AFRL-IF-RS-TR-2003-32
Final Technical Report
February 2003

PERPETUAL TESTING

Perdue University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E099

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-32 has been reviewed and is approved for publication.

APPROVED: Deborah A. Cerino

DEBORAH A. CERINO
Project Engineer

FOR THE DIRECTOR: James A. Collins

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	FEBRUARY 2003	Final Jan 97 – Jul 02

4. TITLE AND SUBTITLE
PERPETUAL TESTING

5. FUNDING NUMBERS
C - F30602-97-2-0034
PE - 62301E
PR - E099
TA - 01
WU - 01

6. AUTHOR(S)
Michal Young

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Prime:
Purdue University
1063 Hovde Hall
West Lafayette IN 47907

Sub:
University of Oregon
5219 University of Oregon
Eugene OR 97401-5219

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-32

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Deborah A. Cerino/IFTB/(315) 330-1445/ Deborah.Cerino@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The Perpetual Testing project was part of the High Assurance Cluster of projects in the DARPA Evolutionary Design of Complex Software Program. The Perpetual Testing project as a whole was a collaboration of Purdue University (continued at the University of Oregon), University of Massachusetts, and the University of California at Irvine, under three separate Air Force contracts. This report describes the portion of work conducted at Purdue University and the University of Oregon under cooperative agreement F30602-97-2-0034 with the Air Force Research Laboratory, Rome NY.

The Perpetual Testing project goal was to develop technologies to support seamless, perpetual analysis and testing of software through deployment and evolution. The current development paradigm treats testing as a phase that succeeds development and precedes delivery. The Perpetual Testing project's focus was to build a foundation for treating analysis and testing as on-going activities to improve quality assurance through generations of a project.

14. SUBJECT TERMS
Software Testing, Flow Analysis

15. NUMBER OF PAGES
34

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1 Introduction

The Perpetual Testing project was a member of the High Assurance cluster of projects in the DARPA program *Evolutionary Design of Complex Software (EDCS)*. The Perpetual Testing project as a whole was a collaboration of Purdue University (continued at University of Oregon), University of Massachusetts, and University of California, Irvine. This report describes only the portion of the work conducted at Purdue and University of Oregon under cooperative agreement F30602-97-2-0034 with U.S. Air Force Research Laboratory, Rome New York 13441-4514.

The current development paradigm treats testing as a phase that succeeds development and precedes delivery. The goal of the Perpetual Testing project was to build a foundation for treating analysis and testing as on-going activities to improve quality assurance through generations of a project. This goal evolved somewhat during the course of the research, and some of it was retargeted and carried forward into the DARPA DASADA program, but overall the focus has remained on key aspects of Evolutionary Development of Complex Software: on methods and supporting technology suitable for evolving software, including analysis and test that continues beyond initial deployment, and on techniques that exploit modularity to permit piecemeal analysis and testing of large, complex systems.

2 Main sub-projects and results

The main parts of the Perpetual Testing project at Purdue and later at University of Oregon were

Residual testing: Extending test coverage monitoring from its conventional place in the development environment to continued monitoring of deployed software.

Design refactoring for analysis: Supporting complex relations between the "as built" structure of an implementation and a logical structure that is more amenable to modular, incremental analysis.

Object protocol specification and checking: Enriching class interfaces in object-oriented programs with precise, checkable ordering constraints on method calls.

Flow analysis as Swiss army knife: Development of lightweight, flexible support for a variety of analyses and transformations using flow analysis algorithms.

These sub-projects reached different stages of development during the project. The residual testing project resulted first in a "proof of concept" prototype tool, some experimental measurements, and a paper. The initial results were promising, but the tool was fragile and completely unsuitable as a vehicle for technology transition or even for use by other researchers. We decided to invest resources in producing a new tool based

on the same concepts but redesigned and re-implemented from the ground up. Fortunately the student programmer assigned this task, Carl Howells, was very knowledgeable and talented. The resulting tool, Gretel, is in use by several commercial organizations and has also been incorporated in a research project at Georgia Tech. The Georgia tech project, led byMary Jean Harrold, is building on a vision consistent with our original goals for residual testing.

Design refactoring for analysis has resulted in a "proof of concept" prototype, and results with some model applications have been reported in the literature. However, we believe there are years of effort required before it will reach a level of maturity comparable to our Gretel tool for residual testing, and it would not even be sensible to devote resources to a production quality tool until more of the fundamental problems are worked through. This project was the dissertation topic of Yung-Pin Cheng, who continues to pursue the research in his new position on the faculty of National Taiwan Normal University.

Object protocol specification and monitoring has also resulted in a "proof of concept" prototype, and has been described in the literature. There has lately been a flurry of research by others with similar goals, most (like ours) implemented as extensions or annotations for Java programs. An aspect of our approach that remains unique and, in our view, crucial is the way static interface compatibility checking is combined with dynamic run-time checks. If continued to conclusion, this work should lead to systems in which architectural descriptions are linked to object protocols in design and implementation and carried forward to dynamic checks during testing and after deployment, as well as richer checking for deviations from architectural design and interfaces during software evolution. This work has been carried out in collaboration with researchers at Ohio State University and continues under the DASADA program. We are seeking other sources of funding to continue it after the conclusion of the DASADA program.

We did not anticipate a major thrust in flow analysis in the original vision for our part of the Perpetual Testing project, although it was a centerpiece in the technology developed by one of our collaborators (the FLAVERS tool at University of Massachusetts) Our first foray in this direction was in service to the redesign for analysis thrust, when we noticed that flow analysis algorithms could be used to extract useful information from legacy concurrent programs. The sort of analysis we envisioned was completely outside what FLAVERS could be used for, even though at some very basic level we were applying the same flow analysis algorithms. An M.S. student, Jeyde Rajamani, produced a pair of prototype tools that carried out these analyses. They were very efficient but each was custom-built for a single task. This led us to design more general tool support in the form of a "little language," which eventually became GenSet. The GenSet effort was continued but retargeted as we turned our attention to the goals of the DASADA program. Development of GenSet will continue until the conclusion of the DASADA program, where currently it is focused on design information fusion (including but not limited to information extracted from implementations). We believe that it will have many other uses, and that in particular it will serve researchers who wish to quickly prototype and test novel applications of flow analysis without building yet another tool from scratch. The maturity of GenSet is between that of Gretel and the two "proof of concept"

prototypes. We have achieved performance much better than the most similar competing tool (Grok, which is based on relational algebra rather than flow analysis), while increasing expressiveness. Some work on packaging remains to be done before a wide public release, and we believe there are years of productive work ahead on applications and extensions.

3 Technical details

Details of the sub-projects are best related through papers. We have attached a set of representative papers to this report.

4 Discussion and Lessons Learned

A variety of technical lessons have been learned on the various sub-projects, but many of them come down to a single rule of thumb: Any technology that will successfully scale for use in complex, evolving software systems must necessarily be usable on small pieces of a system in isolation. Gretel has succeeded partly because, even though it constructs some global data structures, it can be applied equally well to a full system, parts of a system, or a single Java source file. GenSet is explicitly designed to make use of whatever information can be extracted from designs and implementation, however incomplete. Our work on object protocol specifications treats individual superclass/subclass and client/server pairs individually, and for this reason can be very efficient despite the fact that the algorithm used for conformance checking is inherently exponential in the worst case. Our work on redesign for analysis, on the other hand, was conceived initially as a "top down" approach in which one began with an overall view extracted from a design or implementation, and this will be an obstacle to scaling it up until we can devise a version of the approach that is more local and incremental. The original vision for the Perpetual Testing project included integration with a variety of related projects, not only with our direct collaborators at University of California at Irvine and the University of Massachusetts. but also with other EDCS projects producing architecture description languages, hypertext presentations of software, and a variety of other technology. A good deal of energy was invested at EDCS meetings trying to define a shared vision and coordination, but in the end orchestration of such a grand scheme was probably doomed — no project, including ours, could afford to put a large number of other risky projects on their critical development path. But while the grand vision of our technology integrated into a seamless whole with other EDCS technology did not come to pass, it was still quite useful to interact with several other projects with related goals, particularly when visions of how to reach those goals differed. Much was accomplished, in spite of or perhaps even because there was, in the end, no unified vision across projects, but rather a creative tension of many visions.

Our work on Perpetual Testing led, after further development through the DASADA program, to what we believe was a very promising plan for a technology evaluation

experiment on a major, real-world military application (AWACS), in collaboration with the commercial developer of that application. Unfortunately we have learned that the program to which that project was proposed has been canceled. We are very disappointed that we will not be able to test and demonstrate the results of our research effort in this manner. On the other hand, the burgeoning open source software movement opens other avenues for practical impact and "in vivo" evaluation. We have already released the Gretel residual test coverage monitoring tool as an open source tool on SourceForge, where it has spawned two new projects (Hansel and GretAnt) being carried forward by others building on our work; this is in addition to the Georgia Tech research project mentioned above. We expect to release GenSet in a similar manner.

APPENDIX A

Residual Test Coverage Monitoring

Christina Pavlopoulou
Purdue University
Electrical and Computer Engineering
West Lafayette, Indiana 47906 USA
+1 650 494 3499
pavlo@ecn.purdue.edu

Michal Young
University of Oregon
Computer Science Department
Eugene, Oregon 97403-1202 USA
+1 541 346 4140
michal@cs.uoregon.edu

ABSTRACT

Structural coverage criteria are often used as an indicator of the thoroughness of testing, but complete satisfaction of a criterion is seldom achieved. When a software product is released with less than 100% coverage, testers are explicitly or implicitly assuming that executions satisfying the remaining test obligations (the residue) are either infeasible or occur so rarely that they have negligible impact on quality. Violation of this assumption indicates shortcomings in the testing process.

Monitoring in the deployed environment, even in the beta test phase, is typically limited to error and sanity checks. Monitoring the residue of test coverage in actual use can provide additional useful information, but it is unlikely to be accepted by users unless its performance impact is very small. Experience with a prototype tool for residual test coverage monitoring of Java programs suggests that, at least for statement coverage, the simple strategy of removing all probes except those corresponding to the residue of coverage testing reduces execution overhead to acceptably low levels.

Keywords

Testing, coverage, instrumentation.

1 INTRODUCTION

Quality assurance activities in the development environment, including systematic dynamic testing, cannot be performed exhaustively, therefore they always depend on models. Static analysis depends on the fidelity of models extracted for analysis. Statistical testing for reliability estimation depends on models of program usage. Partition testing depends on the models used to divide program behaviors into classes that should be "covered." Discrepancies between these models and actual program behavior are valuable information, even when they don't result in observed program failures, because they indicate how quality assurance activities in the development environment can be improved. For example, having some way of judging when "enough" testing has been done can be valuable in a negative sense. Test adequacy criteria indicate, not when testing is definitely adequate, but when there is evidence that a set of tests is inadequate because some significant class of program behaviors has never been tested.

The family of structural coverage criteria (statement coverage, branch coverage, dataflow coverage, etc.) are based on syntactic models of program control and data flow. These syntactic models are conservative in the sense that they include not only all control and data flows that will occur in any execution, but also many infeasible paths that can never occur. It is (provably) impossible to determine exactly which paths are infeasible. Thus even exhaustive testing would often fail to satisfy structural coverage criteria.[1] When a software product is released without 100% coverage, testers are explicitly or implicitly assuming that the remaining test obligations (the residue) is either infeasible, or occurs in a vanishingly small set of possible executions.

We cannot completely avoid models and assumptions. What we can do is validate the models we use. If we have implicitly or explicitly assumed that a particular path or region in code is never, or almost never executed, then knowing that an execution of that path or region has occurred in actual use is valuable information, even if the software performed correctly in that case. However, in current practice this is not possible since there is a sharp divide between unit, integration, and system testing on the one hand, and feedback from deployed software on the other. While developers have access to a variety of monitoring tools in the development environment, monitoring in the deployed environment is typically limited to error and sanity checks, and the channel from users back to developers is just a list of trouble reports. Residual test coverage monitoring exploits the opportunity provided by increasingly ubiq-

ICSE '99 Los Angeles CA

[1] Frankl [5] has defined variant criteria relative to feasible paths, but practically speaking that does not change the problem considered here.

uitous networking to enrich the feedback channel and validate one kind of model used in the development environment.

The remainder of this paper is organized as follows: Section 2 discusses potential objections to residual test coverage monitoring and motivates investigation of the problem of performance impacts. Section 3 sketches the design of a prototype tool constructed to evaluate a simple approach to minimizing performance impacts in the deployed environment, and Section 4 reports measurements obtained with the tool. Sections 5 discusses related work and open issues, and Section 6 concludes.

2 RESIDUAL TEST COVERAGE MONITORING

The purpose of residual test coverage monitoring is to provide richer feedback from actual use of deployed software to developers, helping developers validate and refine the models they have relied upon in quality assurance. To be successful, run-time monitoring must overcome at least two classes of potential objections from users.

The first class of potential objections is related to security, confidentiality, and privacy. There is probably no complete solution to the confidentiality problem since the most innocuous seeming communications to developers could convey confidential information gathered from users.[2] Communication of any information from actual use will be unacceptable to some users and in some application domains. This class of objections can be partially avoided by targeting the beta test phase, in which users are already used to providing some information to developers, and by limiting communication to forms that are observable and controllable by the end-user (for example small textual e-mail messages that the user can inspect before sending).

The second class of potential objections is related to performance, including degradation of responsiveness and perturbation of real-time behavior. Sensitivity to performance concerns differs widely among different classes of software, and there will be some applications in which no run-time monitoring is acceptable. On the other hand, we believe there is a large class of applications in which some very modest performance degradation is acceptable, particularly in the beta test phase.

In the longer term, we believe it will be useful to provide deployed software with adjustable levels and focus to address performance requirements with user control to address concerns of security, and confidentiality. One can easily imagine uses for very detailed monitoring, sufficient to completely reproduce an unanticipated behavior, and there are no doubt situations in which the inevitable overhead of such detailed monitoring would be acceptable. However, we have chosen to concentrate initially on the other end of the spectrum, establishing that some useful information can be gathered even when the tolerance for performance degradation is small.

