U(v) := merge(U(v), \theta)$;         //merge substitutions
    else
        $U(v) := undefined$;
```

Negations and wildcards can be handled in the same way as for existential queries.

We use the same data structures as for existential queries; the only difference is that T and U can each be represented as an additional field in the structure for each vertex.

The time and space complexities are the same as for the basic algorithm for existential queries.

Memoization and precomputation. We can memoize the result of *match* as for existential queries and obtain the same time and space complexities as for the algorithm with memoization for existential queries.

There are also sets of repeated computations of *match* similar as for existential queries, except that matching information for each outgoing edge $\langle v, el, v_1\rangle$ of v and each state s, rather than just each vertex v and state s, is needed, by the **if**-statement that tests *matched*, thus the target-and-substitution map is not sufficient. We may precompute an auxiliary map, M_{ds}, called *determinism-and-substitution map*, that maps an edge $\langle v, el, v_1, s\rangle$ and state s to the set of pairs $\langle s_1, \theta_1\rangle$ such that there is $\langle s, tl, s_1\rangle$ in P and $match(tl, el) = \theta_1$. This yields pseudo-code similar to (3), except with operations on M_{ts} replaced by operations on M_{ds}, in place of initialization of worklist in (6); and simplified update code, similar to the simplification from (2) to (4) except with the additional tests and uses of *matched* and with the use of M_{ts} replaced by a use of M_{ds}, in place of update of worklist in (6)

The time complexity is the same as for existential queries, except M_{ds} is used in place of M_{ts}. The space complexity is the same as for existential queries.

Nondeterminism. When the determinism condition does not hold, the algorithms above do not apply. We describe two solutions: an algorithm that enumerates all substitutions, and a hybrid algorithm that first does an existential query to reduce the number of substitutions considered.

The enumeration algorithm considers all substitutions from some or all parameters in the pattern to symbols in the graph, and does a universal query without parameters for each substitution. That is, for each substitution, instantiate the pattern, convert the instantiated pattern to a DFA, and use the algorithms described above. The total time complexity is $O(|G| \times maxTrans \times symbs^{pars})$, where $maxTrans$ is the maximum number of transitions in the DFA's for the instantiated patterns, and $symbs^{pars}$ is the number of substitutions. This is asymptotically better than the algorithm above, but enumerating all substitutions is expensive in practice. A clear advantage of the enumeration algorithm is that the space complexity is as small as $O(|G| + maxTrans + verts \times maxStates + verts \times substs)$, where $maxStates$ is the maximum number of states in the DFA's for the instantiated patterns, and the last summand is for the output size. The same is true if we use an enumeration algorithm for existential queries.

The hybrid algorithm refines the enumeration algorithm by first doing an existential query with the same pattern and obtaining a set of substitutions from the result; these substitution are those that are involved in matching on some paths. Then we enumerate all substitutions that are extensions to the substitutions obtained above, and do universal queries without parameters for each of them. This idea is also used in [6]. The asymptotic time complexity is the sum of those for the two steps, and the asymptotic space complexity is the same as for existential queries in the first step.

5. DISCUSSION

We discuss several tradeoffs and extensions for the parametric regular path query framework. We also discuss improvements in the complexity analysis and performance and in the query language and usage.

5.1 Tradeoffs and summary

As in other analysis frameworks, there may be more than one way of formulating an analysis problem using parametric regular path queries. There are many tradeoffs involving efficiency, readability, amount of information in the result, etc.

Several distinct properties of this work allow users to easily test and compare the alternatives: queries can be written declaratively and succinctly; complete algorithms and data structures have been designed and implemented; precise running time and space usage have been analyzed, both in terms of asymptotic complexity and useful parameters in practice; and ways to automatically convert queries to different forms are being investigated.

