Query-Based Debugging

UNIVERSITY OF CALIFORNIA
Santa Barbara

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy in Computer Science

by Raimondas Lencevicius

Technical Report TRCS 99–27

Committee in charge:

Professor Urs Hölzle, co-chair
Professor Ambuj Kumar Singh, co-chair
Professor Anurag Acharya
Professor Teofilo Gonzalez
Professor Martin Rinard
Professor Jianwen Su

August 1, 1999

The dissertation of Raimondas Lencevicius is approved:

_____

Professor Anurag Acharya

_____

Professor Teofilo Gonzalez

_____

Professor Martin Rinard

_____

Professor Jianwen Su

_____

Professor Ambuj Kumar Singh, co-chair

_____

Professor Urs Hölzle, co-chair

June 1999

June 4, 1999

Copyright by
Raimondas Lencevicius
1999

Acknowledgments

The only thing more reliable than magik is one's friends,

Macbeth[1]

Six years spent at the University of California, Santa Barbara is a long time period, and I can only hope to thank most of the people who influenced me over these years. First of all, I want to express my gratitude to my advisors. Dr. Ambuj Kumar Singh invited me to his research group and supported me in every way even when I changed my research direction in midstream. He continued to provide invaluable ideas and critique about my thesis research. Although sometimes our exchanges were as heated as these of a teenager and his parent, Ambuj always found a way to understand me and help me in my research. Dr. Urs Hölzle proposed the idea for my thesis project and continued to help with keen insights and guru-like implementation techniques. I continue to be amazed with Urs's knowledge of systems and their implementation methods, as well as his relentless work on the optimization of object-oriented systems. I want to thank all the members of the OOCSB group and the distributed systems laboratory for comments and discussions on both computer science and El Nino.

My main influence outside the UCSB walls was the keen-eyed teaching of Dae Soen Sa Nim (Zen Master Seung Sahn) and all the teachers and students of the Kwan Um School of Zen. They helped me to discover what this life is all about, and what is my direction in it. Thanks Ji Bong Soen Sa, Paul Park JDPSN, Jeff Kitzes JDPSN, Soeng Hyang Soen Sa, Jane McLaughlin-Dobisz JDPSN, Morgan Riley, Paul Lynch, Julia Murakami, Bridget Duff, Tim Colohan, Joel Feigin, Mu Sang Sunim, Mu Ryang Sunim, Adam Cherensky, Jacob Newell, Dove Woeltjen, Agne Talmantaite. Thanks brothers and sisters, and may you all attain enlightenment and save all beings from suffering.

My deep gratitude extends to my family and friends who nudged, pushed or simply stayed with me over the years preceding the UCSB and during the studies here. My mother always remained a pillar of strength and understanding. We did not always agree, but we always stayed close and supported each other through the radical changes of life here and in Lithuania. I'll always remember my grandmother who died after I left Lithuania for all her kindness in raising me. My father had the vision to direct me into the discipline of computer science and to encourage me to pursue graduate studies at the UCSB. Dr. Jonas Zmuidzinas helped me with the graduate application process and my first months of life in the USA. My landlady Ingeborg Comstock provided a warm German home in Goleta together with companionship in watching movies and arguing about the meaning of life. I lived and breathed my graduate life together with the branch-prediction genius, poet and photographer extraordinaire, the friend who understood me better than I did myself—Dr. Karel Driesen.

[1] Unless otherwise noted, all quotes at the beginning of chapters are from "The Myth Books" by Robert Asprin [17]

Finally, I want to thank all the people whom I met at my various extracurricular activities: Steve Ota Sensei, Peter Slaughter, Claudia Tyler, Michael Little, Faina Khait, Belinda Braunstein, and all Aikido students; Ken Ota Sensei, Ginger Gelhaus, Fascinating Rhythm Dance Center teachers, and all my ballroom dance partners; Maciej Jesmanowicz—movie buff, programming wizard, and bright Polish soul; Dr. Jan Frodesen and Dr. Janet Kayfetz—two ESL teachers not only perfect in their work, but also in their appreciation of student written fiction and non-fiction; merry folks from SCA and especially lady Isabel D'Triana for all the medieval fun; Douglas Chang for our discussions on growth versus value.

Vita

Personalia

Surname:	Lencevicius
Given names:	Raimondas
Place of Birth:	Kaunas, Lithuania
Date of Birth:	September 7, 1969
Nationality:	Lithuanian
Address:	Department of Computer Science University of California Santa Barbara CA 93106 USA
Telephone:	(1-805) 893-4178
Fax:	(1-805) 893-8553
E-mail:	raimisl@cs.ucsb.edu
URL:	http://www.cs.ucsb.edu/~raimisl

Education

Diploma in Applied Mathematics (Vilnius University, Lithuania, June 1992)
Thesis: “3D graphics system prototype for Microsoft Windows”

Experience

Research Assistant. Department of Computer Science, University of California, Santa Barbara. 1994—Current:
Query-based debugging of object-oriented programs.
Aggregation and design patterns in object-oriented systems.

Teaching Assistant. Department of Computer Science, University of California, Santa Barbara. 1993—Current:
Taught discussion sections in lower and upper division undergraduate courses.

Programmer Aide. Laboratory of Infrared and Submillimeter Astronomy, California Institute of Technology. June 1994–September 1994.
Implemented X Windows graphical interface for custom data acquisition hardware. Cooperated in developing data conversion programs between Sybase SQL database and custom astronomic data.

Programmer. ImPro Ltd., Vilnius, Lithuania. January 1992–August 1993.
Applications development for graphics and futures trading systems. Developed 2D and 3D graphics systems and utilities using OO techniques in a team environment. Implemented futures trading system based on personal research on neural networks.

Computer Science Graduate Student Association Officer, Facilities Committee Representative, September 1997–Current.

Publications

"Query-Based Debugging of Object-Oriented Programs," Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA'97), pp. 304-317, Atlanta, GA, October 1997, Published as SIGPLAN Notices 32(10), October 1997.

"Dynamic Query-Based Debugging," Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. In *Proceedings of the 13th Annual European Conference on Object-Oriented Programming* (ECOOP'99), Lisbon, Portugal, June 1999, Published as Lecture Notes on Computer Science 1628, Springer-Verlag, 1999.

"Fault Tolerance Bounds for Memory Consistency," Jerry James, Raimondas Lencevicius and Ambuj K. Singh. Submitted for publication.

Abstract

Object relationships in modern software systems are becoming increasingly numerous and complex. Program errors due to violations of object relationships are hard to find because of the cause-effect gap between the time when an error occurs and the time when the error becomes apparent to the programmer. Although debugging techniques such as conditional and data breakpoints help to find error causes in simple cases, they fail to effectively bridge the cause-effect gap in many situations. Programmers need new tools that allow them to explore objects in a large system more efficiently and to detect broken object relationships instantaneously.

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.

Dynamic query-based debuggers extend query-based debugging by providing instant error alerts. In other words, they continuously check inter-object relationships while the debugged program is running. To speed up dynamic query evaluation, our debugger (implemented in portable Java) uses a combination of program instrumentation, load-time code generation, query optimization, and incremental reevaluation. Experiments and a query cost model show that selection queries are efficient in most cases, while more costly join queries are practical when query evaluations are infrequent or query domains are small.

We thus demonstrate that query-based debugging is a useful method that can be efficiently implemented and effectively used in program debugging.

Table of Contents

List of Figures

List of Tables

1 Introduction

"There are things on heaven and earth, Horatio, Man was not meant to know."

Hamlet

"Man will never reach his full capacity while chained to the earth. We must take wing and conquer the heavens."

Icarus

When object-oriented programming was introduced in 1966 [48], it was viewed as yet another programming paradigm destined for the cobweb-filled corners of the academy and oddball investment banking companies. Even the wide adoption of some of its principles—classes as modules, inheritance as reuse—was marred with compromises and hybrid implementations in languages like C++. Yet, the underlying elegant structure of object-oriented software—objects encapsulate state and behavior, objects exchange and react to messages, objects inherit their parents' features—is more and more recognized as a useful model for a wide range of applications. Commercial software written in C++, Eiffel, Smalltalk and Java; distributed systems using CORBA, DCOM models; web applets, servlets, and Internet agents—all these programs adopt object-oriented or object-based paradigms. The success of object technology has finally driven it into the mainstream of both computer science and popular computing.

Yet at this moment, when the paradigm of object-orientation has swept over the programming world, the programmers find themselves in a curious situation. The programming languages, environments, and tools have improved to accommodate new OO languages and their implementations. However, program complexity has increased at a much faster rate. Millions of lines of code written and maintained each year are almost impossible to comprehend by the humans in charge of them. Users accept buggy software releases as a norm, and programmers aware of the situation can only blame the herculean task of enforcing all requirements of projects on the underlying complex systems.

While the object-oriented paradigm and its tools have tremendously improved the software development and maintenance process, the work is not yet finished. At the beginning of this research project, we felt that there were opportunities to contribute to the field of object-oriented program debugging, and we believe that this dissertation contains such contribution. It is not the final answer to the daunting problem of debugging, but rather a new approach to debugging and a prototypical tool that implements it.

This thesis examines the problem of verifying object relationships in running programs. Programmers writing a piece of code are aware of various constraints that should be preserved at runtime. During the process of debugging, then, how can the programmers be sure that none of these conditions are violated?

1.1 Problem Statement and Motivation

The goal of this thesis is to allow quick and easy checking of object constraints in object-oriented programs. Programmers debugging or trying to understand such programs should be able to check the correctness of a constraint with a simple question. The method to achieve that has to be powerful and simple. The method's implementation should be fast enough to answer a question in less than a second, or, if the query is checked while the program executes, it should not unacceptably slow down the program.

For example, consider the javac Java compiler, a part of Sun's JDK distribution. During compilation, this compiler builds an abstract syntax tree (AST) of the compiled program. Assume that this AST is corrupted and a FieldExpression object no longer refers to the FieldDefinition object that it should reference. Due to an error, the program may create two FieldDefinition objects such that the FieldExpression object refers to one of them, while other program objects reference the other FieldDefinition object (Figure 1). In other words, javac should maintain a constraint that a FieldExpression object that shares the type and the identifier name with a FieldDefinition object must reference the latter through the field field. What happens if this constraint is violated? The compiler traversing the incorrect AST will perform incorrect transformations leading to buggy output code. But even after discovering the existence of an error, the programmer still has to determine which part of the program originally caused the problem. How can debuggers help programmers to find such errors as soon as they occur?

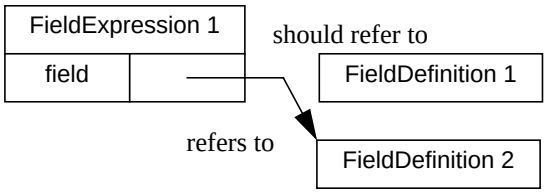

Figure 1. Error in javac AST

This dissertation proposes a technique to solve the problem: query-based debugging, and presents optimized implementations of this technique. Static query-based debugging allows programmers to ask complex questions about interobject relationships using simple queries. A dynamic query-based debugger continually updates the results of queries as the program runs, and can stop the program as soon as the query result changes. In addition, this thesis compares these approaches with existing debugging techniques and shows how queries can be used to debug object-oriented programs and understand runtime object relationships.

1.2 Contributions

The overall contribution of this dissertation is the development of a methodology and the implementation of practical tools for asking questions about object relationships and for checking interobject constraints that exist during the execution of object-oriented programs. The research on query-based debugging contains the following contributions:

- *A new approach to debugging*. Instead of exploring the program 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 of a large number of objects with a single query.

- *A simple yet flexible query model.* The query model extends the simple-to-understand semantics of a programming language expression. Conceptually, a query evaluates its constraint expression for all members of the query's domain variables. The model is simple to understand and to learn, and at the same time it allows a large range of queries to be formulated concisely.

- *A practical interactive query tool.* 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 where the tool takes a long time to produce all results, the first result is often available within a few seconds.

Since static query-based debugging answers users' questions only when the program is stopped, it cannot indicate the exact location in the program where an error happens. To stop the program as soon as the error occurs, we have proposed and implemented dynamic query-based debugging. The research on the dynamic query-based debugging contains the following contributions:

- *An extension of static query-based debugging to include dynamic queries.* Not only does the extended debugger check object relationships, but it also determines exactly when these relationships fail, and it does this while the program is running. This technique closes the cause-effect gap between the error's occurrence and its discovery.

- *Use of dynamic queries for monitoring.* The dynamic debugger helps users to watch changes in object configurations through the program's lifetime. This functionality can be used to better understand program behavior.

- *A practical dynamic query tool.* The implementation of the dynamic query-based debugger has good performance. Selection queries are efficient with less than a factor of two slowdown for most queries measured. We measured field assignment frequencies in the SPECjvm98 suite, and showed that 95% of all fields in these applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation time estimates, our debugger performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. Join queries are practical when domain sizes are small and queried field changes are infrequent.

  Good performance is achieved through a combination of two optimizations: Incremental query evaluation decreases query evaluation overhead by a median factor of 160, greatly expanding the class of dynamic queries that are practical for everyday debugging. Custom code generation for selection queries produces a median speedup of 15, further improving efficiency for commonly occurring selection queries.

In summary, we believe that query-based debugging is a powerful tool to debug large, complex object-oriented programs. Our implementation of the query-based debuggers demonstrates that queries about object relationships can be expressed simply and evaluated efficiently. We expect that the results of this work will provide a foundation for further understanding of program execution and a commercial implementation of advanced debugging tools, simplifying the difficult task of debugging as well as facilitating the development of more robust software systems.

1.3 Overview

This dissertation first presents background and related work in the debugging field in section 2. Then it discusses static query-based debugging, its model and implementation in section 3. Section 4 presents the dynamic query-based debugging method that extends the method proposed in the previous section. The same section gives examples of dynamic query-based debugging, its implementation for Java systems, experimental results, and a query cost model that can predict program slowdown for various queries. Section 5 classifies different query types and their typical use for different programs.

Section 6 outlines open problems and future work. Section 7 presents conclusions indicating that query-based debugging is a novel useful debugging tool.

2 Debugging—Background and Related Work

"Things are not always as they seem."
Mandrake

"No matter what the product or service might be, you can find it somewhere else cheaper!"
E. Scrooge

Debugging of computer programs appeared soon after programming itself. From the beginning of debugging [73] to the present day, researchers and developers have proposed a plethora of tools to find errors in programs. While static program errors can be found automatically through syntactic analysis and semantic checking of language requirements, finding runtime program errors is a much more complicated task. Although runtime errors are directly linked to the program text, they also deal with the different universe of executing instructions, method calls, memory allocations, interobject references, and other relations only implied in the original source code. This duality of the static program text and the dynamic program representation makes runtime debugging a daunting task. A number of runtime debugging methods have been proposed. They can be classified into control flow debugging, data observation, and mixed methods. To better describe the capabilities of different methods, we use the following case studies.

For the first example, consider a graphical user interface program. The program creates objects corresponding to the graphical widgets that reference their parent window, and windows that must in turn reference enclosed widgets. Assume that the program contains an error which makes some windows miss references to their children widgets (Figure 2). As a result of this

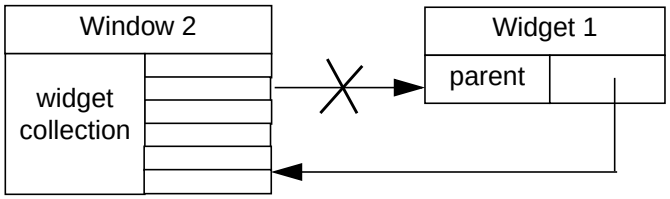

Figure 2. Error in GUI program

error, a program incorrectly redraws widgets contained in a window. Assuming that a program is stopped at a breakpoint, how can a programmer find windows and widgets violating the relationship?

For the second example, consider another error that could occur in an AST built by the javac Java compiler. Assume that this AST is corrupted by an operation that assigns the same expression node to the field right of two different parent nodes (Figure 3) that may be instances

of any subclass of BinaryExpression. The error may not become apparent for some time, and the

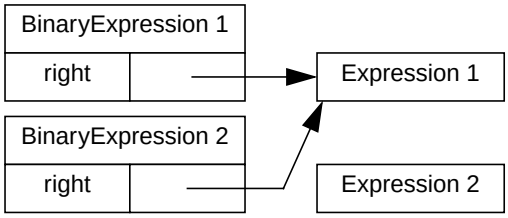

Figure 3. Error in javac AST

compiler may traverse the corrupted AST performing type checks and inlining transformations. Even after discovering the existence of the error, the programmer still has to determine which part of the program originally caused the problem.

The two examples above illustrate errors that can occur in object relationships. The following subsections discuss control flow debugging, data-flow debugging, and debugging related techniques in program visualization in relation to these examples. We show that these techniques do not adequately address the problem of finding errors that involve several related objects.

2.1 Control Flow Debugging

Errors in programs can be caused by an incorrect control flow or an incorrect data flow. This section discusses the first aspect of the program runtime—the control flow. The control flow follows the program text, which may hide errors such as infinite loops, unintended method invocations, or faulty object interactions. Control flow debugging tools help programmers to observe and to manipulate the control flow of the program. This section describes such tools in detail.

2.1.1 Breakpoints and Single Stepping

In the simplest case, programmers want to stop the program when it reaches a certain point or to single step through some of the instructions of a running program. The goal of such actions is to determine what code the program executes before crashing and to provide a foundation for the data observation tools (section 2.2). Breakpoints and single stepping are the original debugging methods that predate even the first paper on debugging by Gill [73] and that are available in most classical (Mesa [165], Cedar [166]) and modern debuggers. Single stepping back in time is confined to a few innovative tools like ZStep 95 [176]. Wilson and Moher [193] propose even more radical concept of a *demonic memory* that would allow programmers to go back to previous process states. Implementing breakpoints for polymorphic calls in object-oriented programming languages is also a less common, though very useful technique [46]. As Gill notes, single stepping is practical only in very limited cases when programmers want to look at a micro-segment of a program, because the slowdown during execution is enormous.

In the case of the GUI program error, a programmer would stop the program at a breakpoint before trying to find the corrupt window and widget objects. To find the javac error using breakpoints, the programmer would have to know which part of code assigns the incorrect link, and only then place a breakpoint close to the error spot, and single step through it using data observation tools. Such debugging is very tedious. To add power to breakpoints and to improve the efficiency of debugging, researchers have proposed conditional breakpoints.

2.1.2 Conditional Breakpoints

Conditional breakpoints [36][109] check a condition at a particular program location and stop the program if this condition is true. The goal of this technique is to let the program run at full speed between the breakpoints, to slow down at breakpoints, and to stop only at interesting breakpoints. The programmer saves a lot of time, because the debugger checks the error condition at each breakpoint automatically. Programmers have to interfere only if the condition is true. Research on conditional breakpoints has led to efficient implementations of breakpoints and conditional breakpoints on modern architectures [109]. However, conditional breakpoints suffer from a drawback—the breakpoint condition cannot easily reference objects which are not reachable from the scope containing the breakpoint. To discover the javac error, the condition has to find an object not reachable directly from the scope containing the breakpoint—the BinaryExpression containing a duplicate reference to the child Expression object. To accomplish this task, the programmer could write custom testing code for use by conditional breakpoints. For example, the javac compiler could keep a list of all BinaryExpression objects and include methods that iterate over the list and check the correctness of the AST. However, writing such code is tedious, and the testing code may be used only once, so the effort of writing it is not easily recaptured. Finally, even with the test code at hand, the programmer still has to find all assignments to the field right and place a breakpoint there; in javac, there are dozens of such statements. In summary, the tool (conditional breakpoints) provides minimal support and the programmer ends up doing all the work “by hand”.

2.1.3 Language Constructs

Some programming languages provide debugging support by allowing assertions [132][133]. Assertions, such as pre-/postconditions and class invariants as provided in Eiffel [134] are similar to conditional breakpoints because they check a given constraint and stop a program or throw an exception if this condition is violated. However, assertions are checked only at specific program execution states. Preconditions are checked at the method entry, postconditions are checked at the method exit, and invariants are checked both at the entry and at the exit. In addition to accessing the object state, postconditions can reference the “old” state of the object at the beginning of the method invocation.

Like conditional breakpoints, the assertions cannot access objects unreachable by references from the checked class.

Some debugging tools such as MrSpidey [62] use static program information to check the assertions before the program is run. MrSpidey provides conservative static assertion analysis indicating potential errors.

Constraint programming languages [90] make assertions first-class objects and structure programs around inter-object constraints. These programming languages change the programmer's viewpoint from objects to constraints. Though constraint programming languages are powerful tools, they do not help to debug programs written in mainstream object-oriented languages.

2.1.4 Breakpoints and Testing Code

To generalize the idea of conditional breakpoints, tools could allow programmers to insert any code at a breakpoint. The implementation of this idea was pioneered by the EDSAC routines [73] and included in other programming environments (Mesa [165], Cedar [166]). Though this approach offers the ultimate versatility, it also shifts all the work onto the programmer's shoulders. The programmer has to gather the data to be printed or displayed by the testing code. Writing test code may consume a considerable amount of time. For example, the Self virtual machine [89] 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 may be inefficient, especially if the conditions it is testing 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. The javac compiler could keep a list of all BinaryExpression objects and include methods that iterate over the list and check the correctness of the AST. With large programs containing thousands of objects, naive testing code may take minutes to execute. Even with the test code at hand, the programmer still has to find all assignments to the field right and place a breakpoint there; in javac, there are dozens of such statements.

2.1.5 Method Call Animation

While breakpoints and testing code provide maximum versatility, there are tools that offer limited control flow debugging with increased ease of use and higher efficiency. One such group of tools deals with the method or function call animation. This group of tools is very important for program debugging because the control flow of the object-oriented programs is difficult to follow due to polymorphic calls. While some tools only display the function call stack, others provide additional functionality. For example, Ovation [50] displays clusters and matrices of interacting classes, histograms of class instances and their method invocation activity. Jinsight [52] extension to Ovation displays the program execution as a modified Jacobson diagram that shows time on vertical axis and message invocations on horizontal axis. Patterns of similar message sequences are compressed into more user-friendly form using several similarity

criteria. The system also allows filtering and searching through the graph. Program Explorer [117][118][119][120] shows method invocations between individual object instances and between objects grouped by classes. It also provides an interaction chart that juxtaposes object lifetime with object interactions. HotWire [116] displays the basic call stack while allowing custom object visualizations. PV [111] extracts events of different levels from program traces and animates them. As a result, programmers can observe object allocation patterns together with the communication library behavior and the operating system activity. Xab [22] intercepts and shows PVM calls in parallel programs.

Jerding, Stasko and others [101][102][155] proposed a number of techniques to visualize messages in object-oriented programs. The first one displays object creation and invocation messages [101]. To use visualization for large real-world systems, the information about a system can be summarized in different ways. Authors propose call graphs, summarized call traces and other representations that carry more information than a call graph, but do not have the information overload of a call trace. Call traces can be summarized in execution murals. Higher level information is obtained by extracting message patterns from these traces.

Consens et al. [44][45] use Hy^+ visualization system to find errors by querying distributed program event sequences. The GraphLog programming language in the Hy^+ system allows to visually specify abstract event groups and find such groups in the event traces. The language is powerful enough to find patterns involving transitive closure.

Kishon et al. [112] propose a formal framework to implement execution monitors in programming environments. Authors use functionals to add monitoring behavior to the programs, and integrate the monitoring semantics by merging it with program's denotational semantics. A system using the paper's theoretical results was implemented in Haskell.

Method call animation and its extensions help to understand object-oriented program execution by displaying the non-trivial control flow of these programs. These methods do not directly solve the GUI and javac errors, but would indicate the control flow anomalies leading to an error.

2.2 Data Observation

The previous section discussed approaches of detecting and changing a program's control flow to discover errors. The second important method of finding errors is data observation. Corrupted memory locations, incorrect values, and faulty object references are a large source of errors. While control flow debugging tries to answer the question *when* something went wrong, data observation tools try to find *what* made the control flow take a wrong path and *what* incorrect data contributed to the incorrect results. This section describes various data observation tools that are widely used in debugging.

2.2.1 Memory Inspection

The simplest tool available for data observation allows users to inspect memory locations and to see their contents in an intelligible way. I.e., the tools make it possible to display memory contents as values of variables or even as structured objects [46]. While most debuggers contain inspector tools, some of them are more powerful than others (DDD[196], Look! [14][143][188]). For example, the ET++ [25][69][183] runtime browser displays a list of all classes, all objects of a selected class and all objects that reference the current object. The last capability is particularly interesting because the system has to track which objects are referencing the current object, which is a costly operation. The ET++ object graph can also show the container objects that store other object instances.

ET++ display capabilities build on the foundation established by Smalltalk implementations that had similar features [76][77]. To allow intelligent display, manipulation, and inspection of objects, Self [41][129][151] and Smalltalk [75] environments adopt the object-focused interaction model. Such environments not only display structured object instances, but also allow users to interactively change them. Similar spatial immediacy is provided by ZStep 95 [176] debugging environment. An implementation of a *demonic memory* [193] would allow users to access and observe past states of a process.

Memory inspection tools can help to find the GUI and javac errors. However, programmers have to do a lot of work to manually look at all GUI window objects, widget objects, and references between them. Similarly, users have to traverse all the javac AST nodes to check whether they are correct. Furthermore, such inspection has to be done every time the program modifies the AST. Consequently, such investigations are very tedious. Data structure display tools described in the next subsection improve the process of data browsing.

2.2.2 Data Structure Display Tools

A straightforward extension of a memory inspection technique is a tool that allows a full-fledged data structure display. Usually the tool displays data structures by using the user specified code. For example, Duel [74] and xDuel [195] display data structures by using user script code. In addition, Duel provides operators for filtering irrelevant collection members, for iteration over collections, and for traversal of data structures. Duel expressions can be intermixed with C expressions accepted by the underlying gdb debugger. Although Duel is not implemented for an object-oriented language, it could be extended to handle object collections. Although these systems simplify data structure display, they still require users to write the traversal and output code. This code can be inefficient for large structures. Duel does not address the problem of finding the execution points at which the structure should be displayed.

2.2.3 Data Filtering and Summary Tools

In large object-oriented programs, the number of objects is so large that programmers would not be able to browse through all instances in a reasonable time. Debugging tools help with the task by filtering and summarizing the data. Ovation [50] includes a compact instance histogram view that shows all instances for all classes and an allocation matrix view that shows different instances created by different classes.

Hotwire [116] shows object instances, class instance counts, and method invocation counts on specific instances. If the method invocation count is zero, the object is not used and probably created erroneously. Similar views are provided in the Java LTK debugger [113].

Look! [14][143] provides instance filtering tools that allow to remove individual objects and classes of objects from a view. The filters are coupled with method-call animation and instance-creation animation tools.

Instance filtering and summary tools do not help to find errors like the GUI and javac errors, because these errors cannot be detected from the summary information nor by creating a simple filter. However, these tools are first steps in the direction of debuggers that mix control flow and data-flow debugging.

2.3 Mixed Constructs

Though some of the tools discussed above contain both control flow and data-flow debugging constructs, most of the time these aspects are separated or interact only occasionally. However, a number of tools integrate both types of debugging to provide additional capabilities.

2.3.1 Data Breakpoints

Data breakpoints combine breakpoints with data monitoring. A breakpoint is triggered when a variable connected to the breakpoint is assigned [106][179][180]. Unconditional data breakpoints can be used to detect unexpected writes to variables. However, just detecting writes may give user too much unnecessary feedback. Errors may occur only when the variable is assigned a certain value. Conditional data breakpoints stop the program only when the variable is assigned a given value. Both variations of data breakpoints allow users to find incorrect value assignments to variables using a single test.

However, even conditional data breakpoints do not help to debug the javac error described in the beginning of this chapter because they are specific to one instance of an object. With hundreds or even thousands of BinaryExpression instances, and in the presence of asynchronous events and garbage collection, the effectiveness of data breakpoints is greatly diminished. In addition, it is hard to express this type of error as a simple boolean expression. The error occurs only if an expression is shared by another parent node—a relationship difficult to observe from the other parent or from the child itself. In other words, by looking just at the field right of some

BinaryExpression object we cannot determine whether this object as well as its new field value are erroneous.

2.3.2 Program Slicing

Often programmers need to decide what operations affect the current program statement. Program slicing [184][185][186] finds such a subset of program statements—a *slice*—that affects the value of a certain variable or the current statement of the program. To find the slice, static slicing uses information inferred from the program text. Static analysis is conservative and uses all possible control flows to find statements affecting the current one. Such a slice may be large, but it is correct for all possible executions of a program. Dynamic slicing [9][104][169] uses program execution history to determine the slice. Such analysis finds a slice accurate only for the current program execution, but the slice may be smaller. In both cases, the analysis combines control flow and data-flow debugging.

Slicing is helpful to find the cause of a detected error. Programmers can investigate the slice of a program affecting the invalid statement and reason about the causes of the error. However, the slice may still contain a large part of the program and make it difficult to identify statements that were original causes of an error. In the case of the javac bug, if the error is discovered late in the program execution, the statement containing the original error may be hard to identify.

Bourdoncle [27] uses a technique similar to slicing to find correctness conditions for Pascal programs. His abstract debugging system uses programmer provided assertions and abstract interpretation of the program to determine how assignments of certain values affect subsequent program statements.

2.4 Program Visualization Systems

Software visualization systems such as BALSA [31], Zeus [32], TANGO/XTANGO/POLKA [159], Pavane [47][147][148], and others [84][137][144][146] offer high-level views of algorithms and associated data structures. They cannot be regarded as pure debugging tools because software visualization systems have different uses than everyday debugging. They aim to explain or illustrate the algorithm [61], so their view creation process emphasizes vivid representation. Consequently the view creation requires substantial user effort. There have been several attempts to use the visualization systems for debugging, some of them using the conventional visualization systems [18], some of them adapting the systems to the new field. For example, Hotwire [116] provides program visualization capabilities through a constraint language. The system separates visualization script from the program allowing succinct visualizations that do not change the program source and can be used in quick debugging sessions.

Hart et al. [82][83] use Pavane for query-based visualization of distributed programs. However, their system displays only selected attributes of different processes and does not allow more

complicated queries. The work done by this group focuses on gathering consistent information from distributed sources in an efficient manner. Similar issues arise in implementing distributed query-based debugging as discussed in section 6.2.2.

Takahashi et al. [167] visualize, animate, and directly modify abstract data structures. Users of their system specify a mapping rule or allow the system to infer it from examples. The interesting feature of their system is the reverse mapping of pictures into data structures which gives users the power of direct manipulation.

Noble [140][141] observes that abstract data structures can be viewed at three levels: abstraction level, implementation level, and contents implementation level. Program visualization systems usually operate at the abstraction level and do not display any details of the ADT's implementation. On the other hand, debuggers display the low level objects (contents) composing the ADT. Noble's Tarraingim program visualization system implemented in Self presents the middle level implementation view. Such view helps programmers to understand and debug ADTs. However, the implementation view is difficult to automatically identify and maintain. It requires ADTs to be implemented in a single class with clearly identified mutator and accessor methods. Though this assumption follows object-oriented design guidelines, programmers may want to observe more general object interaction patterns not supported by the system. For example, the javac AST construction does not follow the strict mutator/accessor requirements of the Tarraingim system.

Though most visualization systems have constructs powerful enough to discover the javac bug, these constructs are as difficult to use as writing custom test code. On the other hand, the systems that facilitate rapid visualization development do so for a limited class of visualizations that do not help in our case.

2.5 Summary

More than fifty years of debugging research has produced a wide variety of control flow and data-flow debugging tools. However, these tools are ill-suited to find the errors involving multi-object relationship violations. Even the best debugging techniques force programmers to adopt either a very low level view of single objects and their properties, or an extremely macroscopic view of class histograms and statistical data. The tools lack capabilities to find small groups of objects satisfying specific constraints.