The prototype tool described in the following sections of this paper selectively monitors execution of Java programs for a simple test coverage criterion, equivalent to statement coverage. Initially, all basic blocks are monitored, but subsequent to a few test runs the program can be instrumented again, removing monitoring of basic blocks that have already been covered and leaving only the probes needed to recognize execution of the "residue" of unexecuted code. Since the high-frequency program paths tend to be executed on almost every program run, the cost of selective reinstrumentation quickly decreases. For the programs we have tested, after a few iterations of testing and reinstrumentation the run-time overhead of execution monitoring becomes insignificant.

3 A RESIDUAL COVERAGE TOOL FOR JAVA

We have implemented a simple residual test coverage monitoring tool for Java applications and applets. The prototype tool provides a record of which basic blocks (hence which statements) have been executed at least once in a series of test runs. An XEmacs library provides a way to view cumulative coverage graphically, by highlighting regions of code that have not been executed. This section sketches the overall design of the tool and a few details of its implementation; full details can be found in the M.S. thesis of the first author [13]. Except for the choice of object code instrumentation, which is specific to Java, the design should be equally applicable to traditionally compiled procedural and object-oriented languages.

Overall process

Figure 1 illustrates the overall process of program instrumentation and coverage monitoring. An object code instrumenter places instrumentation in the program, referring to a cumulative coverage table to place probes only on the as-yet unexecuted "residue." Initially nothing has been covered, so a probe is placed in every basic block of the program. The instrumented class files are executed by an (unmodified) Java interpreter, and as a side effect the instrumentation creates a file recording which basic blocks were executed. After one or several test runs, the instrumenter is invoked again to place probes only in the blocks that remain unexecuted.

Key structures

As one would expect, the prototype tool is designed to

[2] In standard terminology, this information is a potential *covert channel*. The basic block information described in the following sections can be used as a covert channel by including tests of of confidential information in the application, so that execution of a particular block indicates the outcome of the test.

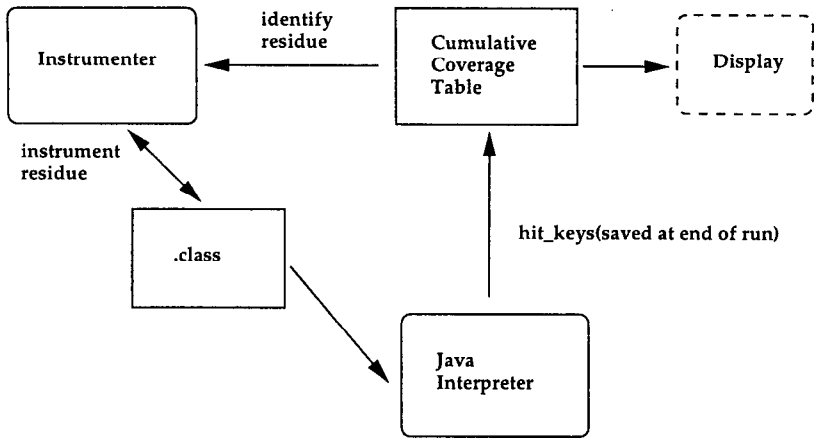

Figure 1: Instrumentation process: in every iteration the basic blocks that were not covered in previous executions are instrumented and the new class file is executed in order to collect coverage information.

minimize execution time overhead by moving as much computation as possible to the instrumentation and post-processing phases. Residual test coverage monitoring provides one extra opportunity for optimization, compared to conventional coverage monitoring: Since the number of blocks monitored may be far fewer than the number of blocks in the program, we can use smaller keys to index blocks at run time. This introduces an extra level of indirection in the auxillary tables which are produced at instrumentation time and interpreted in post-processing.

The key data structures are:

- An *Id Table* which associates a unique identifier (*block id*) with each basic block in a program. The Id Table must be stable in the sense that, if the same program is compiled twice without changes, the same unique identifiers are associated with each basic block.
- A *Coverage Table* recording the basic blocks (elements of the Id Table) that have been covered in previous executions; this corresponds to the cumulative coverage table in Figure 1. It could be a simple list of block identifiers or, as in our implementation, an array of booleans indexed by block id.
- A *Correspondence Table* that associates integers (*hit keys*) with block ids. Hit keys are integers in the range $0 \ldots n$, where n is the number of basic blocks that have not been covered when the instrumenter is run. When n is small, a more efficient code sequence can be used for each run-time probe. The instrumenter creates a new set of hit keys each time the program is reinstrumented, and creates the correspondence table so that a post-processor can update the coverage table from a smaller table of hit keys.
- When an instrumented program is executed, a *Hit-Table* indexed by hit keys is maintained. The hit table is a simple array of booleans, initially all false. `HitTable[i]` will be set to true when the i-th basic block has been executed.

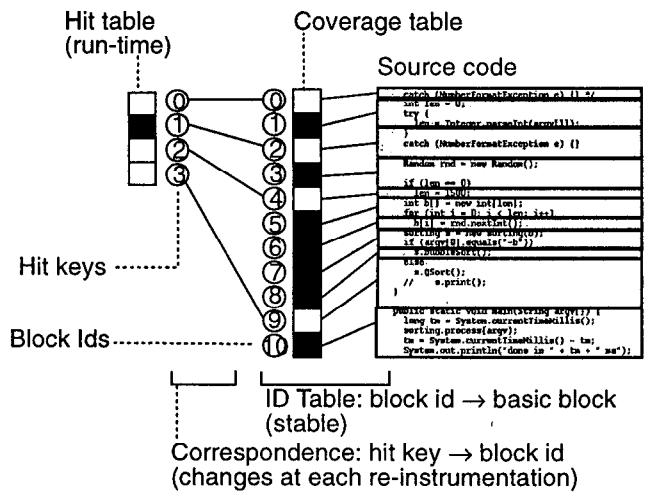

Figure 2: Tables for maintaining the information needed for selective monitoring

The relation among the various tables appears in Figure 2.

Execution of a probe at run time causes a single boolean to be set in the hit table. The table is dumped to a

file at the end of execution, and a post-processor updates the coverage table correspondingly. In the case of deployed software, the usual case would be an empty hit table, and the writing and post-processing phases could be skipped except in the exceptional cases when an untested region of code has been executed; the post-processing phase would also take place in the development environment rather than the field environment. In our prototype, however, we simply dump and process the hit table after every run.

Object Code Instrumentation

While the structures and processing described above should apply to most procedural and object-oriented languages, the strategy we chose for instrumenting programs was strongly influenced by the target language. Java programs and applets are typically compiled to a byte-code format and interpreted by a byte-code interpreter called Java virtual machine [10]. For other languages, an instrumenter based on source-to-source translation or modification of an existing compiler might have been a better choice, but for Java we found direct instrumentation of object code (class files) more attractive.

Java byte code is stack-oriented, whereas the instruction set architectures of the dominant contemporary processors are register-oriented. The only practical way to insert instrumentation that affects register allocation is to insert the instrumentation at the level of source or intermediate code, leaving adjustment of the register allocation to the compiler back end. In contrast, it is relatively easy to insert a few stack-oriented instructions in a stack-oriented instruction stream, leaving the stack unchanged. Equally important is what is not possible in stack-oriented code: In a machine with a generous set of registers, it is worthwhile to work very hard at minimizing the number of memory accesses by making clever use of registers (e.g., as in the path profiling technique of Ball and Larus [2]), but stack code presents no such temptation.

At a more pragmatic level, we had access to standard Java packages for reading, interpreting, and writing Java class files, which greatly reduced the effort required to produce an object-code instrumenter. It is relatively simple to extract control structure and other information from the assembly-language level of information provided by these Java packages, and the files also contain debugging information that served our need for associating regions of object code with regions in the source files.

Basic blocks are identified in the byte codes using standard algorithms [20]. At the head of each basic block that has not previously been executed we insert a call to `Monitor.hit(hit_key)`, a method that stores one value in a boolean array. If the hit key is less than 256, the code is

```
bipush hit_key
invokestatic #index
```

where `index` is the location of the address of method `Monitor.hit(int)` in the constant pool. Class `Monitor` is the run-time library which encapsulates the hit table and provides initialization and finalization. Directly accessing the array might be faster than a method call, but would require more inline code.

In Java, every class is allowed to have a "main" method, and the user can begin execution from any class, so we simply instrument the main methods of all classes. This simple expedient was adequate for our purposes, although obviously inappropriate for a production-quality tool.

Producing valid bytecode

Instrumentation must be inserted in such a way that the Java Virtual Machine Specification is not violated. The Java interpreter checks each class file to determine that it it conforms to the format dictated by the virtual machine specification and that appropriately typed arguments are on the top of the stack when needed. The stack-typing requirement is easily met, since the inserted instrumentation has no net effect on the stack contents (it pushes and then consumes one argument), but the maximum stack depth of each method must be incremented by four to accomodate the added instructions. Method calls in Java are made by indirection through a table of constants (recall the `invokestatic #index` instruction in the sample code above), so entries for the instrumentation methods must also be added to the constant pool of each instrumented class. In addition, the target addresses of control transfer instructions and the exception table must be adjusted. Target addresses of `lookupswitch` and `tableswitch` must also be adjusted and, in some cases, aligned by inserting zero bytes.

Multi-threading

Java programs typically have multiple concurrent threads of control. Execution of a coverage probe at run-time is simple and does not require mutual exclusion, on the fairly conservative assumption that concurrent stores of the same boolean value to a memory location will result in that value being stored. The only real issues we encountered were in ensuring proper initialization and finalization (dumping to a file) of the run-time table.

Multithreaded programs can be applets or programs that use threads or graphics. In the case of applets, execution of the program begins from the constructor

of the applet and ends at the destroy method. Thus, unlike single-threaded programs, we do not need to instrument the main method. While applets are designed primarily for execution in web browsers, we executed them in the appletviewer application to relax the usual web browser security restriction against writing to a file.

The case of multithreaded Java applications which are not applets is more difficult. The beginning of execution is easy to recognize (execution begins at the `main` method of some class), but termination can occur in different places; basically whenever there is a system call to exit the program. In multithreaded testcases the call to dump the file with execution information was inserted manually before the approriate system calls.

4 EXPERIENCE

We have measured the performance impact of residual test coverage monitoring on four applications ranging in size from 55 to 4000 lines. The experiments were conducted in a SPARC 5 processor at 70 MHz, running the Solaris operating system. Two of the applications, ArcTest and Sorting, are an applet and application taken from the examples distributed with the Sun Java Development Kit (JDK), version 1.0. The other two are Java applications that were developed in our laboratory, the larger of these being the residual instrumentation tool itself. In general, no changes were needed in the code to execute the different test cases, except for minor changes to catch the beginning and ending of execution in multithreaded programs as discussed in the previous section. Java applications were executed by the Java interpreter provided in the JDK, and Java applets were executed in the appletviewer application provided in the JDK.

We observed generally that while the execution of a fully instrumented program may have significant overhead, after a few iterations of test execution and reinstrumentation the overhead reduces dramatically. The additional execution time required for the instrumentation statements of a program depends mainly on the size and number of loops as well as on the size of the input data. In practice one would reinstrument only after several test executions, but for measurement purposes we reinstrumented after processing each test case.

The execution time (elapsed wall time) of each test program was measured with the Java system service for time measurement. The execution times in the tables below are averages over ten runs, rounded to the nearest 0.1 second. The first row of each table contains the execution times of the uninstrumented application for different data inputs. The execution times for the instrumented program (second row in the tables) were measured as follows: initially instrumentation is inserted in every basic block and the program is executed (10 times)

	test 1
original	4.5
instrumented	5.0
# blocks instrumented	29

Table 1: ArcTest execution times in seconds and number of blocks executed

with the first test case (time in first column); then the program is reinstrumented, placing probes inserted in those basic blocks that did not execute previously and the program is run with the second test case (time in second column); the process is repeated for all test cases. The last row of each table contains the number of basic blocks that were instrumented for each test case.

The first program, ArcTest, is a simple applet of approximately 80 lines that draws on the screen a number of arcs with random beginning and end. In this case the overhead even from complete instrumentation is small relative to the cost of the graphics operations. Reinstrumentation was not performed on this example; we include it here for comparison because it is the only applet in this group of programs.

The Sorting program (55 lines) sorts an array of randomly generated numbers using either binary or quick sort. The first test case is sorting numbers with binary sort, the second test case is sorting numbers with quicksort and the third is sorting numbers with quicksort again. In the program distributed with the JDK, run times vary considerably depending on the sorting algorithm; we made them comparable by increasing the size of the arrays for the faster algorithms, to make the trend in instrumentation overhead easier to see in the tables (we have also measured the program without this modification, with similar results). In the first test, the instrumentation overhead is nearly 130%. In the second test, which executes a different sorting algorithm and therefore mostly in a different region of code, overhead remains very high at nearly 160%. The third test case uses the same sorting algorithm as the first, and therefore executes in the same region of code. In this case no probes are executed, and the program with residual instrumentation executes in essentially the same time as the uninstrumented program.

Elevator (650 lines) is a simulation program for the operation of two elevators. Unlike the sorting algorithm, it does not consist primarily of tight loops, so even the overhead of complete instrumentation is only about 15%. After two iterations of testing and reinstrumentation the overhead is reduced to 1.5%.

Finally, the instrumentation system itself (approx. 4000

	test 1	test2	test3
original	24	20	20
instrumented	55	52	20
# blocks instrumented	43	25	6

Table 2: Sorting execution times in seconds and number of blocks instrumented

	test 1	test 2	test 3
original	5.5	6.1	6.5
instrumented	6.3	6.8	6.6
# blocks instrumented	323	240	119

Table 3: Elevator execution times in seconds and number of blocks executed

lines) has been instrumented and has been used to instrument the program Sorting. The execution overhead of full instrumentation is approximately 9.3%. After two iterations of testing and reinstrumentation, the overhead of residual coverage monitoring falls below the level that we were able to measure.

5 DISCUSSION

Related work

We are not aware of prior attempts to gather structural test coverage information from deployed software, although Cusamano and Selby report that Microsoft gathers detailed use profiles from specially instrumented versions of its products [3, pp. 377–378].