Consider the uninitialized variables examples in Section 2.2, where forward existential queries $(\neg\mathbf{def}(x))^*\mathbf{use}(x)$ and $(\neg(\mathbf{def}(x)|\mathbf{use}(x)))^*\mathbf{use}(x)$ find all uses and first uses, respectively, of uninitialized variables. These analyses can also be expressed as backward queries. To capture the uses, an additional parameter is used; specifically, $\mathbf{use}(x,l)$ denotes a use of variable x at, say, location l in the program. We add an edge from $v0$ to $v0$ labeled $\mathbf{entry}()$. Then the two backward queries are $\mathbf{use}(x,l)(\neg\mathbf{def}(x))^*\mathbf{entry}()$ and $\mathbf{use}(x,l)(\neg(\mathbf{def}(x)|\mathbf{use}(x,\_)))^*\mathbf{entry}()$, respectively. To find only names of uninitialized variables, we can use a backward query $\_^*\mathbf{use}(x)(\neg\mathbf{def}(x))^*\mathbf{entry}()$. Compared with forward queries, the first two backward queries have a factor $O(symbs^2)$, rather than $O(symbs)$, in asymptotic time because of the extra parameter, but they run much faster because x is bound by $\mathbf{use}(x,l)$ before the negation $\neg\mathbf{def}(x)$, and a single l is paired with a use of x in $\mathbf{use}(x,l)$. The forward queries start with negation $\neg\mathbf{def}(x)$, and x ends up being enumerated for all possible instantiations.

We summarize several of the most important points based on our experience; they hold for all examples we have encountered:

- queries can be written clearly and succinctly; patterns always contain wildcards or negations; individual edge labels contain at most one negation;
- patterns contain a small number of independent parameters; additional parameters can be used easily to capture additional information;
- existential queries are used much more often than universal queries; when both kinds can be used, the existential query is easier to write and understand than the universal query;
- forward and backward queries can be converted to each other by introducing additional parameters, but reversing the order of binding of parameters may significantly affect performance;
- existential queries are much easier to process and more efficient than universal queries; queries that bind parameters positively before negations are much faster than queries that don't;
- memoization and precomputation reduce asymptotic time complexity, but can sometimes be slower in practice, and use much additional space.

5.2 Extensions

To further enhance the expressive power of parametric regular path queries, the framework can be extended to take into account more general relationships, as discussed below.

Tracking equality and points-to information. Simple equality relationships are already captured through multiple occurrences of a parameter, but additional equality relationships can be important. For example, one could (although it is unlikely) open a file and assign it to f, subsequently assign f to g, and then close the file through g. If the analysis does not consider the equality between f and g, it may report false alarms—for example, that f is not closed after it was opened—or miss errors.

To solve such equality problems, we may use the same query but employ a separate module to track equalities.

For example, Field et al. [8] study how to check program properties specified using regular expressions with one-level pointers. We can extend parametric regular path queries to express and check such properties as well. An open problem is precise time and space complexity analysis for the extended frameworks.

Interprocedural analysis. For interprocedural program analysis, equalities relating arguments in calls and parameters in procedure definitions, and relating return values in procedure definition and return values of calls, need to be captured. We have extended the framework of parametric regular path queries for interprocedural analysis by explicitly tracking these two kinds of equalities. An advantage of this approach is that the query pattern can remain unchanged, and the equalities are tracked implicitly. We could make this interprocedural analysis more precise by using context free grammars to restrict the algorithm to consider only valid interprocedural paths [14].

5.3 Complexity analysis and performance

For a given user query, our complexity analysis result corresponds to a formula that gives the worst-case asymptotic running time and space usage for evaluating the query. However, the value of $substs$ in the formula depends on the input and is bounded only by $O(symbs^{pars})$. A natural refinement is to use the sizes of the domains of parameters, such as the number of program variables for parameter x in $\mathbf{def}(x)$, in place of $symbs$, which counts all symbols in the program graph. Also, correlations between the parameters, such as that there is a single program location l associated with a use of x in $\mathbf{use}(x,l)$, can be used to further refine the estimated complexity. Future work may provide even finer complexity characterizations, especially for the effect of negations, by exploiting additional domain knowledge about the input graph and the query.

