Query-Based Debugging of Object-Oriented Programs

Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh
Department of Computer Science
University of California
Santa Barbara, CA 93106
{raimisl,urs,ambuj}@cs.ucsb.edu
http://www.cs.ucsb.edu/{~raimisl,~urs,~ambuj}

Abstract. Object relationships in modern software systems are becoming increasingly numerous and complex. Programmers who try to find violations of such relationships need new tools that allow them to explore objects in a large system more efficiently. Many existing debuggers present only a low-level, one-object-at-a-time view of objects and their relationships. We propose a new solution to overcome these problems: query-based debugging. The implementation of the query-based debugger described here offers programmers an effective query tool that allows efficient searching of large object spaces and quick verification of complex relationships. Even for programs that have large numbers of objects, the debugger achieves interactive response times for common queries by using a combination of fast searching primitives, query optimization, and incremental result delivery.

1. Introduction

As the complexity of object-oriented systems increases, and as the number of objects in a program increases, debugging becomes more and more difficult. Similarly, object relationships become more numerous and complex. Consequently, a graph of objects and their relationships in a large program resembles a giant tangled web. When debugging a program, the programmer must find the few objects that violate their invariants in this giant web, a process that can resemble a search for the proverbial needle in the haystack. For example, assume that a graphical widget references its parent window, and that this parent window must in turn reference the enclosed widget. How can a programmer detect violations of this relationship?

Current debuggers offer little assistance to the programmer when it comes to solving such problems. While some debuggers (e.g., the debuggers in Smalltalk [Gol84, GR91] or ET++ [GWM89, WG94]) provide at least some assistance by finding all instances of a particular class on demand, these debuggers cannot determine the correctness of object relationships. Many graphical debuggers allow an interactive exploration of individual data structures; for example, clicking on a reference displays the referenced object. However, such manual object-by-object exploration does not significantly improve the programmer's productivity in situations where hundreds or even thousands of objects must be tediously inspected.

An alternative way to test object relationships involves using special testing code written for each application. For example, an application could keep track of all created widgets and include methods that iterate over the list of widgets and verify their correctness. Although this approach is more effective for large data structures than is the manual "display-and-click" approach, writing test code may still consume a considerable amount of time. For example, the Self virtual machine [HU96] contains over 10,000 lines of testing-related C++ code. Such testing code can be used only to answer very particular questions while tracking a bug, consequently the effort spent writing this code is considerable compared to the number of times it can be used. In addition, testing code might be inefficient, especially if the constraints are complex. For example, a relatively straightforward assertion, such as "no widget is contained in more than one window," may require the creation of reference counts to be verified efficiently. With large programs containing thousands of objects, naive testing code might take minutes to execute.

We propose a new solution to overcome these problems: query-based debugging. The approach that we describe here offers programmers an effective query tool that allows complex relationships to be formulated easily and evaluated efficiently. Such queries can be formulated during debugging, or stored as a library of predefined debugging queries together with the program's normal code. In our system, verifying that all widgets are referenced back by their containing windows is as simple as entering the query

```
widget wid; window win.
(wid window = win) &&
(win widget_collection includes: wid) not
```


OOPSLA 97- 10/97 Atlanta, Georgia, USA

(The current implementation of the system is based on Self, so query expressions use Self syntax.) This query identifies all objects that violate the containment constraint. Not only does this query save programming time, the query evaluator also applies sophisticated optimization algorithms to speed the execution of the query and to deliver the results incrementally. Typical queries execute in seconds, even for programs involving thousands of objects.

The rest of this paper is structured as follows. Section 2 introduces the query model and demonstrates its usefulness by giving some examples of queries that we used while developing and testing our system. Section 3 discusses the implementation techniques we used to obtain efficient query execution. Section 4 then provides experimental results verifying that the query-based debugger is efficient enough to be a useful tool during program debugging. Finally, we discuss related work and possible future extensions of the system.

2. Query Model and Examples

Our basic premise is that programmers need to verify relationships among objects during debugging. However, any tool for verifying relationships must be both expressive and simple to be widely applicable and to save debugging time. In our system, query expressions look just like expressions in the underlying programming language, Self [US87], so that programmers can use queries without learning a new syntax or language. (Accordingly, we would expect query-based debuggers for different languages to choose a syntax close to that language.) Queries consist of two parts, the *search domain* and the *constraint*. Let us revisit the widget and window query:

```
widget wid; window win.
(wid window = win) &&
(win widget_collection includes: wid) not
```

The first part of the query defines the *search domain* of the query using universal quantification. The domain part of the above example should be read as "for all widgets wid and all windows win...". With Self being a prototype-based language, widget is the name of the widget prototype, and its domain is all objects that contain the same fields as this prototype[†]; in a class-based language, widget would be a class name and its domain would be all instances of the class.

The second part of the query specifies the constraint expression to be evaluated for each tuple of the search domain. Constraints are arbitrary Self expressions that evaluate to a boolean result. In particular, they can contain message sends. Semantically, the expression will be evaluated for each tuple in the Cartesian product of the query's individual domains, and the query result will include all tuples for which the expression evaluates to true (similarly to an SQL select query).

[†] See section 3 for a more precise definition.

The general form of a query is

$X_1\ x_{11} \ldots x_{1n}; \ldots ; X_m\ x_{m1} \ldots x_{mn}.$
$\text{Constraint}_1\ \&\&\ \ldots\ \&\&\ \text{Constraint}_k$

where x_{ij} is a domain variable whose domain is X_i. The definition of domain variables can have single-type domains or domains including subtypes. For example, the widget single-type domain contains widgets but does not contain colorWidgets. In other words, a domain does not include objects of any subclass or subtype (or, in classless Self, subobject). However, using a different syntax, the user can also specify domains that include instances of subtypes, subclasses, or subobjects. In this case the widget domain would contain both widgets and colorWidgets.

We express the constraint part as a conjunction of a number of individual constraints only because this conjunctive form allows a number of optimizations to the query evaluation order as described in section 3. Constraints not in conjunctive form are perfectly valid and can be specified, but our current system performs no optimizations on them. Conjunctive queries occur frequently and are natural to use, so this restriction has not yet proved to be unreasonably limiting.

2.1 Assumptions

Our query model is based on several assumptions. First, it is the programmer's responsibility to ensure that queries are side-effect free—just as expressions in C/C++ assertions must be side-effect free. Second, the programmer must ensure that the queried objects are in a consistent state when the query is evaluated (i.e., that expression evaluation will succeed). Finally, to use efficient hash joins (see section 3.5), we assume that the "=" method in queries has equality semantics; since all methods can be redefined in Self, this is not guaranteed. However, the Self programming guidelines strongly suggest such semantics, and the current Self system contains no method that violates it.[‡]

2.2 Discussion

We chose this particular query model primarily because of its simplicity: a query is a boolean expression prefixed by a search domain specification. Originally we anticipated having to extend the model once its shortcomings would become clear during experimentation, but so far we have not encountered such situations.

An alternative way of specifying queries would use iterators such as Smalltalk's allInstancesDo: method; by predefining more of these iterators, one could potentially avoid

[‡] For environments where this assumption is not valid, section 4.5 shows the effect of not using hash joins.

introducing any query language at all and just use the base language. We rejected this approach for two main reasons. First, it is language-dependent—languages without closures cannot express such iterators succinctly. Second, we felt that declarative domain specification was simpler and more flexible, putting the burden of choosing an efficient iteration and evaluation order on the query optimizer instead of the programmer.

Object-oriented databases typically contain more sophisticated query models [AK89, BKK88, NOT96, SZ90]. Our solution was specifically designed for query-based debugging, and we deliberately traded off some completeness and expressive power in exchange for simplicity and ease of use because we deemed these attributes to be of primary importance.