One class of "residual" monitoring that is already common, though, is run-time checks of assertions. As with coverage monitoring, tolerance of run-time overheads for assertion checking differs between the development environment and the deployed environment. For example, evaluating a quantifier by enumerating elements of a finite set may be acceptable when testing software in the development environment, but unacceptable for deployed software. Some assertion checking systems rule out very expensive predicates entirely (ADL [19] takes this approach), while others like Gnu Nana [12] provide flexible ways to deactivate some checks while leaving others active. The Anna project [11, 18] is the root of much of the recent research in enriching run-time checks [16, 17, 19].

	test 1	test 2	test 3
original	4.3	1.7	4.4
instrumented	4.7	1.8	4.4
# blocks instrumented	1000	614	547

Table 4: Instrumentation program execution times in seconds and number of blocks executed

Instrumentation for cheap run-time coverage monitoring has obvious relations to cheap instrumentation for other purposes, including performance profiling. Coverage monitoring requires less information than performance profiling, since the latter does not distinguish whether code is executed once or one thousand times, and this makes the design of cheap coverage monitoring considerably simpler than cheap performance profiling. Agrawal has shown that the number of program probes needed for basic block coverage monitoring can be reduced considerably by using control flow analysis (pre- and post-dominator information) [1]. The relative savings in the cost of residual coverage monitoring over full monitoring would be correspondingly reduced if Agrawal's technique were applied, and vice versa the savings from Agrawal's technique would be less significant if applied to residual coverage obligations after a few tests. Nonetheless it may be be useful to combine the techniques, not so much to achieve further reductions in execution time overheads as to reduce space overheads, which residual coverage monitoring is less effective at reducing.

The object code instrumentation approach discussed in Section 3 is related to a variety of tools for instrumenting binary machine code [7, 21, 6, 15]. Among tools suited for instrumenting Java byte codes, the most closely related is Lee's Bytecode Instrumenting Tool (BIT) [8, 9]), which was developed contemporaneously but independently. BIT is more general than our tool, providing a way to insert method calls in user class files. In principle, a tool like our residual test coverage monitor could be more simply constructed using a tool like BIT, but several current limitations of BIT prevent us from using it in that way. BIT allows the user to specify the instrumentation statements, but it does not provide the capability of removing monitoring code automatically, nor does it maintain the links we require between source code and bytecode locations. Moreover, BIT does not (yet) properly adjust exception handling code to account for instruction relocation.

Open Issues and Future Work

As stated earlier, our tactic in exploring residual testing is to first establish that some useful information can be gathered even when the tolerance for performance degradation is small before moving on to gather richer and potentially costlier information. We have so far investigated residual monitoring of only the simplest test coverage criterion, albeit the one most used in practice.

Many of the more stringent test coverage criteria involve sub-paths in program control flow, rather than individ-

ual points. The best known of these is data flow coverage testing, in which execution of particular "definition use" pairs (what compiler writers know as "reaching definitions") are monitored. The interested reader may refer to [14] for definitions and an in-depth discussion of data flow testing.

It is not clear whether the run-time performance impact of residual test coverage monitoring can be made insignificant for data flow coverage and other path-based coverage criteria. In the worst case we might have code like the following:

```
if (cond1)
   x = ...; // 1
else
   y = ...; // 2

if (cond2)
   z = x;   // 3
else
   z = y;   // 4
```

with the assumptions that

- the code occurs in a high frequency loop,
- in each of the two "if" statements, the "then" branch is taken 50% of the time and the "else" branch is taken 50% of time,
- when the "then" branch is taken in the first "if" statement, the "else" branch is always taken in the second, and when the "else" branch is taken in the first "if" statement, the "then" branch is always taken in the second.

We observe that the definition-use pairs (1,3) and (2,4) are never executed, even though every point in the path is executed 50% of the time. In this case, unless we transform the code, we cannot avoid monitoring at a point that is executing on 50% of the loop iterations. In some cases (including the example above) such code can be transformed to separate frequently and infrequently executed paths, but such transformations are expensive in space. Empirical evidence is needed to determine how often such pathological cases occur in real programs.

Notification that a user has executed code in a way that was not adequately tested leaves to testers the task of determining how to reproduce a behavior that they have not previously encountered in testing. Even the limited information provided by our current tool should be useful in focusing effort on the presumably small number of reported blocks rather than the whole population of uncovered blocks, but it would be more useful to have additional information such as input data, intermediate data values, or parts of the execution path leading to the newly exercised code. In case testers cannot easily reproduce the behavior, it would be possible to provide selected users with versions of the application that are specially instrumented to provide more information about the particular behaviors of interest.

Security and confidentiality concerns may be more difficult to overcome than the performance issues that would result from providing additional information to developers. As noted earlier, seemingly innocuous information communicated from the users' environment to developers is a potential covert channel which could be used by an unscrupulous developer to obtain confidential information. Increasing the amount of information communicated exacerbates potential security and confidentiality concerns. Even a user who is willing to trust that developers are not encoding confidential information in coverage records may balk at providing input data from actual executions.

6 CONCLUSIONS

We have argued for monitoring of deployed software, particularly in beta testing, that goes beyond correctness checks to provide validation of the models used during quality assurance. In particular, we have described how monitoring of the "residue" of test coverage criteria could be used to validate the thoroughness of testing in the development environment.

A prototype system that implements residual test coverage monitoring has been presented. The system monitors a simple (but widely used) test coverage criterion, statement coverage. By selectively reinstrumenting a program under test to monitor only the coverage obligations that remain unmet, it can dramatically reduce the cost of continued monitoring of programs that have been through development test. Performance measurements made with this tool suggest that the performance impact of residual test coverage monitoring may be low enough to be acceptable in at least some kinds of actual use, such as the beta test phase. We view the simplicity of the approach as a particular virtue.

Only the performance aspect of residual test coverage monitoring has been investigated so far. We have partially side-stepped issues of privacy and security by considering monitoring in the beta test phase of software deployment, but more sophisticated approaches to these issues as well as the actual communication between users and developers deserve attention. Additionally, approaches to minimizing the performance impact of residual path-oriented coverage monitoring remain to be investigated; the prototype tool described here will be useful in gathering empirical data to evaluate possible approaches.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their useful comments.

This effort was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

REFERENCES

[1] Hiralal Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL 94)*, pages 25–34, Portland, Oregon, January 1994.

[2] Thomas Ball and James Larus. Efficient path profiling. In *Proceedings MICRO-29*. IEEE Press, 1996.

[3] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets*. The Free Press, 1995.

[4] Prem Devanbu and Stuart G. Stubblebine. Cryptographic verification of test coverage claims. In *Proceedings of the Fifth ACM/SIGSOFT Conference on Foundations of Software Engineering (FSE)*, Zurich, Switzerland, 1997.

[5] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[6] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J.R. Gongalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, San Francisco, California, November 1997.

[7] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the Fifth ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)*, June 1995.

[8] H. Lee. Bit: A tool for instrumenting java bytecodes. In *Proc. USITS*, 1997.

[9] H. Lee. Bit: Bytecode instrumenting tool. Master's thesis, University of Colorado, 1997.

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

[11] D. Luckham and F. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, 1985.

[12] P. J. Maker. Gnu Nana: improving support for assertions and logging in C and C++ (web page). http://www.cs.ntu.edu.au/homepages/pjm/nana-home/.

[13] Christina Pavlopoulou. Residual coverage monitoring of java programs. Master's thesis, Purdue University, 1998.

[14] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[15] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, August 1997.

[16] D. Rosenblum. Specifying concurrent systems with TSL. *IEEE Software*, 8(3):52–61, 1991.

[17] D. Rosenblum. Towards a method of programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[18] D. Rosenblum S. Sankar and D. Luckham. Concurrent runtime checking of annotated Ada programs. In *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 10–35. Springer-Verlag-Lecture Notes in Computer Science, 241, 1986.

[19] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., April 1994.

[20] A. Aho R. Sethi and J. Ullman. *Compilers: Principles Techniques and Tools*. Addison-Wesley, 1986.

[21] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

Compiler and Tool Support for Debugging Object Protocols

Sergey Butkevich* Marco Renedo* Gerald Baumgartner* Michal Young**

* Dept. of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210–1277
{butkevic,rened,gb}@cis.ohio-state.edu

** Dept. of Computer and Information Science
University of Oregon
120 Deschutes Hall
Eugene, OR 97403–1202
michal@cs.uoregon.edu

ABSTRACT

We describe an extension to the Java programming language that supports static conformance checking and dynamic debugging of object "protocols," i.e., sequencing constraints on the order in which methods may be called. Our Java protocols have a statically checkable subset embedded in richer descriptions that can be checked at run time. The statically checkable subtype conformance relation is based on Nierstrasz' proposal for regular (finite-state) process types, and is also very close to the conformance relation for architectural connectors in the Wright architectural description language by Allen and Garlan. Richer sequencing properties, which cannot be expressed by regular types alone, can be specified and checked at run time by associating predicates with object states. We describe the language extensions and their rationale, and the design of tool support for static and dynamic checking and debugging.

Categories and Subject Descriptors

D.2.5 **[Software Engineering]**: Testing and Debugging—*debugging aids, tracing*; D.2.4 **[Software Engineering]**: Software/Program Verification—*assertion checkers*; D.3.3 **[Programming Languages]**: Language Constructs and Features—*classes and objects*

General Terms

Debugging, protocols, sequencing constraints

1. INTRODUCTION

A repeated pattern in the history of software engineering research is development of underlying principles for specifying certain properties, then development of specification formalisms and automated checks for some part of those properties, and then migration of some efficiently checkable part of those specifications to programming languages. This pattern can be seen in abstract data types, eventually (but only partially) realized in module and class constructs of modern languages, and in module interconnection specifications which likewise were developed first as extrinsic specifications but are now at least partly internalized in the "package" constructs of Java and Ada. Since there is a long thread of research in specifying the sequences of operations accepted at module interfaces [9], and more recently development of extrinsic specifications of operation sequence protocols in architecture description languages [2, 3, 12] as well as the StateChart part of UML [4, 18], it is natural to consider whether and to what extent such protocols can be incorporated directly into programming languages and checked routinely as a part of normal compilation. Recent research in programming language design and semantics has greatly widened the class of interface properties that can be captured as part of type compatibility, and Nierstrasz has shown in principle how operation sequencing can be treated in a type system [13], but to date investigations of protocols as object types have been limited to pencil-and-paper exercises. In this paper, we describe an extension to the Java programming language which supports static protocol conformance checking and dynamic checking of compatibility between actual and declared behavior. The main innovation of the current work is in the way the statically-checkable conformance relation is embedded in a richer formalism for describing sequencing constraints and combined with dynamic checking of behavior. We have implemented the static checking as an extension to the compiler of Sun Microsystem's Java Development Kit, Release 1.1.7, and are close to completion of the implementation of the support for dynamic checking.

SIGSOFT 2000 (FSE-8) 11/00 San Diego, CA, USA
 ISBN 1-58113-205-0/00/0011...$5.00

1.1 Protocols as Part of Types

The interface specifications described here combine concepts of access-right expressions, originally described by Kieburtz and Silberschatz [9] with the regular object types of Nierstrasz [13]. They are interface specifications, distinct and independent from mechanisms used to implement the synchronization for enforcing a particular pattern of operations, such as path expressions [5]. Similar to Liskov and Wing's notion of behavioral subtyping [11], we extend the subtype relationship with behavioral information. Interface specifications are related to architectural description languages (ADLs) such as Wright [2, 3] and Darwin [12]. But while ADLs are language independent and capture higher-level architectural structures, our interface specifications are language specific, which allows some static checking and enables the compiler to generate code for dynamic monitoring. Our approach is partly based on Nierstrasz' regular types for active objects [13]. Similar formal models have been developed for concurrent objects with asynchronous message passing [16]. We adapted Nierstrasz' work to specifying and type-checking object protocols in Java. Furthermore, we extended the specification of protocols to allow dynamic checks of the actual behavior.

The type or interface of a class specifies a set of operations or meth-

ods provided by a class. Often these methods can be called only in a particular order, but the order is not part of the interface and cannot be checked by a Java compiler. (The situation is similar for other strongly-typed, object-oriented languages.) The well-known benefits of static type checking are thus available for such properties as the number and order of arguments to each method, but not for the sequencing of method calls. Protocols add this sequencing information to class and interface declarations and allow a compiler to check whether the declared intent of an object making method calls is compatible with the sequences supported by the object being called.

An extension of Ada that employs behavioral subtyping and is similar in some ways to our approach has been proposed by Puntigam [17]. Our approach differs in two fundamental ways: First, we treat a protocol as a contract between individual objects, whereas Puntigam's behavioral types specify what a set of objects may do collectively. Second, Puntigam's proposal is for static verification of actual behavior through program analysis; our more modest and, we think, more practical approach combines static verification of declarations with dynamic checking of actual behavior. In addition, our protocol specifications are somewhat more expressive, supporting non-determinism that cannot be expressed in Puntigam's behavior specifications.

1.2 Debugging Support for Protocols

Since compile-time checks are limited to checking regular (finite-state) specifications and are not, in general, capable of determining whether actual run-time behavior is consistent with these declarations of intent, additional checking is necessary at run time. To allow this run-time checking, the compiler instruments the generated code such that the run-time behavior is communicated to a debugging tool that compares the dynamic behavior with the declared behavior and either logs protocol violations or generates run-time errors. For checking the method call sequence, the debugging tool employs a labeled transition system, in which each method call triggers a state transition. After each state transition predicates can be evaluated to check the consistency between actual behavior and declared intent. Also, the debugging tool provides support for checking whether a labeled transition system is in a final state.

2. LANGUAGE DESIGN

2.1 Formulation of the Problem

Assume we are given a class `RandomAccess` implementing some interfaces `DataOutput` and `DataInput`.

```
class RandomAccess
      implements DataOutput, DataInput {
  // ...
}
```

Now assume that a client of class `RandomAccess` contains the following piece of code.

```
// ...
DataInput file = new RandomAccess();
file.open();
x = file.read();
file.close();
y = file.read();
// ...
```

This code will compile without errors or warnings. However, it is clearly not what was meant by the author of class `RandomAccess` A client should not read from a file after it has been closed. What is missing in the source code is a description of the order in which the methods of a class or an interface must be called.