3 Static Query-Based Debugging

"In times of crisis, it is of utmost importance not to lose one's head."

M. Antoinette

"Attention to detail is the watchword for gleaning information from an unsuspecting witness."

Insp.Clouseau

3.1 Introduction

To overcome the problems described in chapter 2, we propose a new debugging technique: query-based debugging [123]. This new approach offers programmers an effective query tool that allows complex relationships to be formulated easily and evaluated efficiently. Static queries can be asked whenever a program is stopped at a breakpoint. Queries can be formulated during debugging sessions, or stored in a debugging library together with the program's normal code. A query-based debugger can easily find errors in the examples presented in the beginning of chapter 2. For example, 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
```

(The current implementation of the static query-based debugger is based on Self, so query expressions use Self syntax. On the other hand, the *dynamic* query-based debugger described in section 4 is implemented for debugging Java and uses Java expression syntax [16].) This query identifies all objects that violate the containment constraint. Not only does this query save debugging time, the query evaluator can apply 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.

To discover the javac bug, we could combine conditional breakpoints with a static query-based debugger. For example, the query

```
BinaryExpression* e1, e2.     e1.right == e2.right && e1 != e2
```

would find the objects involved in the above javac error. The breakpoints would then carry the condition that the above query return a non-empty result.

The rest of this chapter discusses the debugger query model, the system implementation, and experimental results.

3.2 Query Model

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 [177], 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. In fact, we use Java expression syntax in the dynamic query-based debugger (section 4).

The query syntax is as follows:

```
<Query>              ::=<DomainDeclaration> { ; <DomainDeclaration> } .
                        <ConditionalExpression>
<DomainDeclaration>  ::=<ClassName> [*]
                        <DomainVariableName> { <DomainVariableName> }
```

The query has two parts: one or more DomainDeclarations that declare variables of class ClassName, and a ConditionalExpression. The first part is called the *domain part* and the second the *constraint part*. Consider 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 database *active domain* semantics ([1], section 4.2). The domain part of the above example should be read as “find all widgets wid and all windows win in a system such that...”. With Self being a prototype-based language, widget is the name of the widget prototype, and its domain are all objects that contain the same fields as this prototype;[1] 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 may 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).

The general form of a query is

$X_1\ x_{11} \ldots x_{1\,n1}; \ldots ; X_m\ x_{m1} \ldots x_{m\,nm}.$
$Constraint_1$ && ... && $Constraint_k$

[1] See section 3.3 for a more precise definition.

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 [35] (or, in classless Self [174], subobject). However, if a "*" symbol in a domain declaration follows the prototype name, the domain includes all objects 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 3.3. 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.

We refer to queries with a single domain variable as *selection queries*; following common database terminology, we call the rest of the queries *join queries* because they involve a join (Cartesian product) of two or more domain variables. Join queries with equality constraints only (e.g., p1.x = p2.x) are *hash joins* because they can be evaluated more efficiently using a hash table (see section 3.3.6).

3.2.1 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. It would be possible (at least in Java) to perform a conservative test whether methods invoked in the query are side-effect free. However, such tests become increasingly difficult for polymorphic method chains. Our system follows the model of assertions in C/C++ and Eiffel that require users to ensure that the methods are side-effect free.

Second, the programmer must ensure that the queried objects are in a consistent state when the query is evaluated. In other words, the expression evaluation should succeed and provide meaningful results. At some program execution points the query evaluation may be unsafe[1]. For example, during an insertion of an element into the list, the list may have an inconsistent state. In Figure 4, if the query is asked just after the program updates the next reference of the OldNode but before it sets the next reference of the NewNode to point to the TailNode, the query evaluation will be unsafe. If the debugger traverses a list, it may crash because the reference NewNode.next is null or it may produce an incorrect output by using a shorter list that does not contain the TailNode. For static queries, we assume that the user is aware of this problem and

[1] The term "unsafe" here follows the programming language terminology and not the database terminology, where safety relates to queries with infinite answers ([1], section 5.3).

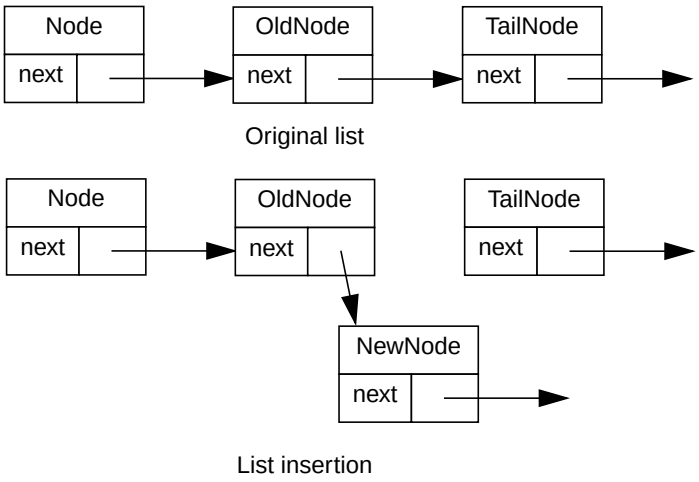

Figure 4. Inconsistent list state

only asks queries when queried objects are consistent. The problem of consistency for dynamic queries is discussed in section 6.2.1.

Finally, to use efficient hash joins (see section 3.3.6), we assume that the "=" method in queries has equality semantics, i.e., that the "=" method has the same meaning and is as strict as equality of the system-defined hash value of the left-hand operand and the hash value of the right-hand operand. If the hash value equality is stricter than the "equals" method or if the two are incomparable, the hash joins would give incorrect results. For example, all methods can be redefined in Self, so the "=" method can violate the constraint above. However, the Self programming guidelines strongly suggest preservation of the equality semantics, and the current Self system contains no method that violates it[1]. On the other hand, both *shallow* and *deep* equalities [110] are supported in the hash joins depending on which equality is used in the "=" method; the hash value equality in a hash join is verified by using the "=" method equality.

3.2.2 Discussion

We chose the given 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.

One possible shortcoming of this query model lies in the nature of program invariants spanning a number of objects and classes. Programmer implied invariants usually have a mixture of universal and existential quantification—"For all widgets, there exists a parent window and this window refers to the widget in its widget collection." Such complex constraints can be violated

[1] For environments where this assumption is not valid, section 3.4.5 shows the effect of not using hash joins.

in a variety of ways. First, the widget may not reference a window at all. Second, the widget may reference a window that does not reference this widget. Finally, the window may reference a widget that does not reference this window. Can a single query find all violations of a given constraint? Unfortunately, that is not the case with the current query model.

Consider a simple constraint that requires an element to belong to a single collection:

```
Collection c; Element e. ∀e ∃c: c contains e
```

Here the constraint is expressed in the notation of mathematical logic and can be written in English as: “For all elements there exists a collection that contains that element.” How can a program violate this constraint? The mathematical logic expression of such constraint violation is:

```
Collection c; Element e. ∃e ∀c: ! (c contains e)
```

Unfortunately, in this case, there is no simple way to express the violation condition as a query. A query

```
Collection c; Element e. ! (c contains e)
```

would give as an answer all tuples such that a given collection does not contain the given element. However, that is not the intention behind the query.

One approach is to define a method in the class Element that iterates through all collections and checks whether the element instance is contained in them. Then the query would be an expression:

```
Element e. ! (e isContainedBySomeCollection)
```

Unfortunately, this solution involves writing testing code which is tedious, inefficient and inelegant. Currently we offer no solution to this problem. Considering that a large class of queries can be expressed in the current model, some queries can be rewritten to conform to the restrictions of the model. If users of a query-based debugger formulate queries by considering some constraint violation instead of formulating the constraint, their queries would likely conform to the model. In some cases, it is possible to consider the constraint and then to check different ways in which it can be compromised. Consequently, finding the violation of an invariant may require asking a few queries to check all possible violations of the constraint.

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 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 a 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.

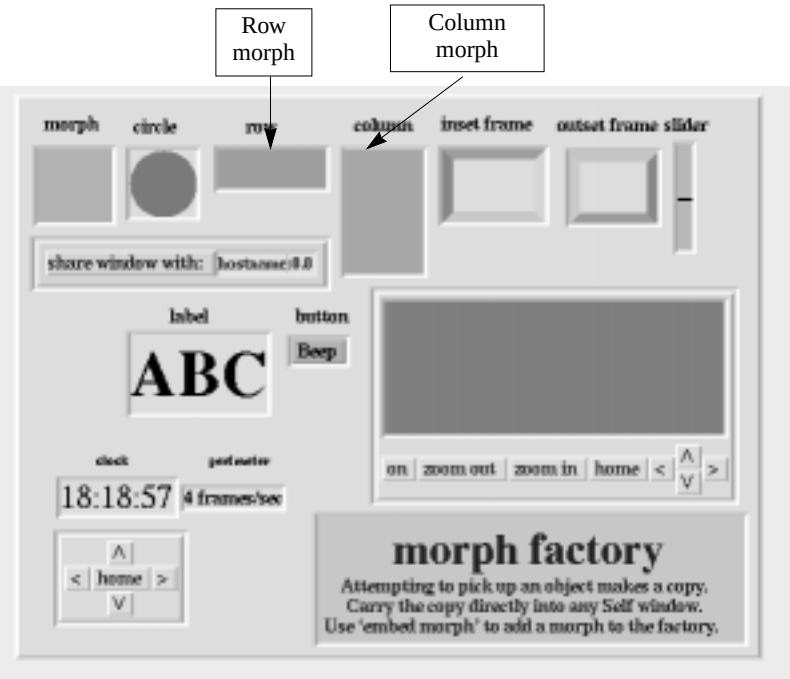

Figure 5. Self morphs

Object-oriented databases typically contain more sophisticated query models [2][19][139][154] including such standard models as OQL from the Object Database Standard [37]. Our solution was specifically designed for query-based debugging, and we deliberately traded off expressive power in exchange for simplicity and ease of use because we deemed these attributes to be of primary importance.

3.2.3 Examples

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

3.2.3.1 The Self Graphical User Interface

We implemented our debugger prototype using the Self user interface based on morphs [129][156]. Morphs are user interface objects (usually having a visual representation) that can perform specified actions and can be moved on the desktop (Figure 5). 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. Note that the last conjunct in the constraint indicates that morph objects a and c should not be identical. This constraint is not enforced implicitly and has to be explicitly specified.

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 (objectOutliner-columnMorph-rowMorph tuples), 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.

3.2.3.2 Understanding the Cecil Compiler

Because our debugger helped us to understand Self GUI objects, we decided to look at a complex system with which we had no previous experience—a prototype Cecil [38] 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 not empty, confirming the hypothesis 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 object-oriented systems.

3.3 Implementation

We implemented the static query-based debugger in Self [177], 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 has a fast Virtual Machine that allows runtime (on-the-fly) code generation and optimization [5][39].

We have also implemented the static query-based debugger in Java and extended it to handle dynamic queries. For a discussion on different implementation techniques and their applicability to various languages, see section 4.3.7.

The query-based debugger's front-end in Self is constructed from morphs (Figure 6). Users can ask questions by typing in a query string or by selecting a query from the history of previously asked queries. The answer is displayed as a collection of tuples that provides access to the

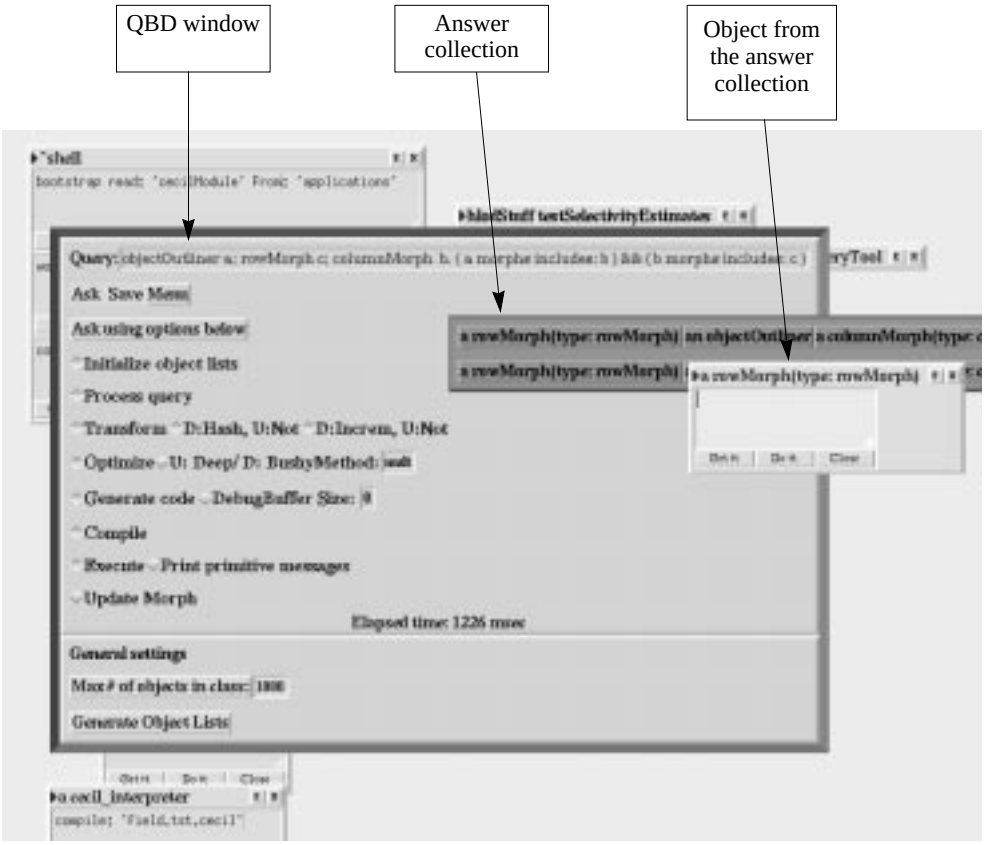

Figure 6. Query-based debugger GUI

corresponding objects via the direct-manipulation interface [41]. For example, in Figure 6, the user has displayed one of the rowMorph objects from the answer set.

The interface for the prototype implementation allows the experimenter to select what parts of query processing to do and what optimizations to apply. It also displays the query evaluation time.

In section 3.3.1, we present the general structure of the system. In the remainder of section 3.3, we discuss in more detail the most important parts of the debugger.

3.3.1 General Structure of the System

Figure 7 shows a data-flow diagram of the debugger. A query string given by the user is parsed by a simple Self expression parser hand implemented in Self. The 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 3.3.2) and returns them in separate arrays to the execution module. The parser also transforms the query string into an intermediate form, which

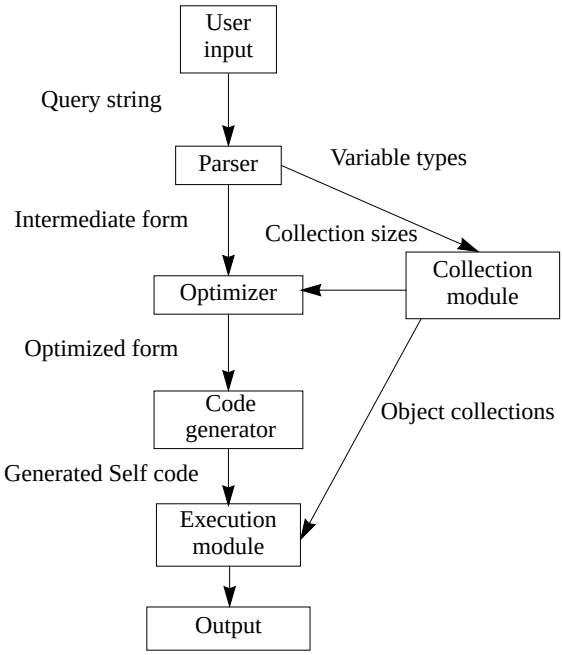

Figure 7. Overview of the query-based debugger

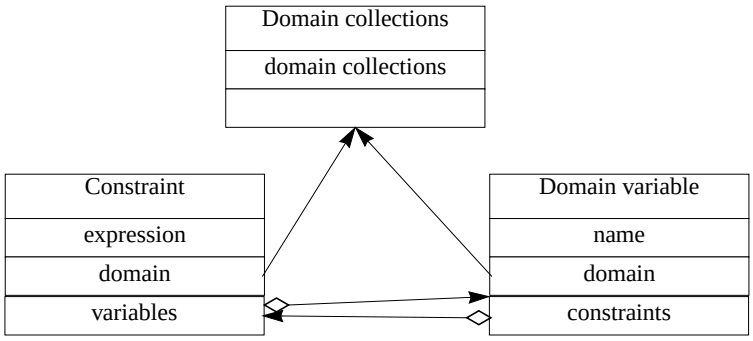

Figure 9. Data structures of the intermediate form of a query

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 3.3.4). 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 runtime environment for the query evaluation, runs the query code, and finally sends the result back to the user. The query evaluation process is summarized in the Figure 8 pseudo-code.

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

// Parsing into the intermediate form
Extract domains from the query string. For each domain create a domain collection.
Extract variables from the query string.
For each variable create a variable object. Store a name, a reference to the variable's domain, and a collection of references to constraints containing the variable.
Extract constraints from the query string.
For each constraint create a constraint object, store the constraint, the code for its evaluation, and collection of references to its variables (Figure 9).

Create domain collections of the query domains by invoking the domain collector primitive. Handle subtypes of the domain type if necessary (section 3.3.2).

// Intermediate form transformation

Go through the constraint list {
    if (equality constraint) {
        Use the left-hand side of the equality to construct a hash table insertion function.
        Use the right-hand side of the equality to construct a hash table lookup function.
        Assign a hash-join evaluation method to the constraint.
        Set left-hand side and right-hand side domains of the hash-join constraint.
    }
    else { Set up the constraint as a nested query }

    Set up the relationship between the constraints and variables in them.
}

// Optimization

Estimate selectivities of all constraints by evaluating constraints for a random sample of domain objects. (Select a small sample size. Evaluate the sample only if all domains are larger than the sample. For small queries set selectivity to high. For no-result highly selective queries, set the selectivity to high. For other selective queries set selectivity to medium. Otherwise, set selectivity to low.)

// Build left-deep tree of joins

Select the first join using maximum selectivity, minimum size heuristic. Assign it index 1. Move it from the unprocessed join table to the processed join table.
While (unprocessed join constraint table non-empty) {
    Find all joins that have at least one domain in common with joins in the processed join table. Find a join with maximum selectivity, minimum size among selected joins. Move this join to the processed join table. Assign it the incremented index.
} // Resulting indices give the join evaluation order.

// Generate code

For each of the join constraints generate its evaluation code. This code will iterate through the input collection and the intermediate result collection to evaluate the current constraint and to produce the tuple collection satisfying this constraint.

// Evaluate code

If (evaluation incremental) { Use pipelined and time-sliced evaluation (section 3.3.7). }
else { Evaluate joins in a straightforward inverted tree way (section 3.3.3). }

Figure 8. Query evaluation pseudo-code

3.3.2 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* [39]), 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.

To find all objects of a type and all its subtypes, the system has to know the subtype tree. This tree in Self is constructed as follows:

```
Iterate through all global objects.
Create two hash tables of all global objects containing object-parent pairs. Object table is
indexed by prototype objects, parent table is indexed by parents.
For each object o in the object table, get all its ancestors.
    For each ancestor, look up the object-parent pair using the parent hash table.
    Put the object o into the pair's subtype collection.
```

This algorithm produces a table indexed by prototype objects. Each element of this table lists prototype objects of all subtypes of a type. This subtype list is, however, imprecise because of Self's support of multiple inheritance and convention-only placement and naming of "class" objects [175]. Multiple inheritance gives additional subtypes of a given type. Objects and "classes" invisible at the base level of global hierarchy will not be processed. Algorithm would need to be extended to handle these cases.

Enumerating all objects of a domain can be achieved in different ways in different systems. The implementation of the dynamic query-based debugger in Java uses two approaches. First approach changes the Java Virtual Machine to provide an additional primitive returning all instances of a certain class. This approach does not work if a Java Virtual Machine cannot be changed. Second approach adds instrumentation to Java class files to call the debugger each time an object of a monitored class is created. In this case, the debugger itself keeps track of existing domain objects (section 4.3.4).

Some programming languages and systems provide a direct way to list all instances of a class. For example, Smalltalk [75][77] provides allInstances: method. In other cases, like C++, tracking objects of a class would be a difficult endeavor unless the debugger instrumented the source code, or tracked only the framework classes [69][183].

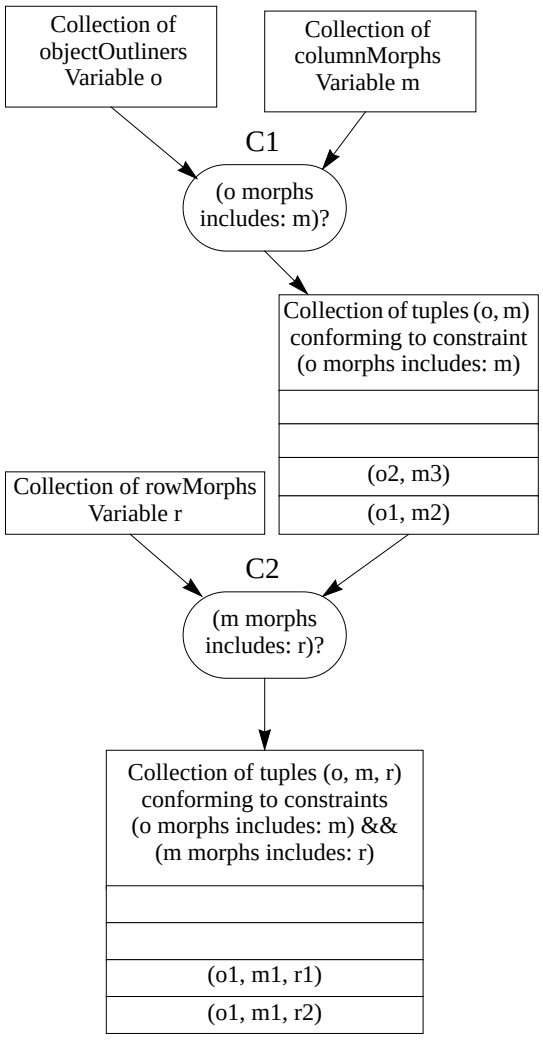

Figure 10. Overview of query execution

3.3.3 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 10 shows the query execution of the query

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

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 10, 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 take longer to process than others. In fact, for some queries the difference is more than an order of magnitute—a bad join ordering may increase the evaluation time of a query from 2 seconds to 10 minutes (see section 3.4.3). Avoiding bad orders and finding good ones is necessary for an acceptable tool performance. The next section investigates the problem of join ordering.

3.3.4 Join Ordering

The execution order of the individual joins of a query significantly influences overall performance. To see why, 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[1].) 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 a 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 [93]. Several algorithms produce optimal or near-optimal orders for restricted cases. For example, the KBZ algorithm [113] computes optimal join orders for left-deep ordering where the join graph is acyclic (for cyclic graphs the algorithm gives approximate results) assuming perfect knowledge of join sizes and selectivities.

Numerous heuristics try to find near-optimal orderings [93][113][160][164]. 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

[1] 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".

cheap joins in the beginning should shrink the intermediate result size, making the expensive (large) joins at the end less expensive to perform.

In addition to using the minimum-cost heuristic, we impose a restriction on the join order by using *left-deep ordering*. Left-deep ordering [153][160] requires each join to have one and only one intermediate result relation as an input (Figure 11). 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 [95][160]. It also simplifies incremental result delivery as explained further below.

3.3.5 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 [113], 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 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).

The query-based debugger knows the precise input sizes of joins because all domains are enumerated before determining the query evaluation order. However, the debugger does not know the selectivities of the joins. To accurately estimate selectivity, the query optimizer would have to sample a large number of tuples [79], a process that 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, this information still allows to distinguish highly-selective constraints (selectivities larger than 0.80) from the constraints that have low selectivities (less than 0.50).

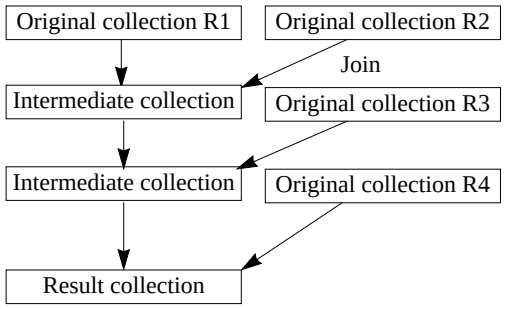

Figure 11. Left-deep join

We decrease the priority of the joins containing low selectivity constraints by multiplying their cost by a factor of 100.

How large a sample should the system use 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.

The dynamic query implementation uses a different join ordering heuristic. Because sizes of domains change during program runtime, and we cannot efficiently determine the selectivities of constraints for changing domain sizes, we simplify the heuristic for join ordering: the systems executes selections first, equality joins next, and inequality constraints last (section 4.3.5.1).

3.3.6 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 a *hash-join* method instead of using a less efficient nested-loop method [53][135]. 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)
```

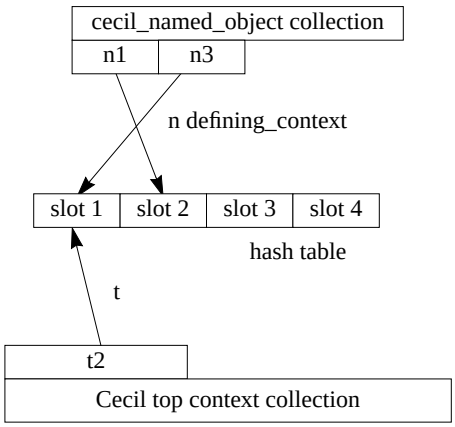

Figure 12. Hash join

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 that maps all result values of expression α to the domain variable(s) that produced these results. In Figure 12, which 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 domain on the right-hand side of the equality 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 α = β. In Figure 12, the cecil_top_context object t2 is mapped to the same slot as the cecil_named_object n3, and t2 = n3 defining_context. Consequently, the tuple (n3, t2) satisfies the constraint and belongs to the result. Using the hash join method, the join result can be 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 3.4.5 demonstrate that hash joins can improve performance for some queries.

In the debugger implementation of a join chain, each join may be a nested-loop join, a hash join, or a selection. Since the join type is decided dynamically and depends on its position in the

chain, the constraint objects in a join chain are assigned the join type through their joinType field. This implementation decision uses a Strategy design pattern [67][68].

3.3.7 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. The intermediate result buffer is implemented using a circular buffer. Figure 13 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 constraint. As in the classical producer-consumer problem, a thread in the pipeline blocks when its output 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 13 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 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 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

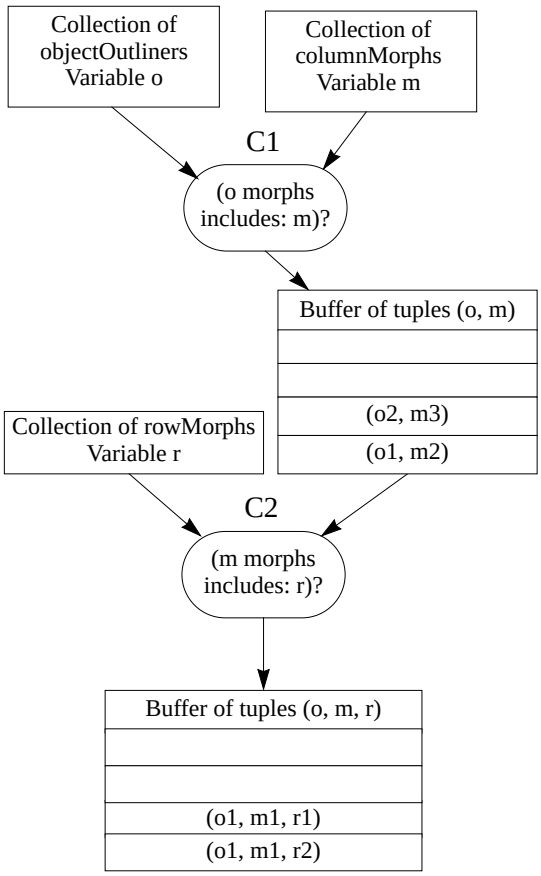

Figure 13. Incremental delivery pipeline

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).

Time-slicing and pipelining work best in concert. Using just time-slicing without thread priorities would lead to inefficient thread scheduling. Omitting limited size buffers would result in additional memory overhead when intermediate results are large.

3.3.8 Related Work

The query optimizations discussed in section 3.3.4 were influenced by research in the area of databases. Ullman [170][171][172] discusses basic insights about efficient join evaluation. De Witt et al. [53] and Lehman and Carey [121] indicate that the hash-join algorithm is efficient in main memory databases, a fact that we used in our implementation. Debugger hash joins are different from the database hash joins because the left-hand side and the right-hand side of equality constraints have to be evaluated for hash table indexing. Regardless of this difference, the cost formula is preserved, and hash joins are efficient for debugging queries. For inequality

joins, an efficient sort-merge join [121] could be used, but it has not been implemented in our system. Numerous researchers [40][93][96][97][113][136][160][164][194] have developed algorithms to find optimal or near-optimal join orderings. Ibaraki and Kameda [93] proved that join ordering problem is NP-complete by reducing CLIQUE [70] to it. The authors also proposed an optimal left-deep join ordering algorithm for acyclic join graphs (for cyclic graphs the algorithm gives approximate results). The KBZ algorithm [113] improves on Ibaraki and Kameda's algorithm. Steinbrunn, Moerkotte and Kemper [42][160] investigate and compare a number of heuristic and randomized algorithms. Swami and others [162][163][164] investigate and propose heuristic and randomized algorithms improving on the KBZ algorithm. Current databases use the exponential exhaustive search of all possible join orders when the number of joins in a query is small. Query-based debuggers also may be able to use this approach.

Unfortunately, the proposed algorithms assume perfect knowledge of join sizes and selectivities. Also, the characteristics of a 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. On the other hand, debugger queries may contain calls to expensive methods. Due to the uncertainty of selectivities and single join costs, the debugger uses simple heuristics drawn from the experience with debugging queries.

3.4 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.

3.4.1 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 contained 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.

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

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 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.

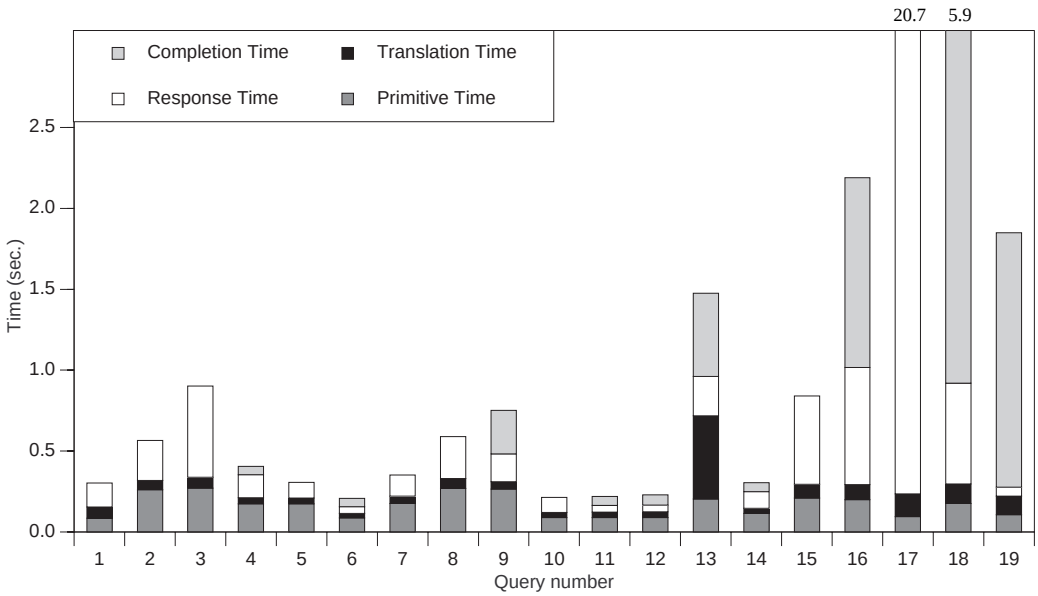

Figure 14. Query execution times

3.4.2 Execution Time

Figure 14 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 3.3.2.
- *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.