2.3 Examples

We now discuss several examples of queries that we used to understand large programs written by others.

2.3.1 The Self Graphical User Interface

We implemented our debugger prototype using the Self user interface which is based on morphs [SMU95]. Morphs are user interface objects (usually having a visual representation) that can perform specified actions and can be moved on the desktop. Morphs can contain other morphs and hierarchically build complex interface objects. Some morphs are primitive. For example, row and column morphs arrange objects horizontally or vertically, frame morphs surround other morphs, and button morphs trigger actions. More complicated morphs such as object outliners are composed of a number of simpler morphs.

During the development of the debugger we encountered numerous questions about the accepted use of morphs. Using our debugger we could easily answer many of these questions. For example, initially we wondered whether one morph could be a part of more than one composite morph. Intuitively, we felt that such a structure would be incorrect, so we asked the debugger to "find all morphs directly contained in at least two morphs":

```
morph a b c.
(a morphs includes: b) && (c morphs includes: b) && (a != c)
```

An empty answer set showed no such morphs in the entire Self system.

Another interesting question arose when we tried to construct "table" morphs. Are row morphs usually embedded into column morphs or vice versa? We looked at the object outliner class that already implements a similar structure, and asked the following two queries:

```
objectOutliner a; rowMorph b; columnMorph c.
(a morphs includes: b) && (b morphs includes: c)
```

```
objectOutliner a; columnMorph b; rowMorph c
(a morphs includes: b) && (b morphs includes: c)
```

The output of the first query was empty, whereas the output of the second query had 23 results, leading us to conclude that object outliners contain column morphs that in turn contain row morphs. However, the query output contained only 23 of the 132 object outliners present in the system at that point; maybe some of them did not contain column morphs or row morphs at all. The query

```
objectOutliner a; columnMorph b.
(a morphs includes: b)
```

returned 130 results. That is, all but two object outliners contained column morphs, but apparently most of these column morphs did not include row morphs. Two outliners were still unaccounted for; perhaps they did not even contain column morphs. The query

```
objectOutliner a.   (a morphs size = 0)
```

confirmed this guess and showed that the two remaining object outliners were special prototype objects. This example demonstrates how our debugger can help to understand both the structure of objects as well as the interactions among them. As we have shown, the debugger can reveal anomalous objects, which might not always be erroneous as witnessed in the last query.

2.3.2 Understanding the Cecil compiler

Because our debugger helped us to understand Self GUI objects, we decided to look at a complex system which we had no previous experience with—a prototype Cecil [Cha93] compiler written in Self by Craig Chambers, Jeff Dean, and David Grove. Here the main goal was to understand the system and in particular to understand relationships among objects in the system. During compilation of a Cecil program, the compiler creates a large number of internal objects representing Cecil language constructs: declaration contexts, Cecil objects and their bindings, methods, and so on.

First, we explored parts of the compiler by finding compiler objects corresponding to Cecil constructs in the compiled Cecil program. We discovered a number of properties of the Cecil types. For example, the simple Cecil program we compiled did not have named types with instantiations:

```
cecil_named_type a.   (a instantiations size != 0)
```

Also, the query below showed that only three Cecil types had subtypes:

```
cecil_named_type a.   (a subtypes size != 0)
```

Can Cecil programs have formals with the same name in different methods?

```
cecil_method a b; cecil_formal c d.
(a formals  includes: c) && (b formals includes: d) &&
(c name = d name) && (c != d) && (a != b)
```

The query result was nonempty, confirming the notion that formals with the same name can indeed occur in different methods.

We made a number of other queries about the compiler objects. Overall, the debugger proved to be a valuable tool in understanding the Cecil compiler. This experience leads us to believe that query-based debuggers will be useful for other programmers trying to understand complex OO systems.

3. Implementation

We implemented the query-based debugger in Self [US87], a prototype-based, pure, object-oriented programming language. We chose Self as our experimental platform because it is a demanding platform for debugging due to the large number of objects in the system, as well as the numerous complex object relationships. In addition, Self provides several features that simplify the implementation of the prototype system. In particular, Self allows run-time (on-the-fly) code generation [CUL89]. Finally, we had access to the Self Virtual Machine source code, which required modification for the purposes of the debugger.

In section 3.1, we present the general structure of the system. In the remainder of section 3, we discuss in more detail the most important parts of the debugger.

3.1 General structure of the system

Figure 1 shows a data flow diagram of the debugger. A parser extracts the domain variables from the query string and passes their types to the collection module which then finds all objects of domain types (see section 3.2) and returns them in separate arrays to the execution module. The parser also transforms the query string into an intermediate form, which is a collection of constraints, each of which in turn corresponds to one constraint of the conjunctive form of the query. The parser then passes this intermediate form to the optimizer which performs order and method optimizations using the knowledge about query constraints and domain sizes (see section 3.4). The code generator module then generates Self source code for the query-specific constraint-checking methods and integrates it with prefabricated constraint evaluation code. In the last phase the execution module sets up the run-time environment for the query evaluation, runs the query code, and finally sends the result back to the user.

The rest of this section discusses the most important parts of the debugger in more detail.

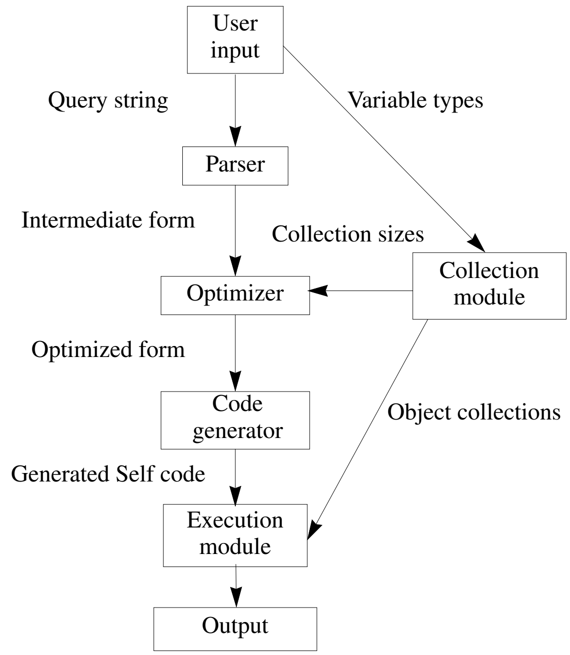

Figure 1. Overview of the query-based debugger

3.2 Enumerating all objects in a domain

To enumerate all objects in a domain, we extended the Self virtual machine with a new primitive that scans the entire heap and returns a vector containing all matching objects. Since the Self implementation already canonicalizes its internal type descriptors (*maps* [CUL89]), it ensures that all objects with the same object layout (i.e., the same slot names and, for constant slots, slot contents) have the same type descriptor. Therefore, the primitive merely needs to find all references to a particular type descriptor, a task that can be accomplished fairly quickly. The cost of the primitive is dependent on the total size of the heap and the number of matches (i.e., the size of the result). On our test machine (a 200 MHz UltraSPARC workstation) the primitive searches about 110 Mbytes per second and can return about 700,000 objects per second. Given this speed, the primitive has never been a bottleneck in our system.