Considering that our algorithms are generic and do not perform optimizations that exploit domain knowledge (although such knowledge can be built into the input graph and the query), the performance is quite good. We can further improve the performance, especially the memory usage, by exploiting the connectivity of graphs. In particular, graph vertices can be visited in a topological order of the strongly connected components (SCCs), and after a SCC is finished, data structures containing information about that SCC can be de-allocated. Also, one can exploit sparsity of relevant edge labels. In particular, since typically relatively few different kinds of edge labels in the graph are relevant to a query, the graph can be compacted by eliminating and collapsing certain edges and vertices before solving the query. Detailed study is needed for these, especially for taking domain knowledge into account.

5.4 Query language and usage

Parametric regular path queries studied thus far can use only single labels on edges, which is why, for model checking problems, we need to transform labeled transition systems to capture the vertices (states) through edge labels. Also, queries do not exploit the values of parameters besides checking equality. Finally, parametric regular path queries can not express analyses that involve non-regular language based patterns, such as analysis of matching acquire and release operations for reentrant locks, which is equivalent to matching parentheses and therefore needs context-free language based patterns.

We are currently extending the framework so queries can use also vertices and vertex labels directly, including multiple labels for each vertex and edge, and can do computations involving the values of parameters. This will lead to a very powerful query language that generalizes XPath [19] by supporting unbounded repeating patterns on paths (via the general Kleene star), not just unbounded skipping of path segments, and by allowing querying of graphs, not just trees. Extending the framework to use context-free language based patterns is a problem open for study.

To facilitate usage of the framework, we are also studying methods that support easier construction of efficient queries. In particular, for certain kinds of problems, we can let the user specify a universal property and automatically generate existential queries for checking different kinds of violations, as well as a merged automata for checking all kinds of violations at the same time. For example, instead of having to write several separate queries for file operations, the user may simply specify that on every path from the entry point to exit point, operations for a file f must follow the pattern $(\mathbf{open}(f)(\mathbf{access}(f))^*\mathbf{close}(f))^*$, allowing other operations to occur anywhere in the middle.

6. IMPLEMENTATION AND EXPERIMENTS

Our tool is divided into front-ends, which take an input and convert it into an edge-labeled graph, and back-ends, which take that graph and an automaton, apply the parametric regular path query, and output the result.

We have multiple front-ends. One reads C programs and creates intraprocedural control flow graphs labeled with **def** and **use**, as in Section 2. The generated control flow graph may optionally include a sequence number with each **use**, allowing the generated graph to be used for forward and backward uninitialized use analysis. This front-end was implemented using CodeSurfer and consists of 682 lines of Scheme.

Another front-end reads transition systems taken from the Very Large Transition Systems web site (`http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`). These transition systems were produced automatically from formal descriptions of real-life concurrent systems, and are suitable for use as input to model checking algorithms. By exploiting existing tools, we were able to implement this front-end with less than a hundred lines of new code.

While the data discussed in this section is generated by the two front-ends described above, we also have others, which support interprocedural file operation analysis for C and def-use analysis for Python. We have been able to use the same automaton to perform uninitialized use analysis for C and Python.

We also have multiple back-ends corresponding to different variants of the algorithms given above. All back-ends are written in a high-level language similar to the pseudo code given in Sections 3 and 4, which is translated to C++ by a Python program. The translator takes care of implementation details such as data structure selection [15] and multiset discrimination [3]. One of the benefits of using a translator to generate the C++ code is that we can easily experiment with changes in precomputation and data structure representation. The high-level source files for the variants range from 50 to 118 lines, and each is translated into over a thousand lines of C++.

To illustrate the performance of the algorithms, we first present running times for the algorithms as applied to program analysis. Table 1 gives performance information for detecting all uses of uninitialized variables in C programs. All timings are recorded on a Pentium 4 running at 2.0 GHz. The enumeration variant was used for the forward query given in Section 2.2, while the basic and precomputation variants were used for the backward query given in Section 5.1. The size of the result set is the number of possibly uninitialized uses found in each program.