3.4.3 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-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 15 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[1], 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

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

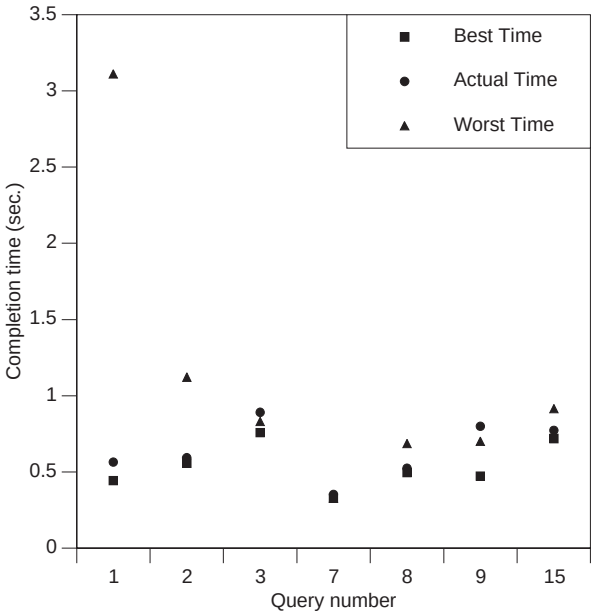

Figure 15. Completion time depending on join ordering (small queries)

orderings, but the outcome of the join ordering may be cushioned by the incremental delivery methods discussed later.

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)

3.4.4 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 14). 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.

3.4.5 Hash Joins

Many queries can use hash joins to avoid computing the full Cartesian product. But as explained in section 3.3.5, 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 system with five times more objects than the original Self system, the slowdowns became substantial (right column of Table 3), ranging from 0.32 to 15.6 with a median of 1.97. The evaluation of query 18 was faster using nested-loop joins in the small system, but much slower in the large system. Query 16 had almost the same evaluation speed regardless of join method.

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).

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

Query number	Original 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

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)

3.5 Related work

Queries containing only object references can be rephrased in a graph-theoretical form [122]. Consider a class of queries in which objects relate to other objects only through references, i.e., consider situations where object constraints do not contain method or expression evaluations. Answering such a query is equivalent to finding a subgraph conforming to the given referential constraints in a graph formed by the runtime objects as nodes and their references as edges. We call the problem of finding such subgraphs the Generalized Pattern Matching (GPM) problem. This problem is a special case of Subgraph Isomorphism [70] that significantly differs from the general problem because the outgoing edges of a vertex have unique labels—a restriction arising from the fact that object fields referring to other objects have unique names. We proved the GPM problem to be NP-complete even for bipartite graphs with only two outgoing edges (Appendix A). We did not further pursue the graph approach because queries are more expressive.

3.6 Summary

Query-based debugging allows programmers to ask queries about the program state, helping to check object relationships in large object-oriented programs. Our implementation of the static 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 programmers 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.

4 Dynamic Query-Based Debugger

"A little help at the right time is better than a lot of help at the wrong time."

Tevye

"Good information is hard to get. Doing something with it is even harder!"

L. Skywalker

4.1 Introduction

Many program errors are hard to find because of a cause-effect gap between the time when the error occurs and the time when it becomes apparent to the programmer by terminating the program or by producing incorrect results [58][59][60]. The situation is further complicated in modern object-oriented systems which use large class libraries and create complicated pointer-linked data structures. If one of these references is incorrect and violates an abstract relationship between objects, the resulting error may remain undiscovered until much later in the program's execution.

For example, consider a possible error in the javac Java compiler discussed in section 2. What would happen if an abstract syntax tree (AST) built during a compilation is corrupted by an operation that assigns the same expression node to the field right of two different parent nodes (Figure 16). The parent nodes may be instances of any subclass of BinaryExpression; for

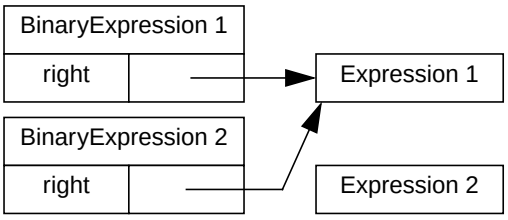

Figure 16. Error in javac AST

example, the parent may be an AssignAddExpression object or a DivideExpression object, while the child could be an IdentifierExpression. The compiler traverses the AST many times, performing type checks and inlining transformations. During these traversals, the child expression will receive contradictory information from its two parents. These contradictions may eventually become apparent as the compiler indicates errors in correct Java programs or when it generates incorrect code. But even after discovering the existence of the error, the programmer still has to determine which part of the program originally caused the problem. How can we help programmers to find such errors as soon as they occur?

As discussed in section 2, data breakpoints [180], conditional breakpoints [109], and other conventional tools do not help in finding the javac error. A more effective way to check an inter-

object constraint would be to combine conditional breakpoints with the static query-based debugger described in section 3 [123]. A static query-based debugger (SQBD) finds all object tuples satisfying a given boolean constraint expression. For example, the query

```
BinaryExpression* e1, e2.    e1.right == e2.right && e1 != e2
```

would find the objects involved in the above javac error. The breakpoints would then carry the condition that the above query return a non-empty result. Unfortunately, even well-optimized SQBD executions would be inefficient for this task. With hundreds or thousands of BinaryExpression objects, each query becomes quite expensive to evaluate, and since the query is reevaluated every time a conditional breakpoint is reached, the program being debugged may slow down by several orders of magnitude. (This claim is substantiated in section 4.4.3.1.)

To overcome this inefficiency, we extend the query-based debugging to allow *dynamic* queries [124]. In addition to implementing the regular QBD query model, a dynamic query-based debugger continually updates the results of queries as the program runs, and can stop the program as soon as the query result changes. To provide this functionality, the debugger finds all places where the debugged program changes a field that could affect the result of the query and uses sophisticated algorithms to incrementally reevaluate the query. Therefore, a dynamic query-based debugger finds the javac AST bug as soon as the faulty assignment occurs, and it does so with a minimal programmer effort and a low program execution overhead.

We have implemented such a dynamic query-based debugger for Java. Our prototype is portable (written in 100% pure Java) and surprisingly efficient. Experiments with large programs from the SPECjvm98 suite [158] show that selection queries are very efficient for most programs, with a slowdown of less than a factor of two in most experiments. Through measurements, we determined that 95% of all fields in the SPECjvm98 applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation times, our performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. More complicated join queries are less efficient but still practical for small query domains or programs with infrequent queried field updates.

4.2 Query Model and Examples

Dynamic query-based debugging uses the query model from section 3.1 adapted for Java syntax and semantics. Consider another javac query:

```
FieldExpression fe; FieldDefinition fd.
fe.id == fd.name && fe.type == fd.type && fe.field != fd
```

The first part of the query is the *search domain* of the query, and should be read as “find all FieldExpressions fe and all FieldDefinitions fd in a system such that...”. FieldExpression is a class name and its domain contains all instances of the class. If a “*” symbol in a domain declaration follows the class name (as in the javac query discussed in the introduction), the domain includes

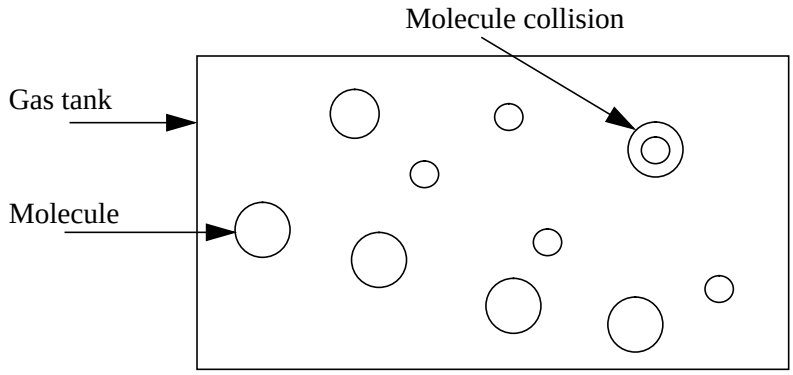

Figure 17. Error in molecule simulation

all objects of subclasses of the domain class; otherwise the domain contains only objects of the indicated class itself.

The second part of the query specifies the constraint expression to be evaluated for each tuple of the search domain. Constraints are arbitrary Java conditional expressions as defined in the Java specification §15.24 [78] with certain syntactic restrictions. Expressions should not contain variable increments which have no semantic meaning in a query. The debugger currently does not handle array accesses. Constraints can contain method invocations; we assume that these methods are side-effect free.

As for the static queries, 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. Conceptually, the dynamic debugger reevaluates a query after the execution of every bytecode, ensuring that no result changes are unnoticed. The debugger stops the program whenever the result changes. In reality, the debugger reevaluates the query as infrequently as possible without violating these semantics. In addition, the debugger will reevaluate only the part of the query that changed since the last evaluation. We describe the incremental reevaluation technique in detail in section 4.3.5.1.

The dynamic query debugger does not allow to query temporal properties of the objects. For example, the queries cannot find token objects that changed their value from a CharacterToken to the UninitializedToken. Such functionality would involve using temporal logic operators and is beyond the scope of this thesis.

4.2.1 Ideal Gas Tank Example

For another example illustrating the need for dynamic query debuggers, consider an applet simulating a tank with ideal gas molecules (Figure 17). Although this applet is a simple simulation of gas molecules moving in the tank and colliding with the tank walls and each other,

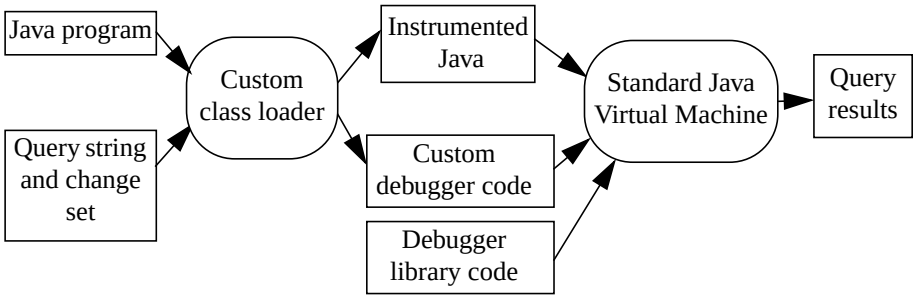

Figure 18. Data-flow diagram of dynamic query-based debugger

it has some interesting inter-object constraints. First, all molecules have to remain within the tank, a constraint that can be specified by a simple selection query:

```
Molecule* m.    m.x < 0 || m.x > X_RANGE || m.y < 0 || m.y > Y_RANGE
```

Another constraint requires that molecules do not occupy the same position as other molecules. Even this simple application may violate the constraint in different places: in the regular molecule move method, in a method that handles molecule bounces from the walls, and so on. The following query discovers the constraint violation:

```
Molecule* m1 m2.    m1.x == m2.x && m1.y == m2.y && m1 != m2
```

This constraint is interesting because its violation is a transient failure. Transient failures disappear after some period of time, so even though the program behaves differently than the programmer expected, queries will not be able to detect failures if they are asked too late. The molecule collision error is such a transient failure—it will disappear as the molecules continue to move. However, the applet will behave erroneously: for example, molecules that should have collided with each other will pass through each other. Dynamic queries are necessary to find transient failures, as a delayed query reevaluation may fail to detect the error entirely.

4.3 Implementation

We have implemented a Java dynamic query-based debugger in pure Java. Java contains a number of features that simplified the implementation. We used the ability to write custom class loaders [125] to perform load-time code instrumentation. Java's bytecode class files proved simple to instrument. The debugger creates custom query evaluation code by using load-time code generation. The debugger can be ported to other languages (e.g., Smalltalk) that have an intermediate level format similar to bytecodes. We discuss issues of such implementation in section 4.3.6 and section 4.3.7.

4.3.1 General Structure of the System

Figure 18 shows a data-flow diagram of the dynamic query-based debugger. To debug a program, the user runs a standard Java Virtual Machine with a custom class loader. The custom class loader loads the user program and instruments the bytecodes loaded, by adding debugger

invocations for each domain object creation and relevant field assignment. The class loader also generates and compiles custom debugger code. After loading, the Java virtual machine executes the instrumented user program. Whenever the program reaches instrumentation points, it invokes the custom debugger code, which calls other debugger runtime libraries to reevaluate the query and to generate query results. The debugger currently does not handle multithreaded code.

Logically, the program control flow can be divided into four important sections:

- Shell—the Java wrapper program that takes the class name and arguments of the original program together with the query string. This section of the system creates a custom class loader instance, loads the program using this class loader, and starts the execution through the reflection interface.
- Class loader—loads the program class and all classes requested by it as defined in section 2.16 of the *Java™ Virtual Machine Specification* [127]. During loading process, instruments class files as described below.
- Original program—executes the original program invoking the debugger at instrumentation points.
- Query evaluator—parses the query expression using a JLex [23] generated lexer and a Java Cup [91] generated parser[1] [12], creates runtime structures for query evaluation, and evaluates the query using them when it is invoked from the original program.

The rest of this section discusses the most important parts of the debugger in more detail: how the debugger instruments a Java program, what parts it instruments, and how it evaluates a query.

4.3.2 Java Program Instrumentation

To enable a dynamic query for a program, the user specifies a query string. The debugger then instruments class files to invoke the debugger after all events that may change the result of the query. The debugger finds assignments to the fields referenced in the query change set

[1] The parser uses a restricted Java expression grammar extracted from the full Java grammar [30].

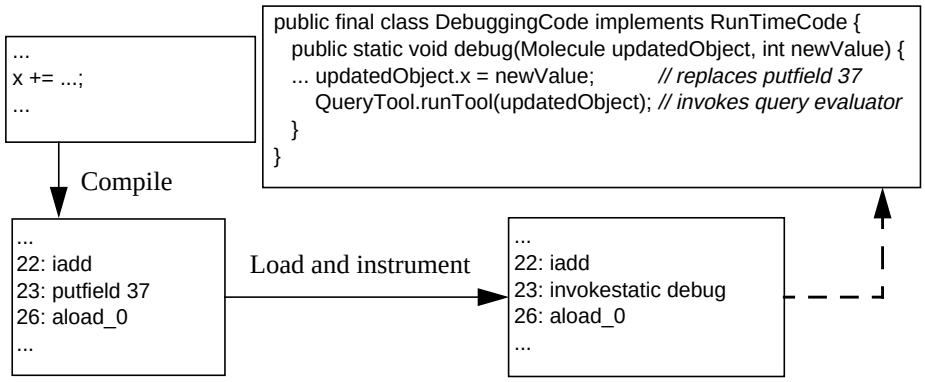

Figure 19. Java program instrumentation

(section 4.3.3) and inserts debugger invocations after each one of them. The system also inserts debugger invocations after each call to a constructor of a domain object.

Figure 19 shows an example of the instrumentation process for a Java method. To instrument class files, the loader transforms them in memory into a malleable format using modified class file handling tools borrowed from the BCA class library [106]. The BCA class library provides functionality to read a class file and parse it in memory into data structure representing its parts: constant pool, fields, methods, interfaces, and attributes. The program handles the class file format described in section 4 of *The Java™ Virtual Machine Specification* [127]. We have extended the framework to parse structures that were not modified in the original BCA, to convert method bytecodes into objects and back into a byte array, to add changes to the constant pool and to the method code, and to write the instrumented class file back to the disk.

The instrumentation method iterates through the code of all class file methods searching for field assignments and object creations that have to be instrumented. In a Java class file, the method code is stored in the *code attribute* of the method. The debugger transforms a byte array of code into a list of Bytecode objects, instruments the code by changing or adding Bytecode objects, and transforms it back into a byte array using the Visitor design pattern [68]. During the instrumentation, the debugger preserves code validity by adjusting branch targets and exception handlers. After instrumentation, the instrumentation program updates the code length and the stack size of a method.

Iterating through the method code, the debugger finds all putfield bytecodes and invokespecial bytecodes. The loader determines all putfield bytecodes that assign to the fields of interest—like field x in Figure 19—and replaces these putfield bytecodes with invokestatic bytecodes invoking the debugger. The system also inserts such debugger invocations after each invokespecial call to a constructor of a domain object.

The debugger determines that a putfield bytecode should be instrumented by checking the type of the field assigned by the bytecode in the constant pool of the class. In Figure 19, the debugger would inspect the constant pool entry 37, and realize that the bytecode assigns to the field x of

the Molecule class. When the debugger replaces a putfield bytecode with an invokestatic call, it also inserts the method name and type information about the invoked debug method of the DebuggingCode class into the constant pool of the instrumented class. The debug method is different for each domain (change set) class, so its type information depends on the putfield replaced. The debug method takes two arguments: the object that the putfield would have updated—a Molecule object in Figure 19—and the newValue value to be assigned to the object field—in this case an integer number. These objects are already on the stack before the execution of the putfield, so they will be correctly passed as arguments to the debug method, and the debugger does no stack manipulation of the instrumented method. The debugger also does not use additional local variables, avoiding data-flow analysis.

Since the original putfield has been replaced by the invokestatic bytecode, the custom debug method performs the assignment originally executed by the putfield. The debugger determines the name of the assigned field and the correct types of objects and values from the class file's constant pool.

To monitor object construction, the debugger inserts a debug method invocation after each bytecode that constructs a domain object. In Java class files, the objects are created in two stages. First, a new object is created using the new bytecode. Then, the constructor method is invoked using the invokespecial bytecode. According to *The Java™ Virtual Machine Specification* ([127] section 4.9.4), the object cannot be used between the creation of the uninitialized object and its initialization. Consequently, we insert debugger invocations after the constructor termination. This approach misses query violations occurring in constructors. However, it may be argued that the values in the fields of uninitialized objects are not legal, so query evaluation with them would not follow the intentions of the programmer. It may be possible to safely insert debugger invocation in the middle of constructor method after object fields are initialized to their "correct" initial values, but before any other housekeeping in the constructor. The data-flow analysis of the constructor methods necessary for such instrumentation could be done in the full-fledged implementation of a query-based debugger.

The debugger invocation after the invokespecial bytecode is similar to the one described above for invocations replacing putfield bytecodes. However, in this case the debugger inserts an additional dup bytecode to duplicate the created object reference on the stack. This reference is the only parameter passed to the debugging method. To account for additional duplication on the stack, the maximal size of the stack is increased.

After instrumentation, the class loader transforms the code back into the class file format and passes the image to the default defineClass method.

The class loader instruments assignments and object constructors that influence the query result. The next section describes how the debugger determines which assignments and constructors to instrument.

4.3.3 Change Monitoring

The dynamic query debugger updates the query result every time the debugged program performs an operation that may affect the query result. Thus, the program being debugged has to invoke the debugger after every event that could change the query result. The query result may change because some object assigns a new value to one of its fields or because a new object is constructed. However, not all field assignments and object creations affect the query. We call the set of constructors and object field assignments affecting the results of a query the query's *change set*. Although we can use all assignments and all constructors as a conservative change set for any query, we are interested in a minimal change set for efficient query evaluation. Such a change set contains only constructors of domain objects and assignments to domain object fields referenced in a query. The change set is used by the class loader to determine which assignments and constructors it should instrument.

Consider the Molecule query:

```
Molecule* m1 m2.    m1.x == m2.x && m1.y == m2.y && m1 != m2
```

The change set of this query consists of the constructors of the Molecule class and its subclasses as well as assignments to Molecule fields x and y. Assignments to other molecule fields such as color do not belong to the change set.

The change set of a query becomes complicated if constraints contain a chain of references. Consider a query for the SPECjvm98 ray tracing program:

```
IntersectPt ip.    ip.Intersection.z < 0
```

The Intersection field is a Point object, and the query result depends on its z value. The query result may change if the z value changes, or if a new value is assigned to the Intersection field. Furthermore, the Point object referenced by the Intersection field may be shared among several domain objects. In this case, a change in one Point object can affect multiple domain objects. A chain of references also occurs when a domain instance method invokes methods on objects referenced in its fields, and these methods in turn depend on the fields of the receiver. The process of tracking which objects accessed through a chain of field references influence which domain objects becomes a complicated task; for example, to do it efficiently, nested objects need to point back to the domain objects that reference them. To simplify the prototype implementation, we support only explicit chains of references in the query, and we do not handle methods that access chains of references. Our debugger rewrites the query by splitting the chain into single-level accesses and by adding additional domains and constraints. For example, the ray tracing query above is rewritten as:

```
IntersectPt ip; Point* __Intersection.
ip.Intersection == __Intersection && __Intersection.z < 0
```

Chain reference splitting adds overhead by introducing additional joins into the query but it also allows users to ask more complex queries. The overhead can be an order of magnitude when a

selection query is rewritten as a join query. We do not handle native methods because their debugging is outside the scope of a Java debugger.

The change set is determined automatically by examining the query string. First, the domain classes are added into the MonitoredClasses structure. Second, the fields of domain classes referenced in the query are added to the MonitoredFields collection of the corresponding MonitoredClass instance. If fields are inherited from superclasses, these classes are also added to the MonitoredClasses collection with fields referenced. Method invocations are not handled automatically, but users can specify fields used by the query methods by hand.

To summarize, we use the change set of the query to instrument the Java program. The instrumented program calls the debugger after every event that could change the result of the query, and the debugger reevaluates the query during each call.

4.3.4 Domain Collection Maintenance

Unlike the static query-based debugger implementation (section 3.3.2), the dynamic query-based debugger does not request the Virtual Machine to provide all objects of query domains at each query reevaluation. Even if such functionality was available, its efficiency would be low. Instead, the debugger tracks all domain objects by maintaining domain object collections. Every time a domain object is created, the program invokes the debugger which places the new domain object into its domain collection. The debugger uses the domain collection in query evaluations to iterate through all domain objects. To facilitate incremental query reevaluation (section 4.3.5.1), the debugger partitions domain collections into changed and unchanged parts after each monitored event. To maintain query correctness and to facilitate garbage collection, the debugger allows the garbage collector [103][173][192] to delete dead objects from domain collections. This behavior is implemented by referring to the objects in domain collections through weak pointers. As a result, the debugger's references to the domain objects are ignored by the garbage collector, and the garbage collector behaves as if no debugger was present. After program-unreachable objects are garbage collected, the debugger discards null weak references that previously pointed to the collected objects.

The debugger does not use weak references in the intermediate result collections. Although it would be easy to use weak references in the intermediate result collections, it is an unnecessary overhead. These collections are discarded and recomputed every time the query is reevaluated, so they themselves are garbage collected. Since the query reevaluation frequency is usually greater than the garbage collection frequency, the current implementation works fine.

For queries that have non-empty results (section 4.6), the debugger should update these results when domain objects are garbage collected. However, to support the result update on GC, i.e., to remove the tuples containing garbage collected objects and to inform the user about the changes in the result, the debugger has to be aware of the garbage collection. This can be

achieved by changing the JVM garbage collector, by using weak reference queues, or by using finalizable weak references. We have not implemented this functionality.

Using weak references adds an additional overhead to the query domain maintenance, so for small programs users can use a system that references objects directly and does not allow garbage collection of domain objects[1].

To confirm that the query results do not contain otherwise dead objects, the system could perform full garbage collection and reevaluate the query again. Such option would be expensive but worthwhile for users who want to be certain about the validity of the query results. This option is not implemented in the current system.

4.3.5 Overview of Query Execution

This section describes query evaluation after an instrumented event in the debugged program. Whenever the program invokes the debugger, it passes the object involved in the event. If the event is a field assignment, the program also passes the new value to be assigned to the field. Figure 20 shows the control flow of the query execution. First, the debugger checks whether the changed object is a domain object. Consider a query that finds Id objects with a negative type code:

```
Id x.    x.type < 0
```

Here, Id is a subclass of the Expression class, and the type field is defined in Expression. Thus, the program may invoke the debugger when the type field inherited from the Expression class is assigned in an object of another Expression subclass. For example, the program invokes the debugger after assigning the type field in an ArithmeticExpression object. This object shares the type field with the domain class objects, but it does not belong to the query domain, so the debugger immediately returns to the execution of the user program without reevaluating the query.

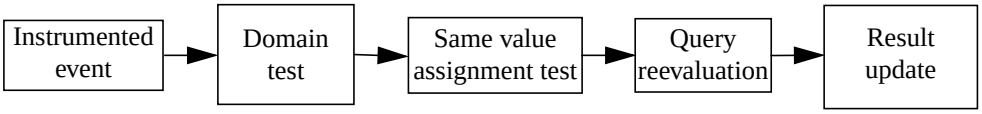

Figure 20. Control flow of query execution

If the object passes the domain test, the debugger checks whether the value being assigned to the object field is equal to the value previously held by the field. For example, some molecules do not move in the ideal gas simulation, yet their coordinates are updated at each simulation step. Such assignments do not change the result of the query and can be ignored by the debugger. The debugger does not perform this test if the invoking event is an object creation.

[1] The experiments in this chapter were performed on such system. See section 4.4.

This test is just one example of tests that quickly verify whether a query result has changed due to the assignment. Assignments that do not change the query result are called *invariant* assignments. If the system can infer that an assignment is invariant, it can skip evaluating the query. Another set of assignments that could be used to optimize the system are *monotonic* assignments. Such assignments either increase the query answer set or decrease it but do not both add and remove some elements to the answer set. Not counting the equality test, other invariant and monotonic assignments depend on the query semantics. For example, if a query has an inequality constraint `x.y < 0`, a decrease of field `y` would be monotonic because it could not decrease the size of the answer set. Future implementations of the debugger could exploit invariant and monotonic events.

After these two tests, the debugger starts reevaluating the query. The non-incremental query evaluation algorithm is described in section 3.3.3. The dynamic query-based debugger uses incremental reevaluation to improve the efficiency of the previous algorithm.

4.3.5.1 Incremental Reevaluation

When a program invokes the debugger, it passes the changed object to the debugger. From the properties of our change sets, we know that this object is the only one that changed since the last query evaluation. Consequently, a full reevaluation of the query for all domain objects is unnecessary. We use incremental reevaluation techniques developed for updates of materialized views in databases [26][34] to speed up the query execution. Consider a query, a join of three domains `A * B * C`, e.g.,

```
A a; B b; C c.    a.x == b.y && b.z < c.w
```

The "*" symbol denotes a Cartesian product with some selection constraint; the "+" symbol below denotes a set union. If an object of domain `B` changes, the new result of the query is

$$A * (B + \Delta B) * C = (A * B * C) + (A * \Delta B * C)$$

The transformation of the result into the formula on the right hand side is correct because the Cartesian product and union operations are distributive. The first part of the result is the result of the previous query evaluation. The debugger stores this result—usually empty for assertion queries—and does not need to reevaluate it. The second part of the result contains only the changed object (ΔB) of the domain `B` combined with objects of the other domains. The debugger evaluates the changed part in the same way as it would evaluate the whole query. Figure 21 shows an incremental evaluation of changes in the query result. The execution starts with the changed object ΔB passed from the user program. Because this is the only object for which the debugger evaluates the first constraint, the intermediate result is likely to be empty. In general, the size of intermediate results is much smaller in the incremental evaluation, speeding up the query evaluation. If intermediate results are not empty, the debugger continues the evaluation in the usual manner and produces the incremental result ($A * \Delta B * C$). The system then merges the result with the previous result to form the complete query result.

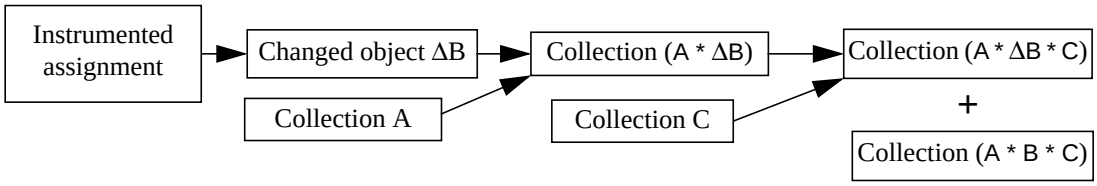

Figure 21. Incremental query evaluation

The query evaluation is further optimized by finding efficient join orders and by using hash joins as described in section 3.3. Because sizes of domains change during program runtime and we cannot efficiently determine the selectivities of constraints, we use a simple heuristic for join ordering: execute selections first, equality joins next, and inequality constraints last.

4.3.5.2 Custom Code Generation for Selection Queries

Constraints of selection queries are usually very simple and can be evaluated very fast. Instead of performing the general query execution algorithm described in section 4.3.5.1, which goes through numerous general steps and calls a number of methods, the debugger can evaluate just the few tests necessary to check the selection constraints. Because these tests depend on the query asked, the code for their evaluation has to be generated at program load time. During the loading of the user program, the debugger generates a Java class with a debug method. We show such a method in Figure 22 for the query

```
Molecule1 m.    m.x > 350
```

The first three statements of the method contain the code common for both unoptimized and optimized versions. This code performs the domain test and the same value assignment test described in section 4.3.5. The optimized code that follows evaluates the selection constraint on

```
public final class DebuggingCode implements RunTimeCode {
    public static void debug (Molecule updatedObject, int newValue) {
        // Code common for both general and optimized versions
        if (! (updatedObject instanceof Molecule1))
         { updatedObject.x = newValue; return; }
        if (updatedObject.x == newValue)  return;
        updatedObject.x = newValue;
        // Instead of calling general query evaluation method,
        // evaluate constraint here
        if (updatedObject.x > 350)
            QueryTool.outputResult(updatedObject);
    }
}
```

Figure 22. Selection evaluation using custom code

the changed object and calls the debugger runtime only if the query has a non-empty result. The debugger uses the debug method as an entry point that the user program calls when it reaches instrumentation points. With custom code generated, the debug method contains all code needed to evaluate a selection, so the reevaluation costs only one static method call. Furthermore, the debug method—a member of a final class—may even be inlined into the instrumentation points by a JIT compiler. We could also inline the bytecodes into the instrumented method.

4.3.6 Related Work

Debugger implementations use a variety of techniques to gather information about objects and to instrument program code. This section discusses runtime information gathering methods in more detail.

4.3.6.1 Runtime Information Gathering Techniques

Debuggers providing data about runtime events use different techniques to detect these events. The implementation of event monitoring can be divided into three categories by the source of event information:

- Program itself. The program is instrumented to provide data about runtime events. Either program text [50][51][116][129], its bytecode form, or the executable [109] can be instrumented.
- Runtime system. The runtime system provides the information about events either by itself or through its modification.
- Operating system debugging interface. The operating system or its debugger is used to gather the information.

Most program debugging and visualization systems such as HotWire [116] and Ovation [50][51] instrument the program text to generate events of interest during the execution. Lange and Nakamura [117][118][119][120] investigated program instrumentation, reflection (metaclass) protocol, and the HeapView Debugger to track objects in the heap. These three methods belong to the three different categories outlined above. According to these authors, source code instrumentation offered the fastest trace generation, but required source code modification and program recompilation.

Consens et al. [44][45] use the Hy^+ visualization system to find errors using post-mortem event traces. De Pauw et al. [52] and Walker et al. [182] use program event traces to visualize program execution patterns and event-based object relationships, such as method invocations and object creation. All these systems use program instrumentation to obtain the event traces, although Kimelman et. al. [111] also use information provided by the underlying operating system.