3.3 Overview of query execution

The debugger finds answers to a query by sequentially evaluating all constraints of the query for the query's domain variables. Constraint evaluation is similar to a relational database join coupled with a selection. The initial inputs are individual domains, but during the evaluation these domains are joined; consequently later evaluations consume tuples of objects. Figure 2 shows the query execution of the query

```
objectOutliner o; columnMorph m; rowMorph r.
(o morphs includes: m) && (m morphs includes: r).
```

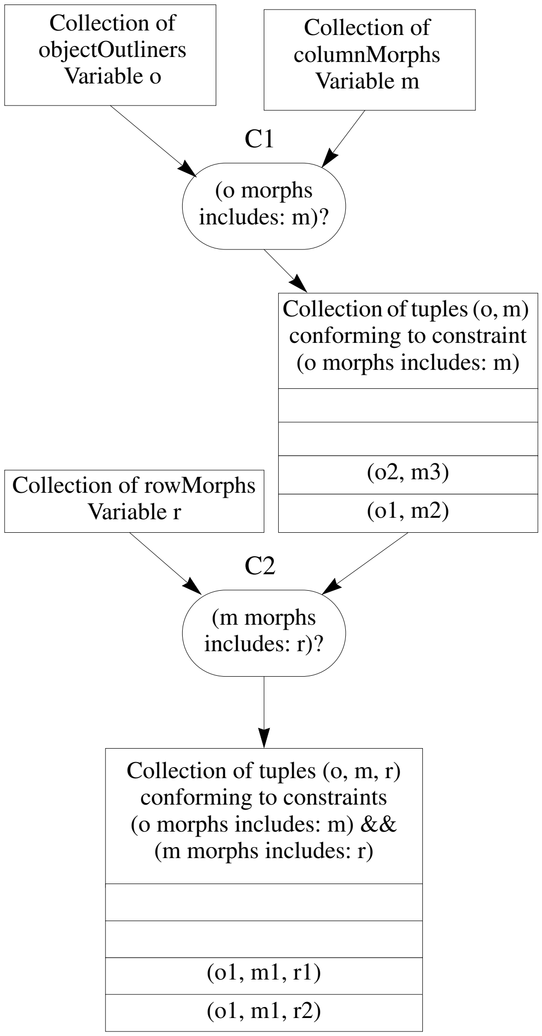

Figure 2. Overview of query execution

This query consists of two constraints, each involving two of the three domain variables. The first constraint, (o morphs includes: m) consumes object outliners and column morphs as input; the second constraint consumes tuples with object outliners and column morphs.

Following the evaluation procedure outlined in Figure 2, the system performs a chain of joins, and ends up with a single output collection containing tuples with all objects. Since only the tuples satisfying earlier constraints are passed along the chain, the output contains only tuples satisfying all constraints. In our example, tuples (o1, m1, r1) and (o1, m1, r2) conform to both constraints and are results of the query.

The system could form the chain of joins in an arbitrary way, but some execution orders are more efficient than others, presenting an opportunity for optimization.

3.4 Join ordering

The execution order of the individual joins of a query significantly influences overall performance. To see why, let us first consider the cost of a single join. The input size of a single join affects its execution time, while its selectivity affects the input sizes of subsequent joins. (The selectivity is the ratio of tuples that do not conform to the constraint to the number of input tuples in the Cartesian product.†) Both the selectivity and the input sizes of a join depend on the join ordering. In particular, evaluating joins with low selectivity at the beginning of join chain can produce large intermediate results that slow down the evaluation of subsequent joins. For example, the Cartesian product of two 10,000-element relations produces a relation with 100,000,000 tuples. Such a relation would not only be costly to store but also very time-consuming to use.

To optimize the query execution, we must minimize the intermediate results by finding an optimal join ordering. There are no general algorithms that find an optimal join ordering in polynomial time—in fact, this problem is NP-complete [IK84]. Several algorithms produce optimal or near-optimal orders for restricted cases. For example, the KBZ algorithm [KBZ86] computes optimal join orders for left-deep ordering where the join graph is acyclic (for cyclic graphs the algorithm gives approximate results), and assumes perfect knowledge of join sizes and selectivities.

Numerous heuristics try to find near-optimal orderings [IK84, KBZ86, SI93, SMK93]. One such heuristic is the *minimum-cost heuristic* that performs the lowest-cost join first. It uses the size of the input relations as the sole cost factor, with the join's cost being the product of the inputs' sizes for the nested loop join. The minimum-cost heuristic is based on the observation that performing cheap joins in the beginning should shrink the intermediate result size, making the expensive (large) joins at the end less expensive to perform.

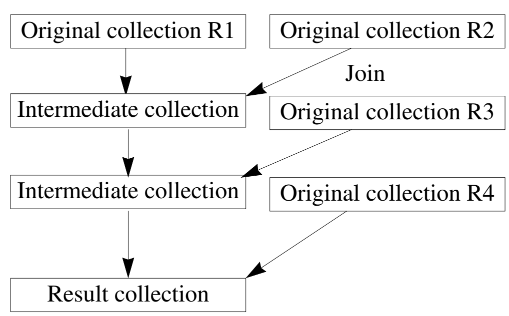

Figure 3. Left-deep join

† The database community uses an inverse terminology where "high selectivity" means "many tuples pass the selection". We find it more intuitive for "highly selective" to mean "only few tuples pass the selection".

In addition to using the minimum-cost heuristic, we impose a restriction on the join order by using *left-deep ordering*. Left-deep ordering [SAC+79, SMK93] requires each join to have one and only one intermediate result relation as an input (Figure 3). Left-deep ordering increases the probability of finding an efficient join order since it has a larger percentage of "good" orderings in its search space [SMK93]. It also simplifies incremental result delivery as explained further below.

3.5 Maximum-selectivity heuristic

Another heuristic used to order joins is the *maximum-selectivity heuristic*. It performs joins with the highest selectivity first because such joins eliminate the most tuples and consequently produce smaller input sizes for subsequent queries. Although the usefulness of selectivity has been well-documented in the database literature (e.g., see [KBZ86]), its use in the query-based debugging is different. Queries asked during debugging typically have modest output sizes of a few dozen objects, despite search domains that may encompass many thousands or even millions of tuples. This is not a coincidence—programmers want to inspect the objects in the query result, and thus are unlikely to pose queries that have large outputs. Consequently, the selectivity of a typical query is very high (e.g., 99.99% of all tuples do not pass the condition).

In our debugger we know the precise input sizes of joins because all domains are enumerated before determining the query evaluation order. However, we do not know the selectivities of the joins. To accurately estimate selectivity, the query optimizer would have to sample a large number of tuples, a process which can negate the potential speedups gained from better query ordering.

To improve query optimization our system uses *imprecise* selectivity information by randomly choosing ten objects from each domain and evaluating the constraint for all tuples formed by these objects. The number of tuples satisfying the constraint indicates the selectivity of the join. While imprecise, it still allows to distinguish highly-selective constraints (selectivities larger than 0.80) from constraints that have low selectivities (less than 0.50). We decrease the priority of the joins containing low selectivity constraints by multiplying their cost by a factor of 100.

How big should the sample be that is used to estimate selectivity? Larger samples allow more precise estimates, but small samples reduce the cost of estimating the selectivity (and thus the overhead of query optimization). We experimented with several sample sizes and found a sample size of 10 to be a good compromise between accuracy and cost. On average, a sample size of ten produces selectivity estimates with a standard deviation of 0.035, which is accurate enough for our high-low distinction. Increasing the sample size to 100 reduces the standard deviation to 0.01 but is 100 times more costly for a two-object constraint.

The debugger uses a modified minimum-cost heuristic that takes into account two-level (high-low) selectivity estimates. For example, for the query

```
cecil_method a b; cecil_formal c d.
(a formals  includes: c) && (b formals includes: d) &&
(c name = d name) && (c != d) && (a != b)
```

the minimum-cost heuristic alone would choose to evaluate the constraints in the following order: (c name = d name), (b formals includes: d), (a != b), (a formals includes: c), (c != d), resulting in a query execution time of 37 seconds. By using selectivity estimates the system recognizes that the join (a != b) has a low selectivity and should be evaluated as late as possible. Consequently it chooses a different evaluation order: (c name = d name), (b formals includes: d), (a formals includes: c), (a != b), (c != d). This improved evaluation order reduces the query execution time to 5.9 seconds.