Table 2 summarizes the performance for deadlock detection on a transition graph. The query is given in Section 2.3. It is a forward query that contains no negation of unbound variables, and all three variants were used for it.

The first finding supported by these experiments is that enumeration is slower than the other two variants of the algorithms. The early introduction of substitutions forces the total amount of work to scale with both the size of the graph and the number of substitutions. This can lead to quadratic behavior where other variants are linear.

A second finding is that the performance increase provided by precomputation is highly dependent on the precise construction of the graph and automaton. In the uninitialized uses analysis, precomputation roughly halves the running time of the program, while in deadlock analysis, precomputation does not provide a consistent benefit. In the deadlock analysis, the main worklist contains only a small number of triples per vertex, and the graph and the automaton have small out-degree. These factors conspire to keep the improvement offered by precomputation small, or in some cases, negative. The larger number of worklist triples encountered in the uninitialized uses analysis allows the cost of precomputation to be amortized, leading to a significant speedup.

The core of our algorithms involves operations on sets and maps, so the design of these data structures strongly affects the memory usage and running time. We tried several variants before settling on the two best. Both are derived from the based representations in [15], but differ in how tuples are represented in sets. When a set is queried for elements matching one or more keys, both representations use the first key to find a unique base containing information about set membership. This base may need to be indexed

		graph	result	basic		precomputation		enumeration		
input	LOC	edges	size	worklist	time	worklist	time	worklist	time	substs
cksum	236	521	20	3,749	0.01s	3,749	0.00s	10,573	0.04s	40
sum	198	714	23	3,620	0.01s	3,620	0.01s	22,551	0.07s	57
expand	317	971	46	7,650	0.03s	7,650	0.02s	40,746	0.14s	75
uniq	406	1,696	147	33,657	0.13s	33,657	0.07s	127,393	0.47s	134
cut	603	2,124	92	37,878	0.15s	37,878	0.07s	181,252	0.67s	146
C-parser	1,847	4,260	97	60,621	0.24s	60,621	0.13s	447,995	1.72s	207
iburg	649	5,672	93	87,606	0.36s	87,606	0.19s	1,206,845	4.57s	377
struct	1,699	6,022	128	148,609	0.61s	148,609	0.31s	1,101,780	4.34s	333
ratfor	1,261	7,617	369	191,809	0.82s	191,809	0.42s	1,405,965	5.48s	361

Table 1: Algorithm performance for the detection of uninitialized variable uses.

		graph	result	basic		precomputation		enumeration		
input	states	edges	size	worklist	time	worklist	time	worklist	time	substs
vasy-0-1	289	1,513	1,224	1,802	0.01s	1,802	0.01s	85,034	0.87s	289
cwi-1-2	1,952	4,339	2,387	6,291	0.04s	6,291	0.05s	3,814,643	26.83s	1,952
vasy-1-4	1,183	5,647	4,464	6,830	0.05s	6,830	0.05s	1,405,136	15.72s	1,183
vasy-5-9	5,486	14,878	9,392	20,364	0.13s	20,364	0.16s	n/d	n/d	5,486
cwi-3-14	3,996	18,548	14,552	22,544	0.15s	22,544	0.18s	n/d	n/d	3,996
vasy-8-24	8,879	33,290	24,411	42,169	0.32s	42,169	0.35s	n/d	n/d	8,879
vasy-8-38	8,921	47,345	38,424	56,266	0.40s	56,266	0.43s	n/d	n/d	8,921
vasy-10-56	10,849	67,005	56,156	77,854	0.60s	77,854	0.60s	n/d	n/d	10,849

Table 2: Algorithm performance for the detection of transition system deadlocks. n/d means that the analysis was not completed in a 180 second time limit.

by a second or later key to determine set membership. This indexing can be performed by using a level of nested arrays for each additional key, or lookup in a hash table of all the additional keys. A variant using hashing without basing was also tried, but was rejected as inferior in both space and time usage.