Laffra [113] discussed Java source code instrumentation by using a preprocessor or by modifying the javac compiler. We have opted for class file bytecode instrumentation at load time.

None of the debugging projects modify compilers to add visualization code. In contrast to the class loader modification, changing a compiler is considered a major effort outside of the scope of debugger implementations. Although using compilers to add debugging code would be roughly equivalent to the preprocessor based code instrumentation, the compiler could be more efficient in added code or more powerful by accessing objects and their state, which is difficult to access from the preprocessor directives. Compilers do implement assertions—an event gathering technology supported in programming languages such as Eifel [134]. However, adding assertions to languages like Java necessitates implementation schemes similar to these of other debugging constructs [56][105] and is rarely done by compiler modification. Both Handshake [56] and *jContractor* [105] use load-time class file instrumentation to implement Java assertions. The BCA tool [107] changes a small part of the javac compiler to incorporate class file loading through the BCA subsystem.

The bytecode instrumentation used in the dynamic query-based debugger is similar to the technique proposed by Kessler [109] to implement fast breakpoints. However, Kessler had to deal with a more difficult problem of instrumenting executable code. In executable code, the system would not be able to replace a short instruction with a long one, or to insert additional instructions, because control flow addresses cannot be adjusted as easily in the executable code as they can be in the bytecode. Executable code instrumentation was used in such areas as software-based fault isolation [181].

To avoid source code modification, some debugging systems use runtime system information for event monitoring. Laffra's [113] LTK Visual Java Debugger uses a patched JVM to receive the method call and exit information. Similarly Hotwire's implementation for Smalltalk uses a patched version of GNU Smalltalk [113]. Lange and Nakamura [117][118][119][120] use reflection capabilities of the IBM SOM to get debugging information. In Java, the reflection package and debugger API provide hooks into the Java VM which are unfortunately insufficient for object-oriented visualizations or query-based debugging. The new *Java™ Platform Debugger Architecture* [98] gives debugger writers access to more information. Unfortunately, the current version of Java™ Platform Debugger Architecture does not yet allow debuggers to retrieve a collection of all objects of a class, nor does it provide code instrumentation facilities.

Yet another way of accessing debugging information is to use available operating system debugging interfaces. In most cases, such an approach has a high cost because the debugger runs in a separate address space and incurs expensive context switches. In spite of this drawback, Lange and Nakamura chose to use the debugger interface in the final version of their system because they deemed source code instrumentation impractical for real world applications.

Classification of the event-gathering debugger implementations is very similar to that of the data breakpoint implementations [108][179][180]. If processors provide support for write monitors, data breakpoints can use this facility. This approach, though available in modern processors, has very limited functionality; for example, only ten locations can be monitored using Intel x86 breakpoint registers. Data breakpoints can also be implemented by placing objects in write-protected virtual memory pages and transferring control to the debugger upon a page trap or by using other write-barrier techniques [87][109]. To implement the query-based debugger using the above two approaches, one would have to modify the Java Virtual Machine, an approach that we avoided for portability reasons.

Since the query-based debugger uses Java bytecode instrumentation, the next section explores alternative load-time instrumentation techniques that can be used to instrument Java programs.

4.3.6.2 Load-Time Code Instrumentation

Our system uses load-time code modification to insert debugger invocations in the debugged program. The instrumentation is done by providing a custom class loader. Other load-time instrumentation alternatives were comprehensively explored by Duncan and Hölzle in [57]. Here the available techniques, their advantages and disadvantages are briefly recapped. (Figures courtesy of Duncan and Hölzle.)

The main goal of load-time adaptation (LTA) is to intercept a file request from the *host* program and to use a *tool* program to adapt the file before providing it to the host program. In the case of debugging event gathering, the host program is the Virtual Machine, while the tool is the class file instrumentor. Here we present different load-time adaptation techniques for JVM-instrumentor combination:

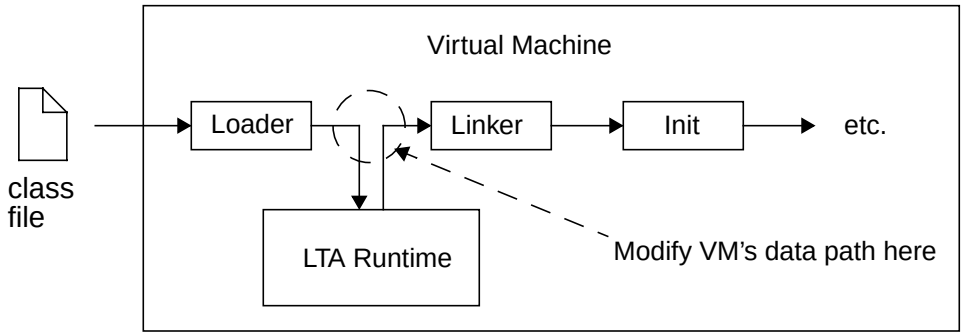

Figure 23. Modifying a VM to implement LTA.

- *Modifying the host (VM) (Figure 23)***.** This approach has been used in numerous projects both in Java world [7][106][138] and beyond it [86]. Since this technique modifies the host, it has limited portability, both because the implementation cannot be reused for other hosts,

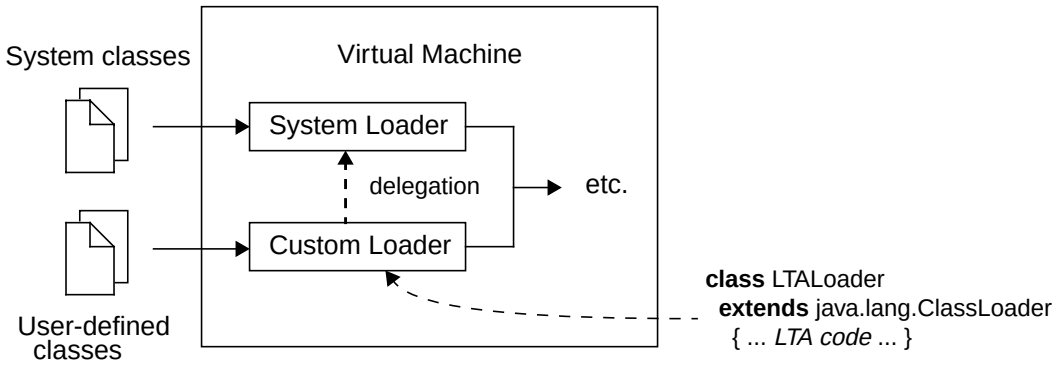

Figure 24. Performing LTA with a custom class loader.

and because it has to be provided for all hosts of the supported class, i.e., all VMs. The tool implementers may not have an access to the host source code and so may not be able to support such a host. The advantage of this approach is that the file adaptation is done inside a host that "understands" the file and has additional host-specific information (e.g., global data-flow), so it can do sophisticated transformations that may not be possible outside the host.

- *Custom class loaders (Figure 24)*. The approach of using custom class loaders to load class files is confined only to the Java Virtual Machines and the class files loaded through the loaders [125]. Although custom class loaders are powerful because they intercept classes loaded both from the files and through the network, they have several limitations. First, the system classes are not accessible to the custom class loaders because they are loaded directly by the JVM system class loader. Second, the interaction of multiple user-defined class loaders is confusing. The advantage of custom class loaders is that they use a hook provided by the Java Virtual Machine definition [127], so tool programming is relatively easy and can be done in Java. We have used this approach in our prototype implementation. This approach was also used in other systems [43][187].

- *Intercepting system calls (Figure 25)*. It is possible for the tool to intercept the operating system calls with which libraries request files. This method does not depend on the host program and on the type of files accessed. A drawback of this approach is that it is operating system specific. Although Solaris provides a /proc interface for the system call interception, other operating systems may not have such convenient hooks [13]. Neither this method nor the following library call interception method can determine the program requesting a file, so they cannot adjust the adaptation behavior to different hosts. The UFO global file system uses this approach for handling file requests [13].

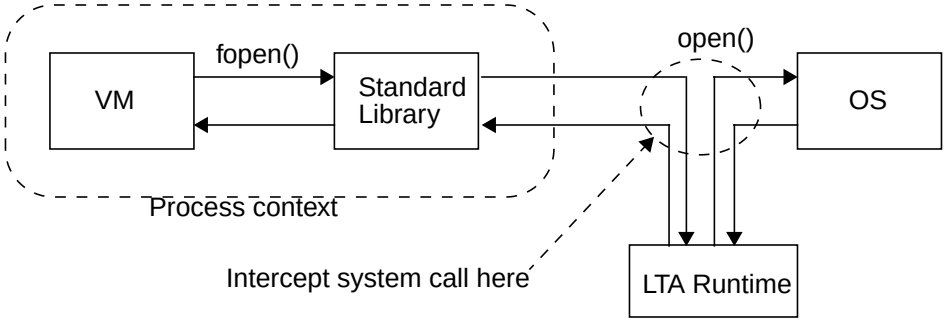

Figure 25. Implementing LTA by intercepting system calls.

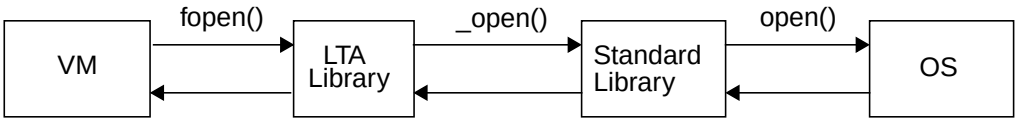

Figure 26. Implementing LTA using dynamic linking.

- *Intercepting library calls (Figure 26).* This approach intercepts file open requests from the host to the standard libraries. It shares most advantages and disadvantages with the previous approach. Though it is operating system independent, it is standard library dependent and can easily be applied only when libraries are dynamically linked. The Proteus system uses this approach to add assertions to the Java language [57]. The Xab PVM program monitoring system [22] also uses library replacement for PVM call interception.

Further investigation of load-time adaptation is beyond the scope of this work. Any of its variations can be applied to the dynamic query-based debugger implementation.

4.3.7 Dynamic Query Debugger Implementations for Other Languages

To be practical for a wide range of debugging tasks, the dynamic query-based debugger should be applicable and implementable for various programming languages. It is clear that the concept of debugging with queries can be used in all object-oriented and object-based languages. Similar principles (although with different implementation concerns) can be used in programming languages with structures or records (procedural languages such as C and Pascal and functional programming languages such as Haskell, Lisp, and ML). How difficult would it be to port the debugger into environments of these languages?

The following key issues influence the difficulty of porting:

- *Instrumentation*. Instrumentation can be done at a source level or at the intermediate format level. The Java bytecode format is well suited for instrumentation. Other intermediate formats such as Smalltalk bytecodes may provide similar ease of instrumentation. Instrumenting source code is more complicated, because this code needs to be parsed and transformed into some temporary format before instrumentation. Instrumenting or handling compiled code is probably prohibitive unless some VM based code standard emerges [4]. In languages allowing unrestricted use of pointers such as C and C++, handling of field assignments through instrumentation may prove difficult if not impractical. For such languages, one would need to use alternative approaches, such as placing objects into the read-only memory and invoking the debugger on write traps.

- *Domain object gathering*. This would be easy in languages like Smalltalk that provide a standard way of retrieving all objects of a class. On the other hand, tracking all objects of the same type in languages such as Pascal may require significant modification of the compiler and the runtime system.

- *Runtime code generation and evaluation*. To evaluate query expressions, the debugger needs to generate and evaluate code during program runtime. Smalltalk and Self support dynamic code generation. For other languages such as C and Pascal, dynamic code generation would be problematic. Although the execution of C statements during runtime is present in the `gdb` debugger, it is not simple to implement.

From our experience, porting the debugger to pure object-oriented languages (e.g. Smalltalk) would be possible without a lot of changes. Support of procedural languages such as C, C++, and Pascal would be more difficult.

4.4 Experimental Results

Ideally, a test of the efficiency of a dynamic query-based debugger would use real debugging queries asked by programmers using the tool for their daily work. Although we tried to predict what queries programmers will use, each debugging situation is unique and requires different queries. To perform a realistic test of the query-based debugger without writing hundreds of possible queries, we selected a number of queries that in complexity and overhead cover the range of queries asked in debugging situations. The selected queries contain selection queries with low and high cost constraints. The test also includes hash-join and nested-join queries with different domain sizes. The queries check programs that range from small applets to large applications and (for stress-tests) microbenchmarks. These applications invoke the debugger with frequencies ranging from low to very high, where a query has to be evaluated at every iteration of a tight loop. Consequently, the experimental results obtained for the test set should indicate the range of performance to be expected in real debugging situations.

Query	Slowdown	Invocation frequency (events / s)
1. Molecule1 z. z.x > 350	1.02	15,000
2. Id x. x.type < 0	1.11	16,000
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	1.25	169,000
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	1.18	1,900,000
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	1.27	
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	1.37	
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	5.83	
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	1.18	933,000
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	1.10	196,000
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	1.83	
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	1.23	787,000
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	1.98	2,300,000
13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	2.13	54,000
14. Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	3.43	25,000
15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	229	350,000
16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	157	1,500,000
17. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	77	2,600,000
18. Test5 z. z.x < 0	6.4	42,000,000
19. TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	228	40,000,000
20. TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)	930	

Table 5. Benchmark queries

For our tests we used an otherwise idle Sun Ultra 2/2300 machine (with two 300 MHz UltraSPARC II processors and 384 MB physical memory) running Solaris 2.6 and Solaris Java 1.2 with JIT compiler (Solaris VM (build Solaris_JDK_1.2_01, native threads, sunwjit)) [99] with a 128 MB heap. Execution times are elapsed times and were measured with millisecond accuracy using the System.currentTimeMillis() method. All executions used only main memory and contained no paging disk I/O. (Full experimental results are reported in Appendix B.)

4.4.1 Benchmark Queries

To test the dynamic query-based debugger, we selected a number of structurally different queries (Table 5) for a number of different programs (Table 6):

- Queries 1 and 13 check a small ideal gas tank simulation applet that spends most of the time calculating molecule positions and assigns object fields very infrequently. It has 100 molecules divided among Molecule1, Molecule2 and Molecule3 classes. The application performs 8,000 simulation steps.
- Queries 2 and 14 check the Decaf Java subset compiler, a medium size program developed for a compiler course at UCSB. The Token domain contains up to 120,000 objects.
- Query 3 checks the Jess expert system, program from the SPECjvm98 suite [158].
- Queries 4–10, and 16–17 check the compress program from the SPECjvm98 suite. Our queries reference frequently updated fields of compress.
- Queries 11–12 and 15 check the ray tracing program from the SPECjvm98 suite. The Point domain contains up to 85,000 objects; the IntersectPt domain has up to 8,000 objects.
- Queries 18–20 check artificial microbenchmarks. These microbenchmarks stress test debugger performance by executing tight loops that continuously update object fields.

Application	Size (Kbytes)	Execution time (s)
1. Compress	17.4	50
2. Jess	387.2	22
3. Ray tracer	55.7	17
4. Decaf	55	15
5. Ideal gas tank	14.3	57

Table 6. Application sizes and execution times

Structurally, queries can be divided into the following classes:

- Queries 1–12 and 18 are simple one-constraint selection queries with a wide range of constraint complexities. For example, query 4 has a very simple low-cost constraint that

compares an object field to an integer. The more costly constraint in query 5 invokes a method to retrieve an object field. Another costly alternative constraint (query 6) invokes a comparison method that takes a value as a parameter. Finally, the most costly constraint in query 7 performs expensive mathematical operations before performing a comparison. Queries 8 and 9 have very similar constraints, but differ 4.8 times in debugger invocation frequency. In this paper, by "debugger invocation frequency" we mean the frequency of events in the original program that would trigger a debugger invocation, i.e., the invocation frequency for a debugger with no overhead. Query 12 compares the parameter of the method to the distance of a point to the origin. This query combines costly mathematical operations with increased debugger invocation frequency, because its result depends on all three coordinates of Point objects.

- Queries 13–17 and 19–20 are join queries. Queries 13–16 and 19 can be evaluated using hash joins. The evaluation of queries 17 and 20 has to use nested-loop joins. For join queries, the slowdown depends both on the debugger invocation frequency and sizes of the domains. Queries 13–14 have low invocation frequencies; queries 15–17, 19–20 have high invocation frequencies. Queries 14 and 15 have large domains.

In the next section, we discuss the performance of these queries. Section 4.4.3 then discusses the efficiency benefits of incremental evaluation, custom selection code, and unnecessary assignment detection.

4.4.2 Execution Time

Figure 27 shows the program execution slowdown for application programs when queries are enabled. The slowdown is the ratio of the running time with the query active to the running time without any queries. For example, the slowdown of query 3 indicates that the Jess expert system ran 25% slower when the query was enabled.

Overall the results are encouraging. All selection queries except query 7 have overheads of less than a factor of 2. The median slowdown is 1.24. We expect overheads of common practical selection queries to be in the same range as our experimental queries; the performance model discussed in section 4.5 supports this prediction.

Join queries have overheads ranging from 2.13 to 229 for applications. Hash queries (which can be used for equality joins) are efficient for queries 13–14, and other joins are practical for query 13 in which the domains contain only 33 objects each. Queries 15–17 have large overheads because of frequent invocations (e.g., 2.6 million times per second for query 16) and large domains. Join query performance is acceptable if join domains are small, and the program invokes the debugger infrequently. For large domains and frequently invoked queries, the overhead is significant.

Microbenchmark stress-test queries 18–20 show the limits of the dynamic query-based debugger. The benchmark updates a single field in a loop 40 million times per second. When queries depend on this field, the program slowdown is significant. Selection query 18 has a

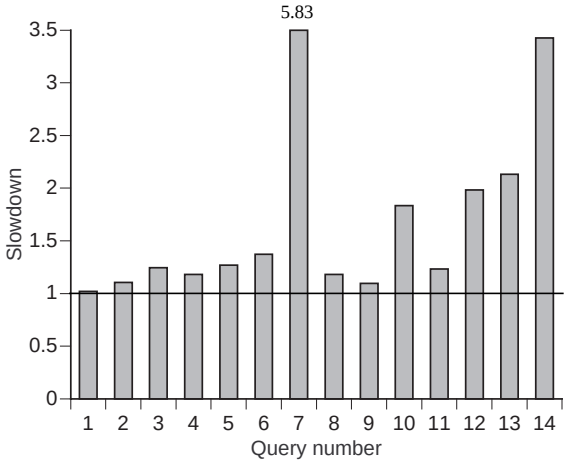

Figure 27. Program slowdown (queries 15–20 not shown)

The slowdown is the ratio of the running time with the query active to the running time without any queries. For example, the slowdown of query 3 indicates that the Jess expert system ran 25% slower when the query was enabled.

slowdown factor of 6.4, the hash-join evaluation has a slowdown of 228 times, and the slower nested-loop join that checks twenty object combinations in each evaluation has a slowdown of 930 times.

Although the microbenchmark results indicate that in the worst case the debugger can incur a large slowdown, these programs represent a hypothetical case. Such frequent field updates are possible only with a single assignment in a loop. Adding a few additional operations inside the loop drops the field update frequency to 3 million times per second which is more in line with the highest update frequencies in real programs. For such update frequencies, the slowdown is much lower as indicated by query 4. The likelihood of high update frequencies is discussed in section 4.5.

There are several parts that contribute to the query overhead:

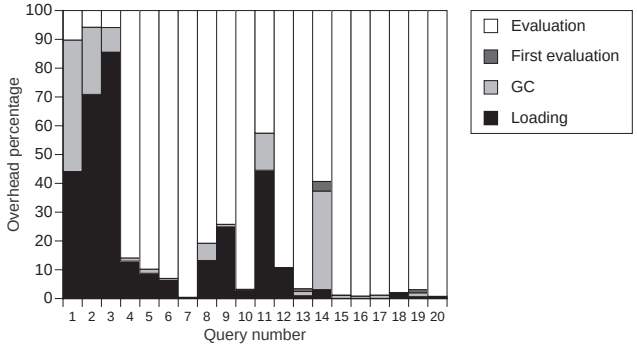

Figure 28. Breakdown of query overhead as a percentage of total overhead

For example, 3% of query 14 overhead is spent on instrumentation, 34% on garbage collection, 3% in the first evaluation, and 60% in subsequent reevaluations.

- *Loading time*, the difference between the time it takes to load and instrument classes using a custom class loader, and the time it takes to load a program during normal execution.
- *Garbage collection time*, the difference between the time spent for garbage collection in the queried program and the GC time in the original program.
- *First evaluation time*, the time it takes to evaluate the query for the first time. For join queries, the first query is the most expensive, because it sets up data structures needed for future query reevaluations. We separate this time from the rest of the query evaluation time, because it is a fixed overhead incurred only once.
- *Evaluation time*, the time spent evaluating the query. This component does not include the first evaluation time. The first evaluation time and the evaluation time together compose the *total evaluation time*.

Figure 28 shows the components of the overhead. For example, 3% of the overhead of query 14 is spent on instrumentation, and 34% on garbage collection. The total evaluation time is 63% of the overhead, with 3% spent in the first evaluation, and 60% spent in subsequent reevaluations. On average, the largest part of the overhead is the evaluation time (75.5%), while loading takes only 17% and garbage collection has a negligible overhead (less than 7%) in most cases[1]. The loading overhead becomes a significant factor when the loaded class hierarchy is large, as in query 3 on the Jess system. The loading overhead also takes a larger proportion of time when query reevaluations are infrequent or fast as in queries 1, 2, 9, and 11. Garbage collection was not a significant factor except in query 14 which creates 120,000 token objects, and in query 1 which has such a small absolute overhead that even a slight increase in GC and loading time becomes a large part of the overhead.

The experiments were executed with a version of the debugger that does not use weak references for domain collections. Using the system with weak references does not change results for selection queries, because the debugger does not track the domain collections for selection queries. Query 17 runs 40% slower because no domain objects are garbage collected, and the weak references only add needless overhead. Users can choose to use the system without weak references if they expect such program behavior. Query 15 executed faster with factor 49 overhead vs. factor 229 overhead. For this query, the domain objects become garbage and are garbage collected. So, in this case, the system with weak references provides a faster and more correct query evaluation.

The evaluation component dominates the overhead, especially in high-overhead, long-running queries, so evaluation optimizations are very important for good performance. We discuss some optimizations already reflected in this graph in the next section. Loading overhead can be further reduced by using efficient class file representations during instrumentation. Since the

[1] Experiments were run with 128M heap, a factor that decreased the GC overhead.

Query	Slowdown versus non-instrumented	Slowdown versus optimized
1. Molecule1 z. z.x > 350	1.19	1.16
2. Id x. x.type < 0	613	554
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	7135	5,725
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	475	402
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	474	373
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	587	428
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	513	88
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	275	233
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	37	33.8
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	40	21.8
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	10,500	8,496
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	17,800	8,972
13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	21.96	10.3
14. Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	1,973	576
15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	12,400	54
16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	1,708	11
17. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	697	9
18. Test5 z. z.x < 0	5,213	821
19. TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	1,491	6.6
20. TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)	5,602	6.02

Table 7. Overhead of non-incremental evaluation

loading overhead is insignificant in most cases, we did not pursue the class file handling optimizations.

4.4.3 Optimizations

To evaluate the benefit of optimizations implemented in the dynamic query-based debugger, we performed a number of experiments by turning off selected optimizations.

4.4.3.1 Incremental Reevaluation

The dynamic query debugger benefits considerably from the incremental evaluation of queries. We disabled incremental query evaluation and reran all queries. Table 7 shows the results of this experiment. The first column of numbers in the table shows the ratio of non-incremental query running time to the running time of the original program. The second column shows the ratio of non-incremental query running time to the running time of fully optimized incremental query evaluation. For example, query 2 had a factor of 613 overhead and ran for 2.5 hours. In contrast, the same query ran 554 times faster using the incremental reevaluation, had only 11% overhead, and finished in 16.4 seconds. Query 1 was the only query that the non-incremental debugger could evaluate in a reasonable time. The overheads of all other queries were enormous; some programs would have run for more than a day. (For queries 3–12 and 14–17, we stopped query reevaluation after the first 100,000 evaluations and estimated the total overhead.) Despite the large overall overhead, the individual non-incremental query evaluations are reasonably fast. For example, even for large join queries 14 and 15, a single query evaluation only took about 50 ms.

The join queries on compress have an overhead of only 9–11 compared to the incremental optimized version. These joins did not benefit much from incremental evaluation and its optimizations because the domains of these joins contain only a single object.

Overall, the experiments with non-incremental evaluation of queries show that incremental evaluation is imperative, greatly reducing the overhead, and making a much larger class of dynamic queries practical for debugging.

4.4.3.2 Custom Generated Selection Code

To estimate the benefit of generating custom code, as discussed in section 4.3.5.2, we ran all selection queries with the optimization disabled. The results of the experiment are shown in Table 8. The first column of numbers shows the slowdown of the unoptimized version compared to the original program. The second column indicates the slowdown of the unoptimized version compared to the optimized version. For example, query 4 ran 68.5 times slower than the original program and 58 times slower than the optimized query.

The ideal gas tank applet and Decaf compiler queries did not benefit from this optimization, because these programs reevaluate the query infrequently, and the optimization benefit is masked by variations in the start-up overhead. All other queries show significant speedups with

Query	Slowdown versus non-instrumented	Slowdown versus optimized
1. Molecule1 z. z.x > 350	1.05	1.03
2. Id x. x.type < 0	1.46	1.34
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	11.70	9.26
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	68.5	58
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	64	51
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	65	47
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	69.6	12
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	43.6	37
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	10.5	9.6
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	11	6
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	21	15
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	61	31
13. Test5 z. z.x < 0	1,952	307

Table 8. Benefit of custom selection code (selection queries only)

the optimization enabled. The benefit of the optimization increases with the frequency of debugger invocations; overall, custom generated selection code produces a median speedup of 15.

4.4.3.3 Same Value Assignment Test

Before evaluating a query after a field assignment, the debugger checks whether the value being assigned to the object field is equal to the value previously held by the field. Such assignments do not change the result of the query and can be ignored by the debugger.

Table 9 shows that the number of unnecessary assignments differs highly depending on the programs and fields. While some programs and fields do not have them at all, others have from 7% to 95% of such assignments. Only the ideal gas tank simulation, the Jess expert system, and the ray tracing application have unnecessary assignments to the queried fields.

To check the efficiency of the same-value test, we disabled it while leaving all other optimizations enabled. The results show that the test does not make much of a difference in query evaluation for most queries. For selections that can be evaluated fast, the cost of the same-value test is similar to the cost of the full selection evaluation. Only when the selection

Query	Slowdown versus optimized	% unnecessary assignments
1. Molecule1 z. z.x > 350	0.99	95%
2. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	0.997	7%
3. spec.benchmarks._205_raytrace.Point p. p.x == 1	0.988	15%
4. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	1.16	40%
5. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	1.61	54%
6. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	1.02	15%

Table 9. Unnecessary assignment test optimization
(excluding queries with no unnecessary assignments)

constraint is costly (as in query 4), does the same-value test reduce the overhead. For joins, the cost reduction is significant for the ideal gas tank query that contains 54% unnecessary assignments. For other joins, the percentage of unnecessary assignments is too low to make a difference.

To summarize, the test whether an assignment changes a value of a field costs only one extra comparison per debugger invocation. It does not change the overhead for most programs, but saves time when the number of unnecessary assignments is large or the query expression is expensive.

4.5 Performance Model

To better predict debugger performance for a wide class of queries, we constructed a query performance model. The slowdown depends on the frequency of debugger invocations and on the individual query reevaluation time. This relationship can be expressed as follows:

$$T = T_{original} (1 + T_{nochange} * F_{nochange} + T_{evaluate} * F_{evaluate})$$

This formula relates the total execution time of the program being debugged T and the execution time of the original program $T_{original}$ using frequencies of field assignments in the program and individual reevaluation times. The model divides field assignments into two classes:

- Assignments that do not change the value of a field. These assignments do not change the result of the query. The debugger has to perform only two comparisons in this case—a domain test and the value equality test, so it spends a fixed amount of time ($T_{nochange}$) in such invocations independent of the query. We calculated $T_{nochange}$ by running a query on a

program that repeatedly assigned the same value to the queried field; for the machine/JVM combination we used, $T_{nochange}$ = 66 ns.

- Assignments that lead to the reevaluation of a query. The time to reevaluate a query $T_{evaluate}$ for such an assignment depends on the query structure and on the cost of the query constraint expression. For each query, we calculate $T_{evaluate}$ by dividing the additional time it takes to run a program with a query into the number of debugger invocations. This calculation gives an exact result for programs that have no unnecessary assignments ($F_{nochange}$ = 0). For example, for query 18 $T_{evaluate}$ is 131ns. $T_{evaluate}$ for query 4 is 140 ns, which is close to the time to evaluate a similar query in a microbenchmark. When constraints are more costly, $T_{evaluate}$ increases; for example, for the highest cost selection query (query 10) it is 4.26 μs. It is even higher for join queries where it depends on the size of domains in joins; for example, for query 16 it is 60 μs, and for query 15 which has large domains, it is 546 μs.

Using the values of reevaluation times and the frequency of assignments to the fields of the change set, we can estimate the debugging overhead. First, we determine the typical field assignment frequency.

4.5.1 Debugger Invocation Frequency

The debugger invocation frequency is an important factor in the slowdown of programs during debugging. The program invokes the debugger after object creation and after field assignments. For most queries, the field assignment component dominates the debugger invocation frequency. To find the range of field assignment frequencies in programs, we examined the microbenchmarks and the SPECjvm98 application suite. We instrumented the applications to record every assignment to a field. Table 10 shows results of these measurements.

The maximum field assignment frequency in microbenchmarks is 40 million assignments per second, but that would be difficult to reach in an application because the microbenchmarks contain a single assignment inside a loop. The compress program has the highest field assignment frequency in the SPECjvm98 application suite, 1.9 million assignments per second. Other SPEC applications, as well as the Decaf compiler and the ideal gas tank applet, have much lower maximum field assignment frequencies.

Figure 29 shows the frequency distribution of field assignments in the SPECjvm98 applications. The left graph indicates how many fields have an assignment frequency in the range indicated on the x axis. For example, only four fields are assigned between one million and two million times per second. The right graph shows the cumulative percentage of fields that have assignment frequencies lower than indicated on the x axis; 95% of all fields have fewer than 100,000 assignments per second.

To predict the overhead of a typical selection query, we can now calculate the overhead as a function of invocation frequency. Figure 30 uses the minimum (130 ns) and maximum (4.26 μs) values of $T_{evaluate}$ from Table 10 to plot the estimated selection query overhead for a range of

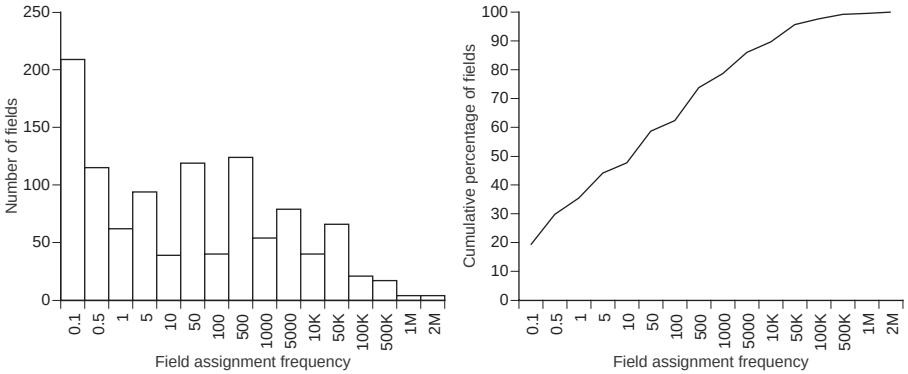

Figure 29. Field assignment frequency in SPECjvm98

invocation frequencies. For example, a selection query on a field updated 500,000 times per second would have an overhead of 6.5% if its reevaluation time was 130 ns. If the reevaluation time was 4.26 μs, the overhead will be a factor of 3.13. The graph reveals that selection queries on fields assigned less than 100,000 times a second—95% of fields—have a predicted overhead of less than 43% even for the most costly selection constraint. For less costly selections, the query overhead is acceptable for all fields.