2.2 Protocol Declarations

We introduce a new language construct[1], a protocol declaration, or, briefly, a protocol. A protocol declaration can appear in an interface or in a class. Syntactically, a protocol is introduced by the keyword `"protocol"` and contains a block of protocol statements. (We are using double quotes to denote literals and symbols.) Unlike methods, classes, and interfaces, a protocol does not have a name but is associated with its enclosing class or interface.

In the simplest case, a protocol contains just a single regular expression over the alphabet of all public method names. For the interface `DataInput` the protocol might be:

```
interface DataInput {
  protocol { open, read*, close; }
  // ...
}
```

This means that an object of a class that implements this interface is allowed to call the method `open` once, then call the method `read` zero or more times, and then call the method `close` before being destroyed (garbage-collected).

A reasonable protocol for the class `RandomAccess` would be the following:

```
class RandomAccess
      implements DataOutput, DataInput  {
  protocol { open, (read|write)*, close; }
  // ...
}
```

The latter protocol allows more functionality than the former. We say that the protocol of class `RandomAccess` *conforms* to that of interface `DataInput`. An object X conforms to an object Y, if X is *request substitutable* for Y. I.e., if a client of Y expects Y to accept a sequence of requests s, and we substitute X for Y, then X will accept the same sequence s. (A more formal definition of the notion of conformance will be given later, when we describe more general types of protocols.)

The conformance relation is a partial ordering among types. It has to be consistent with the subtype relation, i.e., if a class or interface X is a subtype of another class or interface Y, then the protocol of X must conform to the protocol of Y. Otherwise, the compiler should generate an error. If an interface or class X has no protocol declaration, the default protocol is assumed — i.e., methods of such a class can be called in any order. Such a protocol represents a minimal element with respect to our conformance relation, i.e., it conforms to any other protocol. If we use the symbol $\prec$ to mean "conforms to," then we have:

$$\texttt{Default} \prec \texttt{RandomAccess} \prec \texttt{DataInput}$$

[1] In the initial version we chose to extend the syntax directly. A future version may encapsulate the construct in a formal JavaDoc comment, as done in iContract [10].

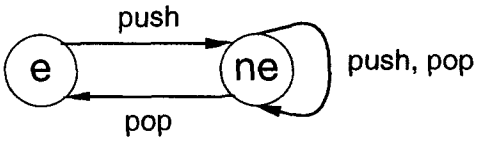

Figure 1: The LTS for interface `Stack`

where `Default` denotes a default protocol.

Can the allowed sequences of operations always be expressed as a single regular expression? The following example shows that, unfortunately, this is not possible. Consider a simple interface for a stack.

```
interface Stack {
  public void push(int i);
  public int pop();
}
```

We would like to write a protocol for this class that would allow sequences of requests such as `(push, pop)` or `(push, push, pop)`, but would disallow, for example, the sequences of requests `(pop)` or `(push, pop, pop)`. A regular expression or a deterministic finite automaton (DFA) cannot do that since it cannot keep track of the number of elements on the stack. Other finite-state specifications share the same fundamental limitation in expressiveness. Thus, we would need a richer language, such as a context-free grammar. The conformance check for context-free languages, however, is undecidable, which makes it unsuitable for use in the type system of a programming language. Following the idea introduced by Nierstrasz [13], we use a *labeled transition system* (LTS) over the alphabet of all public methods of a class or interface to describe the protocols, which, in general can be non-deterministic. This allows writing protocols that represent a reasonable approximation for possible object behaviors and, at the same time, are simple enough that the conformance check can be performed at compile time.

Using this approach, we can write the protocol for the interface `Stack` as follows:

```
protocol {
  start final state e;
  final state ne;
  <*> push <ne>;
  <ne> pop <*>;
}
```

Figure 1 shows the LTS defined by this protocol. Clearly, this protocol is only an approximation for the stack behavior. It disallows the sequence `(pop)` but still allows the sequence `(push, pop, pop)`. *Internal* non-determinism (as opposed to external non-determinism) is introduced as an artifact of modeling, i.e., deterministic choices of the service are modeled as arbitrary choices. It is for this reason that the protocol must be modeled as a labeled transition system (LTS) with failure semantics, and not as a language acceptor in which non-determinism can be removed by transformation to a deterministic finite-state acceptor using the subset construction (see Section 2.8). Because internal non-determinism is an artifact of abstraction in the finite-state model and not a feature of the actual system, these same internal choices are interpreted differently in run-time checks. Using these run-time checks, the sequence `(push, pop, pop)` can be disallowed as well (see Section 2.9).

A formal protocol syntax specification is given in Figure 2. Below, we outline its main features. Some details were intentionally left out for the sake of brevity.

A protocol declaration consists of a series of protocol statements. Each protocol statement is either a state declaration, a regular expression declaration, or a sequencing statement.

2.3 State Declarations

A state declaration declares one or more state identifiers that subsequently can be used in sequencing statements. Final states can be identified with the modifier `"final"`. The start state can be identified with the modifier `"start"`. Each state identifier is followed by an optional `"="` sign followed by a boolean expression, which represents a *state predicate*. Its meaning will be explained later. There are two implicitly defined states — the default start state and the default final states that are represented by empty state expressions on the left side and on the right side, respectively, of a sequencing statement.

In the `Stack` example above, we defined two states — `e` and `ne` (corresponding to empty and non-empty states of the stack). Both states are final, which means that an object implementing this interface is allowed to be destroyed at every state. In the example of `DataInput` there are no explicitly defined states.

2.4 Regular Expression Declarations

A regular expression declaration defines one or more names for regular expressions. This might be thought of as a macro definition and might be useful when writing complex protocols.

2.5 State Lists

A state list is either the literal `"*"` or a list of one or more identifiers separated by commas. The literal `"*"` is interpreted as the list of all explicitly declared state identifiers.

2.6 Sequencing Statements

In the simplest case, a sequencing statement is just a regular expression (as in the `DataInput` example).

More generally, a sequencing statement consists of an optional state list, followed by a regular expression over the alphabet of public method names, another optional state list, and a semicolon.

A sequencing statement defines state transition in the LTS defining the protocol. An individual regular expression describes a language of allowed sequences, and the appropriate semantics for this is language acceptance (also called trace semantics). There is no internal non-determinism, so we can use the standard subset construction [1, p. 117] to represent each individual regular sequencing statement as a deterministic acceptor, while still maintaining the failure semantics of the protocol LTS as a whole.

If the left-hand side (LHS) and right-hand side (RHS) state lists specify only one state each, the start state of the DFA is the LHS state, and the final state of the DFA is the RHS state. An empty LHS represents the default start state. An empty RHS represents the default final state. If there are multiple LHS states, multiple

```
⟨ProtocolDeclaration⟩  ::= "protocol" "{" {⟨ProtocolStatement⟩} "}"
⟨ProtocolStatement⟩    ::= ⟨StateDec⟩ | ⟨RegExpDec⟩ | ⟨SeqStatement⟩
⟨StateDec⟩             ::= ["start"]["final"]"state" ⟨JavaId⟩ ["=" ⟨JavaBoolExp⟩]
                           {"," ⟨JavaId⟩["=" ⟨JavaBoolExp⟩]} ";"
⟨RegExpDec⟩            ::= "regexp" ⟨JavaId⟩ "=" ⟨RegExp⟩ {"," ⟨JavaId⟩} "=" ⟨RegExp⟩ ";"
⟨SeqStatement⟩         ::= ["<" ⟨StateList⟩ ">"] ⟨RegExp⟩ ["<" ⟨StateList⟩ ">"] ";"
⟨StateList⟩            ::= "*" | ⟨JavaId⟩ {"," ⟨JavaId⟩}
⟨RegExp⟩               ::= ⟨MethodCallPattern⟩
                           | ["~"] "[" [⟨MethodCallPattern⟩ {"," ⟨MethodCallPattern⟩ } "]" ]
                           | ⟨RegExp⟩ "*" | ⟨RegExp⟩ "+" | ⟨RegExp⟩ "?"
                           | ⟨RegExp⟩ "|" ⟨RegExp⟩ | ⟨RegExp⟩ "," ⟨RegExp⟩ | "(" ⟨RegExp⟩ ")"
⟨MethodCallPattern⟩    ::= ⟨JavaId⟩ ["(" [⟨PatternArgumentList⟩] ")" ]
⟨PatternArgumentList⟩  ::= ⟨PatternArgument⟩ {"," ⟨PatternArgument⟩}
⟨PatternArgument⟩      ::= "*" | ⟨JavaType⟩
```

Figure 2: Protocol Grammar Definitions

RHS states, or both, the sequencing statement is equivalent to a series of sequencing statements with the same regular expression and all possible LHS-RHS state pairs. The protocol LTS is constructed by connecting the DFAs resulting from individual sequencing statements.

If a public method of a class or interface is not mentioned in any sequencing statement or regular expression definition, it is assumed that no restrictions are imposed on its use. In other words, not mentioning a public method `foo` in the protocol declaration is equivalent to every state in the LTS having a transition on `foo` onto itself.

2.7 Regular Expressions

We use a conventional syntax for regular expressions except that the comma operator corresponds to concatenation in `lex`-style regular expressions. A vertical bar represents a choice between two subprotocols. The operators `"*"`, `"+"`, and `"?"` denote zero or more, one or more, and zero or one occurrences of the regular factor, respectively. A list of method call patterns between brackets is equivalent to the same list of patterns separated with vertical bars. A bracketed list of patterns with the literal `"~"` in front denotes the list of all possible method call patterns of any public method of the class or interface *except* the ones listed. Parentheses are used to group terms together.

The simplest method call pattern, an identifier `foo`, indicates that any public method `foo` can be called by a client. By providing types as arguments, a smaller set of methods out of the set of all overloaded methods can be selected. The literal `"*"` inside a method call pattern acts as a wild card.

2.8 Conformance

How should a conformance relation between two protocols be formally defined? We employ the principle of request substitutability introduced in [13]. Protocol Y conforms to protocol X if all sequences of requests supported by X will be also supported by Y and, moreover, that any request refused by Y after accepting one of those sequences *might* also have been refused by X. More formally, $Y \prec X$ if

$$\text{traces}(X) \subseteq \text{traces}(Y) \quad (1)$$

$$\text{failures}_X(Y) \subseteq \text{failures}(X) \quad (2)$$

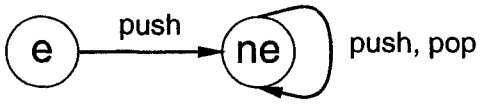

Figure 3: The LTS for interface `Var`

E.g., suppose a client makes method calls according to interface protocol X on an object implementing the class protocol Y. Condition (1) specifies that any sequence of method calls the client might make is understood by the object. Condition (2) specifies that if after accepting a sequence of method calls, the object fails to accept the next method call, then this failure is also possible according to the interface protocol.

Without non-determinism, condition (2) is redundant, but if internal non-determinism is present, as in the case of our `Stack` example, it is necessary to check both conditions.

As an example, assume we have an interface for an uninitialized variable that has two public methods, which we also call `push` and `pop`.

```
interface Var {
  protocol {
    final start state e;
    final state ne;
    <e> push <ne>;
    <ne> pop | push <ne>;
  }
  public void push(int i);
  public int pop();
}
```

Figure 3 shows the LTS defined by this protocol. The protocol of interface `Var` allows more freedom than that of `Stack`, and we would expect that `Var` conforms to `Stack`, but not vice versa. Note though that traces(`Var`) = traces(`Stack`), so we cannot distinguish between the two protocols by their traces only. However, if we compare the failure sequences, the difference between the two protocols becomes clear. The protocol of `Var` will always accept the call sequence `push, pop, pop`, whereas the protocol of `Stack` *might* not. This means that the set of the relative

failures of Stack with respect to Var is not a subset of the failure set of Var. Hence, Var $\prec$ Stack, but Stack $\not\prec$ Var.

In [13], an algorithm for conformance checking between two LTSs was given. We use this algorithm as part of the type checking phase of the compiler. If a class/interface Y extends/implements class/interface X, then the protocol of Y must conform to that of X. Otherwise, a compilation error is reported. For example, if we declare interface Stack as extending interface Var with the protocols described above, we will receive a compilation error saying that the protocol of Stack does not conform to that of Var.

There is a serious deficiency in describing protocols with finite-state LTSs — they are only approximations of real protocols, as seen in the Stack protocol. This protocol does not rule out the sequence of calls push, pop, pop. It only tells that it *might* fail. As we noted earlier, if we tried to specify protocols more precisely, we would not be able to perform the conformance check during compile time and their language would become too complicated for them to be useful. However, we can do better at *run time* by attaching predicates to the states, choosing among branches of the LTS at run time.

Part or all of the internal non-determinism in protocol specifications, which plays a role in static conformance checking, is removed by evaluating the predicates at run time. The remaining non-determinism is interpreted as external choice. While we use failure semantics for static checks of the conformance relation between declared protocols, language (trace) acceptance is the appropriate semantics for run-time checks of the consistency between actual behavior and declared intent.

2.9 State Predicates

A state predicate is a Java boolean expression that is an optional part of a state declaration and is associated with a state. It is stored in the LTS and is evaluated at run time to choose between several non-deterministic transitions in the LTS. A state predicate has class scope (syntactically it is the same as an initializer of a class field).

As an example, we can add another method isEmpty() to our Stack interface and rewrite the interface as follows:

```
interface Stack {
  protocol {
    start final state e = isEmpty();
    final state ne = !isEmpty();
    <ne> pop <*>;
    <*> push <ne>;
  }
  public void push(int i);
  public int pop();
  public boolean isEmpty();
}
```

The state predicates are only used at run time. They have no effect on the compile-time conformance check. Moreover, it is not possible for the compiler to check whether they are reasonably implemented and not self-contradictory. However, they provide essential information for debugging.

2.10 Debugging

For demonstrating the use of debugging, consider the following example (from some file foo.java) in which the protocol of interface Stack is violated by calling the method pop on an empty Stack:

```
Stack a = new StackImplementation();
a.push(3);
int x = a.pop();
int y = a.pop();
```

Since, in general, the compiler cannot detect protocol violations as in the second call of pop(), we provide run-time debugging support to detect such protocol violations.