3.6 Hash Joins

Many query constraints have the form $\alpha = \beta$, where α and β are expressions involving two different domains. In such cases the system can compute the corresponding join using the *hash-join* method instead of using less efficient nested-loop method [ME92]. Consider the following query about Cecil structures:

```
cecil_named_object n; cecil_top_context t;
cecil_object_binding o.
(n defining_context = t) &&
(t varBindings includes: o) && (o value = n)
```

In this Cecil query we can use a hash join to evaluate the (n defining_context = t) constraint. Hash joins do not affect the join semantics; they are simply a more efficient way to evaluate a join.

To evaluate a constraint using a hash join, we first construct a hash table mapping all result values of expression α to the domain variable(s) that produced these results. In Figure 4 that uses the above example, the hash table maps contexts (the results of applying n defining_context expression to objects n1 and n3) to cecil_named_object objects n1 and n3. Then, we evaluate the expression β for each object of the corresponding domain and probe the hash table with each result. If the probe is successful (i.e., if the hash table contains one or more tuples that evaluated to the same value using expression α), all of these tuples are added to the result because they satisfy the condition $\alpha = \beta$. In Figure 4, the cecil_top_context object t2 is mapped to the same slot as the cecil_named_object n3, and t2 = n3. Consequently the tuple (n3, t2) satisfies the constraint and belongs to the result. Using the hash join method, the join result can be

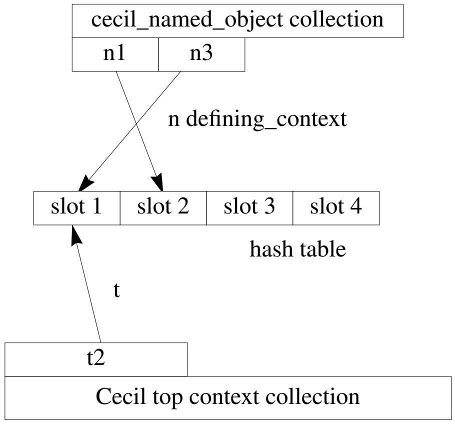

Figure 4. Hash join

constructed with a single scan through all tuples in β's domain.

On average hash joins are more efficient than nested loop joins because they execute in O(|α| + |β|) time if the output is small, compared to O(|α| * |β|) time required for a nested loop join regardless of the output size. If the output is close to the size of Cartesian product then the cost of a hash join will also be O(|α| * |β|). Our join ordering algorithm uses |α| + |β| to estimate the cost of a hash join. For example, if there are 4 cecil_top_context objects and 69 cecil_named_object objects, the estimated cost of the hash join (n defining_context = t) is 73.

The experiments in section 4.5 demonstrate that hash joins can dramatically improve performance for some queries.

3.7 Incremental delivery

Because a query-based debugger is a part of an interactive programming environment, we would like to achieve interactive performance for the widest possible class of queries. But sometimes it is impossible to compute the entire query result within a short time, for example, the input relations may be very large or the individual query expressions may involve time-consuming operations. Incremental delivery improves the response time of long-running queries by delivering the first result(s) as quickly as possible, so that the user can inspect them while the rest of the result is being computed. To achieve this goal, we developed a technique that dynamically adjusts the CPU time allocated to individual constraints to produce the first result as quickly as possible.

In our system, each join executes as a Self thread and is connected to the next join by a limited-size intermediate result buffer. Figure 5 shows buffers of size 4, and both buffers contain two tuples. This arrangement resembles a pipeline where intermediate results flow along the pipeline toward the next query. As in the classical producer-consumer problem, a thread in the pipeline blocks when its output

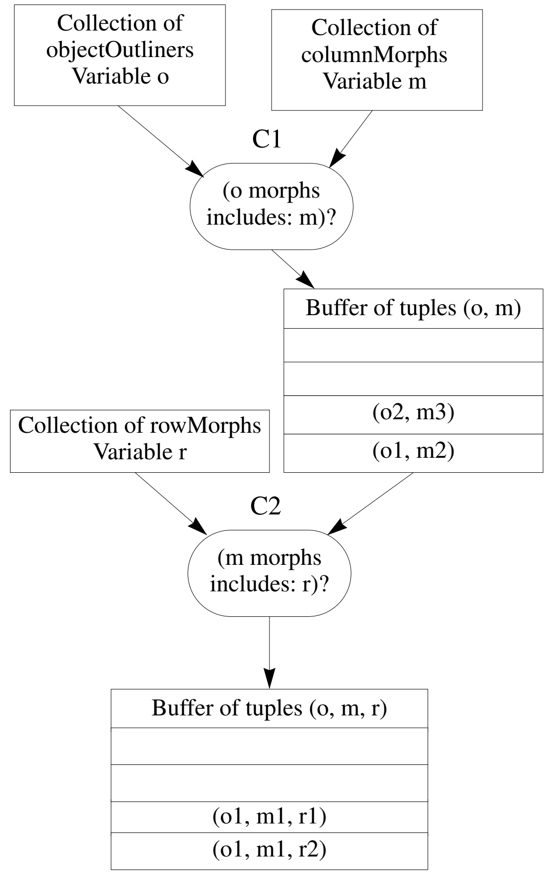

Figure 5. Incremental delivery pipeline

buffer is full or when its input buffer is empty. Whenever a thread blocks, the query scheduler picks the next thread to run. By keeping the size of the intermediate buffers small—on the order of 100 elements—the system can "push" intermediate results down the pipeline towards the output before early joins compute their complete results.

The join threads must be scheduled correctly to minimize the response time. We use a simple queue scheduling scheme that prefers threads closer to the end of the pipeline. As a result, intermediate results flow towards the end of the pipeline because consumers closer to the end will run before joins earlier in the pipeline will produce new intermediate results. For example, in Figure 5 the thread executing join C1 will block when it fills the remaining slots in the intermediate buffer or when it finishes scanning through input collections. Then the scheduler will select and run the thread executing join C2 because it is closer to the end of the pipeline and it has non-empty input and non-full output buffers.

Although the thread pipeline by itself increases the speed of output generation, we optimize it further by time-slicing the threads. Time-slicing prevents a slow (or highly selective) join at the beginning of the pipeline from running for a long time before filling its output buffer. Without time-slicing, if

Query	Input	Output
1. morph a b c. (a morphs includes: b) && (c morphs includes: b) && (a != c)	37*37*37	0
2. objectOutliner a; rowMorph b; columnMorph c. (a morphs includes: b) && (b morphs includes: c)	12*146*370	0
3. objectOutliner a; rowMorph c; columnMorph b. (a morphs includes: b) && (b morphs includes: c)	12*146*370	1
4. objectOutliner a; columnMorph b. (a morphs includes: b)	12*370	11
5. objectOutliner a; rowMorph b. (a morphs includes: b)	12*146	0
6. objectOutliner a. (a morphs size = 0)	12	1
7. objectOutliner a; smallEditorMorph b. (a titleEditor = b) && (b owner = a)	12*16	0
8. objectOutliner a; columnMorph b; labelMorph c. (a morphs includes: b) && (c owner = b) && (a moduleSummary = c)	12*370*1006	0
9. cecil_named_object a; cecil_top_context b; cecil_object_binding c. (a defining_context = b) && (b varBindings includes: c) && (c value = a)	69*4*79	68
10. cecil_named_type a. (a instantiations size != 0)	198	0
11. cecil_named_type a. (a subtypes size != 0)	198	3
12. cecil_method a. (a resultTypeSpec printString = 'int')	167	2
13. point a; rectangle b. (a x = b origin y) && (a x = 6)	11195*4579	6,780
14. point a. a x = 256	11195	2
15. point a; rectangle b b1. (a x = b origin y) && (b height = b1 height) && (b != b1) && (b1 height = 1000)	11195*4579*4579	0
16. point a; rectangle b b1. (a x = b origin y) && (b height = b1 height) && (b != b1) && (b1 height > 1000)	11195*4579*4579	12,467
17. rectangle b b1. (b height > (b1 height + 800)) && (b width < (b1 width - 900))	4579*4579	0
18. cecil_method a b; cecil_formal c d. (a formals includes: c) && (b formals includes: d) && (c name = d name) && (c != d) && (a != b)	167*167*179*179	7042
19. mutableString a. (a asSlotIfFail: [abstractMirror]) isReflecteeSlots	15540	11,281

Table 1: Sample queries with their input and output sizes

the first join does not completely fill its output buffer it would run to completion before a thread switch occurs, and the first query output would not appear before the entire first join is completed. Instead, the scheduler preempts threads after they have used up their time slice and schedules the new highest-priority thread. Thus, if the first join produced any intermediate results during its time slice, these results are pushed down the pipeline, shortening the time to produce the first result. To keep thread switching overhead low, we keep time slices reasonably long (e.g., 100 ms per slice).