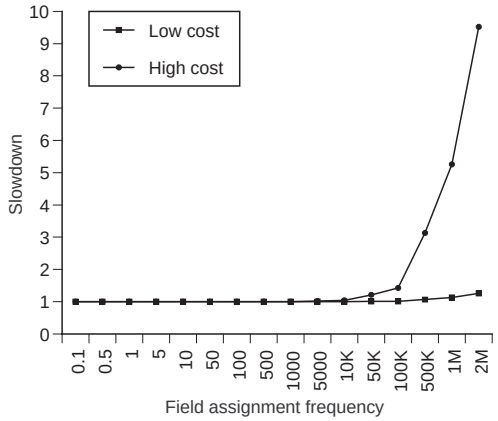

Figure 30. Predicted slowdown

The graph shows the predicted overhead as a function of update frequency. For example, the predicted overhead of a low-cost selection query on a field updated 500,000 times per second is 6.5%; the predicted overhead of a high-cost query with the same frequency is a factor of 3.13.

The worst case frequency scenario for a selection query evaluation would occur if a query referenced all fields of an application. This would imply that the application has only one class—an uncommon case. In this case, the query would be evaluated every time every field is assigned. The frequency of such evaluations is given in Table 11. Except of the compress program, the total field assignment frequencies are within the range of individual field

Query	$F_{evaluate}$ (assignments per second)	$T_{evaluate}$ (μs)
1. Molecule1 z. z.x > 350	N/A	N/A
2. Id x. x.type < 0	16,000	3.73
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	169,000	3
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	1,900,000	0.140
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0		0.208
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)		0.286
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)		3.7
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	933,000	0.193
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	196,000	0.488
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)		4.26
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	787,000	0.486
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	2,300,000	0.461
13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	N/A	N/A
14. Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	25,000	56.8
15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	350,000	546
16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	1,500,000	60
17. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	2,600,000	51
18. Test5 z. z.x < 0	42,000,000	0.131
19. TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	40,000,000	5.7
20. TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)		23

Table 10. Frequencies and individual evaluation times

assignment frequencies (lower than 2.6 million assignments per second). Consequently, we can argue that the overhead of most selection queries would be acceptable for debugging.

Application	Maximum single field assignment frequency (field assignments per second)	Total field assignment frequency (field assignments per second)	Original program execution time (s)
1. Compress	1,900,000	7,800,000	50.4
2. Jess	169,000	1,100,000	22.45
3. Db	254	897	75
4. Javac	217,000	2,600,000	38
5. Mpegaudio	495,000	2,600,000	57.4
6. Jack	27,000	214,000	27
7. Ray tracer	787,000	2,200,000	17
8. Decaf	56,000	528,000	15
9. Ideal gas tank	23,150	70,000	57
10. Microbenchmark	40,000,000	40,000,000	2.4

Table 11. Maximum field assignment frequencies

In the current model, the evaluation time $T_{evaluate}$ models all sources of query overhead. This time includes the actual reevaluation time as well as the additional garbage collection time, the class instrumentation cost, and the first evaluation cost. It would be more exact to model each of these overheads separately. However, for long running programs the evaluation time dominates the total cost, so the values of $T_{evaluate}$ are likely to fall in the range we have covered.

In summary, the performance model predicts that most selection queries will have less than 43% overhead. The model can be used as a framework for concrete overhead predictions and future model refinements.

4.6 Queries with Changing Results

So far we have discussed using dynamic queries for debugging, where the program stops as soon as the query returns a non-empty result. However, programmers can also use queries to monitor program behavior. For example, in the ideal gas tank simulation, users may want to monitor all molecule near-collisions with a query:

```
Molecule* m1 m2.    m1.closeTo(m2) && m1 != m2
```

Programmers may use this information to check the frequency of near-collisions, to find out if near-collisions are handled in a special way by the program, or to check the correspondence of program objects with the visual display of the simulation. In this case, the debugger should not

stop after the result becomes non-empty, but instead should continue executing the program and updating the query result as it changes. Such monitoring, perhaps coupled with visualization of the changing result, can help users understand abstract object relationships in large programs written by other people. How can a debugger support continuous updating of query results while the program executes?

Query	Slowdown
1. Molecule1 z. z.x < 200	1.05
2. Id x. x.type == 0	1.23
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == 0	1.3
4. spec.benchmarks._201_compress.Compressor z. z.OutCnt == 0	1.19
5. spec.benchmarks._201_compress.Compressor z. z.out_count == 0	1.09
6. Molecule1 z; Molecule2 z1. z.x < z1.x && z.y < z1.y (33x33 join)	1.47
7. Lexer l; Token t. l.token == t && t.type == 0 (120,000x600 hash join)	4.09
8. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. (p.z == ip.t) && (p.z > 100) (85,000x8,000 hash join)	212.4
9. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.out_count (1x1 hash join)	9.07
10. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < z.InCnt (1x1 join)	127
11. Test5 z. z.x % 2 == 0	45

Table 12. Benchmark queries with non-empty results

The dynamic query-based debugger described above needs only a few changes to support monitoring queries. The basic scheme and the implementation of the dynamic query-based debugger discussed in section 4.3 remain the same. The only new component of the debugger is a module that maintains the current query result. As discussed in section 4.3.5.1, the debugger reevaluates only the changed part of the query. Consequently, the result handling module must store the query result from the previous evaluation and then merge it with the new partial result. To achieve that, after query execution the debugger deletes all tuples from the previous result that contain the changed domain object and inserts the new tuples generated by the incremental reevaluation.

Experiments with queries similar to the ones in Table 5 show that adding the query result update functionality does not significantly change the query evaluation overhead (Table 12). The only exception is the microbenchmark selection query 11 which updates the query result during each

reevaluation. Consequently, the overhead of the selection increases from 6.4 times to 45 times, although part of this increase can be attributed to the more costly selection constraint. However, such frequent result updates are unlikely for most monitoring queries: programmers can only absorb infrequent result changes, so, if results change rapidly, the display will be unintelligible unless it is artificially slowed down or used off-line.

The dynamic debugger reevaluates queries whenever their results may change. Another approach useful for monitoring queries would be to reevaluate queries at regular time intervals regardless of changes. This method may be advantageous when a change set is difficult to determine or to achieve better efficiency for costly queries. However, the timer based reevaluation does not guarantee an efficient reevaluation nor does it produce all results. The transient failures may be lost if animating reevaluation is applied. We decided that such loss of results is unacceptable for debugging. However, an imprecise animation may be helpful for programmers trying to understand program dynamics. Consequently, a full implementation of the debugger could include an efficient way to reevaluate queries at regular time intervals.

To summarize, monitoring queries are useful for understanding and visualizing program behavior. With slight modifications our debugger supports monitoring queries. Unless the result changes very rapidly, the additional overhead of monitoring query execution is insignificant when compared to similar debugging queries.

4.7 On-the-fly Debugging

The current implementation of the dynamic query-based debugger requires users to specify queries before the program execution starts. Queries are enabled from the beginning of the program execution and remain active until its end. These requirements diminish the usefulness of the debugger because users cannot restrict queries to parts of the program execution and cannot ask new queries in the middle of a program run. An on-the-fly debugging implementation removes these two restrictions.

The original debugger implementation could not support on-the-fly debugging because the debugger had to know a query and its change set to instrument class files at load time. The class loader then instrumented the assignments to the monitored fields and the creations of the domain objects while loading Java class files. Class files cannot be instrumented after loading

```
    16: getstatic 133 // Get debugger activation flag
    19: ifeq 14 (33) // If debugger disabled go to bytecode 33
// If debugger enabled get debugger parameters,
// perform putfield, and invoke debugger
    22: dup2
    23: putfield 35
    26: pop
    27: invokestatic 127
    30: goto 6 (36) // Go to bytecode 36
    33: putfield 35 // Perform original putfield
    36: // end of instrumented block
```

Figure 31. On-the-fly debugging instrumentation

without changing the Java Virtual Machine. On the other hand, changing the JVM would compromise the portability of the debugger across different Virtual Machines.

On-the-fly debugging is implemented by instrumenting all constructors and all field assignments during class load time. Around each putfield code the debugger inserts a test and a call to the debugger if the debugger is enabled. If the debugger is not enabled, the program executes only two additional bytecodes per each putfield bytecode: a load of a debugger flag and a conditional jump to the original putfield. Figure 31 shows the instrumentation performed on a single putfield bytecode. The "fast path" has only two extra bytecodes. However, if the debugger is enabled, the overhead is higher. In this case, the debugger has to replicate the reference to the updated object, pass it to the debugger's run method and then invoke that method.

To support on-the-fly debugging, the debugger has to keep collections of objects belonging to all classes. These collections are necessary to evaluate queries given by users. Since the current debugging API does not allow debuggers to retrieve all objects of a class, debuggers have to track creation of all program objects. Program object tracking, although inexpensive by itself, becomes costly because of the excessive memory use—for each object created by a program, the debugger has to maintain a WeakReference object and space in the domain collection. Referring to domain objects through weak references allows the Java Virtual Machine garbage collector to collect all objects that are referenced only by the debugger. However, even though domain objects are garbage collected, the weak references themselves remain in the collection, so the collection grows as the program runs. Some programs like the gas tank simulation create so many temporary objects that weak references fill all available memory. To prevent such internal garbage, a more sophisticated implementation uses an internal "garbage collector" to recycle the weak references no longer pointing to the reachable objects. Unfortunately, the

internal garbage collection of weak references adds an additional overhead and should be used only when the program runs out of memory without it.

4.7.1 Alternative Implementations

On-the-fly debugging could be implemented using alternative techniques that may possibly increase the debugger efficiency. One approach would be to change the Java Virtual Machine. Even though we did not pursue this approach because of its lack of portability, JVM changes may lead to the most efficient implementations. These changes could be simple or sophisticated. A simple JVM change[1] would allow the debugger to retrieve all objects of a class. Such capability would remove the necessity to track all objects of all classes and would reduce both the direct object tracking overhead and the excessive memory use by weak references.

More sophisticated JVM changes would allow to instrument already loaded classes and avoid the overhead of extra bytecodes surrounding each putfield bytecode.

As mentioned above, JVM changes are not portable. An alternative technique to speed up a debugger would be to use shadow classes. In other words, while the debugger is not enabled, the program would execute the code that is instrumented to check the debugger activation only at the beginning of the methods and possibly at the back branches of the loops. When the debugger is enabled, it would generate fully instrumented versions of the classes. Such fully instrumented shadow methods would be invoked through the redirection at the beginning of the regular methods. This method reduces the overhead of the instrumented putfield execution but does not solve the problem of object tracking. Also, the debugger activation would be delayed until the instrumentation point is reached. Due to this delay, the debugger may miss some errors.

4.7.2 Experimental Results

To evaluate the on-the-fly debugger, we performed the following measurements. First of all, since programs instrumented by the debugger suffer a slowdown even when the debugger is not enabled, we measured this slowdown. Table 13 shows slowdowns together with the total field assignment frequencies for SPECjvm98 programs as well as microbenchmarks. This table indicates that adding two bytecodes after each putfield costs less than 70% for applications with a median overhead of 25% and 3.3 times for a microbenchmark[2].

If the debugger is enabled, but the query is never evaluated, for example, because domains contain only non-instantiated classes, programs suffer a larger slowdown. In this case, the instrumented byte code invokes the debugger run method. This method at the very least checks whether the changed object is a domain object. With the debugger enabled, but no query ever evaluated, the applications have a slowdown ranging up to 3.14 with a median overhead of 62%.

---

[1] Implemented for JDK 1.1.5 during the initial design of the query-based debugger.

[2] In the current JVM/JIT, the insertion of the same two bytecodes *after* the putfield bytecode instead of *in front* of it reduces overhead from 70% to 40% for compress. This phenomenon does not occur with the JIT disabled, and we cannot explain it.

Application	Total number of field assignments	Total assignment frequency (field assignments per second)	Original program execution time (s)	Disabled debugger slowdown	Enabled debugger slowdown
1. Compress	392,000,000	7,800,000	50.4	1.70	3.14
2. Jess	25,000,000	1,100,000	22.45	1.30	1.54
3. Db	67,000	897	72	1.0	1.0
4. Javac	100,000,000	2,600,000	38	1.27	1.62
5. Mpegaudio	148,000,000	2,600,000	49.5	1.25	1.96
6. Jack	5,700,000	214,000	26	1.15	1.19
7. Ray tracer	44,000,000	2,200,000	17	1.12	1.62
8. Decaf	7,900,000	528,000	15	1.15	1.40
9. Ideal gas tank	4,000,000	70,000	57	1.27	2.0
10. Microbenchmark	100,000,000	40,000,000	2.4	3.28	11.14

Table 13. On-the-fly debugging overhead

The microbenchmark slowdown is 11.14, a number increased by the fact that the microbenchmark assigns only to a long integer field which costs more to instrument. Both experiments above do not include the object tracking overhead.

Finally, if a query needs to be reevaluated, the additional slowdown to reevaluate the query depends on the query. A large part of the query reevaluation time is consumed by the domain collection maintenance and by extra garbage collection. For example, in selection query 11, 36% of the query evaluation time was due to the object collection and additional GC overhead, 17% of the time was consumed by the domain class check. Overheads for all queries are given in Table 14. Selection overhead ranges up to factor 9.5 with a median of 5.5. It is noticeable that selection query overheads almost totally depend on the program executed and neither on the query itself, nor on the query reevaluation frequency. The low cost of selection reevaluation seems to be overshadowed by large overheads of on-the-fly instrumentation, domain collection maintenance and garbage collection.

Join query overheads are very high. Query 15 was aborted after running for more than a day. However, on-the-fly debugging may be usable when programmers only need to check query results during a part of program execution.

Query	Slowdown	Invocation frequency (events / s)
1. Molecule1 z. z.x > 350	3.23	15,000
2. Id x. x.type < 0	1.83	16,000
3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	4.05	169,000
4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	6.3	1,900,000
5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	5.48	
6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	5.72	
7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	9.36	
8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	5.58	933,000
9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	5.54	196,000
10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	9.54	
11. spec.benchmarks._205_raytrace.Point p. p.x == 1	4.82	787,000
12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	4.82	2,300,000
13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	21.82	54,000
14. Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	6.4	25,000
15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	Inf	350,000
16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	384	1,500,000
17. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	263	2,600,000
18. Test5 z. z.x < 0	28	42,000,000
19. TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	935	40,000,000
20. TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)	935	

Table 14. On-the-fly query overhead

4.8 Related Work

We are unaware of other work that directly corresponds to dynamic query-based debugging. Extensions of object-oriented languages with rules as in R++ [128] provide a framework that allows users to execute code when a given condition is true. However, R++ rules can only reference objects reachable from the root object, so R++ would not help to find the javac errors we discussed. Due to restrictions on objects in the rule, R++ also does not handle join queries.

A significant amount of work exists on general rule-based extensions of OO languages. Differently from R++, such systems use OPS5 [33][63] for rules, which allow full join semantics including negation (non-existence of objects satisfying the given constraint) in rules. Most systems extend C++ or Java [29][65][80][94][145]. Such systems could serve as foundations for query-based debugger implementations, though currently they are not used for debugging. However, rule-based extensions of object-oriented languages significantly differ from query-based debugging. Most of these systems do not handle method invocations in rules. Only some of them, like RAL/C [65] and OPSJ [145] do allow method invocations. Most implementations are based on source code instrumentation. These systems also require prescribed programming styles or necessitate writing interface code. For example, ILOG and RETE++ generate class skeletons that have to be used by programmers for their classes; OPSJ preprocesses Java code and recompiles affected classes. The CLIPS compiler and ILOG rules require additional interface code. In ILOG, programmers have to supply and indicate methods setting and reading object fields. On the other hand, the dynamic query-based debugger does not require source code to be available; it also automatically determines change set and instruments necessary constructors and field assignments.

The systems usually implement RETE [64] rule-matching and activation algorithm or its extensions [28] and optimizations [3][55]. This algorithm does not consider different join orders. However, some other algorithms proposed in rule-based language implementations optimize join ordering or even execute all joins in parallel [142]. Rule-based systems also employ specialized code generation for efficient rule evaluation.

Speed comparisons between the query-based debugger and rule-based language extensions are difficult. Most of the rule system benchmarks focused on the number of simultaneously supported rules in slowly changing rule-only programs. Such an environment is very different from rapidly changing object graphs in object-oriented programs. Only recent measurements [145] try to evaluate the performance of object programs extended with rules.

In addition to the research discussed in the background section (section 2), Sefika et al. [149][150][151][152] implemented a system allowing limited, unoptimized dynamic selection queries about objects in the Choices operating system. The Choices visualizer shows users all instances of a given class that satisfy a certain property. The view is animated as the program executes, however the granularity of animation has to be at a method-call level or coarser. Choices implementation uses internal Class objects to track all objects of all classes. The object

classes themselves are “aware” of the visualization and provide views corresponding to their function, e.g. CPU activity shows utilization ratio. Unlike a query-based debugger, the application (Choices) is specifically instrumented to allow queries.

Dynamic query-based debugging extends work on data breakpoints [180]—breakpoints that stop a program whenever an object field is assigned a certain value. Pre-/postconditions and class invariants as provided in Eiffel [132] can be thought of as language-supported dynamic queries that are checked at the beginning or end of methods. Unlike dynamic queries, they are not continuously checked and they cannot access objects unreachable by references from the checked class. Dynamic queries could be used to implement class assertions for languages that do not provide them. The current implementation of dynamic queries cannot use the “old” value of a variable, as can be done in postconditions. We view the two mechanisms as complementary, with queries being more suitable for program exploration as well as specific debugging problems.

Consens et al. [44][45] use the Hy^+ visualization system to find errors using post-mortem event traces. De Pauw et al. [52] and Walker et al. [182] use program event traces to visualize program execution patterns and event-based object relationships, such as method invocations and object creation. This work is complementary to ours because it focuses on querying and visualizing runtime events while we query object relationships.

Dynamic queries are related to incremental join result recalculation in databases [26][34]. Buneman and Clemons [34] introduced the notion of database *alerter*, a program that monitors database for changes. The authors discuss complex alerters that monitor several relations. Such alerters are similar to join queries in debugging. Buneman and Clemons observe that some updates are ignorable independently of the relation contents. Our debugger uses similar ideas to ignore updates to irrelevant domains and fields. Blakeley, Larson, and Tompa [26] discuss the conditions under which a change in the database does not affect the view. The authors also propose the algorithms to incrementally update database views. We use insights of this work to implement the incremental query evaluation scheme. First, the dynamic debugger tries to detect and discard irrelevant events. Second, it uses the incremental reevaluation techniques to update the query result. Stonebraker [161] proposes query (update) rewriting techniques to preserve integrity constraints in databases. Such integrity constraints are related to database views and dynamic debugging queries. Coping with inter-object constraints in the extended ODMG model [24] may require methods similar to dynamic query-based debugging.

Active databases (POSTGRES, HiPAC) [21][130] include the notion of triggers (event-condition-action triples) that are similar to the dynamic queries. Some active databases allow only conditions relating to a single relation or a class in object-oriented databases. Ariel [81] contains OPS5 style rules with optimized A-TREAT matching algorithm. The A-TREAT incorporates index based selection evaluations and space-saving join optimizations. Since Ariel’s rules essentially mimic the OPS5 rule structure, comments made about rule-based programming languages apply to it as well. ADAM [54], Ode [10][71][72][126], and Sentinel

[15] integrate triggers with object-oriented programs. ADAM and Sentinel establish flexible frameworks for events and rules by treating them as first-class objects. Ode associates triggers with object classes. However, Ode's triggers cannot refer to objects outside of a class, i.e., Ode does not allow "join" triggers. Sentinel supports multi-class queries, but join evaluation is left to the implementor of the rules. Consequently, the join-query support is not adequate.

Ode and Sentinel allow method invocations in triggers, a feature similar to debugging queries. In addition, triggers in these systems can contain temporal relationships that are not supported in dynamic queries. For example, the trigger can be activated if two events occur one after another. Ode's triggers can be also set on "read" and "transaction" events. Such events are not used for dynamic queries, because they do not change the system state. Finally, Ode implements triggers by compiling the associated O++ (C++ extension) code.

Starburst [189][190][191] is a relational database system with active rules that can contain general SQL queries in their conditions. Starburst does not allow incremental evaluation of rule conditions, although an extension allowing incremental evaluation has been proposed [20]. Unlike dynamic queries and most other active databases, Starburst rules are evaluated at the end of operation blocks usually corresponding to transactions. Such semantics provide one way of coping with the consistency problem (Section 6.2.1).

While it is difficult to evaluate the efficiency of rule matching in active databases, it is clear that they have low overhead because they are invoked infrequently. First, the events contain only certain method activations. Second, the databases change much more slowly than object-oriented programs. In contrast, in query-based debugging, the cost of all reevaluations is important. I.e., an optimization that reduces the cost by even a little can save a lot of overhead if this reduction affects all reevaluations. For that reason, customized selection code significantly speeds up query reevaluations. With rapidly changing objects and initially unknown domain selectivities, a full-fledged debugger could optimize the join ordering during runtime. Costs and benefits of such optimization have not been evaluated.

4.9 Summary

The cause-effect gap between the time when a program error occurs and the time when it becomes apparent to the programmer makes many program errors hard to find. The situation is further complicated by the increasing use of large class libraries and complicated pointer-linked data structures in modern object-oriented systems. A misdirected reference that violates an abstract relationship between objects may remain undiscovered until much later in the program's execution. Conventional debugging methods offer only limited help in finding such errors. Data breakpoints and conditional breakpoints cannot check constraints that use objects unreachable by references from the statement containing the breakpoint.

We have described a dynamic query-based debugger that allows programmers to ask queries about the program state and updates query results whenever the program changes an object

relevant to the query, helping programmers to discover object relationship failures as soon as they happen. This system combines the following novel features:

- An extension of static query-based debugging to include dynamic queries. Not only does the debugger check object relationships, but it determines exactly when these relationships fail while the program is running. This technique closes the cause-effect gap between the error's occurrence and its discovery.

- Implementation of monitoring queries. The debugger helps users to watch the changes in object configurations through the program's lifetime. This functionality can be used to better understand program behavior.

The implementation of the query based debugger has good performance. Selection queries are efficient with less than a factor of two slowdown for most queries measured. We also measured field assignment frequencies in the SPECjvm98 suite and showed that 95% of all fields in these applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation time estimates, our debugger performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. Join queries are practical when domain sizes are small and queried field changes are infrequent.

Good performance is achieved through a combination of two optimizations:

- Incremental query evaluation decreases query evaluation overhead by a median factor of 160, greatly expanding the class of dynamic queries that are practical for everyday debugging.

- Custom code generation for selection queries produces a median speedup of 15, further improving efficiency for commonly occurring selection queries.

On-the-fly dynamic debugging has been implemented but currently suffers a high overhead. Selection slowdowns range up to factor 9.5 with a median of 5.5. Further optimizations could reduce this overhead.

By combining a novel idea of query-based programming with dynamic query result updates and by providing an efficient implementation, we have shown that dynamic query-based debugging is a practical debugging tool.

5 Query Analysis and Classification

"Is it just me, or does it seem to you that I get more than my share of troubles?"

Job

"What did I do wrong?"

Lear, Rex

5.1 Introduction

Previous chapters of this dissertation motivated and proposed use of queries for program debugging, introduced a query model, and described implementations of static and dynamic query debuggers. This section explores the breadth of object relationships and corresponding queries in software systems. By listing numerous queries applicable to the programs from different domains, this section investigates typical use of queries by programmers. Even though queries in this section were conceived solely by the author (sometimes in a discussion with other programmers), the query list covers a wide variety of domains and structures. Such a list is helpful in selecting queries for experiments, for establishing common query structures and requirements for debugger implementations. According to the requirements, different debugger features may have different implementation and optimization priorities. Furthermore, classes of similar queries from different programming domains can be summarized into query patterns that may warrant special debugger support.

Our investigation of the query catalogue suggests that query-based debugging is widely applicable, and that the current query model and its implementation support most queries. The research also reveals additional features that should be supported in a full implementation of a debugger.

5.2 Queries in Software Systems

This section explores object relationships and related queries in a wide range of applications. The queries cover loosely defined domains, such as networks, graphical user interfaces, programming systems, simulations, and resource management systems. The queries were asked both on real programs and as thought experiments. The author designed most queries by investigating programs or system models. However, some queries were suggested by other programmers or were paraphrased from the existing testing code.

Query grouping by domain indicates that although queries do not strictly depend on the application domain, similar queries can be asked about different applications of the same domain.

A table summarizing all queries can be found at the end of the section. The queries that were asked about Self programs are written in Self syntax. All other queries use Java syntax.

5.2.1 Networks

The following sections illustrate query use in various network simulations and network protocols.

5.2.1.1 Simulation of a Cellular Communication Network.

The program in this section simulates a cellular communication network [168]. The program was not implemented but only outlined. The main components of the network are mobile users and base stations in a cell grid. Users periodically request and release communication channels. Base stations allocate channels for users using one of the simulated protocols. To avoid interference, the same channel cannot be used in the neighboring cells. Additionally, various simulated faults can occur in the network.

In this system, there are a number of interesting queries that can be either answered at a breakpoint or monitored while the program is running. The query

```
baseStation b; channel c.  b.usedChannels.contains(c)
```

shows the basic view of all base stations and channels that are currently used by base stations. This query can be answered during a breakpoint or used to visualize the station-channel assignments. The query

```
baseStation b1 b2; channel c.
b1.usedChannels.contains(c) && b2.usedChannels.contains(c) && b1.isNeighbor(b2)
```

checks whether the neighborhood cells do not use the same channel. This query can be used to verify station-channel assignments while the program is running. Furthermore, queries can check whether clients and base stations have the same view on the channel assignment:

```
assignments a; baseStation b; channel c1 c2; client cl.
b.assignments.contains(a) && a.client == cl && a.channel == c1 &&
cl.assignedChannel == c2 && c1 != c2
```

This query can be used to verify the consistency of station-channel assignment with client-channel assignment. To detect handling of client failures one may ask a query

```
client cl. cl.active && (cl.currentTime - cl.channelRequestTime > LIMIT)
```

that shows all clients holding channels longer than a set limit. This query would show the clients that are faulty during the program execution.

5.2.1.2 Token-Based Network

In a token-based network system, the queries would check the correctness of token use. The token in a network should be unique. If there are different tokens for different groups on the network, there should be only one token per group. Enforcing these constraints can be done with the following dynamic queries.

Is there more than one token in the system?

```
Token t1 t2. t1 != t2
```

Is there more than one active token in the system?

```
Token t1 t2. t1.active && t2.active && t1 != t2
```

Is there more than one active token belonging to the same group? There are two different queries that can answer this question:

```
Token t1 t2. t1.active && t2.active && t1.group == t2.group && t1 != t2
```

```
Group g1; Token t1 t2. t1.active && t2.active && g1.contains(t1) && g1.contains(t2) &&
t1 != t2
```

Does the same node have different tokens?

```
Node n; Token t1 t2. n.tokens.contains(t1) && n.tokens.contains(t2) && t1 != t2
```

```
Node n; Token t1 t2. t1.belongsTo(n) && t2.belongsTo(n) && t1 != t2
```

5.2.2 Graphical User Interfaces

Graphical user interfaces consist of numerous objects interacting with each other. Consequently, they provide a rich source for query-checked constraints.

5.2.2.1 The Self Graphical User Interface

The queries about the Self user interface were discussed in section 3.2.3.1. This section recaps the queries. Finding all morphs directly contained in at least two morphs:

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

Are row morphs usually embedded into column morphs or vice versa? The following two queries give insight into this question:

```
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)
```

We can find object outliners that contain column morphs and the ones that contain row morphs:

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

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

The count of tuples in the result can be used to calculate the number of outliner referenced column morphs that do not contain row morphs.

At last we find object outliners containing no morphs at all:

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

Additional queries about morph organization into hierarchical structures:

```
objectOutliner a; smallEditorMorph b. (a titleEditor = b) && (b owner = a)
```

```
objectOutliner a; columnMorph b; labelMorph c.
(a morphs includes: b) && (c owner = b) && (a moduleSummary = c)
```

Apart from the Self GUI, queries can be asked about other GUI systems. Relationships among windows, widgets, rulers, and menu bars can be explored. For example, assume that a graphical widget references its parent window, and that this parent window must in turn reference the enclosed widget. This relationship can be verified by the query

```
widget wid; window win.
wid.window == win &&
! win.widget_collection.contains(wid)
```

5.2.2.2 Graphical Object Properties

Graphical programs (user interfaces, painting, CAD, and picture manipulation programs) use simple graphical objects such as points, rectangles, and lines to build more complicated objects. Even though low level graphical objects are simple, relationships between them can be complex. Additionally, points and lines are usually largest classes in graphical programs, so finding objects violating constraints by hand is not easy. For example, the Self graphical user interface may at times contain more than ten thousand points and rectangles. Here are some queries that verify various interobject constraints.

Find a point and a rectangle, such that the point has the same x coordinate as the y coordinate of the origin of the rectangle, and the point's x coordinate is fixed:

```
point a; rectangle b. (a x = b origin y) && (a x = 6)
```

Find a point with a certain x coordinate:

```
point a. a x = 256
```

Find a point and a rectangle with the same relationship as in the first query, but the rectangle has to also have height of 1000, and there exists another rectangle of the same height:

```
point a; rectangle b b1. (a x = b origin y) && (b height = b1 height) && (b != b1) &&
    (b1 height = 1000)
point a; rectangle b b1. (a x = b origin y) && (b height = b1 height) && (b != b1) &&
    (b1 height > 1000)
```

Find two rectangles such that one of them has much smaller height than the other, but much larger width:

```
rectangle b b1.  (b height > (b1 height + 800)) && (b width < (b1 width - 900))
```

5.2.2.3 SPECjvm98 Ray Tracer

The ray tracing program from the SPECjvm98 benchmark suite creates 85,000 point objects and 8,000 intersection point objects while rendering a scene. Several queries can be asked about the program objects.

Can a z coordinate of an intersection point be negative?

IntersectPt ip. ip.Intersection.z < 0

Are there any points with coordinate x equal to 1?:

spec.benchmarks._205_raytrace.Point p. p.x == 1

Are there any points with the distance from the origin greater than 100 million?

spec.benchmarks._205_raytrace.Point p. p.farther(100000000)

Are there such points with a negative z coordinate equal to the t field of IntersectPt?

spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip.
p.z == ip.t && p.z < 0

5.2.3 Programming Systems

Programming systems such as compilers and runtime libraries perform complicated data structure creations and transformations. It is important that the structures remain correct through the program's execution because errors could get compounded by later transformations. In addition, runtime system errors may be difficult to detect because they appear only under certain conditions. Subtle errors may only influence the underlying system's efficiency without breaking its "correctness." This section discusses several programming systems.