Table 3 shows how these representations provide different time and space tradeoffs. Using nested arrays is fastest only when enumeration is used, and it uses a large amount of space to represent sparse sets, such as the target-and-substitution map. If a set is very sparse, the cost to initialize the memory may outweigh the savings from eliminating hashing. The space usage of hashing is less than that of nested arrays, and the running time is similar, generally making a based hash representation (the second option above) the best, when it fits into memory. Table 3 also shows that an advantage of using enumeration is that the memory usage is much smaller than the other two variants, at the cost of much larger running time.

The running time of the algorithms is affected by the size and composition of the graph and automaton. Figure 3 plots the size of the worklist and the running time of the basic algorithm against the size of the program. Both plots look identical, providing another strong indication that, on practical graphs, running time is strongly tied to worklist size. In practice, the size of this worklist appears to grow at a rate slightly greater than linear, with the precise amount of work done highly dependent on the actual input data.

7. RELATED WORK AND CONCLUSION

Existential regular path queries without parameters have been studied and used for querying databases and semi-structured data [20, 1], but parameters are essential for expressing correlations of information in different parts of the data. For example, parameters are needed in querying system logs for intrusion detection [16], and in many other applications that use regular expression packages for matching strings, even though not graphs. Many interesting applications of regular path queries in program analysis and model checking require parameters, as shown by the examples in this paper.

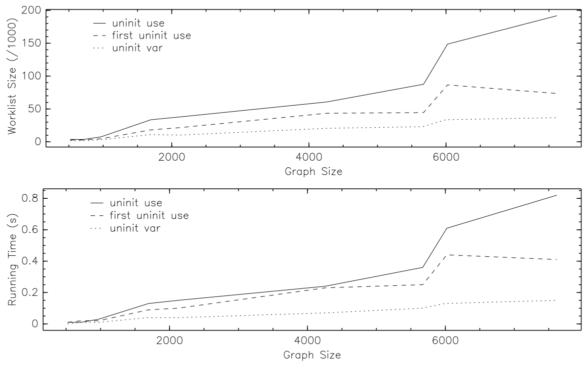

Figure 3: Worklist size and running time, respectively, versus graph size, for uninitialized variables and uses.

Engler et al. demonstrated that program analyses expressed as state machines can be very effective at finding defects in software [10]. Global state machines and variable-specific state machines in their framework roughly correspond to existential regular path queries with zero parameters and one parameter (which gets bound to the associated variable), respectively; this is illustrated by the freed memory example in Section 2. ESP [5] and CQUAL [9] deal with a similar class of properties. These projects focus on defect detection and verification for C programs. In contrast to these and

	basic				precomputation				enumeration			
	hashing		nested		hashing		nested		hashing		nested	
input	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time
cksum	149k	0.01s	324k	0.02s	181k	0.00s	390k	0.01s	23k	0.04s	22k	0.03s
sum	146k	0.01s	461k	0.02s	197k	0.01s	582k	0.01s	32k	0.07s	31k	0.07s
expand	315k	0.03s	1,118k	0.03s	359k	0.02s	1,172k	0.02s	43k	0.14s	44k	0.12s
uniq	1,294k	0.13s	4,292k	0.13s	1,370k	0.07s	4,303k	0.07s	75k	0.47s	91k	0.43s
cut	1,429k	0.15s	6,135k	0.16s	1,531k	0.07s	5,639k	0.09s	90k	0.67s	83k	0.63s
C-parser	2,300k	0.24s	16,521k	0.29s	2,486k	0.13s	15,730k	0.14s	163k	1.72s	162k	1.57s
struct	5,540k	0.61s	36,921k	0.66s	5,989k	0.31s	38,191k	0.35s	233k	4.34s	247k	4.05s
ratfor	7,201k	0.82s	70,949k	0.94s	7,831k	0.42s	65,196k	0.48s	215k	5.48s	272k	5.31s

Table 3: Time and memory usage for hashing and nested array implementations of uninitialized uses.