4. Experimental results

We tested the debugger on a number of realistic and synthetic queries. For our tests we used an otherwise idle Sun Ultra 2/2200 machine (with a 200 MHz UltraSPARC processor) running Solaris 2.5.1 and a modified version of Self 4.0. Execution times reported are elapsed times. Times were measured with millisecond accuracy. We observed a variance of about 10% in the measurements due to various asynchronous events in the Self VM and user thread scheduling effects. We also measured the total CPU time to verify that no other processes disrupted the measurements. We chose the lowest time observed during three repetitions of a measurement.

4.1 Benchmark queries

We ran the tests using both realistic queries as well as artificial queries. Table 1 lists the queries and their sizes. We selected a number of realistic queries and tried to present a fair sample in terms of query complexity, query evaluation methods, and query input and output sizes. The realistic queries (1–12) dealt with inputs ranging from 10 to 1,000 objects per type (see Table 1). Queries 1-8 involve the Self GUI, and queries 9-12 involve the Cecil compiler.

To test the limits of the debugger's performance, we also evaluated difficult synthetic queries (queries 13-19) that are less likely to be asked in real life. These queries involved larger inputs that contain tens of thousands of objects (except query 18). Some of them were difficult to evaluate efficiently because the system could not use hash joins (query 17), or because they had an empty result set (query 15), where incremental delivery techniques did not help. Queries 18 and 19 are realistic but time-consuming, so they were grouped together with other queries that place considerable stress on the debugger's performance.

Queries can be divided into the following broad classes:

- Queries 6, 10, 11, 12, 14, and 19 are simple one-constraint selection queries. Since these queries along

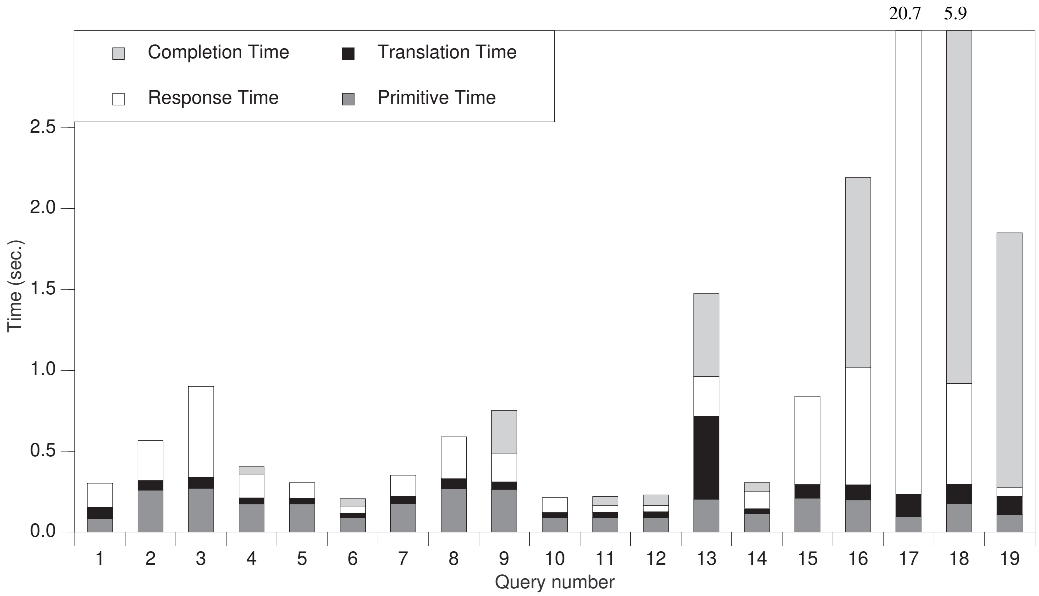

Figure 6. Query execution times

with queries 4 and 5 have only one constraint, their evaluation time does not depend on join optimization.

- Queries 1, 2, 5, 7, 8, 10, 15, and 17 are assertion queries that have an empty result set. Incremental delivery techniques do not affect their response time.
- Queries 7, 8, 9, 13, 15, 16, and 18 have some equality constraints that can be evaluated using hash joins, while the other queries use only nested-loop joins.

4.2 Execution time

Figure 6 shows the execution times of all queries, split into four components:

- *Primitive time*, the time spent computing the input domains of the query using the primitive described in section 3.2.
- *Translation time*, the time spent choosing a query evaluation order and generating the code to execute the query.
- *Response time*, the time spent producing the first result (not including the time needed to display it in the graphical user interface). For queries producing an empty answer, the response time includes the entire query evaluation because the user has to wait until the end of the query evaluation to learn the outcome.
- *Completion time*, the remaining execution time needed to complete the query and produce all results.

For example, for query 16 it took less than 0.2 seconds to collect all points and rectangles, and less than 0.1 seconds to translate the query into Self code. The first result appeared after an additional 0.7 seconds, and it took another 1.2 seconds to complete query evaluation.

Overall, the results are encouraging, with most of the realistic queries (1-12) taking less than a second to evaluate. For these queries, the median response time was 0.33 seconds, and the median completion time was 0.33 seconds. Query 18 was the only realistic query that took a long time to evaluate, but even there the first result appeared in less than a second.

Some of the artificial queries (13-16) also executed in the subsecond range. For these queries, the median response time was 0.89 seconds, and the median time to complete the query was 1.14 seconds. Query 17 took 20.7 seconds to complete. This query is difficult for the system to evaluate efficiently because it does not contain equality constraints. Consequently the system can not use hash joins. The debugger had to execute a nested-loop join for 4579×4579 rectangles, evaluating the first constraint 21 million times. Query 19 took 2 seconds to complete. This query is a simple selection query, but it performs a time-consuming operation—compilation of the Self code string—for each mutable string.

We found that many queries had such enormous outputs that our machine ran out of memory trying to produce results. Usually, such queries are asked by mistake, since the programmer probably would not want to look at millions of tuples in a result. A straightforward restriction on the result size eliminates system crashes and alerts users to this type of mistake.