5.2.3.1 Self Virtual Machine

One way to test object relationships involves using special testing code written for each application. For example, the Self virtual machine [88][89] contains over 10,000 lines of testing-related C++ code. After examination of the Self VM testing code, it appears that queries could have been used to verify the VM more efficiently. Some verification code iterates through the objects of a class checking simple properties. Such code could have been expressed by queries. For example, the verification code of ByteVectorOopClass checks whether the object has correct length and byte array. Other testing methods check for length and sizes of various data structures. These properties can be tested using simple queries

ByteVectorOop a. a.length() < 0

ByteVectorOop a. a.bytes() == NULL

objVectorOop a. a.length() < 0

oopsOop a. a.size() < 1

Similarly fctProxyOop is tested for

foreignOop a. a.addrs()->noOfArgs->is_smi()

Verifying that C pointers point to correct objects:

C_pointer a. a.hi->is_smi()

memOop a. a.mark()->is_mark()

Queries and testing code can coexist through the query use of verification methods.

```
memOop a. a.verify_oop()
stringOop a. !a.is_old()
stringOop a. a != a.Memory->string_table->lookup(a.bytes(), a.length())
vFrameOop a. a.is_live() && (!a.method()->has_code())
vFrameOop a. a.is_live() && (!a.oop(a.locals())->is_smi())
```

An important part of answering these queries is reporting. When some of these invariants are violated, the programmer may not want to see just the resulting object collection, but may want to receive an explanatory message. To support such reporting, the full implementation of the debugger should have auxiliary output facilities that integrate the result collection with custom messages.

5.2.3.2 Understanding the Cecil Compiler

The analysis of a prototype Cecil [38] compiler written in Self by Craig Chambers, Jeff Dean, and David Grove is reported in section 3.2.3.2. This section summarizes the queries used. The compiler can be explored by finding compiler objects corresponding to Cecil constructs in the compiled Cecil program and determining common use patterns. For example, a simple Cecil program compiled by the compiler 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)
```

Query finding Cecil methods returning integers:

```
cecil_method a. (a resultTypeSpec printString = 'int')
```

All of these queries could be asked while the program is running to visualize the result set.

Another query shows that Cecil programs can 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)
```

Finally, a query checks whether a Cecil object's context includes a variable binding such that the bindings' value is the same Cecil object:

```
cecil_named_object a; cecil_top_context b; cecil_object_binding c.
(a defining_context = b) && (b varBindings includes: c) && (c value = a)
```

Queries can be used to monitor object relationships in other object-oriented compilers and parsers. Monitoring relationships between tokens in the input stream may help users to understand the compiler's error detection techniques.

5.2.3.3 Javac Compiler

The javac Java compiler, a part of Sun's JDK distribution builds an abstract syntax tree (AST) of the compiled program. Several queries about the state of the AST were discussed in the introduction and section 4.1. The first query finds an AST corrupted by an operation that assigns the same expression node to the field right of two different parent nodes:

```
BinaryExpression* e1, e2.    e1.right == e2.right && e1 != e2
```

The javac compiler also maintains a constraint that a FieldExpression object that shares the type and the identifier name with a FieldDefinition object must reference the latter through the field field:

```
FieldExpression fe; FieldDefinition fd.
fe.id == fd.name && fe.type == fd.type && fe.field != fd
```

5.2.3.4 Decaf Compiler

The Decaf Java subset compiler was written at UCSB for an undergraduate compiler course. This compiler parses a Java program and generates executable byte code. Queries about the compiler check properties of objects corresponding to the compiled program objects. For example, the following queries check whether there are identifier objects with negative and zero type fields:

```
Id x.    x.type < 0
Id x.    x.type == 0
```

The following query checks whether the lexical analyzer finds any error tokens:

```
Lexer l; Token t.    l.token == t && t.type == ERROR
```

Similar query checks whether the lexical analyzer finds any uninitialized tokens:

```
Lexer l; Token t.    l.token == t && t.type == UNINITIALIZED
```

The number of tokens in a compiled program may be large, making these queries difficult to check by hand.

5.2.3.5 Jess Expert System

The expert system belonging to the SPECjvm98 benchmark suite reads in a rule-based program and executes it. The following query finds tokens with negative sortcodes:

```
spec.benchmarks._202_jess.jess.Token z.    z.sortcode == -1
```

5.2.4 Games and Simulations

Computer games and computer simulations create webs of objects rich with constraints. Simulations of airline routing systems and manufacturing plants should be performed precisely to avoid costly design updates during their physical implementation. This section describes errors that can be found in simple games and more complicated simulations.

5.2.4.1 Tic-Tac-Toe

Some queries can be asked about an implementation of the tic-tac-toe game. To make a move the program has to find an empty cell. The query can check whether such a cell exists:

```
Cell a. a.value == Empty
```

In certain situations, there exist moves that would win the game. A query can be used to find such moves. The following query determines the winning move that would fill a column:

```
Cell a b c. a.value == b.value && a.x == b.x && b.x == c.x && c.value == Empty &&
    a.value != Empty
```

Another three queries would be necessary to check for winning moves in the rows and diagonals.

5.2.4.2 Chess

Chess is a game that has numerous interactions between different pieces on the board. This section checks only some positions that can occur on a virtual chessboard. For example, a query can be used to find out if one side has put the other in “check”:

```
King k; Figure * f. f.attacks(k)
```

Another query can be used to check whether both rooks are in the same column:

```
Rook r1 r2. r1.color == r2.color && r1.x == r2.x && r1 != r2
```

5.2.4.3 Ideal Gas Simulation

There are several ideal gas simulation programs implemented in Self and Java. These applets simulate a tank with ideal gas molecules moving in the tank and colliding with the tank walls and each other. Some of the queries were discussed in section 4.2.1. All gas molecules have to remain within the tank:

```
Molecule* m.    m.x < 0 || m.x > X_RANGE || m.y < 0 || m.y > Y_RANGE
```

Molecules should not occupy the same position as other molecules:

```
Molecule* m1 m2.    m1.x == m2.x && m1.y == m2.y && m1 != m2
Molecule1 z; Molecule2 z1.
    z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius
atom a b. a.center == b center && a != b
```

5.2.5 Resource Management Systems

A loosely coupled collection of applications that can be called resource management systems deals with entities managing different resources. The following sections describe some of the applications.

5.2.5.1 Views and Users

In this section, we consider a system in which users simultaneously subscribe, read, and edit different views. One such system is the Usenet news system. As it is known, the number of views can be measured in thousands, and the number of users can reach tens and hundreds of thousands. A programmer trying to debug such a system needs to be able to check how users interact with the views. For example, to visualize users and views to which the users subscribe, programmer may ask a query:

```
View v; User u. u.subscribesTo(v)
```

All views subscribed to or edited by at least two users are given by:

```
View v; User u z. u.subscribesTo(v) && z.subscribesTo(v) && u != z
View v; User u z. u.edits(v) && z.edits(v) && u != z
```

Are users reading different views at the same time?

```
View v1 v2; User u. u.reads(v1) && u.reads(v2) && v1 != v2
```

Do different users subscribe to multiple same views?

```
View v1 v2; User u z. u.subscribesTo(v1) && z.subscribesTo(v1) && u != z &&
    u.subscribesTo(v2) && z.subscribesTo(v2) && v1 != v2
```

Are the same views edited by at least two users from the same site?

```
View v; User u z. u.edits(v) && z.edits(v) && u != z && u.site == z.site
Site s; View v; User u z. u.edits(v) && z.edits(v) && u != z && s.hosts(u) && s.hosts(z)
```

5.2.5.2 Room Scheduling System

A scheduling system of university rooms has reservation relationships between rooms and courses during different time slots. For example, the debugger can check whether the same group reserved two different rooms at the same time:

```
Room r1 r2; Group g1; Slots s1 s2. r1.slots.contains(s1) && r2.slots.contains(s2) &&
    r1 != r2 && s1.time == s2.time && g.reservations.contains(s1) &&
    g.reservations.contains(s2)
Room r1 r2; Slots s1 s2. r1.slots.contains(s1) && r2.slots.contains(s2) &&
    r1 != r2 && s1.time == s2.time && s1.group == s2.group
```

5.2.5.3 Process and Resource Simulation

Find all tasks with indices less than ProcessedIndex that have not been executed by any processes:

```
task t; process p. ∀p (! p.execute.contains(t)) && t.index < ProcessedIndex
```

This query requires extension of the query model to include a universal quantifier.

5.2.5.4 Airline Plane Routing Service

The software system discussed here is a flight reservation and plane routing system. Queries may check an assignment of a plane to a flight. For example, a query may require a plane to have enough seats for all passengers and a return flight to the same airport:

```
Flight f f1; Plane p. f.seatsBooked < p.seatsInPlane && f.assigned(p) &&
    f1.seatsBooked < p.seatsInPlane && f1.unassigned && f1.departure > f.arrival &&
    f.destination == f1.startingPoint && f1.destination == f.startingPoint
```

Simpler queries may find tickets issued by the airline to a destination not served by the airline and tickets in which flight arrival time in a multi-leg trip is later than the next flight departure time:

```
Ticket t; Airline a. ! a.services(t.destination)
Ticket t. t.firstFlight.arrival > t.secondFlight.departure
```

5.2.6 Miscellaneous Programs

This section covers programs that did not fit into the categories above.

5.2.6.1 VLSI Layout Programs

A simple gate and path layout program coded in Smalltalk allows users connect gates into a circuit and test it with various inputs. The following query checks for connections attached to both input and output of a gate.

```
gate a; connection b. a.output == b && a.inputs.contains(b)
```

The program might be intelligent enough to prohibit connecting a “true” value to an “or” gate, and a “false” value to an “and” gate. The following queries check these conditions:

```
trueElement a; connection b; orGate c. a.output == b && c.inputs.contains(b)
falseElement a; connection b; andGate c. a.output == b && c.inputs.contains(b)
```

5.2.6.2 Java Animator

The Java Animator applet shows a slide show of series of images stored in a collection. Queries check whether the image collection, the image duration collection, and the image name hash table have the same size:

```
Animator an; Vector images; Hashtable imageNames.
    images.size() != imageNames.size() && an.images == images &&
    an.imageNames == imageNames
Animator an. an.images.size() != an.imageNames.size()
Animator an. an.images.size() != an.durations.size()
```

Both image and image name collections should contain the same images:

```
Animator an; Image im. an.images.contains(im) && (!an.imageNames.containsKey(im))
```

5.2.6.3 SPECjvm98 Compress

Compress is a straightforward compression and decompression implementation that does not use a lot of object-oriented programming. However, programmers still can use queries to debug this program. For example, a query can find whether the output count of the output buffer is negative:

```
spec.benchmarks._201_compress.Output_Buffer z.    z.OutCnt < 0
spec.benchmarks._201_compress.Output_Buffer z.    z.count() < 0
```

These two queries check the same constraint using two different methods: comparing a field against a constant and invoking a method.

Similar queries can be asked about input and output counts of a Compressor object:

```
spec.benchmarks._201_compress.Compressor z.    z.in_count < 0
spec.benchmarks._201_compress.Compressor z.    z.out_count < 0
```

Similar queries check for the program points where the out count is equal to zero:

```
spec.benchmarks._201_compress.Compressor z.    z.OutCnt == 0
spec.benchmarks._201_compress.Compressor z.    z.out_count == 0
```

Queries can check the relationships between the input count of the input buffer and the output count of the output buffer. The first query checks whether there is a point in the program when the output count is less than 100, the input count is greater than 0, and they are equal:

```
spec.benchmarks._201_compress.Input_Buffer z;
    spec.benchmarks._201_compress.Output_Buffer z1.
    z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0
```

We can also check when the output count is smaller than the input count:

```
spec.benchmarks._201_compress.Input_Buffer z;
    spec.benchmarks._201_compress.Output_Buffer z1.
    z1.OutCnt < z.InCnt
```

Another query finds out whether the output count of the output buffer is less than 100, the output count of the compressor is greater than 0, and the output buffer output count is ten times larger than the output count of the compressor:

```
spec.benchmarks._201_compress.Compressor z;
    spec.benchmarks._201_compress.Output_Buffer z1.
    z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count
```

The following query checks when the output counts of compressor and buffer are equal:

```
spec.benchmarks._201_compress.Compressor z;
    spec.benchmarks._201_compress.Output_Buffer z1.    z1.OutCnt == z.out_count
```

5.2.7 Query Summary

Table 15 summarizes the queries posed in previous sections. Query attributes are shown using letters in tables' columns. The first column of the table contains the query string. The second column categorizes queries as assertions or visualizations. Queries that have no results unless an error occurs are called *assertion (A)* queries, while the queries that have results even in correct executions are called *visualization (V)* queries. The distinction between two sets is important for the debugger implementation, because the debugger does not have to maintain the query result set for assertion queries. Consequently, assertion and visualization query evaluation costs may differ.

The third column indicates whether the query is a *selection (S)*, a *hash (H)* join, or a *nested-loop (N)* join. As discussed in section 3.3, join queries are more costly to evaluate and pose an optimization challenge. The query catalogue may be biased towards the join queries because the author tried to come up with complex queries.

The special requirement column specifies requirements for the query evaluation in a debugger. The requirements are grouped into four groups. Some queries can be evaluated only if the debugger can execute the *methods (M)* of the underlying programming language. Without method execution, the expressiveness of the query language would be reduced. The current implementation can perform method evaluations with a user-supplied change set. Some queries contain references to object *collections (C)*. Though the static Self query-based debugger allows the use of collections, the dynamic Java debugger does not currently support collections because most collections are system classes. The Java debugger also does not handle arrays. The *system class (S)* support is necessary to debug applications that interact with library data structures. The Java debugger does not support debugging of system classes, while in the Self world there is no distinction between library and user classes. Finally, some queries need universal or existential *quantifiers (Q)*. The current query model does not provide such support.

The last column of the table classifies queries according to a scheme discussed in the next section.

The table provides a handy reference for queries of different domains, structures and requirements. For example, query 1 is a visualization query that uses nested-loop join, and requires method invocations and collection handling.

Query	Assertion/ Visualization	Selection/Join (Nested/Hash)	Special requirements	Query classification
1. baseStation b; channel c. b.usedChannels.contains(c)	V	N	C, M	P
2. baseStation b1 b2; channel c. b1.usedChannels.contains(c) && b2.usedChannels.contains(c) && b1.isNeighbor(b2)	A	N	C, M	D

Table 15: Query examples

Query	Assertion/ Visualization	Selection/Join (Nested/Hash)	Special requirements	Query classification
3. assignments a; baseStation b; channel c1 c2; client cl. b.assignments.contains(a) && a.client == cl && a.channel == c1 && cl.assignedChannel == c2 && c1 != c2	A	N, H	C, M	D, P
4. client cl. cl.active && (cl.currentTime - cl.channelRequestTime > LIMIT)	V	S		D
5. Token t1 t2. t1 != t2	A, V	N		D
6. Token t1 t2. t1.active && t2.active && t1 != t2	A, V	N		D
7. Token t1 t2. t1.active && t2.active && t1.group == t2.group && t1 != t2	A	N, H		D
8. Group g1; Token t1 t2. t1.active && t2.active && g1.contains(t1) && g1.contains(t2) && t1 != t2	A	N	M	D
9. Node n; Token t1 t2. n.tokens.contains(t1) && n.tokens.contains(t2) && t1 != t2	A, V	N	C, M	D
10. Node n; Token t1 t2. t1.belongsTo(n) && t2.belongsTo(n) && t1 != t2	A, V	N	M	D
11. morph * a b c. (a morphs includes: b) && (c morphs includes: b) && (a != c)	A	N	C, M	P
12. objectOutliner a; rowMorph b; columnMorph c. (a morphs includes: b) && (b morphs includes: c)	V	N	C, M	D
13. objectOutliner a; rowMorph c; columnMorph b. (a morphs includes: b) && (b morphs includes: c)	V	N	C, M	D
14. objectOutliner a; columnMorph b. (a morphs includes: b)	V	N	C, M	D
15. objectOutliner a; rowMorph b. (a morphs includes: b)	V	N	C, M	D
16. objectOutliner a. (a morphs size = 0)	V	S	C, M	P
17. objectOutliner a; smallEditorMorph b. (a titleEditor = b) && (b owner = a)	V	H		P
18. objectOutliner a; columnMorph b; labelMorph c. (a morphs includes: b) && (c owner = b) && (a moduleSummary = c)	V	H, N	C, M	P
19. widget wid; window win. wid.window == win && (! win.widget_collection.contains(wid))	A	H, N	C, M	P
20. point a; rectangle b. (a x = b origin y) && (a x = 6)	V	H		P
21. point a. a x = 256	V	S		P
22. point a; rectangle b b1. (a x = b origin y) && (b height = b1 height) && (b != b1) && (b1 height = 1000)	V	H, N		P
23. point a; rectangle b b1. (a x = b origin y) && (b height = b1 height) && (b != b1) && (b1 height > 1000)	V	H, N		P
24. rectangle b b1. (b height > (b1 height + 800)) && (b width < (b1 width - 900))	V	H, N		P
25. IntersectPt ip. ip.Intersection.z < 0	A	S		P
26. spec.benchmarks._205_raytrace.Point p. p.x == 1	A	S		P

Table 15: Query examples

Query	Assertion/ Visualization	Selection/Join (Nested/Hash)	Special requirements	Query classification
27. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	A	S	M	D
28. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0	A	H		P
29. ByteVectorOop a. a.length() < 0	A	S	M	P
30. ByteVectorOop a. a.bytes() == NULL	A	S	M	P
31. foreignOop a. a.addrs()->noOfArgs->is_smi()	A	S	M	D, P
32. C_pointer a. a.hi->is_smi()	A	S	M	D, P
33. memOop a. a.mark()->is_mark()	A	S	M	D, P
34. memOop a. a.verify_oop()	A	S	M	D
35. objVectorOop a. a.length() < 0	A	S	M	P
36. oopsOop a. a.size() < 1	A	S	M	P
37. stringOop a. !a.is_old()	A	S	M	D
38. stringOop a. a != a.Memory->string_table->lookup(a.bytes(), a.length()	A	S	M	D
39. vFrameOop a. a.is_live() && (!a.method()->has_code())	A	S	M	D
40. vFrameOop a. a.is_live() && (!a.oop(a.locals())->is_smi())	A	S	M	D
41. cecil_named_type a. a instantiations size != 0	V	S	C	D, P
42. cecil_named_type a. a subtypes size != 0	V	S	C	D, P
43. cecil_method a. a resultTypeSpec printString = 'int'	V	S	M	D, P
44. cecil_method a b; cecil_formal c d. (a formals includes: c) && (b formals includes: d) && (c name = d name) && (c != d) && (a != b)	V	H, N	C, M	D, P
45. cecil_named_object a; cecil_top_context b; cecil_object_binding c. (a defining_context = b) && (b varBindings includes: c) && (c value = a)	V	H, N	C, M	D, P
46. BinaryExpression* e1, e2. e1.right == e2.right && e1 != e2	A	H, N		D, P
47. FieldExpression fe; FieldDefinition fd. fe.id == fd.name && fe.type == fd.type && fe.field != fd	A	H, N		D, P
48. Id x. x.type < 0	A	S		P
49. Id x. x.type == 0	V	S		P
50. Lexer l; Token t. l.token == t && t.type == 27	A	H		D
51. Lexer l; Token t. l.token == t && t.type == 0	V	H		D
52. spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	A	S		P
53. Cell a. a.value == Empty	V	S		D
54. Cell a b c. a.value == b.value && a.x == b.x && b.x == c.x && c.value == Empty && a.value != Empty	V	H, N		D

Table 15: Query examples

Query	Assertion/ Visualization	Selection/Join (Nested/Hash)	Special requirements	Query classification
55. Rook r1 r2. r1.color == r2.color && r1.x == r2.x && r1 != r2	V	H		D
56. King k; Figure * f. f.attacks(k)	V	N	M	D
57. Molecule* m. m.x < 0 || m.x > X_RANGE || m.y < 0 || m.y > Y_RANGE	A	S		D
58. Molecule* m1 m2. m1.x == m2.x && m1.y == m2.y && m1 != m2	A	H, N		D
59. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius	A	H		D
60. atom a b. a.center == b.center && a != b	A	H		D
61. View v, User u. u.subscribesTo(v)	V	N	M	D
62. View v, User u z. u.subscribesTo(v) && z.subscribesTo(v) && u != z	V	N	M	D
63. View v, User u z. u.edits(v) && z.edits(v) && u != z	V	N	M	D
64. View v1 v2; User u. u.reads(v1) && u.reads(v2) && v1 != v2	V	N	M	D
65. View v1 v2; User u z. u.subscribesTo(v1) && z.subscribesTo(v1) && u != z && u.subscribesTo(v2) && z.subscribesTo(v2) && v1 != v2	V	N	M	D
66. View v; User u z. u.edits(v) && z.edits(v) && u != z && u.site == z.site	V	N	M	D
67. Site s; View v; User u z. u.edits(v) && z.edits(v) && u != z && s.hosts(u) && s.hosts(z)	V	N	M	D
68. Room r1 r2; Group g1; Slots s1 s2. r1.slots.contains(s1) && r2.slots.contains(s2) && r1 != r2 && s1.time == s2.time && g.reservations.contains(s1) && g.reservations.contains(s2)	V	H, N	M	D
69. Room r1 r2; Slots s1 s2. r1.slots.contains(s1) && r2.slots.contains(s2) && r1 != r2 && s1.time == s2.time && s1.group == s2.group	V	H, N	M	D
70. task t; process p. ∀p (! (p.execute.contains(t)) && t index < ProcessedIndex	A	N	M, Q	D
71. Flight f f1; Plane p. f.seatsBooked < p.seatsInPlane && f.assigned(p) && f1.seatsBooked < p.seatsInPlane && f1.unassigned && f1.departure > f.arrival && f.destination == f1.startingPoint && f1.destination == f startingPoint	V	H, N	M	D
72. Ticket t; Airline a. ! a.services(t.destination)	A	N	M	D
73. Ticket t. t.firstFlight.arrival > t.secondFlight.departure	A	S		D
74. gate a; connection b. a.output == b && a.inputs.contains(b)	A	H, N	M	D
75. trueElement a; connection b; orGate c. a.output == b && c.inputs.contains(b)	A	H, N	M	D

Table 15: Query examples

Query	Assertion/ Visualization	Selection/Join (Nested/Hash)	Special requirements	Query classification
76. falseElement a; connection b; andGate c. a.output == b && c.inputs.contains(b)	A	H, N	M	D
77. Animator an; Vector images; Hashtable imageNames. images.size() != imageNames.size() && an.images == images && an.imageNames == imageNames	A	H, N	M, S	P
78. Animator an. an.images.size() != an.imageNames.size()	A	N	M	P
79. Animator an. an.images.size() != an.durations.size()	A	N	M	P
80. Animator an; Image im. an.images.contains(im) && (! an.imageNames.containsKey(im))	A	N	M	P
81. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	A	S		P
82. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	A	S	M	P
83. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	A	S		P
84. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	A	S		P
85. spec.benchmarks._201_compress.Compressor z. z.OutCnt == 0	V	S		P
86. spec.benchmarks._201_compress.Compressor z. z.out_count == 0	V	S		P
87. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0	A	H		P
88. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < z.InCnt	V	N		P
89. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count	A	N		P
90. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.out_count	V	H		P
91. mutableString a. (a asSlotIfFail: [abstractMirror]) isReflecteeSlots	V	S	M	P

Table 15: Query examples

5.3 Query Classification

The analysis of queries in the catalogue shows that queries check constraints of program domains, abstract data structures and program objects. The following classification groups

queries according to semantic intention. Queries belonging to different groups may have different creation times during a program's lifecycle and different goals. This section discusses the query categories indicated in the last column of Table 15 to give programmers a framework for query creation, use, and maintenance. Differences between query classes provide an opportunity for debugger customization.

Domain specific queries (D). The queries in this class check domain specific constraints, rules and invariants. Since the constraints checked by queries belong to the problem domain, they have to be satisfied independently of the program implementation of the problem domain. In other words, any program implementing a certain domain has to satisfy its restrictions. Domain constraints usually are identified in the program design stage, and consequently queries would be constructed at the same stage. Ideally, a debugger could automatically generate queries from the program design in a modeling language [66]. Domain queries would constitute a part of program design and would be used and verified during the program implementation.

Violation of domain specific queries indicates a design error and informs the programmer that some functionality of the application domain is implemented incorrectly or not implemented at all. For example, in a VLSI design program, a "true" input to an "or" gate makes the gate's output permanently true. Though a naively implemented program may not check this constraint and may allow such connections, such a configuration is a domain error.

Domain specific queries also include visualization queries that illustrate the program execution at the domain level. Results of these queries are similar to the results obtained from the program visualization systems.

Abstract data structure queries (A). Some queries check properties of abstract data structures [11][131] such as stacks, hash tables, trees, and so on. These queries are not domain queries, because the data structures can hold data of any domain. These queries are also different from the programming construct queries, because they check the constraints of well-defined abstract data structures. For example, a query about a binary tree may find the number of its nodes that have only one child. On the other hand, programming construct queries usually span different data structures. Abstract data structure queries can usually be expressed as class invariants and could be packaged with the class that implements an ADT. However, the queries that provide information rather than detect violations are best answered by dynamic queries. For example, monitoring B+ trees using queries may indicate whether this data structure is efficient for the underlying problem.

Program construct queries (P). Program construct queries verify object relationships that are related to the program implementation and not directly to the problem domain. Such queries verify and visualize groups of objects that have to conform to some constraints because of the lower level of program design and implementation. For example, in a graphical user interface implementation, every window object has a parent window, and this window references its children widgets through the widget_collection collection (section 5.2.2). Such construct is not

required by the domain, but it is used in the design and implementation, so it must be verified. It is also not an abstract data structure constraint because the object relationship does not form a clearly defined abstract data type. Sometimes queries have qualities of both programming construct queries and domain queries. Program construct queries are posed during the detailed program design and implementation stages. Design patterns [68], when used in programs, enforce program level constraints. Similarly to the design patterns, queries having similar basic structure can be summarized into query patterns (Table 16). Specifying and understanding widely used query patterns can allow developers of query-based debuggers to implement efficient algorithms for common queries. For example, developers could use the configuration of query pattern 1 to keep track of all elements already belonging to collections and to prohibit assignments of such elements to other collections. Such pattern specific implementation techniques would make query reevaluation less expensive.

Query	Description
1. collection a b; element c. a.contains(c) && b.contains(c)	Element must belong to a single collection.
2. collection a b; element c. a.contains(c) && ! b.contains(c)	Element must belong to both collections at the same time.
3. collection a b. a.size != b.size	Two collections must have the same size.
4. object a b. a.field == b.field && a != b	Objects of a class must have unique field.
5. object a b. a.field == b && b.field2 != a	Objects must mutually reference each other.
6. object a b. a.field == b.field && b.field2 != a	Object must reference another object that has the same value in a certain field.
7. object a b c. a.field == b && b.field2 == c	Monitor a chain of references usually with selection constraints on objects. Mostly used as a static-query program understanding tool.

Table 16: Query patterns

Although programming construct queries are difficult to check by conventional means and usually require specialized testing code, they can be easily checked using a dynamic query-based debugger.

To summarize, classification of queries gives insight in use of queries during different stages of a project lifecycle. The identification of query patterns promises potential speed benefits using query pattern specific optimizations.

5.4 Query Analysis and Classification Conclusions

The analysis of queries presented above indicates that 73 out of 91 queries can be handled by the current dynamic debugger implementation. Being aware of the query model and the debugger implementation may have precluded the author from posing more queries that cannot

be handled by the debugger. An important area that is not supported by the prototype is handling of collections (16 queries) and system classes (1 query, not counting system class collections). System class instrumentation can be done by using a load-time adaptation method different from class-loader based instrumentation (section 4.3.6.2). Handling of collections presents a more complicated challenge discussed in section 6.

Though 57 out of 91 queries are join queries, this observation may be biased by the effort of the author to come up with complicated queries. Another observation is that the queries rarely involve more than one or two joins. Perhaps writing down and understanding the semantics of more complex queries takes too much user effort. Programmers may use more complex queries when query libraries are available or when programmers are comfortable with tools provided.

It is unclear whether extending the query model with explicit universal and existential quantifiers would increase the tool's appeal. Without a field study it is difficult to decide whether programmers would understand and use such quantifiers.

One more area where the tool can be easily extended is the ability to identify individual object instances and use them as singleton domains. This would allow programmers to restrict queries only to certain instances of classes, and would increase the tool's efficiency. Such an extension would be easy in an integrated debugger environment that already maps object instances to variables or in a debugging Virtual Machine. If implemented without VM modification, the object instances would need to be identified by using unique hash codes.

The query list strongly suggests that method invocation support is necessary in any query debugging tool (53 out of 91 queries contain method invocations). The same can be said about dynamic queries. Most of the queries have results that change as the program executes, so static queries are not sufficient to detect error conditions.

5.5 Summary

Static and dynamic queries can be used to efficiently verify and monitor object relationships in object-oriented programs. This section has presented queries covering a dozen programs from five different domains. The classification of such queries and identification of query patterns has allowed us to better understand the query classes. Most of the queries are supported by the current dynamic query-based debugger. The full debugger implementation should support system class and collection debugging because these structures are frequently used in queries.

6 Future Work and Open Problems

"I didn't come all this way to sit out the fight!"
R. Balboa

"To survive, one must be able to adapt to changing situations."
Tyrannosaurus Rex

The query-based debugger model and implementation can be extended in a number of ways. First, the query model could be extended to include projection on "columns" of the result corresponding to the domain variables. The query model could allow computations that involve the result objects. For example, it would be useful to write queries that calculate the average length of certain lists. Although it is possible to do this 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 would no longer need to construct and then consume the output tuples.

A second avenue for the future work is to extend the dynamic queries to handle collections and system classes. System classes can be instrumented using a different load-time adaptation method (section 4.3.6.2). Handling of collections presents a more complicated challenge. First, collections are usually based on arrays, and the current implementation does not handle arrays. Second, collection interfaces are exported through accessor methods. For such methods the automatic change set determination may present problems. Third, collection support should be coupled with individual instance identification in queries. Only some collections may belong to the debugged program while most would be a part of the library code. Without filtering out these extraneous collections, the query results may be confusing and the debugger may become inefficient.

The rest of this chapter discusses two open problems: efficient automatic change set generation and safe query reevaluation. Section 6.2.2 also outlines the ongoing research on distributed query-based debugging.

6.1 Automatic Change Sets

Though the current debugger implementation automatically determines change sets of queries without method invocations (section 4.3.3), the general problem of automatic change set determination is not easy to solve. The first problematic area lies with method invocations that make it harder to determine which fields and objects affect the query result. The second problem is reference chains that introduce additional overhead in chain splitting. This section discusses possible solutions for both problems.

6.1.1 Automatic Change Sets for Method Invocations

Automatically determining a change set of the query when the query invokes methods is a difficult problem. To determine a change set, the system needs to perform a static or dynamic analysis of method invocations and field accesses. Such analyses would find the methods invoked during the query evaluation and objects referenced by these methods. These objects belong to the change set. The conservative starting point of the analysis are all objects transitively reachable through the fields of the domain class, and through all static fields of classes in the system. The analysis attempts to reduce this set.

The result of a static analysis that determines the methods involved in a query evaluation is equivalent to a static program slice [185] at the end of the query expression evaluation. To determine methods invoked, the system needs to use a type inference algorithm [6] because, in dynamically dispatched languages, a target method depends on the receiver type. The debugger already knows types of domain objects and their fields, allowing the system to determine the first methods invoked. If an ideal type inference engine were available, the system would know exactly which methods are called. Unfortunately, even the best type-inference methods require large amounts of memory, are relatively slow, and may not be precise enough to reduce the change set into an efficient size for monitoring purposes.

The second part of the static analysis would extract objects and fields accessed from the executed methods and so would determine which objects to monitor. The relationship between these objects and concrete instances of the domain class would need further identification as discussed in section 6.1.2.