There are two main design issues involving debugging. First there is the problem of how to implement the LTS tracing so that it can be used in already existing code, and second what action should be performed when a protocol violation is detected.

With respect to the first problem, one alternative would be modifying the Java Virtual Machine so that it traces the LTS. A second alternative would be modifying the compiler so that it inlines additional functionality in the client code. The approach we adopted is to introduce a Wrapper (as in the Decorator design pattern [7]) in the first line between reference a and the StackImplementation object. In this way, any time there is a method call to object a, the Wrapper object can trace the protocol (perform an LTS transition) as a side effect, while calling the same method on the object of class StackImplementation. We can automate this modification and make it invisible to the user by modifying the compiler. For every assignment in which the left-hand side type is an interface type with a protocol, the compiler inserts a Wrapper constructor call on the right hand side. This approach has significant run-time overhead but the user does not need a special Java Virtual Machine to take advantage of this functionality. In any case, this aspect of the implementation strategy is independent of the overall approach to specifying and checking object protocols.

With respect to the second problem, only the user knows exactly what to do in case of a protocol error. For maximum flexibility, we provide a mechanism for selecting the error handling behavior. Following the Strategy design pattern [7], we provide an interface ErrorHandler, in which each method corresponds to a possible type of protocol violation, and allow the user to select an appropriate implementation of this interface. We provide standard error handler implementations for logging protocol violations and for raising run-time exceptions. Using a simple API users can write custom error handlers.

Run-time tracing of the protocol can serve at least four purposes. It is up to the user to decide what role protocols should play in the debugging process:

- **Finding errors in the client's implementation.** This is potentially the most useful application of the run-time checking of protocols. If there is a misuse of a class or interface, it will be reflected in a violation of the protocol of that class or interface and the user will be able to detect this violation by tracing its protocol.
- **Debugging the protocol.** The user can write test harnesses and check if the LTS generated from the protocol behaves as expected at run time.
- **Using exceptions to control the application.** By using an error handler that throws exceptions in case of a protocol vio-

lation and by catching these exceptions in the client, the LTS can be (mis)used as part of the control of the application.

- **Finding errors in the server's implementation.** Using an interface with the same protocol as the class and with appropriate state predicates, it is possible to build a test harness for the class such that protocol violations indicate errors in the class.

3. EXAMPLE

To demonstrate the usefulness of protocols in practice, consider the class `java.util.zip.ZipOutputStream`:

```
public class ZipOutputStream
       extends DeflaterOutputStream {
  public ZipOutputStream(OutputStream out);
  public static final int DEFLATED;
  public static final int STORED;
  public void close() throw IOException;
  public void closeEntry() throw IOException;
  public void finish () throws IOException;
  public void putNextEntry(ZipEntry e)
         throws IOException;
  public void setComment(String comment);
  public void setLevel(int level);
  public void setMethod(int method);
  public synchronized void
         write(byte[] b, int off, int len)
         throws IOException;
}
```

For using an object of this class properly, the user must invoke its methods in a particular order, as described, for example, in the book *Java in a Nutshell* [6]:

> This class is a subclass of `DeflaterOutputStream` that writes data in API file format to an output stream. Before writing any data to the `ZipOutputStream`, you must begin an entry within the ZIP file with `putNextEntry()`. The `ZipEntry` object passed to this method should specify at least a name for the entry. Once you have begun an entry with `putNextEntry()`, you can write the contents of that entry with the `write()` methods. When you reach the end of an entry, you can begin a new one by calling `putNextEntry()` again, or you can close the current entry with `closeEntry()`, or you can close the stream itself with `close()`.
>
> Before beginning an entry with `putNextEntry()`, you can set the compression method and level with `setMethod()` and `setLevel()`. The constants `DEFLATED` and `STORED` are the two legal values for `setMethod()`. If you use `STORED`, the entry is stored in the ZIP file without any compression. If you use `DEFLATED`, you can also specify the compression speed/strength tradeoff bypassing a number from 1 to 9 to `setLevel()`, where 9 gives the strongest and slowest level of compression. You can also use the constants `Deflater.BEST_SPEED`, `Deflater.BEST_COMPRESSION`, and `Deflater.DEFAULT_COMPRESSION` with the `setLevel()` method.

Not only is this text hard to read, it is also of little use in debugging a client of class `ZipOutputStream`. Given the simple protocol declaration

```
protocol {
  ((setMethod | setLevel)*,
   putNextEntry,
   write*,
   closeEntry?
  )*, close;
}
```

the appropriate use of the class becomes much more understandable. Furthermore, by adding an interface and an adapter class with this protocol, we enable our debugging tool to detect protocol violations, such as a call to `write()` that immediately follows a call to `setMethod()`.

The protocol can be made more precise by using the class states `DEFLATED` and `STORED` as protocol states:

```
protocol {
  start state DEFLATED;
  state STORED;
  final state DONE;

  <DEFLATED>
    putNextEntry, write*, closeEntry?
    <DEFLATED>;
  <STORED>
    putNextEntry, write*, closeEntry?
    <STORED>;
  <DEFLATED,STORED> setMethod
    <DEFLATED,STORED>;
  <STORED> setLevel <STORED>;
  <DEFLATED,STORED> close <DONE>;
}
```

Note that the original verbal description of the protocol failed to mention in which state (`DEFLATED` or `STORED`) an object of type `ZipOutputStream` is originally constructed. A quick glance at the source code shows that the start state should be `DEFLATED`. Also, the above protocol should be extended to mention the methods `setComment()` and `finish()`, whose descriptions have been left out of the book.

4. IMPLEMENTATION

This section describes the changes we made or plan to make to the Java Development Kit, Release 1.1.7 [19] for implementing compile-time and run-time support for protocols. The compile-time conformance check is implemented and fully functional, also in the case of separate compilation. In particular, all the examples that we consider in the text would compile with our modified Java compiler. However, we did not yet implement support for method signatures in method call patterns, negation in regular expressions, and regular expression (macro) definitions. The run-time debugging support is also fully functional. We are currently in the process of finishing the implementation in the compiler of generating and instantiating the wrapper classes.

4.1 Compile-Time Implementation

The compile-time implementation consists of four main parts

1. **Parsing protocols**. We modified the `javac` compiler so that it recognizes the new keyword `protocol`, parses protocols,

creates a parse tree for each protocol, and reports syntax errors.

2. **Semantic analysis**. In this stage we generate an LTS from the parse tree. Each regular expression is first translated into an NFA using Thompson's construction and then this NFA is translated into a DFA using the subset construction [1, pp. 122, 117].[2] Then a protocol LTS is built by connecting these individual DFAs. This automaton is non-deterministic, in general. We do not convert it to a DFA, since the conformance relation that we use is not preserved under such conversion. The protocol LTS has two types of states — the states that were generated as a result of conversion of regular expressions to DFAs and the states that were explicitly defined by the programmer in the protocol. The data structure representing the latter type of states contains extra information, such as state predicates and source file line and position number.

3. **Conformance checking**. If a class or interface extends or inherits another class or interface and both of them have protocols, the compiler performs the conformance check between the protocols. For this conformance check, we use the algorithm proposed by Nierstrasz [13]. Since that algorithm assumes that all states of the LTS are final, we made a simple modification to the algorithm so that it can compare two LTSs that possibly have non-final states. In the worst case, the running time of this algorithm is exponential in the number of states. However, in a typical case, it is much faster. If the LTSs are deterministic, the running time is quadratic [13]. Since we do not expect typical protocols to be overly non-deterministic, the exponential worst-case behavior should not be a problem. Note that we check the conformance of the protocol LTSs without taking state predicates into account (since the task of conformance checking would become undecidable otherwise). If the protocol of the subtype does not conform to the protocol of the supertype, a type error is reported.

4. **Storing protocols in binary code**. If the conformance check was successful, then a representation of the protocol LTS is stored in the class file as a user-defined class attribute. This allows performing the conformance check between protocols from different source files, which is necessary for separate compilation. The class file so created is readable by a standard Java compiler and by a standard Java virtual machine. However, only a modified compiler is able to read the protocol LTS back from the class file.

4.2 Run-Time Implementation

When a piece of code (the *client code*) assigns an object (the *server*) to a reference, and this reference has as type an interface with a declared protocol, the tool initializes an LTS that corresponds to that protocol.

For example, if file `foo.java` contains the line:

```
I a = new C();
```

then an LTS is initialized when the client code `foo.java` assigns the server `new C()` to the reference `a`. The initialized LTS corresponds to the protocol of interface `I`. The simulation of the LTS is stopped when the reference is garbage collected.

The main purpose of the run-time tool is to detect violations of the protocol specified in the interface when calling a method on a variable of the interface type.

Every time the client code calls a server method through the reference, the tool checks if any of the possible current states of the LTS allows that method call. If so, a state transition is performed in the LTS and the server method is executed. After the server returns from executing the method, the LTS states for which the corresponding state predicate is false are removed from the set of possible states.

Implementation Overview

A `Wrapper` object is inserted between the reference and the server in the statement where the assignment occurs. The client code of the previous example is modified as follows:

```
I a = new Wrapper(new C(),...);
```

In this way, any call to a method of the server through the reference first has to go through a method call on the `Wrapper` object. So far this insertion has to be done manually, but we are working on modifying the compiler so the insertion will be done automatically when the client code is compiled. The `Wrapper` class is tailored to the interface, that is, for each interface *I* there is an *I*`_Wrapper` class. We are working on modifying the compiler so the wrapper class will be created automatically when compiling the interface. The common code among the `Wrappers` is contained in the superclass `Tracer`. Additional data structures are the `ProtocolInformation` class, used to store the specification of an LTS, and the `TraceState` class, used to store the state of an LTS. Sometimes reporting is desirable after the reference has been garbage collected, so a list of `TraceStates` is kept separately for that purpose. See Figure 4 for a UML diagram of the class hierarchy.

Below we explain in more detail the components of the run-time implementation.

Data Structures

The data structures involved in the run-time implementation are:

- Several classes *I*`_Wrapper`, one for each interface *I* that contains a protocol declaration. The class *I*`_Wrapper` will be created by the compiler when compiling the interface *I*.
- Class `Tracer`: an instance of this class simulates an LTS. It is a superclass for the `Wrapper` classes.
- Interface `ErrorHandler`, which can be implemented by the user for controlling protocol error handling and reporting. Some standard error handler classes are provided, which can be extended by the user.
- Class `TraceState`: an instance of this class holds the current state and the history of an LTS.

[2] Recall that the regular expression parts of protocol specifications describe language acceptance, and involve no internal non-determinism; converting them (individually) to deterministic automata thus gives the correct semantics in the overall LTS representation.

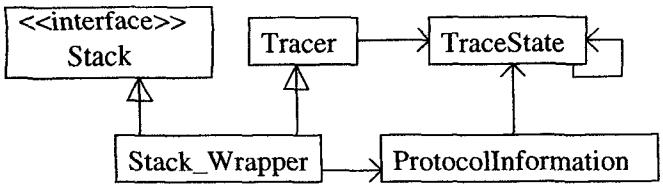

Figure 4: Class diagram for the run-time tool

- Interface `TraceFilter`, which can be implemented by the user to select `TraceStates` currently in memory that satisfy specified conditions.
- Class `ProtocolInformation`: an instance of this class holds the specification of an LTS at run time.

Class Wrapper

The main role of this class is to enrich any method call to the server object with operations to help trace the states of the LTS. A `Wrapper` contains code specific to a particular interface. Everything else is implemented in class `Tracer`, which is a superclass of the `Wrappers`.

The `Wrapper` also creates the run-time description of the protocol as a static object of type `ProtocolInformation`.

Assume that class `StackImpl` implements the interface `Stack` defined above, and that line 15 of file `foo.java` contains the following assignment:

```
Stack a = new StackImpl();
```

When compiling file `foo.java`, our compiler will (eventually) replace the assignment by

```
Stack a = new Stack_Wrapper(new StackImpl(),
  this, "foo.java: line 15.");
```

The class `Stack_Wrapper` would have been generated previously by the compiler when the interface `Stack` was compiled:

```
public class Stack_Wrapper extends Tracer {
  public Stack_Wrapper(Stack server,
           Object client,
           String lineAndFile) {...}
  public void push(int i) {...}
  public int pop() {...}
  public boolean isEmpty() {...}
  public boolean verify() {...}
}
```

The *I*_`Wrapper` class for an interface *I* implements every method of that interface as a sequence of three method calls:

1. A call to the method `announce()`, which is inherited from class `Tracer`, to check if the method is valid, i.e., if there are any edges with that method's name out of any of the current states of the LTS.
2. A call to the corresponding method of the server.
3. A call to the method `advance()`, which is inherited from class `Tracer`, to perform the transition in the LTS to the new states, and to check state predicates.

Method `push()` of class `Stack_Wrapper` is implemented as

```
public void push(int i) {
  announce(METHOD_PUSH);
  server.push(i);
  advance(METHOD_PUSH);
}
```

The method `verify()` removes all the states that do not satisfy their state predicates from the list of current states. This method is interface dependent because it evaluates the state predicates.

Class Tracer

Class `Tracer` is the simulator of the LTS. The algorithm used to simulate the LTS involves three phases:

- **Announce phase.** The `Wrapper` announces to the `Tracer` that a method is going to be called. The `Tracer` checks if any of the current states allows that method call. If there are none we say that *the method is invalid.*
- **Advance phase.** After the method has been called on the server, the `Tracer` computes the list of new current states by finding the states that we can jump to from the old states by calling that method. Then the `Tracer` takes off this list all the states that do not satisfy their state predicate. If the list is now empty we say that *the state is invalid.*
- **Finalize phase.** This phase occurs when the `Wrapper` is garbage-collected. One would like to detect if the protocol terminated in a final state or not. According to the Java Language Specification [8] any method `finalize()` implemented in a class is always called by the virtual machine when an object of this class is about to be garbage collected. We employ this feature and insert an appropriate algorithm in the method `finalize()` of the `Tracer` so that when the `Wrapper` is garbage-collected, it checks if any of the current states is a final state. If none of the current states is a final state we say that *the protocol is not in a final state.*

To allow users to query the state of a protocol at run time, class `Tracer` maintains a static list of protocol states, with an object of class `TraceState` per wrapper, and provides a general mechanism to collect and filter information from this list (see Figure 5).