4.3 Join ordering

To see how well the query optimizer works, we compared its performance against the best and worst join orderings. In this experiment, we executed the queries using all possible left-

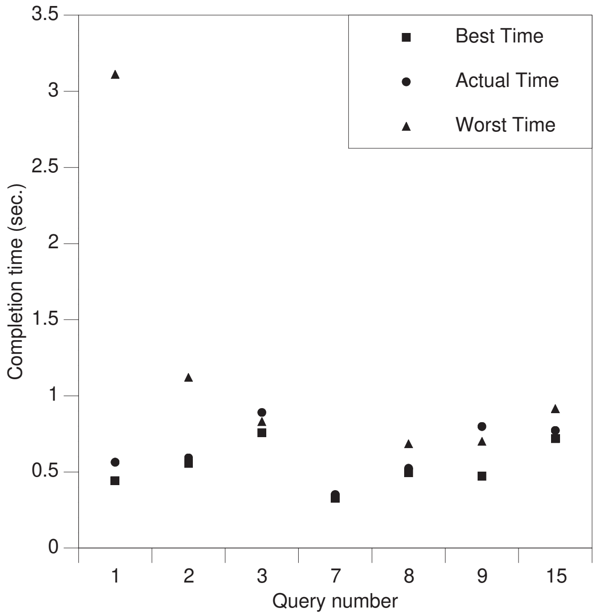

Figure 7. Completion time depending on join ordering (small queries)

deep orderings that perform selections first. Such orderings correspond to the search space of our join ordering algorithm. In all cases the best ordering in this search space was indeed the globally optimal ordering, so the search space limitation did not influence the results. We did not include queries with only one constraint in this experiment.

Figure 7 compares the best and the worst query completion times to the completion time using the ordering found by our system. The results show that query optimization works well for small queries[†], but the join ordering does not matter very much for such queries since even in the worst case they take less than a second to execute.

However, larger and more complicated queries have much wider range of best and worst times (Table 2). Queries 16 and 18 clearly demonstrate the need for join order optimization. Our system evaluates query 16 in 2.2 seconds, but the worst-case execution time is more than 10 minutes. Our system does not perform optimally all the time; it sometimes finds fairly bad orderings but the outcome of the join ordering may be cushioned by the incremental delivery methods discussed later.

[†] Sometimes, though, the debugger has an execution time slightly higher than the "worst" one due to optimization overhead and random system events.

Query number	Best	Actual	Worst
16	1.1	2.2	631
17	17	20.7	27.2
18	3	5.9	130

Table 2: Completion time depending on join ordering (large queries)

4.4 Incremental delivery

Incremental delivery significantly reduces the response time of the system. For example, query 18 took 5.9 seconds to complete but produced the first result in less than a second. Overall, the response time ranged from 12% to 100% of the completion time with a median of 87% (recall Figure 6). Excluding queries with empty answer sets (which cannot benefit from incremental delivery) the median ratio is 73%, i.e., incremental delivery reduces the waiting time for the programmer by about 27%.

The use of incremental delivery also limits sizes of intermediate collections. For example, query 18 runs out of memory when executing in non-incremental mode because it produces enormous intermediate results. In incremental mode, it evaluates successfully because intermediate result sizes are limited by buffer sizes.

To check the influence of time-slicing on the response time of different queries, we evaluated all queries using time slices ranging between 100 ms and infinite time slice. The experiments showed that time-slicing does not have visible effect on most queries. However, time-slicing becomes important for large queries with small result sets. For example, if we take assertion query 17 that has an empty result set, and add an object to the system that violates the assertion, the query execution pattern changes. The assertion originally took 20 seconds to verify, but the object violating the assertion is found early in the execution. Consequently, the response time is 3 seconds, while the completion time remains 20 seconds. If the time-slicing is disabled, the first join in the query does not fill the intermediate buffer, so no thread switches occur almost until the end of the execution. As a result, without time-slicing the response time becomes equal to the completion time of 20 seconds.

In summary, incremental delivery can substantially shorten the response time; large queries and queries with nested loop joins benefit most from incremental delivery. Time-slicing is a useful enhancement when dealing with large queries.

4.5 Hash joins

Many queries can use hash joins to avoid computing the full Cartesian product. But as explained in section 3.5, some joins must be computed the hard way, using nested loops. How much slower are such joins? To answer this question we re-executed all queries that use hashed joins with hashing disabled. Table 3 shows the slowdowns of these query evaluations. In our test system, slowdowns ranged from 0.6 to 2 with a median of 0.98. Query 15 experienced a speedup because the optimizer found a better join ordering for nested loop joins than the one used for hash joins. When we increased the size of each relevant domain by five times to simulate a larger system, the slowdowns became substantial (right column of Table 3), ranging from 0.32 to 15.6 with a median of 1.97.

The results of these preliminary experiments indicate that hash joins can be more efficient than nested-loop joins, although they slow down several of our test cases. With large input domains, the performance advantage of hash joins can exceed an order of magnitude (e.g., in query 18).

Query number	Small system	5x larger system
7	0.93	1.13
8	1.18	1.97
9	2	6.42
13	1.94	5.76
15	0.63	0.32
16	0.98	0.91
18	0.6	15.6

Table 3: Slowdown of nested queries vs. hash queries

Fortunately, even if hash joins could not be used, incremental delivery can mask much of the nested loop join overhead by producing the first result quickly. Table 4 shows the response times both with and without hashing. For these queries the first results can be produced in less than a second, and the response time varies by less than a factor of two between the two configurations.

Query number	Nested (sec.)	Hash (sec.)	Ratio
7	0.33	0.35	0.93
8	0.7	0.6	1.18
9	0.7	0.5	1.46
13	0.89	0.96	0.92
15	0.5	0.8	0.63
16	0.93	0.98	0.95
18	0.72	0.92	0.78

Table 4: Response time (time to first result)

5. Future work

The current query-based debugger can be extended in a number of ways. First, the query model can be extended to include projection or even arbitrary computations that involve the selected objects. For example, it would be useful to write queries that calculate the average length of certain lists. Although this is possible to do now by iterating over the tuples in the query result, integrating such functionality into the query model could make such computations easier to express and potentially more efficient. In this way the system no longer needs to construct and then consume the output tuples.

A second avenue for future work is to extend the system to provide dynamic queries; i.e., queries whose results are dynamically updated while a program is running. For example, dynamic queries could help maintain class invariants [Mey88]. A query whose condition is the negated class invariant should have an empty result. If query results were updated dynamically the user could stop the program as soon as the result became non-empty, i.e., as soon as the class invariant was violated.

Unfortunately, dynamic queries are quite difficult to implement, especially if the system does not require programmers to annotate their programs to enable dynamic queries. Major difficulties in implementation arise in two main areas: the first area of difficulty is to determine when the input domains change, and the second area of difficulty is to determine when it is safe to reevaluate an expression. Consider the query

```
p: Point, r: Rectangle.   (p = r origin).
```

Its result may change if the program creates new points or rectangles or if old objects become garbage. Furthermore, the result can also change if a rectangle's origin changes, either when a new point is assigned to the `origin` instance variable or when the existing origin is mutated (e.g., by assigning to the `x` instance variable). In addition, the system would also have to analyze the equality method to determine which objects and fields it transitively references, and then to add all of these to the set of objects that need to be monitored for changes. Efficiently keeping track of these dependencies during millions of object mutations per second will be a challenging task. This is especially true for a language as dynamic as Self which is less amenable to static analyses like slicing [Tip95, Wei81].

Even if the above problem can be solved efficiently, dynamic queries still face another challenge: how can the query evaluator know when it is safe to update the query result? Consider a query involving list operations such as the Cecil queries in section 2.3.2. If the system evaluates such a query expression at the same time that a new element is being inserted into the list, the query may lead to an error because the list at this moment in the execution is in a temporarily inconsistent state. Therefore, queries should be updated only when all abstractions involved are in a consistent state. For the system to determine consistent states automatically is probably impossible. It might be difficult even to formulate a heuristic that handles many common cases satisfactorily, especially in the presence of multiple abstractions that cooperate to maintain a higher-level invariant.

Clearly, an implementation of dynamic queries is difficult, but their many attractive features encourage us to study their feasibility more closely. We plan to identify a subset of dynamic queries that is restricted enough to be implemented efficiently yet flexible enough to be useful during debugging.