The system could also find change sets by using dynamic analysis. In this case, the debugger would determine a change set at the time when a query is evaluated. Unlike the static analysis, no type inference algorithm would be needed, since the system would wait to detect method invocations until runtime. However, the system would incur a runtime overhead with each query reevaluation.

The dynamic analysis determines the query change set by marking all objects accessed during the evaluation. The marking can be done using a number of techniques proposed in the research on data breakpoints and on software fault isolation [108][180][181]. One way to determine all objects in the change set is to track all method calls. When a method is invoked, the system would mark the receiver object. This technique may be too expensive because each method invocation would need to be tracked. It can be improved by using code patching.

Another technique is a *virtual memory* approach. In this case all objects are placed in a read-protected area. Control is transferred to the debugger each time one of the objects is accessed. Then the object is marked and moved to an unprotected area. For example, while evaluating the query:

```
Ticket t; Airline a.      ! a.services(t.destination)
```

the system would access Airline objects, AirlineDestinations objects and so on. The virtual memory method would be faster if an efficient read trap were available. It may be possible to achieve a greater efficiency by increasing the granularity of the change set detection. For example, if a read-protected page is accessed, the system could mark all objects in it as belonging to a change set. However, this approach increases the overhead of monitoring falsely shared objects.

A variation of this method is Keppel's *patch-on-trap* method [108]. When a trap is hit the first time, the system not only deals with the object access but also inserts patch code before the memory access instruction. This patch code contains instructions that check the address of an object being accessed. When this memory access instruction is executed again, the patch code is invoked before the instruction itself. If the instruction would access an object in the protected page, the patch code marks and moves this object without causing a trap. Then the instruction can access the moved object without a trap. This technique decreases the number of page faults to one page fault per memory access instruction. This approach also saves time spent trying to find and patch memory access instructions if the system were to use pure patching.

A system using pure code patching technique would dynamically patch all code executed during the query evaluation. For example, memory access instructions in the method services() would be patched when this method is invoked during the evaluation of the airline query above.

Dynamic analysis solves the issue of identifying methods called in the constraint part of the query because the system either patches the methods when they are called or ignores them and marks objects that generate read exceptions. However, dynamic analysis needs to be performed at each query reevaluation because a change in some object may change the dynamically determined change set. Furthermore, virtual-memory and code patching based approaches would require Java Virtual Machine modifications, something that we would like to avoid.

Dynamic analysis could potentially be done with an acceptable efficiency. Wahbe and others [181] have shown that the software fault isolation (both read and write) costs only 22% of execution speed. Efficient data breakpoints implemented by the same authors [180] slow down the execution about 30%. The slowdown should not be significantly larger in a debugger implementation. On the negative side, dynamic analysis has to be done for each tuple evaluated during the query execution. It would be preferable to use methods that depend on the number of domain objects but not on the number of tuples. Techniques used in the system should have lower penalty for accesses to already marked objects than the penalty for accesses to unmarked objects.

To summarize, though static and dynamic analyses could be used to determine change sets for method invocations, the viability of these approaches still needs to be investigated.

6.1.2 Reference Chains

Although reference chains appear in queries directly, the change sets of queries with method invocations are even more likely to contain reference chains. Efficient handling of reference chains is important for good debugger performance on complicated queries.

Section 4.3.3 shows the use of reference chain splitting to handle change sets of queries with reference chains:

```
IntersectPt ip.    ip.Intersection.z < 0
```

The Intersection field is a Point object, and the query result depends on its z value. The query result may change if the z value changes, or if a new value is assigned to the Intersection field. Furthermore, the Point object referenced by the Intersection field may be shared among several domain objects. In this case, a change in one Point object can affect multiple domain objects. Our debugger rewrites the query by splitting the chain into single-level accesses and by adding additional domains and constraints. For example, the ray tracing query above is rewritten as:

```
IntersectPt ip; Point* __Intersection.
ip.Intersection == __Intersection && __Intersection.z < 0
```

Another approach to handle change sets of such queries would be to keep *backward references* from the change set objects to the domain objects. When a change set object changes, the debugger is informed which domain objects were affected by the event. Assume that two IntersectPt objects share the same point object through the Intersection field. When the z coordinate of this point is changed, the debugger would be informed which two IntersectPt objects are affected. However, keeping and maintaining backward references is a complicated and potentially expensive task. Future research could calculate the overhead of backward references and their contribution to the efficient query evaluation.

Alternatively, the system could keep references from each domain object to the change-set objects affecting this domain object. These references are called *forward references*. If forward references are used, after an event occurs, the debugger would find whether a domain object was affected by traversing these references. If objects changed during the event are in the forward reference set of some object, this domain object is affected by changes. This approach may be slower than using backward references but it may consume less memory, because the change set is usually referenced by the fields of the domain object. Such (forward) references already exist in the program and do not need to be stored separately, while backward references have to be stored explicitly.

6.2 Safe Reevaluation and Distributed Debugging

The problem of safe query evaluation was discussed in section 3.2.1. This section proposes some ways to deal with a problem and introduces an extension of the query-based debugger to the distributed setting.

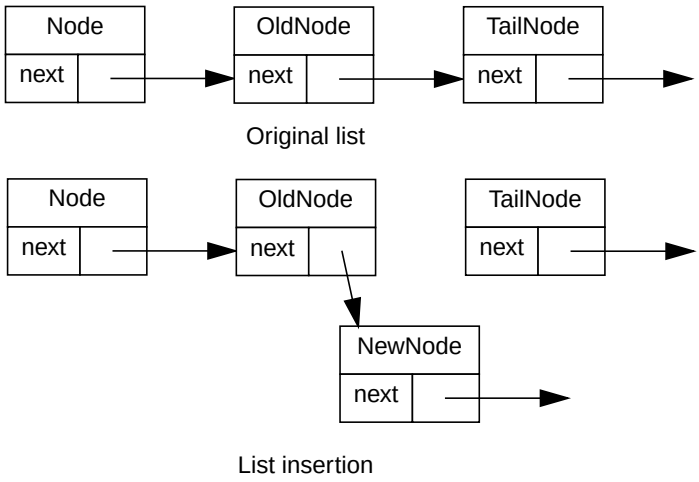

Figure 32. Inconsistent list state

6.2.1 Safe Reevaluation

Section 3.2.1 outlined the problem of safe evaluation. To recap, the queries should be reevaluated only when queried objects are in consistent states. In other words, the expression evaluation should succeed and provide meaningful results. At some program execution points the query evaluation may be unsafe. For example, during an insertion of an element into the list, the list may have an inconsistent state. In Figure 32, if the query is asked just after the program updates the next reference of the OldNode but before it sets the next reference of the NewNode to point to the TailNode, the query evaluation will be unsafe. If the debugger traverses a list, it may crash because the reference NewNode.next is null or it may produce an incorrect output by using a shorter list that does not contain the TailNode.

Therefore, query results can be updated only when all abstractions involved are in a consistent state. What are the consistent states? The program state is *inconsistent* with respect to a query, if either the underlying program objects are inconsistent, or if the design-level abstraction is temporarily broken. The list example shows the situation where the inconsistency lies with program objects. The following gas tank query illustrates a temporary design-level inconsistency:

```
Molecule* m1 m2.     m1.x == m2.x && m1.y == m2.y && m1 != m2
```

This query finds molecule objects that erroneously occupy the same position. One way to preserve this constraint in a program is to check whether molecules have collided and then adjust the molecule coordinates to reflect the collision. However, such a solution would be perceived incorrect by the debugger, because the molecules “occupy the same position” after the collision and before the fix. The debugger should not evaluate the query in this inconsistent region.

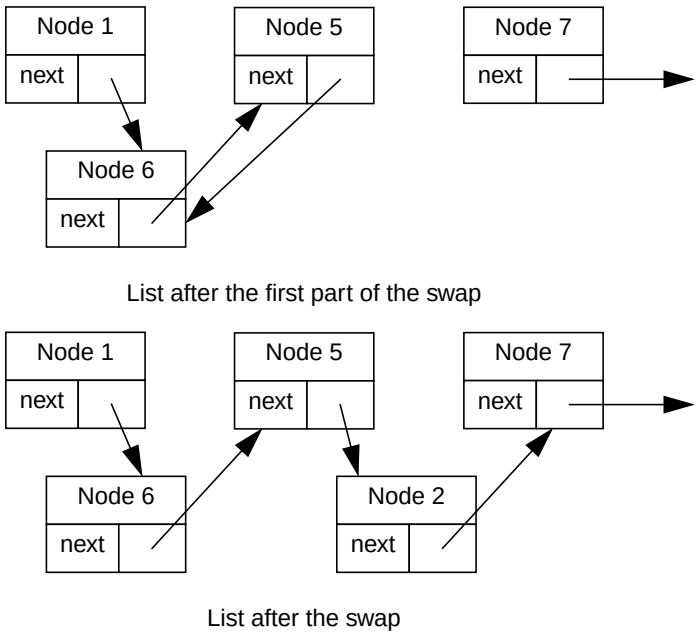

Figure 33. Inconsistent intermediate list state

Consider another scenario—recursive sorting of a list by deletion and insertion. The system needs to reevaluate a query involving the list being sorted. The system cannot reevaluate this query as soon as the execution leaves the method that changed the list, because the list's state may still be inconsistent at this point. Other operations on the list or operations on cooperating objects that are on the execution stack may still be in the process of modifying the list's state—even though the list link update is finished, the list may still be unconnected. For example, if the list class uses the swap method to change places of two elements, the first part of the swap method may just move one of the elements to its new place (Figure 33). The list may be inconsistent until the second part of swap finishes.

To avoid inconsistency at the program level, the debugger could wait with the query reevaluation until there are no domain object operations on the execution stack. Unfortunately, this approach may cause a long delay. In the list sorting example, the program spends all its time in the list code, making the debugger wait until the end of the execution to update the result of the query. This is clearly unacceptable. Even if the debugger waited until all domain methods were off the execution stack, the consistency could still be violated at the logical level. Objects cooperating with the domain objects (multi-object transactions) could create inconsistencies. For example, in a cellular network simulation (section 5.2.1.1), when a user contacts a cellular base station, the simulation program needs to update both the list of cellular base-station users and the list of the assigned channels. If the debugger waits only until the list of users is off the execution stack, the query reevaluation will violate the transaction, because the list of assigned

channels is not yet updated. These scenarios show that simple strategies cannot deal with complex consistency situations.

Another problem of dealing with inconsistent regions is that some of these regions may hide genuine errors. Prohibiting query execution from a part of the runtime makes it more difficult for programmers to understand the query and the program. Consequently, such region exclusion should probably be left to a strict supervision of programmers, especially since automatic determination of inconsistent states is probably impossible. One way to give users control over excluding inconsistent regions is to use guarded queries. With each query users would provide a "when" or "while" clause that could be consistently evaluated at any time. To provide such a guarantee, the guard clause has to be a simple expression involving simple methods. The rest of the query is evaluated only when (or while) the guard is true. Guards allow to declaratively specify program regions where the query reevaluation is safe. Even though programmers have to spend time writing guards, such programmer assistance seems to be reasonable when a guard is available or easy to construct. For example, the cellular network base station should have no active channels when it has no active users. Checking the size of the user list would be a natural guard for queries involving the empty list.

Guards in general are as powerful as direct commands to reevaluate a query in the program text. Such a command can be simulated using a guard by adding a flag variable to the program and setting this flag in the place where a command would have been invoked. The guard would check the flag and perform a query reevaluation when the flag is set. If guards are available in the system, users can force query reevaluation at any moment of the program execution.

In addition to the transactional model, active database systems apply different methods to solve the safe reevaluation problem. Most databases use a direct command method for triggering a condition in the ECA (event-condition-action) model. Ariel [81] contains atomic (guarded) regions where rules should not be fired. Similarly, Starburst [189][190][191] reevaluates rules only at the end of operation blocks usually corresponding to transactions. These techniques correspond to the safe evaluation methods proposed above.

6.2.2 Distributed Query-Based Debugging

The implementation of query-based debugging for distributed systems would help programmers to debug distributed applications well known for their difficult-to-understand semantics. However, such an implementation would have to deal with several problems. First, the domain objects have to be collected from distributed nodes. Second, these objects have to be in a consistent state. Methods for achieving consistency discussed in the previous section would have to be adapted to the distributed setting. A distributed query-based debugging project is currently under way at UCSB [114]. The debugger will be implemented on the Kan distributed object system [100]. Consistent reevaluation points will correspond to the transaction commit points and to the entry/exit points of user indicated methods. Work on distributed query-based

debugging could build on the research describing efficient view updates in distributed data warehouses [8][197].

Another aspect of distributed debugging would be dealing with shared multi-user spaces such as Kansas [157]. Debugging in such spaces would have to deal both with the problem of distributed environment and with the problems of user separation and interaction.

7 Conclusions

"By persevering over all obstacles and distractions, one may unfailingly arrive at his chosen goal or destination."

C. Columbus

"Just because something doesn't do what you planned it to do doesn't mean it's useless."

T. Edison

Understanding and debugging complex software systems is difficult. The cause-effect gap between the time when a program error occurs and the time when it becomes apparent to the programmer makes many program errors hard to find. This situation is further complicated by the increasing use of large class libraries and complicated reference-linked data structures in modern object-oriented systems. A misdirected reference that violates an abstract relationship between objects may remain undiscovered until much later in the program's execution.

Conventional debugging methods offer only limited help in finding such errors. Most conventional debuggers offer only a limited low-level view of the program state and provide little support for the exploration of large data structures. Data breakpoints and conditional breakpoints cannot check constraints that have objects unreachable by references from the statement containing the breakpoint.

The research in this dissertation proposes a system that allows to ask queries about the program state, helping to check object relationships in large object-oriented programs. The current implementation of the query-based debugger[1] 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.

[1] Available from the author as a prototype implementation in Java.

An extension of the static query-based debugger handles dynamic queries that update query results whenever the program changes an object relevant to the query, helping to discover object relationship failures as soon as they happen. This system provides the following new features:

- Dynamic queries: Not only does the debugger check object relationships, but it determines exactly when these relationships fail in the program execution. This technique closes the cause-effect gap between error's occurrence and its discovery.
- Implementation of monitoring queries: the debugger helps users to watch the changes in object configurations through the program's lifetime. This functionality can be used to understand the program behavior.

The implementation of the dynamic query-based debugger has good performance. Selection queries are efficient, with a slowdown smaller than a factor of two for most queries measured. We also measured field assignment frequencies in the SPECjvm98 suite, and showed that 95% of all fields in these applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation time estimates, our debugger performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. Join queries are practical when domain sizes are small and queried field changes are infrequent. Additional optimizations of join queries could possibly improve their evaluation efficiency.

On-the-fly dynamic debugging has been implemented but currently suffers a high overhead. Selection slowdowns range up to factor 9.5 with a median of 5.5. Further optimizations could reduce this overhead.

We believe that query-based debugging adds a powerful tool to the programmer's tool chest for tackling the complex task of debugging. Our implementation of the dynamic query-based debugger demonstrates that dynamic queries can be expressed simply and evaluated efficiently. The debugger implementations in Self and Java show that the system can be adapted to different programming languages and environments. We hope that future mainstream debuggers will integrate similar functionality, simplifying the difficult task of debugging and facilitating the development of more robust object-oriented systems.

8 Glossary

“It’s useless to try to plan for the unexpected... by definition!”

A. Hitchcock

“It all hinges on your definition of ‘a good time’!”

L. Borgia

Active database. A database that can change reacting to events. Usually with event-condition-action rules. Section 4.8.

Assertion. A conditional expression that is claimed to be true at some time during the program’s execution. Includes class and method preconditions, postconditions, and invariants. Section 2.1.3.

Backward reference. A reference from a change-set object to a domain object that is affected by changes of this change-set object. See **forward reference**. Section 6.1.2.

Bytecode. An instruction in the intermediate format, in particular, the format used by Java Virtual Machine. Section 4.3.2.

Change set. A set of objects and their fields that can affect the result of a query. Section 4.3.3.

Class file. An intermediate format file containing a Java class description compiled into bytecodes. Section 4.3.2.

Class loader. A program that controls the class loading process. Section 4.3.2.

Code generation (load-time or runtime). Producing a runnable code during the program’s load-time or runtime. Section 4.3.5.2.

Conditional breakpoint. A breakpoint that stops a program when a condition at the breakpoint insertion point evaluates to true. Section 2.1.2.

Consistent state. A program execution state at which a query can be executed without crashing the program and without producing logically incorrect results. Section 3.2.2.

Constraint. An object relationship that holds for a (part of a) program execution. Section 2.

Data breakpoint. A breakpoint that stops a program when the program modifies a selected variable. Section 2.3.1.

Debugger invocation frequency. The frequency of events in the original program that would trigger a debugger invocation, i.e., the invocation frequency for a debugger with no overhead. Section 4.4.1.

Domain. A collection of objects on which a query is evaluated. Typically a class. Section 3.2.

Dynamic query. A query that is answered while a program is running. Section 4.

Forward reference. A reference from a domain object to its change-set object. See **backward reference**. Section 6.1.2.

Guard. A simple expression that can be evaluated and is consistent at every program execution point. Used to determine **consistent states** of a query. Section 6.2.1.

Hash join. A join performed by hashing objects corresponding to a left-hand side of an equality constraint into a hash table, and retrieving matches by using objects corresponding to a right-hand side of a constraint. Section 3.3.6.

Instrumentation. Changing the source or the intermediate code of a program. Can be done to add debugger or profiler invocations. Section 4.3.2.

Invariant. A constraint that is correct at every execution of class, method, or loop. Section 2.1.3.

Java Virtual Machine (JVM). See **Virtual Machine**.

Join query. A query with more than one variable. Section 3.2.

Load-time adaptation (LTA). A program change at load-time. Used for debugging, profiling, program behavior extension. Section 4.3.6.1.

Nested-loop join. A join performed by checking all combinations of domain tuples in a nested loop. Section 3.3.4.

Query. A question about program structure satisfying the query model. Section 3.2.

Safety (of query evaluation). Evaluation of queries at **consistent states**. Section 3.2.2.

Selection query. A query with a single variable. Section 3.2.

Slicing. Finding a subset of program statements—a slice—that affects the value of a certain variable or the current statement of the program. Section 2.3.2.

Soft (weak) references. References that are not considered by the garbage collector when deciding whether an object is alive. Section 4.3.4.

Static query. A query that is answered at a program breakpoint. Section 3.

Transaction. An atomic group of statements that are either all executed or not executed at all. Section 6.2.1.

Transient failures. Failures that disappear after some period of time. Section 4.2.1.

Virtual Machine (VM). Runtime system for running programs in an intermediate format understood by the virtual machine. Section 4.3.

Visualization. On-screen display of program information usually changing in time. Section 2.4.

9 References

"Old heroes never die; they reappear in sequels"
M. Moorcock

[1] Abiteboul, S., Hull, R., Vianu, V., *Foundations of Databases*, Addison-Wesley, 1995.

[2] 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.

[3] Acharya, A., Scalability in Production System Programs, *Ph.D. Thesis*, Computer Science Department, Carnegie Mellon University, November 1994.

[4] Adl-Tabatabai, A.-R., Langdale G., Lucco S., and Wahbe R., Efficient and Language-Independent Mobile Programs, *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation 1996*, Philadelphia, May 1996. Published as *ACM SIGPLAN Notices*, vol. 31, (no.5), pp. 127–136, May 1996.

[5] Agesen, O., Bak, L., Chambers, C., et al. *The Self 4.0 Programmer's Reference Manual*.

[6] Agesen, O., Concrete Type Inference: Delivering Object-Oriented Applications. *Ph.D. Thesis*, Stanford University 1995.

[7] Agesen, O., Freund, S.N., and Mitchell, J.C. Adding Type Parameterization to the Java Language. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA'97*, Published as *SIGPLAN Notices 32(10)*, pp. 304-317, Atlanta, GA, October 1997.

[8] Agrawal, D., El Abbadi, A., Singh, A.K., Yurek, T., Efficient View Maintenance at Data Warehouses, *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, *SIGMOD Record*, pp. 417–427, 1997.

[9] Agrawal, H., Horgan, J.R., Dynamic Program Slicing, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'90*, White Plains, N.Y., pp. 246–256, June 1990.

[10] Agrawal, R., Gehani, N.H., ODE (Object Database and Environment): The Language and the Data Model, *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, pages 36-45, May 1989.

[11] Aho, A.V., Hopcroft, J.E., Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley 1974.

[12] Aho, A.V., Sethi R., Ullman J.D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, 1986.

[13] Alexandrov, A., Ibel, M., Schauser, K., and Scheiman, C. Extending the Operating System at the User Level: the Ufo Global File System. *Proceedings of the USENIX 1997 Technical Conference*, January 1997.

[14] Anderson E., Dynamic Visualization of Object Programs written in C++, *Objective Software Technology Ltd.*, http://www.objectivesoft.com/, 1995.

[15] Anwar, E., Maugis, L., Chakravarthy, S., A New Perspective on Rule Support for Object-Oriented Databases, *Proceedings of 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, USA, May 26–28, 1993. *SIGMOD Record*, vol.22, (no.2), pp. 99–108, May 1993.

[16] Arnold, K., Gosling, J., *The Java Programming Language*, Addison-Wesley Pub. Co., 1996.

[17] Asprin R., The Myth-ing Books: *Another Fine Myth*, *Myth Conceptions*, *Hit or Myth*, *Myth-ing Persons*, *Little Myth Marker*, *M.Y.T.H. Inc. Link*, *Myth-nomers and Im-pervections*, *Myth Directions*, *M.Y.T.H. Inc. in Action*, *Sweet Myth-tery of Life*, Ace Books, 1978–1998.

[18] Baecker, R., DiGiano, C., Marcus, A., Software Visualization for Debugging, *Communications of the ACM*, Vol. 40., No. 4, pp. 44–55, April 1997.

[19] 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.

[20] Baralis E., and Widom, J., Using Delta Relations to Optimize Condition Evaluation in Active Databases. *Proceedings of the Second International Workshop on Rules in Database Systems*, *Lecture Notes in Computer Science* 985, pp. 292–308, Springer-Verlag, Berlin, September 1995.

[21] Beeri, C., Milo, T., A Model for Active Object-Oriented Database, P*roceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, pages 337–349, September 1991.

[22] Beguelin, A., Dongarra, J., Geist, A., Sunderam V., Visualization and Debugging in a Heterogeneous Environment, *IEEE Computer* 26(6), pp. 88–96, June 1993.

[23] Berk E., *JLex: A Lexical Analyzer Generator for Java*^{TM}, version 1.2.3, http://www.cs.princeton.edu/~appel/modern/java/JLex/ , 1996.

[24] Bertino, E., Guerrini, G., Extending the ODMG Object Model with Composite Objects, *Proceedings of OOPSLA'98*, pp. 259-270, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

[25] Bischofberger, W. R., Kofler, T., Schäffer, B., Object-Oriented Programming Environments: Requirements and Approaches, *Software—Concepts and Tools*, Vol. 15, No. 2, Springer-Verlag, 1994

[26] 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.

[27] Bourdoncle, F. Abstract Debugging of Higher-Order Imperative Languages. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'93*, Albuquerque, N.M., pp. 46–55, June 1993.

[28] Brant, D.A., Grose, T., Lofaso, B., Miranker, D.P., Effects of Database Size on Rule System Performance: Five Case Studies, *Proceedings of the 17th International Conference on Very Large Data Bases* (VLDB), Barcelona, Spain, pp. 287–296, September 1991.

[29] BrightWare, *ART*Enterprise*, http://www.brightware.com/products/art.html, 1999.

[30] Bronnikov, D., *Java 1.1 grammar*, version 1.03, http://home.inreach.com/bronikov/grammars/java.html , November 1997.

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

[32] 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.

[33] Brownston, L., Farrell, R., Kant, E., Martin, N., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Reading, MA, 1985.

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

[35] Cardelli L., Wegner P., On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, Vol. 17, No. 4, pp. 471-522, December 1985.

[36] Cargill, T.A; Locanthi, B.N.; Cheap Hardware Support for Software Debugging and Profiling. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 1987. pp. 82–83, ACM Press 1987.

[37] Cattell, R.G.G., edited by, *The Object Database Standard: ODMG-93, Release 1.2*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

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

[39] 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.

[40] Chandra, A.K., Merlin P.M., Optimal Implementation of Conjunctive Queries in Relational Data Bases, *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, Boulder, Colorado, May 1977, pp 77–90, 1977.

[41] Chang, B.-W., Ungar, D., Smith, R. B., Getting Close to Objects: Object-Focused Programming Environments, *Visual Object Oriented Programming*, Burnett, M., Goldberg, A., Lewis, T., eds., Prentice-Hall, 1995, pp. 185-198.

[42] Cluet S., Moerkotte G., On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products, *Proceedings of the 5th International Conference on Database Theory*, Prague, Czech Republic, volume 893 of Lecture Notes in Computer Science, Springer-Verlag, pp. 54-67, January 1995.

[43] Cohen, G.A., Chase, J.S., and Kaminsky, D.L. Automatic Program Transformation with JOIE. *Proceedings of the 1998 USENIX Annual Technical Symposium*. 1998.

[44] 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.

[45] 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.

[46] 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.

[47] Cox, K. C.; Roman G.-C.; *Experiences with the Pavane Program Visualization Environment*, Technical Report, WUCS-92-40, October 1992.

[48] Dahl, O., and Nygaard, K., Simula: An Algol-based simulation language. *Communications of the ACM*, 9(9), pp. 671–678, September 1966.

[49] Detlefs D., Dosser A., Memory Allocation Costs in Large C and C++ Programs, Software - Practice and Experience, Vol. 24 (6), June 1994, pp. 524 - 542.

[50] 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.

[51] 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.

[52] De Pauw, W.; Lorenz, D.; Vlissides, J.; Wegman, M. Execution patterns in object-oriented visualization. *Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, Sante Fe, NM, USA, 27-30 April 1998, USENIX Association, 1998. pp.219-34.

[53] De Witt, D..J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood D., Implementation techniques for main memory database systems, *Proceedings of 1984 ACM SIGMOD International Conference on Management of Data*, pp. 1–8, May 1984.

[54] Diaz, O., Paton, N., Gray, P., Rule Management in Object-Oriented Databases: A Uniform Approach, P*roceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, pages 317–326, September 1991.

[55] Doorenbos, R.B., Production Matching for Large Learning Systems, *Ph.D. Thesis,* Computer Science Department, Carnegie Mellon University, January 1995.

[56] Duncan, A., Hölzle, U.; *Adding Contracts to Java with Handshake*, Technical Report TRCS98-32, December 1998.

[57] Duncan, A., Hölzle, U.; *Load-Time Adaptation: Efficient and Non-Intrusive Language Extension for Virtual Machines*, Technical Report TRCS99-09, April 1999.

[58] Eisenstadt, M., My Hairiest Bug War Stories, *Communications of the ACM*, Vol. 40., No. 4, pp. 30–38, April 1997.

[59] Eisenstadt M., *Tales of Debugging from The Front Lines*, Technical Report 106, Human Cognition Research Laboratory, 1993.

[60] Eisenstadt M., Why Hypertalk Debugging Is More Painful Than It Ought To Be, *in J. Alty, D. Diaper and S.P. Guest (Eds.), People and Computers VIII*. Cambridge, U.K.: Cambridge University Press, 1993.

[61] Eisenstadt M., Price B. A., Domingue J., Software Visualization As A Pedagogical Tool: Redressing Some ITS Fallacies, *Instructional Science,* 21, pp. 335–365, 1993.

[62] Flanagan, C., Flatt, M., Krishnamurthi, S., Weirich, S., Feilleisen, M., Catching Bugs in the Web of Program Invariants, *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation 1996*, Philadelphia, May 1996. Published as *ACM SIGPLAN Notices*, vol. 31, (no.5), pp. 23–32, May 1996.

[63] Forgy, C.L., *OPS5 User's Manual*, Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July 1981.

[64] Forgy, C.L., RETE: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, No. 19, pp. 17–37, 1982.

[65] Forgy, C.L., RAL/C and RAL/C++: Rule-based extensions to C and C++. *Position Papers for the OOPSLA'94 Embedded Object-Oriented Production Systems Workshop (EOOPS)*, October 1994.

[66] Fowler, M., Scott, K., *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.

[67] Gamma E., Design Patterns Elements of Reusable Object-Oriented Software, *Tutorial Notes of TOOLS'95 Conference*, 1995.

[68] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[69] 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.

[70] Garey M.R., Johnson D.S., *Computers and Intractability A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, NY 1979, 1979.

[71] Gehani N.H. and Jagadish H. V. Ode as an Active Database: Constraints and Triggers. P*roceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, pages 327-336, September 1991.

[72] Gehani, N.H., Jagadish, H.V., Shmueli, O., Event Specification in an Active Object-Oriented Database, *Proc. ACM-SIGMOD 1992 Int'l Conf. on Management of Data*, 1992.

[73] Gill, S., The diagnosis of mistakes in programmes on the EDSAC, *Proceedings of the Royal Society Series A Mathematical and Physical Sciences*, 206(1087), pp. 538–554, Cambridge University Press, May 1951.

[74] 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.

[75] 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.

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

[77] Goldberg, A.; Robson, D.; *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[78] Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley 1996.

[79] Haas, P.J.; Naughton, J.F.; Seshadri, S.; Swami, A.N., Selectivity and Cost Estimation for Joins based on Random Sampling, *7th Annual Conference on Computational Learning Theory. Journal of Computer and System Sciences*, June 1996, vol.52, (no.3):550-69.

[80] The Haley Enterprise, RETE++ and Eclipse, http://www.haley.com, 1999.

[81] Hanson, E.N., Rule Condition Testing and Action Execution in Ariel, *Proceedings of 1992 ACM SIGMOD International Conference on Management of Data*, pp. 49–58, June 1992.

[82] Hart D., Kraemer E., Roman G.-C., Interactive Visual Exploration of Distributed Computations. *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, pp.121-127, April 1997.

[83] Hart D., Kraemer E., Roman G.-C., *Interactive Visual Exploration of Distributed Computations*. Pre-International Parallel Processing Symposium Technical Report,1997.

[84] Hao, M.C.; Karp, A.H.; Waheed, A.; Jazayeri, M., VIZIR: an integrated environment for distributed program visualization. *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '95*, pp.288–92, Durham, NC, USA, January 1995.

[85] Henry, R. R., Whaley, K. M., Forstall B., The University of Washington Illustrating Compiler, *Proceedings of The ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, ACM Press, New York, 1990, pp. 223-233.

[86] Hölzle, U.; Chambers, C., Ungar, D., Debugging Optimized Code with Dynamic Deoptimization, *Proceedings of The ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, Published as *SIGPLAN Notices 27(7)*, ACM Press, pp. 32–43, 1992.

[87] Hölzle, U., A Fast Write Barrier for Generational Garbage Collectors, *Proceedings of OOPSLA'93 Workshop on Garbage Collection*, Washington, D.C., September 1993.

[88] Hölzle, U., *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*, Ph. D. Thesis, Sun Microsystems Laboratories Technical Report TR-95-35, 1995.

[89] 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.

[90] Horn, B. Constraint Patterns as a Basis for Object Oriented Programming. *Proceedings of SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '92*, Vancouver, BC, Canada, 20-22 Oct. 1992, Published as *SIGPLAN Notices*, vol.27, (no.10), pp. 218–233. October 1992.

[91] Hudson, S.E., *CUP Parser Generator for Java*, version 0.10i, http://www.cs.princeton.edu/~appel/modern/java/CUP/ , February 1999.

[92] Hyrskykari A., Development of Program Visualization Systems, *2nd Czech British Symposium of Visual Aspects of Man-Machine Systems*, March 27, 1993, Praha

[93] 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.

[94] ILOG, *ILOG Rules*, White Paper, http://www.ilog.com/products/rules/whitepaper.pdf, March 1997.