Class TraceState

A `TraceState` object must contain the information necessary to provide a full report of the state of a protocol even after the `Wrapper` and its `Tracer` part have been garbage-collected. It should contain, for example, the last method calls performed, a reference to the protocol specification, error flags and the class names of the server and client, as well as the current states the LTS might be in.

The current states of the LTS are represented as an array of type `boolean`: if the `boolean` at index *i* has value `true` then the LTS might be in state number *i*. A `TraceState` object also contains

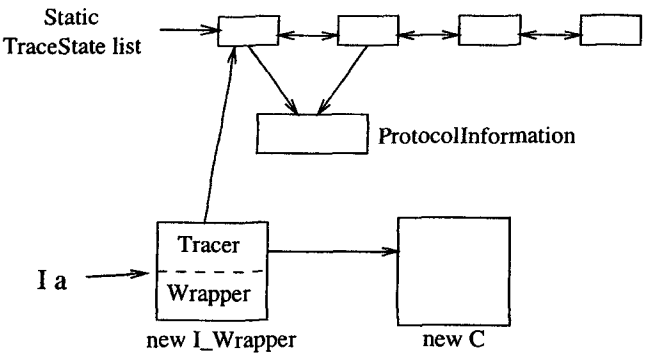

Figure 5: Objects at run time.

a reference to the `ErrorHandler` object that the `Tracer` uses any time it finds an error.

The information in a `TraceState` object is updated with each LTS transition. The information is queried by error handlers or if the users traverses the static list of trace states to dump the information.

Class ErrorHandler

The tool is designed to give maximum flexibility as far as error handling is concerned. We apply the **Strategy** design pattern [7] in this situation. When the `Tracer` encounters an error it defers the error to an object that implements the `ErrorHandler` interface. The user can create classes that implement the interface `ErrorHandler` and pass an error handler object to the `Tracer` through the static method `Tracer.SetDefaultOptions()`. All `Wrappers` that are created from then on will defer errors to that `ErrorHandler` implementation until the next call to that method.

There are three kinds of errors that can be found at run time by the `Tracer`:

- **Invalid method.** This error can happen during the announce phase. Typically it will by caused by an error in the interface protocol or in the client code.
- **Invalid state.** This error can happen during the advance phase. Typically it will be caused by an error in the server code.
- **Not in final state.** This error can happen in the finalize phase. There are no current states that are final states. Typically some method calls on the client code are missing to bring the protocol to a closure.

Class ProtocolInformation

A `ProtocolInformation` object stores the run-time description of a protocol. There is only one protocol information object per interface with a protocol. It is created statically by the corresponding `Wrapper`. This data structure also has to contain information about the interface and the protocol necessary for reporting errors: it contains the name of the interface, the names of the methods and the names of the states. For each state declared in the protocol, the declared name of that state is stored. States that were not declared are named with the line number and character position of the regular expression from which the state originates.

5. CONCLUSIONS

We have described an extension of Java with a `protocol` construct for specifying sequencing constraints on the order in which methods may be called. Protocols can be specified as part of a class definition or an interface declaration. We have extended the compiler of Sun Microsystem's Java Development Kit, Release 1.1.7, to check the conformance of a class protocol to an interface protocol as part of the interface conformance type check and to generate wrapper classes for the user code to interface with a debugging tool. An alternative implementation would have been to embed protocols in formal JavaDoc comments and to implement the conformance check and the wrapper generation in a preprocessor to the Java compiler. Using similar implementation strategies, protocols could be added to other object-oriented languages.

We have also described the design of a debugging tool for testing the conformance of a client's code to the protocol declared in an interface. The tool runs a labeled transition system (LTS) generated by the compiler from the protocol declaration. For every method call by the client, the LTS checks whether the method call is allowed according to the protocol. For specifying sequencing constraints that cannot be captured by an LTS, we allow associating predicates with states of the LTS. By testing these predicates at run time, object states can be mapped to LTS states.

We have illustrated the usefulness of protocols using as an example class `java.util.zip.ZipOutputStream`. We will experiment with protocols to determine whether finite-state specifications and state predicates are sufficiently expressive in practice. Possible extensions to the language would need to be designed such that there continues to be a finite-state specification as a subset that allows decidable type-checking. The dynamic checks could then be more precise. Another possible extension would be support for expressing two-way collaborations between objects or protocols involving more than two participating objects.

We will also experiment with the debugging tool to evaluate its practicality and benchmark the run-time overhead of the wrappers and of running the LTS.

The run-time debugging tool, as we have described it, only allows testing the conformance of the client's code to the protocol specified in the interface. The conformance of the class protocol to the interface protocol is checked at compile time. What is missing is testing the conformance of the class's code to the declared class protocol. In future research, we will explore the automatic generation of a test harness from the class protocol for testing this latter conformance.

Currently we combine static checking of protocol declarations with run-time checking of actual behaviors; there is no static analysis of code. While we believe that complete verification of protocol conformance through code analysis, as proposed by Puntigam [17], is likely to be too computationally expensive and too conservative to be useful, it is possible that program analysis may play a useful role in combination with dynamic checking. We will explore how data flow analyses, such as those described by Olender and Osterweil for checking sequencing constraints [14, 15], can be used to find some violations at compile time and reduce the amount of checking left for run time.

Acknowledgments

The effort of Michal Young was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

6. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1988.

[2] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 10th ICSE International Conference on Software Engineering*, pages 71–80. IEEE, 1994.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997. cf. errata in *TOSEM* 7(3), July 1998.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, 1998.

[5] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Proceedings of the International Symposium on Operating Systems*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102, Rocquencourt, France, 23–25 April 1974. Springer-Verlag, Berlin, New York.

[6] D. Flanagan. *Java in a Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, Sebastopol, California, 2nd edition, 1997.

[7] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[9] R. B. Kieburtz and A. Silberschatz. Access-right expressions. *ACM Transactions on Programming Languages and Systems*, 5(1):78–96, Jan. 1983.

[10] R. Kramer. iContract – the Java design by contract tool. In *Proceedings of the 1998 on Technology of Object-Oriented Languages and Systems (TOOLS '98)*, Santa Barbara, California, 3–7 August 1998.

[11] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[12] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Enginering Conference (ESEC 95)*, Sitges, Spain, Sept. 1995.

[13] O. Nierstrasz. Regular types for active objects. In *Proceedings of the OOPSLA '93 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15. Association for Computing Machinery, 1993. *ACM SIGPLAN Notices*, 28(10), October 1993.

[14] K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16:268–280, Mar. 1990.

[15] K. M. Olender and L. J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, Jan. 1992.

[16] C. Peter and F. Puntigam. A concurrent object calculus with types that express sequences. In *Proceedings of the ECOOP Workshop on Semantics of Objects as Processes (SOAP '99)*, Lisbon, Portugal, June 1999.

[17] F. Puntigam. Types that reflect changes of object usability. In S.Gjesing and K.Nygaard, editors, *Proceedings of the Joint Modular Languages Conference (JMLC'97)*, number 1204 in Lecture Notes in Computer Science, pages 55–77, Linz, Austria, Aug. 1994. Springer-Verlag.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1998.

[19] Sun Microsystems. Java Development Kit, Release 1.1.7. Available at $< http://java.sun.com/products/jdk/1.1/ >$.

APPENDIX C

Flow Equations as a Generic Programming Tool for Manipulation of Attributed Graphs

John Fiskio-Lasseter and Michal Young*
Department of Computer &
Information Science
University of Oregon
Eugene, OR 97403

{johnfl,michal}@cs.uoregon.edu

ABSTRACT

The past three decades have seen the creation of several tools that extract, visualize, and manipulate graph-structured representations of program information. To facilitate interconnection and exchange of information between these tools, and to support the prototyping and development of new tools, it is desirable to have some generic support for the specification of graph transformations and exchanges between them.

GENSET is a generic programmable tool for transformation of graph-structured data. The implementation of the GENSET system and the programming paradigm of its language are both based on the view of a directed graph as a binary relation. Rather than use traditional relational algebra to specify transformations, however, we opt instead for the more expressive class of flow equations. Flow equations—or, more generally, systems of simultaneous fixpoint equations—have seen fruitful applications in several areas, including data and control flow analysis, formal verification, and logic programming. In GENSET, they provide the fundamental construct for the programmer to use in defining new transformations.

Categories and Subject Descriptors

D.1.0 [**Programming Techniques**]: General; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages,Constraint and logic languages*

General Terms

Languages, Verification

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0620 and F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

PASTE'02, November 18–19, 2002, Charleston, SC, USA.

1. INTRODUCTION

Many problems of software analysis can be usefully modelled by viewing the structure of the problem data as a directed, attributed graph and defining analyses and transformations on this structure. Consequently, the past three decades have seen the creation of several tools that extract, visualize, and manipulate graph-structured representations of program and design information, for applications spanning a broad range of fields. This includes, on the one hand, specialized programs such as data/control-flow analyzers for optimizing compilers and model checkers. On the other hand, it includes a number of tools designed for more general tasks of program analysis, such as reverse engineering, program comprehension, design, and visualization.

In reverse engineering, for instance, *fact extractors* generate raw information about a software system (call graphs, directory structure, etc.). These are ultimately displayed by *visualization tools* such as AT&T's `dotty`, but only after considerable processing to elide detail, as well as transformation to the graph notation supported by the tool.

In a different, hypothetical domain (although the example is inspired by a real verification method of de Alfaro [5]), we might use a *web crawler* to traverse the pages of a web site, storing the results as a crawl-graph. If the crawler categorizes web pages according to, say, access control attributes, we can inspect the displayed crawl graph to look for security violations of private-data pages (in the form of paths from the home page to private pages that do not go through control pages). Ideally, we could use a tool that would transform the crawl graph so that insecure browsing paths were clearly displayed by a graph viewer.

Three themes run through these examples. First is the use of multiple graph-based tools in combination, for analyses ranging from established industrial practices to experimental techniques. With this comes the second theme: Each composition of tools is predicated on the ability of one tool's input format to be compatible with the other's output. The third theme is displayed in the web crawler example by the wish for an automatic display of security violations: the occasional need for rapid creation of a hitherto unconceived analysis. Taken together, these themes underscore the following: To support the prototyping and development of new tools, and facilitate the interconnection and exchange of information between existing tools, *it is desirable to have some generic support for the specification of graph transformations and exchanges between them.*

We believe that flow equations provide an attractive generic technique for programming such transformations. Viewing the typed

edges and attributes of a digraph as binary relations, we can define a system of simultaneous fixpoint equations whose solution is a set of new relations corresponding to the edges and attributes of the desired transformed graph. This approach gives a nice combination of declarative programming character, expressive power, sound theoretical foundations, and modest implementation cost.

To explore this approach, we have created GENSET, a generic programmable tool for the transformation and exchange of graph-structured data using flow analysis. At the heart of GENSET is an interpreter for a domain-specific language of flow equations in which the desired graph transformations can be programmed. Both the implementation of the GENSET system and the programming paradigm of the GENSET language are based on the view of a directed graph as a binary relation, a viewpoint that is extended to the attributes on the nodes of the graph.

The remainder of this paper is organized as follows. In the next section we give an overview of the GENSET language. This is followed in §3 by a discussion of some interesting aspects of our implementation. We follow this in §4 with a discussion of the possible applications for a tool such as GENSET. A demonstration of our approach on a real example is given in §5.Finally, we survey related work in §6. Possibilities for future work are offered in the conclusion of §7.

2. GENSET—AN OVERVIEW

GENSET is best thought of as a "little language": a compact, special-purpose, declarative language designed specifically as a utility for programming transformations of edge-typed, directed, attributed graphs. Such graphs are the primary data value in GENSET, and the only kind of value a GENSET script produces.

Graphs in GENSET are defined over a single, finite universe of untyped nodes, called *items*. Items are defined inductively to be either *atoms* (roughly as in Lisp) or *pairs* of items. Edges may be thought of as ordered pairs (src, tgt) of items, directed from src to tgt. Every edge in a graph has a *type* T, out of a finite set of edge types. Equivalently, we can view each edge type T as a graph with label T.

As with graph transformation approaches based on relational algebra [9, 11], the graphs input to and constructed by a GENSET program are represented internally as a collection of binary relations, one for each edge or node attribute type. The edges of a graph T are then represented as pairs belonging to relation T. A node may also have a finite number of *attributes*, and every attribute may itself be understood as a relation. For example, a node n with $Color$ attribute of red is represented by the edge (n, red) in the $Color$ relation.

The basic programming construct in GENSET is a block of *flow equations*: simultaneous equations over a first-order predicate calculus, extended with fixpoint operators. Although syntactically similar to iteration in an imperative language, there are important conceptual differences.

The general form can be illustrated by this example:

```
for x in IterExp_x, y in IterExp_y do
  R(x) := least E_R;
  S(y) := most E_S;
od;
```

Each "statement" can be understood as an equation defining a new relation: the expression on the right-hand side is used to compute, for each item a in the relation's domain, the set of items in the relation's image to which a maps. The domain from which a is drawn is the value of the corresponding $IterExp$ expression in the initialization of the block. In the above example, the equation `R(x) := least` E_R; defines the relation

$$R = \{\ (a, b) \mid a \in v_x, b \in v_R\ \}$$

where v_x and v_R are the values that result from evaluating the expressions $IterExp_x$ and E_R, respectively.

The scope of an "iterator variable" such as x is limited to the right-hand sides of the equations in which it appears on the LHS. For example, x is in scope in the expression E_R, while y is not. On the other hand, the scope of identifiers denoting relations is the entire GENSET script.

Note that `R(`x`)` can occur recursively in the equation E_R that defines it, and that R and S can be mutually recursive. In contrast with ordinary iteration, this means that each "statement" in a for-block is evaluated *at least* once for each element in the value of its controlling iterator expression, but it may be re-evaluated as many times as necessary to reach a *fixed point*. A finer point here is that `for`-blocks are themselves evaluated in source code order, and so forward references—references to relations that are defined by blocks following the current one—are disallowed. This avoids the problem of mutually recursive definitions inter-block, allowing each block of equations to be individually iterated to a solution.