6. Related work

We are unaware of other work that directly corresponds to query-based debugging. Sefika et al. [Sef96, SSC96] implemented a system allowing limited, unoptimized selection queries about high-level objects in the Choices operating system. The system shows program state and run-time statistics at various levels of abstraction. Unlike our query-based debugger, the application (Choices) was specifically instrumented to allow queries.

While no one has investigated the query-based debugging specifically, various researchers have proposed a variety of enhancements to conventional debugging [And95, Cop94, DHKV93, DKV94, GH93, KRR94, Laf97, LM94, LN97, LN95a, LN95b, LN95c, ZL96]. The debuggers most closely related to query-based debugging allow the visualization of relationships—usually references—among objects. Duel [Gol93] allows users to display data structures by writing script code. HotWire [LM94], Look! [And95], Object Visualizer [DHKV93, KRR94, DKV94], and Program Explorer [LN97, LN95a, LN95b, LN95c] provide numerous graphical and statistical run-time views with class-dependent filtering but do not allow general queries. Our debugger can gather statistical data through queries ("How many lists of size greater than 500 exist in the program?") but does not display animated statistical views.

Software visualization systems such as BALSA [Bro88], Zeus [Bro91], TANGO/XTANGO [Sta90], POLKA and Pavane [RC93, Rom92] offer high-level views of algorithms and associated data structures. Software visualization systems have different uses than everyday debugging. They aim to explain or illustrate the algorithm, so their view creation process emphasizes vivid representation. Hart et al. [HKR97] use Pavane for query-based visualization of distributed programs. However, their system only displays selected attributes of different processes and does not allow more complicated queries. Consens et al. [CHM94, CMR92] uses Hy$^+$ visualization system to find errors using post-mortem event traces. This work is complementary to ours because it focuses on querying and visualizing run-time events while we query object relationships. Noble's observations regarding the use of Self for the Tarraingim program visualization system [NGB95] coincide with our own observations.

The query optimizations discussed in section 3.4 were influenced by research in the area of databases. Ullman [Ull82] discusses basic insights about efficient join evaluation. Numerous researchers [IK84, KBZ86, SI93, SMK93] have developed algorithms to find optimal or near-optimal join orderings. The characteristics of the query-based debugger are different from databases—the number of objects per domain is smaller than in databases, but all objects reside in the main memory, so input/output time is not a factor. The debugger has to deliver interactive, subsecond performance, and cannot spend much time finding the optimal join order. Dynamic queries are related to incremental join result recalculation [BC79, BLT86]. Slicing [Wei81, Tip95] and data breakpoints [WLG93] could provide a foundation for dynamic queries, although they would need to be extended and adapted for this purpose.

Pre-/postconditions and class invariants as provided in Eiffel [Mey88] can complement queries and are checked at the beginning or end of methods. Unlike queries, they cannot check multi-object constraints nor can they invoke arbitrary methods. We view the two mechanisms as complementary, with queries being more suitable for program exploration as well as specific debugging problems.

7. Conclusions

Understanding and debugging complex software systems is difficult. Most conventional debuggers offer only a limited low-level view of the program state and offer little support for the exploration of large data structures. Our work provides a system that allows programmers to ask queries about the program state, helping to check object relationships in large object-oriented programs. Our implementation of the query-based debugger combines several novel features:

- A new approach to debugging: Instead of exploring a single object at a time, a query-based debugger allows the programmer to quickly extract a set of interesting objects from a potentially very large number of objects, or to check a certain property for a large number of objects with a single query.
- A flexible query model: Conceptually, a query evaluates its constraint expression for all members of the query's domain variables. The present model is simple to understand and to learn, yet it allows a large range of queries to be formulated concisely.
- Good performance: Many queries are answered in one or two seconds on a midrange workstation, thanks to a combination of fast object searching primitives, query optimization, and incremental delivery of results. Even for longer queries that take tens of seconds to produce all results, the first result is often available within a few seconds.

Although the system described here is based on the Self system, the query model as well as its implementation could be adapted fairly easily to other languages. We believe that the system demonstrates the usefulness of query-based debugging, and hope that similar functionality will be integrated into future mainstream debuggers, making the daunting task of debugging easier for programmers and facilitating the development of more robust object-oriented systems.

Acknowledgments

The authors would like to thank Karel Driesen for his comments on an early version of this paper. This work was supported in part by National Science Foundation CAREER grant CCR-9624458, an IBM Faculty Development Award, and by Sun Microsystems. Some work was performed on the SGI workstations acquired under NSF instrumentation grant CDA-9421978.

8. References

[AK89] Abiteboul, S.; Kanellakis, P.C., Object identity as a query language primitive. 1989 *ACM SIGMOD International Conference on Management of Data*, Portland, OR, USA, 31 May-2 June 1989). *SIGMOD Record*, vol.18, (no.2), pp. 159-73, June 1989.

[And95] Anderson E., Dynamic Visualization of Object Programs written in C++, *Objective Software Technology Ltd.,* http://www.scotnet.co.uk/ost, 1995.

[BC79] Buneman, O.P.; Clemons E.K., Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems,* 4(3), pp. 368-382, September 1979.

[BKK88] Banerjee, J.; Kim, W.; Kim, K.-C., Queries in object-oriented databases. In: Proceedings Fourth International Conference on Data Engineering, Los Angeles, CA, USA, 1-5 Feb. 1988. Washington, DC, USA: IEEE Comput. Soc. Press, pp. 31-8, 1988.

[BLT86] Blakeley, J.A.; Larson P.-A.; Tompa F. Wm.; Efficiently Updating Materialized Views. *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 61-71, Washington, D.C., USA, May 1986. Published as SIGMOD Record 15(2), June 1986.

[Bro88] Brown, M.H., Exploring Algorithms Using Balsa-II, *IEEE Computer* 21(5), pp. 14-36, May 1988.

[Bro91] Brown, M.H., Zeus: A System for Algorithm Animation and Multi-View Editing, *Proceedings of IEEE Workshop Visual Languages*, pp. 4-9, IEEE CS Press, Los Alamitos, CA., 1991.

[CHM94] Consens, M. P., Hasan M.Z., Mendelzon A.O., Debugging Distributed Programs by Visualizing and Querying Event Traces, *Applications of Databases, First International Conference*, ADB-94, Vadstena, Sweden, June 21-23, 1994, Proceedings in Lecture Notes in Computer Science, Vol. 819, Springer, 1994.

[CMR92] Consens, M.; Mendelzon, A.; Ryman, A., Visualizing and querying software structures, *International Conference on Software Engineering*, Melbourne, Australia, May 11-15, 1992, ACM Press, IEEE Computer Science, p. 138-156, 1992.

[CUL89] Chambers, C., Ungar, D., Lee, E., An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes, In *Proceedings of OOPSLA'89*, pp. 49-70, New Orleans, LA, October 1989. Published as SIGPLAN Notices 24(10), October 1989.

[Cha93] Chambers. C. Cecil language: specification and rationale. *UW-CS Technical Report* 93-03-05, 1993.

[Cop94] Coplien, J.O., Supporting truly object-oriented debugging of C++ programs., In: Proceedings of the 1994 USENIX C++ Conference, Cambridge, MA, USA, 11-14 April 1994. pp. 99-108, Berkley, CA, USA: USENIX Assoc, 1994.

[DHKV93] De Pauw, W.; Helm, R.; Kimelman, D.; Vlissides, J. Visualizing the behavior of object-oriented systems. In *Proceedings of the 8th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1993*, Washington, DC, USA, 26 Sept.-1 Oct. 1993. SIGPLAN Notices, Oct. 1993, vol.28, (no.10):326-37.