[95] Ioannidis Y. E., Kang Y. C., Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization, *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 168-177, Denver, USA, May 1991.

[96] Ioannidis Y. E., Kang Y. C., *Randomized Algorithms for Optimizing Large Join Queries*, Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 312-321, Atlantic City, USA, May 1990.

[97] Jarke, M., Koch, J., Query Optimization in Database Systems, *ACM Computing Surveys*, vol. 16, No. 2, pp. 111-152, June 1984.

[98] *Java™ Platform Debugger Architecture*, http://developer.java.sun.com/developer/earlyAccess/jbug/index.html, 1999.

[99] *Java™ 2 SDK Production Release*, http://www.sun.com/solaris/, 1999.

[100] James, J., *The Kan Project—Reliable Concurrent Objects*, http://www.cs.ucsb.edu/~dsl/Kan/, 1999.

[101] Jerding, F.J., Stasko J.T., *Using Visualization to Foster Object-Oriented Program Understanding*, Technical Report GIT-GVU-94-33, July 1994

[102] Jerding, D.F., Stasko, J.T., Ball, T., *Visualizing Message Patterns in Object-Oriented Program Executions*, Technical Report GIT-GVU-96-15, May 1996.

[103] Jones R., Lins R., *Garbage Collection Algorithms for Automatic Dynamic Memory Management*, Wiley, 1996.

[104] Kamkar, M., An Overview and Comparative Classification of Program Slicing Techniques, *Journal of Systems and Software*, vol. 31, pp. 197–214, July 1995.

[105] Karaorman, M., Hölzle, U.; Bruno, J.; *jContractor: A Reflective Java Library to Support Design By Contract*, Technical Report TRCS98-31, December 1998.

[106] Keller, R., Hölzle, U.; Binary Component Adaptation, *Proceedings ECOOP'98*, Springer Verlag Lecture Notes on Computer Science, Brussels, Belgium, July 1998.

[107] Keller, R., Hölzle, U.; *Implementing Binary Component Adaptation for Java*, Technical Report TRCS98-21, August 1998.

[108] Keppel, D., *Fast Data Breakpoints*. Technical Report UWCSE 93-04-06, University of Washington, April 1993.

[109] Kessler, P., Fast Breakpoints: Design and Implementation. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation* 1990, Published as *SIGPLAN Notices* 25(6), pp. 78–84, ACM Press, June 1990.

[110] Khoshafian, S. N., Copeland, G. P., Object Identity, *Proceedings of OOPSLA'86*, pp. 406–416, Portland, OR, November 1986. Published as SIGPLAN Notices 21(11), November 1986.

[111] 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.

[112] Kishon, A., Hudak, P., Consel, C., Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Reasoning about Execution Monitors, *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation* 1991, Toronto, Ontario, Canada, June 1991, pp. 338–352, ACM Press 1991.

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

[114] Kulkarni, S., *Distributed Debugging*, http://www.cs.ucsb.edu/~somil/thesis/proj.html, 1999.

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

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

[117] 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.

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

[119] 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.

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

[121] Lehman, T.J., Carey, M.J., Query Processing in Main Memory Database Management Systems, *Proceedings of 1986 ACM SIGMOD International Conference on Management of Data*, pp. 239–250, May 1986.

[122] Lencevicius, R.; Hölzle, U.; Singh, A.K., *High-Level Debugger for Object-Oriented Programs*, Unpublished report, November 1995.

[123] Lencevicius, R.; Hölzle, U.; Singh, A.K., Query-Based Debugging of Object-Oriented Programs, *Proceedings of OOPSLA'97*, pp. 304-317, Atlanta, GA, October 1997. Published as *SIGPLAN Notices* 32(10), October 1997.

[124] Lencevicius, R.; Hölzle, U.; Singh, A.K., Dynamic Query-Based Debugging, *Proceedings of the 13th European Conference on Object-Oriented Programming'99*, (ECOOP'99), Lisbon, Portugal, June 1999, Published as *Lecture Notes on Computer Science 1628*, Springer-Verlag, 1999.

[125] Liang, S., Bracha, G.; Dynamic Class Loading in the Java™ Virtual Machine, *Proceedings of OOPSLA'98*, pp. 36-44, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

[126] Lieuwen, D., Gehani, N., and Arlein R., The Ode Active Database: Trigger Semantics and Implementation. *Proceedings of Data Engineering*, February–March 1996.

[127] Lindholm, T., Yellin, F., *The Java™ Virtual Machine Specification*, Addison-Wesley 1996.

[128] Litman D.; Mishra A.; Patel-Schneider P.F., Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules, *Proceedings of OOPSLA'97*, pp. 77-92, Atlanta, GA, October 1997. Published as *SIGPLAN Notices* 32(10), October 1997.

[129] Maloney J., *Morphic: The Self User Interface Framework*, Sun Microsystems and Stanford University, 1995.

[130] McCarthy, D. R., Dayal, U., The Architecture Of An Active Data Base Management System. *Proceedings of 1989 ACM SIGMOD International Conference on Management of Data*, pp. 215–224, 1989.

[131] McHugh, J.A. *Algorithmic Graph Theory*, Prentice-Hall 1990.

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

[133] Meyer B., Applying Design by Contract, *IEEE Computer*, vol. 25, no. 10 pp. 45–51, October 1992.

[134] Meyer B., *Eiffel: The Language*, Prentice-Hall, 1992.

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

[136] Mitchell, G., Dayal, U., Zdonik, S.B., Control of an Extensible Query Optimizer: A Planning-Based Approach, *Proceedings of the 19th VLDB Conference*, 1993.

[137] Mössenböck, H., Films as graphical comments in the source code of programs. *Proceedings of the International Conference on Technology of Object Oriented Systems and Languages, TOOLS-23*, pp. 89-98, Santa Barbara, CA, USA, July-August 1997.

[138] Myers, A.C., Bank, J.A., and Liskov, B. Parameterized Types for Java. *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[139] 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.

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

[141] Noble R. J., Groves L.J., Tarraingim - A Program Animation Environment, *Proceedings of the 12th New Zealand Computer Conference*, Dunedin, August 14-16, 1991

[142] Oflazer, K., Partitioning in Parallel Processing of Production Systems, *Ph.D. Thesis*, Computer Science Department, Carnegie Mellon University, March 1987.

[143] OST, *Source vs. Object Level Debugging*, Objective Software Technology, White Paper, 1993.

[144] Price B.A., Baecker, R.M., and Small, I.S. A Principled Taxonomy of Software Visualization, *Journal of Visual Languages and Computing*, 4(3), p.211-266.

[145] Production Systems Technologies, *OPSJ, RETE II*, http://www.pst.com/, 1999.

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

[147] 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.

[148] Roman G.-C.; Cox, K. C.; Wilcox, C.D.; Plun, J.Y; *Pavane: A System for Declarative Visualization of Concurrent Computations*, Technical Report, WUCS-91-26, April 1991.

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

[150] Sefika M., Campbell R.H., An Open Visual Model For Object-Oriented Operating Systems, *Fourth International Workshop on Object Orientation In Operating Systems*, Lund, Sweden, August 1995.

[151] 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.

[152] Sefika M., Sane A., Campbell R.H., Monitoring Compliance of a Software System With Its High-Level Design Models, *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, March 1996.

[153] 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.

[154] 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.

[155] Shilling J.J, Stasko J.T., *Using Animation to Design, Document and Trace Object-Oriented Systems*, Technical Report GIT-GVU-92-12, 1992

[156] 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.

[157] Smith, R.B., Wolczko, M., Ungar, D., From Kanzas to Oz: Collaborative Debugging When a Shared World Breaks, *Communications of the ACM*, Vol. 40., No. 4, pp. 72–79, April 1997.

[158] Standard Performance Evaluation Corporation, *SPEC JVM98 Benchmarks*, http://www.spec.org/osg/jvm98/, 1998.

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

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

[161] Stonebraker, M., Implementation of Integrity Constraints and Views by Query Modification, *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data*, June 1975.

[162] Swami, A., Optimization of Large Join Queries: Combining Heuristics and Combinational Techniques, *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 367-376, Portland, USA, May 1989.

[163] Swami, A., Gupta A., Optimization of Large Join Queries, *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 8-17, Chicago, USA, 1988.

[164] 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.

[165] Sweet, R.E., The Mesa Programming Environment. *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, ACM Press 1985, (ACM SIGPLAN Notices 20(7), July, 1985), pp. 216-229

[166] Swineheart, D.C., Zellweger, P.T., Hagmann, R.B., The Structure of Cedar. *ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, ACM Press 1985, (ACM SIGPLAN Notices 20(7), July, 1985), pp. 230–244

[167] Takahashi, S., Matsuoka, S., Miyashita, K., Hosobe, H., Yonezawa, A., Kamada, T., A Constraint-Based Approach for Visualization and Animation, *Constraints* 3(1): pp. 61–86, 1998.

[168] Tekinay S., Jabbari B., Hand-over and Channel Assignment in Mobile Cellular Networks, *IEEE Communications Magazine*, vol. 29, no. 11, November 1991, p. 42 - 46.

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

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

[171] Ullman, J.D., *Principles of Data and Knowledge Bases*, vol. I - II, Computer Science Press, Woodland Hills, CA, 1988.

[172] Ullman, J.D., Widom J., *A First Course in Database Systems*, Prentice Hall, Upper Saddle River, NJ, 1997.

[173] Ungar, D. M., Generation scavenging: A non-disruptive high-performance storage reclamation algorithm, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM Press 1984, (*ACM SIGPLAN Notices* 19(5), May, 1987), pp. 157-167.

[174] Ungar, D. M., Chambers, C., Chang, B.-W., Hölzle, U., Organizing Programs Without Classes, *Lisp and Symbolic Computation*: An International Journal, vol. 4., No. 3, 1991.

[175] Ungar, D. M., Chambers, C., Chang, B.-W., Hölzle, U., Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF, *Lisp and Symbolic Computation*: An International Journal, vol. 4., No. 3, 1991.

[176] Ungar, D., Lieberman, H., Fry, C., Debugging and the Experience of Immediacy, *Communications of the ACM*, Vol. 40., No. 4, pp. 38–44, April 1997.

[177] 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.

[178] Vion-Dury J.-Y., Santana M., Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems, *OOPSLA'94*, ACM Press 1994, pp. 65–84, 1994.

[179] Wahbe R., Efficient Data Breakpoints. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992. pp. 200–212, ACM Press 1992.

[180] 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.

[181] Wahbe R., Lucco S., Anderson, T.E., Graham S.L., Efficient Software-Based Fault Isolation. *Proceedings of the Symposium on Operating System Principles* 1993.

[182] Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J., Visualizing Dynamic Software System Information through High-level Models, *Proceedings of OOPSLA'98*, pp. 271-283, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

[183] 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.

[184] 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.

[185] Weiser, M., Program Slicing. *IEEE Transactions of Software Engineering*, Vol. SE-10, No. 4, pp. 352–357, July 1984.

[186] Weiser, M., Programmers Use Slices When Debugging, *Communications of the ACM*, Vol. 25, No. 7, pp. 446–452, July 1982.

[187] Welch I. and Stroud R., Dalang—A Reflective Java Extension. *Proc. of Workshop on Reflective Programming in C++ and Java.* UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan, ISSN 1344-3135, October 1998.

[188] West, A. Animating C++ Programs, Objective Software Technology, White Paper, 1993.

[189] Widom, J., Cochrane R.J., and Lindsay B. Implementing Set-Oriented Production Rules as an Extension to Starburst. *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pp. 275–285, Barcelona, Spain, September 1991.

[190] Widom J. and Finkelstein S.J. Set-Oriented Production Rules in Relational Database Systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 259–270, Atlantic City, New Jersey, May 1990.

[191] Widom J. The Starburst Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering*, 8(4), pp. 583–595, August 1996.

[192] Wilson, P.R., Uniprocessor Garbage Collection Techniques, *Proceeedings of International Workshop on Memory Management IWMM 9*, St. Malo, France, 17-19 Sept. 1992, Edited by: Bekkers, Y.; Cohen, J. Berlin, Germany: Springer-Verlag, p. 1–42, 1992.

[193] Wilson, P.R.; Moher, T.G. Demonic Memory for Process Histories. *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, USA, 21-23 June 1989, Published as *SIGPLAN Notices*, vol.24, (no.7), pp. 330–343, July 1989.

[194] Wong, E., Youssefi, K., Decomposition - A Strategy for Query Processing. *ACM Transactions on Database Systems*, vol. 1., No. 3, pp. 223-241, September 1976.

[195] *xDuel*, http://www.math.tau.ac.il/~frangy/, *currently inaccessible*, 1999.

[196] 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.

[197] Zhuge, Y., Garcia-Molina, H., Hammer, J., and Widom, J., View Maintenance in a Warehousing Environment. 1995 *ACM SIGMOD International Conference on Management of Data*, 1995. *SIGMOD Record*, vol. 24, pp. 316–327, 1997.

Appendix A Generalized Graph Matching

"Meanwhile, back at reality..."
G. Lucas

"If you're too busy to help your friends, you're too busy!"
L. Iacocca

Generalized Pattern Matching problem

The Generalized Pattern Matching problem is defined as follows:

INSTANCE: Two arbitrary directed graphs *G* and *H* in which each vertex is labeled with an element of the set *L* (note that two or more vertices may have the same label) and every edge is labeled with an element of the set *I* (note that all the outgoing edges of a single vertex have a unique label, but the outgoing edges of different vertices may have the same label).

QUESTION: Does *G* occur in *H*, i.e., is it possible to delete edges and vertices from *H* in such a way that the resulting graph is identical to *G*?

Theorem 1. The GPM problem is NP-Complete.

Proof: The GPM problem is in NP: a nondeterministic algorithm can guess the edges to be deleted from graph *H* to get a graph identical to *G* and then the algorithm can check that the guess was correct in time polynomial with respect to the number of edges and vertices in *G*.

We now show that 3-SAT polynomially reduces to GPM.

Given an arbitrary 3-SAT problem *(C,X)*, we construct the directed graphs *G* and *H* as follows. Let *C* consist of clauses $C_1, C_2, ..., C_m$, and let *X* consist of boolean variables $X_1, X_2, ..., X_n$.

In *G* there is one vertex labeled X_i for each variable X_i; call these *variable* vertices. For each clause $C_j = (x_{j1}, x_{j2}, x_{j3})$, where x_{j1}, x_{j2}, x_{j3} are literals over *X*, there are four vertices $C_{j1}, C_{j2}, C_{j3}, C_{j4}$ with their respective names as labels; we call all vertices with labels C_{ji} *clause* vertices. We also add the following edges:

$(C_{j1},X_{j1}), (C_{j2},X_{j2}), (C_{j3},X_{j3}), (C_{j1}, C_{j4}), (C_{j2}, C_{j4}), (C_{j3}, C_{j4})$

Pictorially the graph *G* constructed for clause C_j is shown in Figure 34.

In *H* there are two vertices x_i and $\bar{x}_i$, labeled X_i for each variable X_i; we call them *value* vertices. For each clause $C_j = (x_{j1}, x_{j2}, x_{j3})$ there are 28 vertices:

$11:C_{j1}, 21:C_{j1}, ..., 71:C_{j1},$

$12:C_{j2}, 22:C_{j2}, ..., 72:C_{j2},$

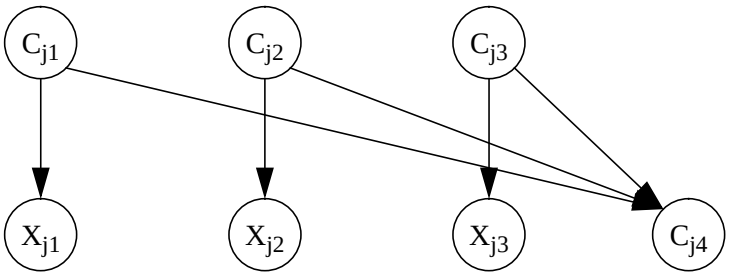

Figure 34. Subgraph corresponding to clause C_j in graph G

13:C_{j3}, 23:C_{j3}, ..., 73:C_{j3}, and

14:C_{j4}, 24:C_{j4}, ..., 74:C_{j4}.

The label of vertex *mn:C_{ji}* is C_{ji}.

We also add the following directed edges:

(11:C_{j1}, $\overline{x}_{j1}$:X_j),(12:C_{j2}, $\overline{x}_{j2}$:X_j),(13:C_{j3},x_{j3}:X_j),(11:C_{j1},14:C_{j4}), (12:C_{j2},14:C_{j4}), (13:C_{j3},14:C_{j4}),

(21:C_{j1}, $\overline{x}_{j1}$:X_j),(22:C_{j2},x_{j2}:X_j),(23:C_{j3}, $\overline{x}_{j3}$:X_j),(21:C_{j1},24:C_{j4}), (22:C_{j2},24:C_{j4}), (23:C_{j3},24:C_{j4}),

(31:C_{j1}, $\overline{x}_{j1}$:X_j),(32:C_{j2},x_{j2}:X_j), (33:C_{j3},x_{j3}:X_j), (31:C_{j1},34:C_{j4}), (32:C_{j2},34:C_{j4}), (33:C_{j3},34:C_{j4}),

(41:C_{j1},x_{j1}:X_j),(42:C_{j2}, $\overline{x}_{j2}$:X_j),(43:C_{j3}, $\overline{x}_{j3}$:X_j),(41:C_{j1},44:C_{j4}), (42:C_{j2},44:C_{j4}), (43:C_{j3},44:C_{j4}),

(51:C_{j1},x_{j1}:X_j),(52:C_{j2}, $\overline{x}_{j2}$:X_j), (53:C_{j3},x_{j3}:X_j), (51:C_{j1},54:C_{j4}), (52:C_{j2},54:C_{j4}), (53:C_{j3},54:C_{j4}),

(61:C_{j1},x_{j1}:X_j),(62:C_{j2},x_{j2}:X_j), (63:C_{j3}, $\overline{x}_{j3}$:X_j), (61:C_{j1},64:C_{j4}), (62:C_{j2},64:C_{j4}), (63:C_{j3},64:C_{j4}),

(71:C_{j1},x_{j1}:X_j), (72:C_{j2},x_{j2}:X_j), (73:C_{j3},x_{j3}:X_j), (71:C_{j1},74:C_{j4}), (72:C_{j2},74:C_{j4}), (73:C_{j3},74:C_{j4}).

All the edges of graphs G and H are assumed to be labeled by a pair composed of the labels of initial and terminal vertices. The set of all such labels is the set I. Set L contains elements C_{11}, ..., C_{m1}, C_{12}, ..., C_{m2}, C_{13}, ..., C_{m3}, C_{14}, ..., C_{m4}, X_1, X_2, ..., X_n.

The construction above is used to show that even if the graph G is bipartite and each vertex of it has no more than two outgoing edges, the problem still remains NP-complete. The construction can be simplified to prove the theorem for general graphs.

We claim that the 3-SAT problem *(C, X)* has a solution if and only if the instance of GPM constructed from it is satisfiable.

Assume that the instance of the 3-SAT problem *(C,X)* has a solution. Then there exists a selection of either x_i or $\overline{x}_i$ for each i = 1..n such that all the clauses are satisfied. That means that for each clause C_j there is a triplet of value vertices which satisfies it as a part of the solution. Moreover this triplet will correspond to one of the seven sextuplets in the graph H as introduced above. So for each clause C_j we can select a set of four clause vertices corresponding to the

triplet satisfying the clause. It is clear that the graph induced by these vertices together with the appropriate set of value vertices forms an occurrence of graph *G*.

Now assume that GPM has a solution. Then we have a set of clause vertices with labels C_{11}, ..., C_{m1}, C_{12}, ..., C_{m2}, C_{13}, ..., C_{m3}, C_{14}, ..., C_{m4} and a set of value vertices with labels X_1, X_2, ..., X_n that form a graph *G* in graph *H*. From the construction of the graphs *G* and *H*, we have a selection of either x_i or $\bar{x}_i$ for each i = 1..n, and for each C_j we have four value vertices with labels C_{i1}, C_{i2}, C_{i3}, C_{i4} that are connected between themselves and to the corresponding triplet of value vertices. It is clear that the selection of x_i's is the solution to the corresponding 3-SAT problem. Q.E.D.

Appendix B Detailed Data

"What am I doing here?"
Any recruit, any army

"How come I get all the hard questions?"
O. North

This appendix contains detailed data collected during the experiments. Table 17 gives the field assignment numbers for all SPECjvm98 benchmark applications. The "Frequency" column gives the frequency in assignments per second. The "Total" column gives a total number of fields across all programs that fall into a given assignment frequency range. The next seven columns give the total number of fields for a particular benchmark that fall into a given assignment frequency range. The last column expresses the total number as percentage of all fields. E.g. 209 fields or 19.22% are assigned less than 0.1 times per second. 3 fields of compress, 92 fields of jess, etc., have less than 0.1 assignment frequency.

The second section of the table gives the numbers for finer frequency ranges. Finally, the last section of the table provides the data in cumulative form. For example, it shows that 519 fields or 47.74% are assigned less than 10 times per second, though the first part shows that only 12.23% of the fields are assigned between one and ten times a second.

Frequency	Total	Compress	Jess	Db	Javac	MPEG	Jack	Ray	% of all
0.1	209	3	92	13	0	79	2	20	19.22
1	177	28	16	1	44	29	29	30	16.28
10	133	2	29	1	66	0	34	1	12.23
100	159	5	21	0	98	2	20	13	14.62
1K	178	0	11	4	130	15	17	1	16.37
10K	119	0	1	0	74	18	12	14	10.94
100K	87	2	15	0	30	25	9	6	8.00
1M	21	4	2	0	6	6	0	3	1.93
2M	4	4	0	0	0	0	0	0	0.36

Table 17. Field assignment frequency in SPECjvm98 applications

Frequency	Total	Compress	Jess	Db	Javac	MPEG	Jack	Ray	% of all
0.1	209	3	92	13	0	79	2	20	19.22
0.5	115	14	15	1	32	29	1	23	10.57
1	62	14	1	0	12	0	28	7	5.70
5	94	2	19	1	44	0	28	0	8.64
10	39	0	10	0	22	0	6	1	3.58
50	119	5	9	0	86	2	17	0	10.94
100	40	0	12	0	12	0	3	13	3.67
500	124	0	9	4	86	11	14	0	11.40
1K	54	0	2	0	44	4	3	1	4.96
5K	79	0	1	0	58	2	11	7	7.26
10K	40	0	0	0	16	16	1	7	3.67
50K	66	2	6	0	24	20	9	5	6.07
100K	21	0	9	0	6	5	0	1	1.93
500K	17	3	2	0	6	6	0	0	1.56
1M	4	1	0	0	0	0	0	3	0.36
2M	4	4	0	0	0	0	0	0	0.36
0.1	209	3	92	13	0	79	2	20	19.22
0.5	324	17	107	14	32	108	3	43	29.80
1	386	31	108	14	44	108	31	50	35.51
5	480	33	127	15	88	108	59	50	44.15
10	519	33	137	15	110	108	65	51	47.74
50	638	38	146	15	196	110	82	51	58.69
100	678	38	158	15	208	110	85	64	62.37
500	802	38	167	19	294	121	99	64	73.78

Table 17. Field assignment frequency in SPECjvm98 applications

Frequency	Total	Compress	Jess	Db	Javac	MPEG	Jack	Ray	% of all
1K	856	38	169	19	338	125	102	65	78.74
5K	935	38	170	19	396	127	113	72	86.01
10K	975	38	170	19	412	143	114	79	89.69
50K	1041	40	176	19	436	163	123	84	95.76
100K	1062	40	185	19	442	168	123	85	97.70
500K	1079	43	187	19	448	174	123	85	99.26
1M	1083	44	187	19	448	174	123	88	99.63
2M	1087	48	187	19	448	174	123	88	100

Table 17. Field assignment frequency in SPECjvm98 applications

Table 18 shows the breakdown of the query overhead and the overhead itself. The overhead is given in the rightmost column as a ratio of execution time with a query enabled to the execution time of the original program. The queries are numbered in the same order as they appear in Table 5 in section 4.4.1. The loading overhead is the difference between the time it takes to load and instrument classes using a custom class loader, and the time it takes to load a program during normal execution. The garbage collection time is the difference between the time spent for garbage collection in the queried program and the GC time in the original program. The first evaluation time is the time it takes to evaluate the query for the first time. The evaluation time is the time spent evaluating the query. This component does not include the first evaluation time. The first evaluation time and the evaluation time together compose the total evaluation time. For example, 3.1% of query 14 overhead is spent on instrumentation, 34.27% on garbage collection, 3.26% in the first evaluation, and 59.35% in subsequent reevaluations. The overhead of query 14 was 3.42.

Query #	Loading	GC	First evaluation	Evaluation	Overhead
1	44.09	45.75	0	10.14	1.02
2	70.95	23.31	0	5.73	1.10
3	85.63	8.54	0	5.82	1.24

Table 18. Breakdown of query overhead

Query #	Loading	GC	First evaluation	Evaluation	Overhead
4	13.05	1.04	0	85.89	1.18
5	8.80	1.49	0	89.70	1.27
6	6.38	0.68	0	92.92	1.37
7	0.49	0.01	0	99.48	5.82
8	13.20	6.02	0	80.76	1.18
9	24.94	0.85	0	74.19	1.09
10	2.85	0.38	0	96.76	1.83
11	44.63	12.87	0	42.48	1.23
12	10.56	0.22	0	89.21	1.98
13	1.04	1.44	0.95	96.55	2.13
14	3.104	34.27	3.26	59.35	3.42
15	0.04	1.13	0.07	98.74	229.24
16	0.01	0.92	0.01	99.05	157.25
17	0.03	1.17	0.01	98.7	77.21
18	1.93	0.21	0	97.8	6.35
19	0.71	1.28	1.11	96.88	227.51
20	0.17	0.32	0.27	99.22	930.06

Table 18. Breakdown of query overhead

Tables 19–21 give more detailed data on various aspects of query evaluation. Table 19 lists the total program execution time, the original program execution time, the query overhead time, the program execution time with loading overhead only (no query evaluation), the loading overhead itself, the garbage collection overhead, and the first query overhead. All times are given in milliseconds. The last two columns give the number of debugger invocations and the time to evaluate a single query ($T_{evaluate}$) in microseconds.

Query #	Total time	Original time	Overhead	Time with loading	Loading overhead	GC	FQ overhead	Invocations	$T_{evaluate}$
1	58526	57324	1202	57854	530	550	0		
2	16356	14786	1570	15900	1114	366	0	420000	3.73
3	26760	21471	5289	26000	4529	452	0	3794911	1.39
4	59596	50406	9190	51606	1200	96	0	65616330	0.14
5	64042	50406	13636	51606	1200	204	0	65616330	0.20
6	69191	50406	18785	51606	1200	129	0	65616330	0.28
7	293699	50406	243293	51606	1200	42	0	65616330	3.70
8	59495	50406	9089	51606	1200	548	0	47052800	0.19
9	55216	50406	4810	51606	1200	41	0	9861615	0.48
10	92450	50406	42044	51606	1200	160	0	9861615	4.26
11	20923	16971	3952	18735	1764	509	0	13381240	0.29
12	33665	16971	16694	18735	1764	37	0	39264920	0.42
13	122186	57324	64862	58000	676	939	620	82543	785.79
14	50672	14786	35886	15900	1114	12300	1171	631000	56.87
15	3890531	16971	3873560	18735	1764	44040	2988	7088454	546.46
16	7926776	50406	7876370	51606	1200	73000	621	131232660	60.01
17	3892187	50406	3841781	51606	1200	45000	720	75477945	50.89
18	15116	2380	12736	2626	246	28	0	100000000	0.12
19	57107	251	56856	655	404	730	635	10000000	5.68
20	233446	251	233195	655	404	753	648	10000000	23.31

Table 19. Execution times, overhead times, and invocation frequency

Table 20 lists the results of two experiments. The first three columns give the program execution times and overheads when custom selection code was not generated. For example, query 4 ran 68.5 times slower than the original program and 57.93 times slower than the optimized query. The three columns on the right side of the table give the results of queries executed with the

same value test disabled. For example, query 13 runs 3.43 times slower than the original program and 61% slower than the optimized query.

Query #	No fast selections time	No fast ratio to original	No fast ratio to optimized	No change test time	No change ratio to original	No change ratio to optimized
1	60443	1.05	1.03	58082	1.01	0.99
2	21993	1.48	1.34			
3	257305	11.98	9.61	26684	1.24	0.99
4	3452858	68.50	57.93			
5	3239434	64.26	50.58			
6	3270410	64.88	47.26			
7	3509850	69.63	11.95			
8	2197827	43.60	36.94			
9	531086	10.53	9.61			
10	561512	11.13	6.07			
11	359111	21.16	17.16	20680	1.21	0.98
12	1032903	60.86	30.68	39200	2.30	1.16
13				197089	3.43	1.61
14						
15				3985049	234.81	1.02
16						
17						
18	4646048	1952.12	307.35			
19						
20						

Table 20. Results for evaluations with no fast selections and no change tests

Table 21 shows results of evaluating queries without incremental reevaluation. The first three columns indicate the program execution times and overheads without incremental reevaluation. For example, query 2 ran 613 times slower than the original program and 554 times slower than the optimized query. The four columns on the right side of the table give the results of queries

that ran so long that we stopped query reevaluation after the first 100,000 evaluations and estimated the total overhead. The first two columns give the time it took to execute the program for 100,000 query evaluations and the estimated full execution time. The last two columns show the overheads. For example, query 3 has estimated overhead of 7135 over the original program and 5725 over the optimized query.

Query #	Non incremental	Noninc ratio to original	Noninc ratio to optimized	Non incremental 100K assignments	Non incremental calculated time	Noninc ratio to original	Noninc ratio to optimized
1	67932	1.18	1.16				
2	9064124	613.02	554.17	5756038	24128044	1631.81	1475.18
3				4057943	153201990	7135.29	5725.03
4				86822	23945248	475.04	401.79
5				86750	23898004	474.11	373.16
6				95443	29602032	587.27	427.83
7				89726	25850746	512.85	88.01
8				79799	13880635	275.37	233.30
9				68838	1868098	37.06	33.83
10				70319	2014149	39.95	21.78
11				1345266	177759312	10474.29	8495.88
12				786181	302046662	17797.81	8972.12
13	1258899	21.96	10.30				
14				4636672	29178886	1973.41	575.83
15				2986276	210494790	12403.20	54.10
16				115974	86097036	1708.07	10.86
17				96914	35153688	697.41	9.03
18	12407634	5213.29	820.82				

Table 21. Non-incremental evaluation results

Query #	Non incremental	Noninc ratio to original	Noninc ratio to optimized	Non incremental 100K assignments	Non incremental calculated time	Noninc ratio to original	Noninc ratio to optimized
19	374151	1490.64	6.55				
20	1406216	5602.45	6.02				

Table 21. Non-incremental evaluation results

Table 22 gives the predicted query overhead as a function of update frequency. For example, the predicted overhead of a low-cost selection query on a field updated 500,000 times per second is 6.5%; the predicted overhead of a high-cost query with the same frequency is a factor of 3.13.

Frequency	Low cost	High cost
0.1	1.000000013	1.000000426
0.5	1.000000065	1.00000213
1	1.00000013	1.00000426
5	1.00000065	1.0000213
10	1.0000013	1.0000426
50	1.0000065	1.000213
100	1.000013	1.000426
500	1.000065	1.00213
1000	1.00013	1.00426
5000	1.00065	1.0213
10K	1.0013	1.0426
50K	1.0065	1.213
100K	1.013	1.426
500K	1.065	3.13
1M	1.13	5.26
2M	1.26	9.52

Table 22. Predicted slowdown