Whether the least or greatest solution is calculated for an equation depends on which of the keywords, `least` or `most`, is given at the beginning of the RHS. All equations in GENSET are qualified by one of these two *fixpoint operators*, which extends over the entire RHS of an equation. If no operator is specified, the default choice is `least`. These keywords function like the μ and ν operators of the μ-calculus [15], but, for simplicity, they are limited to one use per equation: each one quantifies its whole RHS and they cannot be nested. For the least fixpoint of an equation, the value of the corresponding LHS is initialized to the empty set. For the greatest fixpoint, the LHS is initialized to the "universe," a value that must be implemented with some care. See §3.3 for a discussion of this issue and some of the restrictions that it carries.

Genset expressions evaluate to sets of items. The syntax is given by seven general classes in Figure 1.

Direct set construction is limited to the empty and singleton sets, as well as some limited uses of the identifiers denoting relations (as $filter$ or u_op operands). When the `single` keyword is applied to a variable x, the value is the singleton set consisting of the node to which x is currently bound. If applied to the constant `"x"`, the value is the set {x}. Node constants have global scope, and are created on-the-fly if necessary.

The *binary operations* are set union, intersection, difference, and cross-product, each with the expected meaning. *Unary operations* are defined only on an expression e whose value v_e is a set of pairs (*i.e.*, a relation); they facilitate the extraction of a relation's domain, image, or both. One can also select a subset of the pairs in a relation r, using a *filter* expression e on either the domain or image of r.

The two *appl* constructs are borrowed from relational algebra approaches (see [11], for example) for their utility: in the traversal of a graph r, we will often need to know the successors (resp. predecessors) of a node x. From the relational viewpoint, this corresponds to a projection of x through r (or vice-versa), an operation that we will term relation *application* (resp. *inverse application*). Relations can be applied either to variables or (by enclosing the identifier in `" "`) node constants.

The *combination expressions* are (very) loosely modeled on the syntax of the reduction operator `/` of APL, but are more similar in intent to the application in dataflow analysis of a "combination operator" for combining the flow information collected along dif-

e	:=	$direct \mid b\_op \mid u\_op \mid filter$ $\mid appl \mid combine \mid cond$	*(expressions)*
$direct$	:=	`null`	$\emptyset$
		`single("x")`	{x}, x a constant
		`single(`x`)`	$\{n_x\}$, where n_x is the current binding for x
		r	$\{(x, x') \in r\}$, r a relation
b_op	:=	e `union` e	*(set union)*
		e `intersect` e	*(set intersection)*
		e `-` e	*(set difference)*
		e `cross` e	*(cross product)*
u_op	:=	`dom` e	$\{ x \mid \exists x'.(x, x') \in v_e \}$
		`img` e	$\{ x' \mid \exists x.(x, x') \in v_e \}$
		`base` e	(`dom` e) $\cup$ (`img` e)
$filter$	:=	r `ofdomain` e	$\{ (x, x') \mid (x, x') \in r \wedge x \in v_e \}$
		r `ofimage` e	$\{ (x, x') \mid (x, x') \in r \wedge x' \in v_e \}$
$appl$	:=	$r(x) \mid r$`("x")`	$\{ x' \mid (x, x') \in r \}$
		$\_r(x) \mid \_r$`("x")`	$\{ x' \mid (x', x) \in r \}$
$combine$	:=	`/union` x `in` e_1 `:` e_2	$\bigcup_{x \in e_1} e_2$
		`/intersect` x `in` e_1 `:` e_2	$\bigcap_{x \in e_1} e_2$
$cond$	:=	`if` p `then` e_1 `else` e_2 `fi`	*(conditional evaluation)*

Figure 1: Summary of GENSET constructs

ferent paths. With either operator, the scope of variable x is the expression e_2.

Finally, GENSET includes a construct for *conditional evaluation* of an expression, based on the value of predicate p. Support is available for predicates that test emptiness of a set and set containment/comparison, and predicates may be combined using the standard propositional boolean connectives (`not`, `and`, `or`).

To illustrate, suppose we have extracted from a program the control flow graph `Flow` along with suitable `Kill` and `Gen` relations. We can use these to construct, for example, the classical *reaching definitions* analysis[2]:

```
for x in (base Flow) do
  RD(x) := /union w in _Flow(x):
              (RD(w) - Kill(w)) union Gen(w);
od;
```

which is notation in GENSET for the familiar textbook flow equation

$$RD(x) = \bigcup_{w \in Flow^{-1}(x)} (RD(w) \setminus Kill(w)) \cup Gen(w)$$

Another example, using the `most` operator, is the computation of dominators. For any node n in the $Flow$ graph, the *dominators of n* are those nodes which must be traversed on any path from the root of $Flow$ to n:

```
for x in (base Flow) do
  DomsOf(x) :=
   most (if (empty _Flow(x)))
          then single(x)
          else (single(x) union
            (/intersect y in _Flow(x): DomsOf(y)))
         fi);
od;
```

This example also demonstrates the use of an `if-then-else` expression. The standard dominators equation is defined by a pair of simultaneous equations, the choice of which depends on whether x is the root node of $Flow$.

3. IMPLEMENTATION

Implementation of an interpreter for the GENSET language involved several interesting challenges. We summarize the main features in this section.

3.1 Generic Worklist Algorithm

Interpretation of a GENSET program takes the form of an iterative dataflow analysis [13], using a worklist algorithm:

$while\ (Worklist\ not\ empty)$
  $(EQN, a) \leftarrow Worklist.extract()$ // `R(a) :=` e;
  $e \leftarrow EQN.getRHS()$
  $R \leftarrow EQN.getLHS()$
  $v \leftarrow eval(e, a)$
  $if\ (R(a) \neq v)\ then$
    $\forall\ dependent\ (equation, item)\ pairs\ (Q, b)$
      $Worklist.add(Q, b)$
    $R(a) \leftarrow v$

Note that we perform a *noninflationary* update, $R(a) \leftarrow v$ (instead of $R(a) \leftarrow v \bigsqcup R(a)$) [1]. Furthermore, this update is performed on *any* change to $R(a)$. As a consequence, the existence of a fixed point for any equation block, and hence termination of the worklist algorithm, can only be guaranteed by the equation itself.

The usual approach to ensuring termination is to guarantee that each equation is *monotonic*, from which the existence of a fixpoint follows. Although there are ways to give a static guarantee of monotonicity, or to enforce it at runtime, the current implementation of GENSET does neither. Instead we lay the responsibility for

termination of evaluation in the programmer's hands. The reason for this lies in the desire to provide as much flexibility to the programmer as possible. Divergence is not a desirable trait, of course, but a requirement of monotonicity placed on every equation would eliminate some analyses that nonetheless have fixpoint solutions. One example of this is the *partial dead code elimination* of Knoop *et al* [14]. Their algorithm involves an analysis based on functions that individually are not themselves monotonic, but when considered as a family enjoy weaker properties sufficient to guarantee the existence of a fixpoint solution [10].

3.1.1 Variable Dependencies

The other departure from the usual worklist algorithm lies in the determination of dependent $(equation, item)$ pairs to requeue—*i.e.*, those equations whose value might change because of the new value v. The intuition here is that when re-evaluation of an equation R results in a new value, the change will affect every equation Q whose right-hand side depends on R. More to the point, we do not need to re-evaluate anything else, and so can avoid a complete re-evaluation of the equation system. Although this optimization does not offer an improvement in worst-case complexity, in practice, it can produce significant performance increases.

When the set of equations is fixed syntactically (*e.g.*, the μ-calculus), dependence consists of one or more occurrences of R in the RHS of Q, and can be determined entirely from the source code. But in the case of data-flow equations (and also logic programming), we have the added challenge that the equations are parameterized over one or more variables, defining a *schema* of equations. For example,

`for` x `in` E_1 `do ... R(x) :=`E_2`; ... od;`

defines one R equation for each element of the value of E_1. Thus, when some $R(x)$ changes value, we need to know not just that some of the equations defined by $R(x)$ could be affected, but which ones. If this cannot be determined, we must adopt the conservative approach of re-evaluating R on *every* node in the value of E_1.

While this necessitates a dynamic component to dependency determination, in traditional bit-vector dataflow analyses, the *form* of the analysis is always the same. When, for some node a, the value of $R(a)$ changes, the affected equations are those corresponding to the adjacent nodes (either successors or predecessors) in the program's control flow graph, such as `Flow`, in the RD example above: $\{b \mid b \in \texttt{Flow}(a)\}$. In imperative programs, this graph is fixed before analysis and remains static throughout.

A GENSET script, however, represents an even more general case. While the language can be used to formulate bit-vector analyses over fixed graphs, this is not a necessary assumption. An equation block's iterator expression, for example, can be any legal GENSET expression, not just the nodes in the (pre-computed) `Flow` graph, and there is no requirement that the RHS of an equation combine information from successors or predecessors in `Flow` or any other single relation.

Our approach is based on a static analysis of the GENSET source code. During construction of the abstract syntax tree, we associate with each equation definition R, a list of $(equation, expression)$ pairs, (Q, d_R^Q), one for each equation definition Q in the same block as R. As mentioned in §2 above, each block is independently iterated to a fixed point, so no equations defined outside the block can be affected by a change to R. Moreover, in practice the number of equation definitions in a block is likely small, so this overhead, although quadratic in the number of equations defined in the block, should be manageable.

The expression d_R^Q (not to be confused with a GENSET expression) represents the computation necessary to determine the items on which Q will need re-evaluation, when the value of $R(a)$ changes for some item a: this set is given by $d_R^Q(a)$. Its construction is essentially a partial evaluation of the RHS of Q. For an equation definition $R(x) := E_R$, the dependency expression for a definition $Q(y) := E_Q$ (defined over iterator domain Y), is the function $d_R^Q = \lambda x.dep(R, E_Q, x)$. $dep(R, E_Q, x)$ is defined inductively on the structure of E_Q as in Fig. 2. Note that for the classical form

```
A(x) := /<op> w in _Flow(x):
              (A(w) - Kill(w)) union Gen(w)
```

we have

$$dep(A, \texttt{(A(w) - Kill(w)) union Gen(w)}, x) = \{x\}$$

and so the dependency expression is the special case

$$\begin{aligned}\lambda x.(Flow(x) \cup dep(A, \texttt{\_Flow(x)}, x)) &= \lambda x.(Flow(x) \cup \emptyset) \\ &= \lambda x.Flow(x)\end{aligned}$$

3.2 Set and Relation Data Structures

As discussed above, the edges of each graph (or equivalently, the members of each edge type in a graph) are represented in GenSet as elements of a binary relation. Since relation application and inverse application are perhaps the most commonly used expressions in a GENSET script, it is important that these operations be as fast as possible. Internally, relations consist of two hash tables, one for the forward and one for the inverse image. Each entry in the forward (*resp.* inverse) table corresponds to an item in the domain (*resp.* image), and it is stored in the table with a set of the items in the image (domain) to which it is related.

Ordinary sets are implemented using the Java `HashSet` class, which maximimizes the speed of insertion and retrieval of elements. One cost for this choice is that we are committed to the generic semantic view of nodes as being unordered. Except for equality, we cannot compare one node with another, nor can we easily choose any particular iteration order over elements (*e.g.* reverse postorder traversal).

3.3 "Infinite" Sets

The specification of a greatest fixpoint equation (using the `most` keyword) requires that we have some way to initialize the equation to the "universe" value and perform set operations (*e.g.*, set difference) on this value. Unfortunately, it will not do simply to use the set of nodes contained in input graphs, nor even adding to this the set of named constants in the source code. On a pragmatic level, even if this top value could be determined in advance, it may be very large and hence impractical to use directly, if we can avoid it.

A more significant problem is that in many cases, it is the wrong value to use. The problem arises from the cross-product operator, as in this (silly but illustrative) example:

```
for x in (single("a") union single("b")) do
  S(x) := most (S(x) intersect
                (single("c") cross single("d")));
od;
```

This should give the relation $\{(a, (c, d)), (b, (c, d))\}$, but if we iterate from the universe $\{a, b, c, d\}$, we will instead get $S = \emptyset$. The point here is that while the number of *atoms* is fixed by the graphs given as input, the number of *items* is not. As a consequence, the universe in general cannot be statically determined: it is finite but of indeterminate size.

$$
\begin{array}{lcl}
dep(R, \texttt{null}, x) & = & \emptyset \\
dep(R, \texttt{single(\_)}, x) & = & \emptyset \\
dep(R, Q', x) & = & \begin{cases} \mathrm{Y} & , \text{ if } Q = Q' \\ \emptyset & , \text{ if } Q \neq Q' \end{cases} \\
dep(R, u\_op\ e, x) & = & dep(R, e, x) \\
dep(R, e_1\ b\_op\ e_2, x) & = & dep(R, e_1, x) \cup dep(R, e_2, x) \\
dep(R, e_1\ filter\ e_2, x) & = & dep(R, e_1, x) \cup dep(R, e_2, x) \\
dep(R, r(y'), x) & = & \begin{cases} \{\mathrm{x}\} & , \text{ if } r = R \\ \emptyset & , \text{ otherwise} \end{cases} \\
dep(R, \_r(y'), x) & = & \begin{cases} R(x) \cup \langle\, old\ value\ of\ R(x) \,\rangle & , \text{ if } r = R \\ \emptyset & , \text{ otherwise} \end{cases} \\
dep(R, \texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi}, x) & = & dep(R, p, x) \cup dep(R, e_1, x) \cup dep(R, e_2, x) \\
dep(R, /op\ w\ \texttt{in}\ e_1 : e_2, x) & = & \begin{cases} dep(R, s(y), x) & , \text{if } e_1 \equiv s(y) \text{ and } dep(R, e_2, x) = \emptyset \\ dep(R, \_s(y), x) & , \text{if } e_1 \equiv \_s(y) \text{ and } dep(R, e_2, x) = \emptyset \\ \_s(x) \cup dep(R, s(y), x) & , \text{if } e_1 \equiv s(y) \text{ and } dep(R, e_2, x) \neq \emptyset \\ s(x) \cup dep(R, \_s(y), x) & , \text{if } e_1 \equiv \_s(y) \text{ and } dep(R, e_2, x) \neq \emptyset \\ \mathrm{Y} & , \text{otherwise} \end{cases}
\end{array}
$$

Figure 2: Static determination, for equation $Q(y) := E_Q$, of the nodes on which E_Q must be re-evaluated

Our approach is to treat the universe as if it were infinite. For such a value, we use a "symbolic infinity" by working instead with the *co-enumeration* (*i.e.* the complement) of an infinite set.