[DKV94] De Pauw, W.; Kimelman, D.; Vlissides, J. Modeling object-oriented program execution. In: *Proceedings of the 8th European Conference on Object-Oriented Programming, ECOOP '94*, Bologna, Italy, 4-8 July 1994. pp. 163-82, Edited by: Tokoro, M.; Pareschi, R. Berlin, Germany: Springer-Verlag, 1994.

[Gol84] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1984.

[GH93] Golan, M.; Hanson, D.R. Duel-a very high-level debugging language. In: USENIX Association. *Proceedings of the Winter 1993 USENIX Conference*. San Diego, CA, USA, 25-29 Jan. 1993. Berkley, CA, USA: USENIX Assoc, 1993. p. 107-17.

[GR91] Gold, E.; Rosson, M.B., Portia: an instance-centered environment for Smalltalk. *OOPSLA '91. Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, AZ, USA, 6-11 Oct. 1991. Published as *SIGPLAN Notices*, vol.26, (no.11), pp. 62-74, November 1991.

[GWM89] Gamma E., Weinand A., Marty R., Integration of a Programming Environment into ET++ - a Case Study, *Proceedings ECOOP'89* (Nottingham, UK, July 10-14), pp. 283-297, S. Cook, ed. Cambridge University Press, Cambridge, 1989.

[HKR97] Hart D., Kraemer E., Roman G.-C., Interactive Visual Exploration of Distributed Computations. To appear in the *International Parallel Processing Symposium* 1997.

[HU96] Hölzle, U.; Ungar, D., Reconciling Responsiveness with Performance in Pure Object-Oriented Languages, *ACM Trans. Programming Languages and Systems* 18(4), pp. 355-400, 1996.

[IK84] Ibaraki T., Kameda T., On the Optimal Nesting for Computing *N*-Relational Joins., A*CM Transactions on Database Systems*, Vol. 9, No. 3, pp. 482-502, September 1984.

[KBZ86] Krishnamurthy, R., Boral, H., Zaniolo, C., Optimization of Nonrecursive Queries., *Proceedings of International Conference on Very Large Data Bases* (VLDB), pp. 128-137, 1986.

[KRR94] Kimelman D., Rosenburg B., Roth T., Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior, Proceedings of Visualization'94, pp. 172-178, IEEE 1994.

[Laf97] Laffra C., *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*, pp. 229-252, Prentice Hall 1997.

[LM94] Laffra C., Malhotra A., HotWire: A Visual Debugger for C++, *Proceedings of the USENIX C++ Conference*, pp. 109-122, Usenix Association 1994.

[LN95a] Lange, D.B., Nakamura Y. Program Explorer: A Program Visualizer for C++, *Proceedings of USENIX Conference on Object-Oriented Technologies'95*, pp. 39-54, June 1995.

[LN95b] Lange, D.B., Nakamura Y. Object-Oriented Program Tracing and Visualization, *IBM Research Report*, July 1995.

[LN95c] Lange, D.B., Nakamura Y. Interactive Visualization of Design Patterns Can Help in Framework Understanding, *Proceedings of OOPSLA'95*, pp. 342-357, Austin, TX October 1995. Published as SIGPLAN Notices 30(10), October 1995.

[LN97] Lange, D.B., Nakamura Y. Object-Oriented Program Tracing and Visualization, IEEE Computer, vol. 30, no. 5, pp. 63–70, May 1997.

[Mey88] Meyer B., *Object-oriented Software Construction*, pp. 111 - 163, Prentice-Hall, 1988.

[ME92] Mishra, P., Eich, M. H., Join Processing in Relational Databases, *ACM Computing Surveys*, vol. 24, No. 1, pp. 63-113, March 1992.

[NGB95] Noble J., Groves L., Biddle R., Object Oriented Program Visualisation in Tarraingim, *Australian Computer Journal*, 27:4, November 1995.

[NOT96] Nishimura, S.; Ohori, A.; Tajima, K., An equational object-oriented data model and its data-parallel query language. In: *OOPSLA '96: Eleventh Annual Conference on Object Oriented Programming Systems Languages and Applications*, San Jose, CA, USA, 6-10 Oct. 1996). Published as *SIGPLAN Notices*, vol.31, (no.10), pp. 1-17, October 1996.

[RC93] Roman G.-C., Cox K.C., A Taxonomy of Program Visualization Systems, *IEEE Computer* 26(12), pp. 11-24, December 1993.

[Rom92] Roman, G.-C. et al., Pavane: A System for Declarative Visualization of Concurrent Computations, *Journal of Visual Languages and Computing*, Vol. 3, No. 2, pp. 161-193, June 1992.

[SAC+79] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., Price, T. G., Access Path Selection in a Relational Database Management System, *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 23-34, Boston, USA, June 1979.

[Sef96] Sefika M., Design Conformance Management of Software Systems: An Architecture-Oriented Approach. *Ph.D. thesis*, University of Illinois at Urbana-Champaign, July 1996.

[SMK93] Steinbrunn, M., Moerkotte, G., Kemper, A., Optimizing Join Orders, *Technical Report MIP 9307,* Universität Passau, FMI, September 1993.

[SMU95] Smith, R.B.; Maloney, J.; Ungar, D. The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. *OOPSLA '95: Conference on Object Oriented Programming Systems Languages and Applications. Tenth Annual Conference*, Austin, TX, USA, 15-19 Oct. 1995. Published as *SIGPLAN Notices*, vol.30, (no.10), pp. 47-60, October 1995.

[SSC96] Sefika M., Sane A., Campbell R.H., Architecture-Oriented Visualization, In *Proceedings of OOPSLA'96*, pp. 389-405, San Jose, CA, October 1996. Published as SIGPLAN Notices 31(10), October 1996.

[Sta90] Stasko, J., TANGO: A Framework and System for Algorithm Animation, *IEEE Computer* 23(9), pp. 27-39.

[SI93] Swami, A., Iyer, B., A Polynomial Time Algorithm for Optimizing Join Queries, *Proceedings of the IEEE Conference on Data Engineering*, pp. 345-354, Vienna, 1993.

[SZ90] Shaw, G.M.; Zdonik, S.B., A query algebra for object-oriented databases. In: Sixth International Conference on Data Engineering, Los Angeles, CA, USA, 5-9 Feb. 1990. pp. 154-62, Los Alamitos, CA, USA: IEEE Comput. Soc, 1990.

[Tip95] Tip, F., A survey of program slicing techniques. *Journal of Programming Languages*, vol.3, (no.3) pp. 121-89, Sept. 1995.

[Ull82] Ullman, J.D., *Principles of Database Systems*, pp. 268-316, Computer Science Press 1982.

[US87] Ungar, D., Smith, R.B., Self: The Power of Simplicity, *Proceedings of OOPSLA'87*, pp. 227-243, Orlando, FL, October 1987. Published as SIGPLAN Notices 22(12), October 1987.

[Wei81] Weiser, M., Program slicing. In: *5th International Conference on Software Engineering*, San Diego, CA, USA, 9-12 March 1981. New York, NY, USA, pp. 439-49, IEEE, 1981.

[WLG93] Wahbe R., Lucco S., Graham S.L., Practical Data Breakpoints: Design and Implementation. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation* 1993, Albuquerque, June 1993. ACM Press 1993.

[WG94] Weinand, A.; Gamma, E. ET++-a portable, homogenous class library and application framework. In: *Computer Science Research at UBILAB, Strategy and Projects. Proceedings of the UBILAB Conference '94*, Zurich, Switzerland, 1994. pp. 66-92. Edited by: Bischofberger, W.R.; Frei, H.-P. Konstanz, Switzerland: Universitätsverlag Konstanz, 1994.

[ZL96] Zeller A., Lütkehaus D. DDD - A Free Graphical Front-End for UNIX Debuggers. *ACM SIGPLAN Notices*, Vol. 31, No. 1, pp. 22-28, January 1996.