other dataflow frameworks and type systems for analyzing program properties and ensuring program safety, our focus is on a general graph analysis framework that can easily be applied to C programs, Python programs, XML data, labeled transition systems, etc., and that supports a precisely defined and more expressive query language with negation, multiple parameters, and universal queries.

de Moor et al. showed that universal parametric regular path queries are useful for compiler analysis and optimizations [6]. They give a high-level algorithm for solving universal queries without parameters and describe an implementation of it as a logic program. Liu and Yu designed a complete algorithm with detailed data structures for solving universal queries without parameters [12]. That algorithm requires the finite automaton to be complete, which usually means adding explicit transitions to a trap state; this can significantly increase actual space usage of the analysis. The algorithm in this paper handles incomplete automata directly, saving space. Drape et al. describe how to program universal parametric queries as logic programs [7] but do not give a direct algorithm for solving such queries.

A major contribution of this paper is the first complete algorithm and data structures for directly and efficiently solving universal parametric regular path queries. Other major contributions of this paper compared to all of the work cited above are very detailed complexity analysis of algorithms for solving existential and universal queries, detailed performance comparison of variations of the algorithms and data structures, and investigation of efficiency tradeoffs between different formulations of a query. We have derived the basic algorithms using a methodology based on formal specification and transformation, which helps ensure correctness and supports precise complexity analysis. Due to space limitations, details of the derivations are not given in this paper.

Reps et al. showed how to perform precise interprocedural dataflow analysis by using graph reachability to eliminate infeasible interprocedural paths [14]. This technique can be incorporated into many program analysis frameworks, including ours. While their work focuses on identifying feasible interprocedural paths, our work focuses on a simple and powerful language for expressing analyses and on the detailed design and complexity analysis of the analysis algorithms.

Parametric regular path queries are not as powerful as some other analysis frameworks, such as set constraints [11, 2] and temporal logic [17], but they are more perspicuous and convenient, and they are sufficiently powerful to express many interesting properties in many application domains, and they can easily be used within more powerful frameworks when necessary. Parameters support correlation of information along paths. This is also the idea underlying trace-based program analysis [4], which uses powerful but heavy-weight symbolic execution techniques.

8. REFERENCES

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, 1997.

[2] A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, 1999.

[3] J. Cai and R. Paige. "look ma, no hashing, and no arrays either". In *Conference Record of the 18th Annual ACM Symposium on Principles of Prog. Lang.*, pages 143–154, 1991.

[4] C. Colby and P. Lee. Trace-based program analysis. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Prog. Lang.*, pages 195–207, 1996.

[5] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In PLDI 2002 [13], pages 57–68.

[6] O. de Moor, D. Lacey, and E. V. Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1-2), 2003.

[7] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .net intermediate language using path logic programming. In *Proceedings of the 4th International Conference on Principles and Practice of Declarative Programming*, Oct. 2002.

[8] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Proceedings of the 10th International Static Analysis Symposium*, volume 2694 of *LNCS*, pages 439–462. Springer-Verlag, Berlin, 2003.

[9] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In PLDI 2002 [13].

[10] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In PLDI 2002 [13].

[11] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, pages 281–298.

Springer-Verlag, Berlin, 1994.

[12] Y. A. Liu and F. Yu. Solving regular path queries. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, volume 2386 of *LNCS*, pages 195–208. Springer-Verlag, Berlin, 2002.

[13] *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, 2002.

[14] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Prog. Lang.*, 1995.

[15] E. Schonberg, J. Schwartz, and M. Sharir. An automatic technique for the selection of data representations in SETL programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, Apr. 1981.

[16] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, pages 63–78, 1999.

[17] B. Steffen. Data flow analysis as model checking. In A. M. T. Ito, editor, *Theoretical Aspects of Computer Science (TACS'91),*, volume 526 of *LNCS*, pages 346–364. Springer-Verlag, September 1991.

[18] The World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/.

[19] The World Wide Web Consortium. XML Path Language (XPath). http://www.w3.org/TR/xpath.

[20] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the ACM Symposim on Principles of Database Systtems*, pages 230–242, 1990.