However, this itself raises another problem. Nontermination of expression evaluation is a separate consideration from the divergence that results when no solution exists for an equation. Unlike the absence of a fixed point, divergence of the expression evaluator cannot be considered an acceptable possiblity—it would be a bug in the interpreter itself, not the programmer's code. In the case of a cofinite set, this means that we must be careful about enumeration of any value. In particular, the termination of expression evaluation with co-enumerated infinite sets requires some restrictions on the possible forms of a relation: for every set, we must be able either to enumerate the set itself, or else its complement. Moreover, when evaluation of a `for`-block has been iterated to a fixpoint, each relation defined in the block must be finite. Further restrictions prevent the possiblity of enumerating an infinite set (the initializations of all iterators must evaluate to finite sets), or of taking the cross product with an infinite set operand. As with all other typing properties in GENSET, these restrictions are checked dynamically.

4. APPLICATIONS

Although GENSET has enough expressive power to write equations for dataflow analysis or CTL properties, it was designed for neither optimizing compilation nor model checking, and is not intended to compete with more specialized tools used in these areas (*e.g.*, SUIF [25]). Indeed, as it lacks the optimizations that render tractable the storage and rapid traversal of enormous control flow or state space graphs, it is likely unsuitable for either domain.

GENSET is intended rather to fill a niche similar to the role that Awk plays in the world of Unix text streams: The use of flow equations as a specification medium enables the programmer to draw from a class of graph manipulations which are too complex for simpler, less flexible tools, and with significantly less programming effort than would be required to write a special-purpose tool in, say, C or Java.

Hopefully, the near future will see the adoption of a standard graph exchange format for analysis and visualization tools; for example, the graph-based GXL format for XML [12]. In the presence of such a standard, we foresee a natural application for GENSET in the construction of "pipe-fitting" components that facilitate the composition of off-the-shelf tools. In support of this goal, we designed GENSET to be as flexible as possible with respect to support of new exchange formats. The current implementation includes support for RSF (Rigi Standard Format) and AT&T "dot" format, as well as the early untyped form of GXL described in [12]. We are currently working to extend support to include the recent GXL changes.

5. EXAMPLE

Flow equations have traditionally been associated with problem domains in compilers and logic programming. To illustrate their utility as a basis for graph transformation tasks relevant to the analysis of programs and software systems, we apply our approach to a transformation known in the reverse engineering community as *lifting*.

As a basis for our analysis, we chose the Linux kernel example from the PBS Guinea Pig repository [18]. From this, we chose as an input graph a selection of edge types from the raw example: 3 relations used in the calculation (`contain`, `funcdef`, and `sourcecall`), and one that did not play a part. GENSET does not yet have a parameterization mechanism for input arguments[1], so these are read into a global symbol table, along the lines of `extern` variables in C. Edges we actually used represented 8136 function definitions, 2068 source calls, and 1149 containment edges (directory structure). There were edges included in the file but not used in the transformation in the form of 6749 `include` edges.

A lifting transformation is used to produce a high-level view of a relation. This is done by lifting a relation between low-level entities (such as the original function to function call graph stored in the factbase by the `sourcecalls` relation) to a more abstract version of the relation, between higher-level entities (such as the directory to directory level). Those entities considered at the top level are represented as edges in the `toplevel` relation. The hierarchy is given here by the `contain` relation, which in the original factbase described containment of files in directories as well as directory nesting.

In the usual form of lifting, the `toplevel` relation would consist of exactly those entities that are at or above some level in the

[1]This is under development. See the comments in §7.

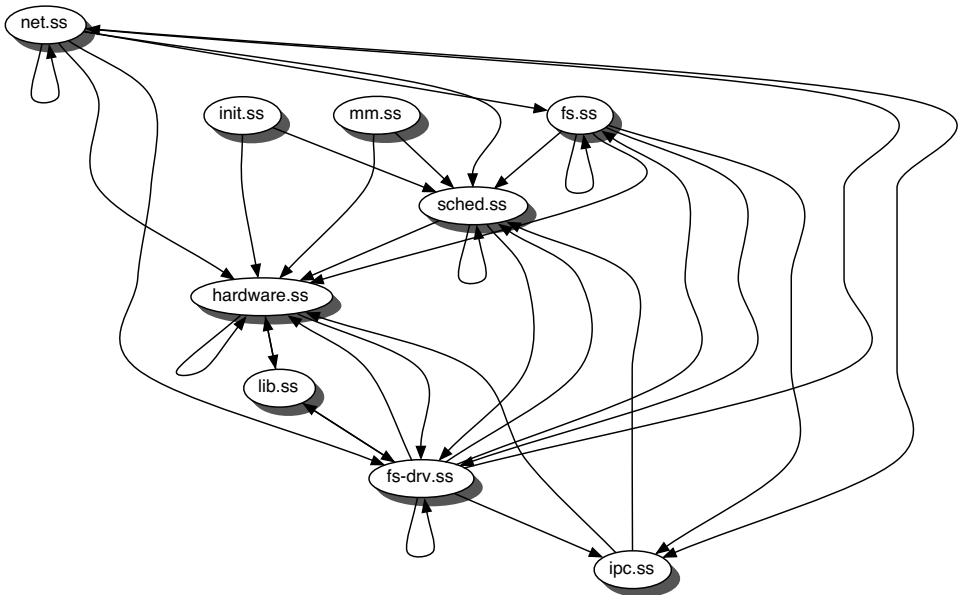

Figure 3: Lifted top-level calls—the `tlcall` graph

`contain` hierarchy. If we are interested only in the calls between a few different subsystems regardless of their position in the hierarchy, this approach will give either too much information, too little, or both. To illustrate the flexibility of our approach here, we took a slight departure from the usual transformation, and defined the `toplevel` relation manually to consist of 9 edges between subsystems chosen at varying levels in the directory hierarchy. The resulting graph analyzed consisted of 9,059 nodes and 17,832 edges.

The analysis is specified in 21 lines of GENSET code as a set of three equation blocks:

```
for ss in (base contain) do
  tlcontain(ss) := (/union child in contain(ss) :
            single(child) union inherit(child));
  unmarkedChildren(ss) :=
                    contain(ss) - dom toplevel;
  inherit(ss) := (/union unmarked in
                        unmarkedChildren(ss):
                          tlcontain(unmarked));
od;

for func in (base sourcecall),
    dir in (dom tlcontain) do
  inDir(func) :=
         /union file in _funcdef(func):
                                _tlcontain(file);
  dirCalls(dir) :=
     (/union file in tlcontain(dir):
        (/union f in funcdef(file) :
            (/union g in sourcecall(f): inDir(g))
        ));
od;

for tldir in (base toplevel) do
  tlcall(tldir) := dirCalls(tldir)
                        intersect (dom toplevel);
od;
```

In the first block, we collapse nodes corresponding to files into the nearest ancestor directory node which has been marked as (*i.e.* included in) `toplevel`: this is the `tlcontain` relation.

In the second block the `sourcecall` relation representing calls between functions is lifted to become the `dirCalls` relation, representing the presence of a call between two directories. This is computed for *all* directories, not just those marked as `toplevel`. The extraneous calls from non-`toplevel` directories are filtered out in the third block, leaving the `tlcall` relation from `toplevel` directory to `toplevel` directory.

Analysis was run on a Macintosh Quicksilver with 900 Mhz G4 and 1.2 GB of RAM. The resulting graph is given in Fig. 3. The slowest step was the 4 seconds required to read the input graph (in RSF format). After the graph was represented in memory, execution of the GENSET script completed in less than 2 seconds.

6. RELATED WORK

A variety of programmable graph transformation tools have been developed, with explicit application to problems of program and software system analysis.

The approach closest to our own is the specification of transformations with *relational algebra* (RA). Given the tight mathematical correspondence between binary relations and directed graphs [22], it is unsurprising that the use of RA has very natural applications in graph transformation; we have borrowed a few of the operators ourselves.

A number of programmable graph transformation systems based on extensions of this algebra have been presented in the literature [3, 4, 9, 11, 16, 17, 19]. One of the common extensions is to augment RA with a transitive closure operator (this is used in all of the works cited here). With this extension, several important graph transformations have been implemented, notably in the area of software architecture. Holt, for example, shows in [11] how to implement six important architectural transformations using his Grok system, including the lifting transformation we demonstrated in this paper.

We do not yet know how the performance of GENSET compares with that of the extended relational algebra systems across the whole range of transformations expressible in RA (with or without transitive closure). Thus far, however, our experimental results (such as the example presented in this paper) suggest that the flow equation approach embodied in GENSET will be competitive

with RA approaches. (The question of greater programming convenience is still a matter of debate.)

The primary difference between a flow equation and relational algebra approach lies in the expressive power of the underlying languages. RA by itself cannot express any kind of recursion (hence the need to add transitive closure as a magic operator). Even with transitive closure, the GENSET language is strictly more expressive than RA languages. On the other hand, we will suffer from a worse worst-case, since some queries expressible in GENSET will be of a higher complexity—indeed, since one can write divergent analyses in GENSET, our worst case is unbounded.

The other major approach to graph transformation comes from the research in *graph grammars*, particularly systems for graph-rewriting-based transformations. One of the best known of these is the PROGRES project [20, 23]. In this system, transformations are specified with graph-rewriting rules which are using to generate C code for a stand-alone prototype. This approach offers greater expressive power than flow equations (it is computationally complete). However, it appears to suffer from the general complexity of graph rewriting—for example, the necessity of using subgraph isomorphism detection to implement generalized pattern matching. Using PROGRES to specify several common architectural transformations, Fahmy and Holt [6] report that the system, while effective at small prototypes, is impractical for transforming the large graphs associated with real software systems.

Generic iterative equation solvers have been developed primarily within the logic programming field. Our approach to the problem of dynamic determination of dependencies has been influenced in part by work presented in [8]. Another generic dataflow analysis tool is the Fixpoint-Analysis Machine described by Steffen *et al* [24], which handles generic instances from several classes of flow analysis whose reductions to equivalent model checking problems are known.

Recently Rayside and Kontogiannis [21] have developed another generic worklist algorithm that, like ours, is designed for towards graph-based analyses. Their work is targeted specifically toward generic support for graph reachability problems, which leads to three significant differences from our version. First of all, their test for re-evaluation is specifically for a *monotonic* change, while we base re-evaluation on a test for *any* changes, allowing for the possibility of non-monotone functions to be evaluated. Further, the user of their algorithm must specify manually the lattice to be used, and in particular, must define the partial order relation between elements, which is necessary for detecting monotonic change. Even if we were to enforce monotonicity, we always use the same lattice—the power set of the set of possible nodes in the relation's image, ordered by subset/superset inclusion. Finally, because their algorithm is targeted towards reachability analyses on a fixed graph, the determination of dependent equations is always done in the traditional static fashion: by taking the successors in the graph of the current node. As discussed in §3.1 above, this approach to dependency determination does not work in our more general setting.

An interesting alternative solution to the problem of representing infinite sets is Alfaro's constructive μ-calculus [5] in which GFP equations are restricted by the requirement that the universe of discourse be both finite and explicitly stated. We are still investigating a comparison of this approach with our own.

7. CONCLUSION AND FUTURE WORK

The expressiveness and relative efficiency of flow analysis appears to be a useful point in the design space of graph transformation tools for program analysis. Meanwhile, there remain a number of opportunities for further development.

Two aspects of the existing implementation of GENSET need improvement. First, the requirement that equation blocks be evaluated in source code order avoids the mess of mutual recursion between blocks, but is a shortcoming in the declarative character of the language. The alternative is to add a "program-wide" iteration, along with the use of dependency graphs both within and between equation blocks to determine, if possible, a better evaluation order than that given by the source code. Second, the language lacks effective schemes for parameterization and library construction. Relations that are not explicitly defined by equations in a script are presumed to exist in a global symbol table before evaluation, with the symbol value assumed explicitly in the source code. We are currently developing a procedure-definition facility for the reuse of commonly used equations (*e.g.* transitive closure) and will remove assumptions about the global symbols with a top-level "main" construct.

From a philosophical point of view, the paradigm of the GENSET language is compatible with relational algebra and it may benefit from the inclusion of many of the ordinary RA operators. As it stands, the language is strictly more expressive than RA, and such additions would therefore be "syntactic sugar," but may offer more convenience for programmmers.

More challenging is the possibility of developing a static type system to guarantee finiteness and union-compatibility properties of relations, eliminating the need for many of the runtime checks that are performed in the present version. In addition to potential improvements in the runtime performance of interpretation, this would make the possibility of a compiler for the language more appealing.

Finally, it is important to understand the tradeoffs between expressiveness and efficiency among the variety of graph transformation approaches available for manipulating representations of programs and software systems. Fahmy *et al* [6, 7] have begun this task with a comparison of manipulation using relational algebra and graph rewriting. We expect that our flow analysis approach will fall somewhere between these two, but more benchmark analyses are needed with representative, practical examples. Widespread adoption of a single exchange format such as GXL may help; additionally, we are developing conversions among some of the more widely used graph representations to facilitate comparisons.

Acknowledgements
Craig Kaes, Sachiko Honda, and Lam Nguyen designed and implemented parts of GENSET. We also wish to thank Dr. Jan Hidders for a helpful discussion of relative expressiveness and Dr. Luciano Baresi for invaluable constructive feedback. Finally we thank the anonymous reviewers for their many helpful suggestions.

8. REFERENCES

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

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

[3] O. Ciupke. Analysis of object-oriented programs using graphs. In *ECOOP'97 Workshop Reader*, LNCS 1357, pages 270–271. Springer, 1998. Extended version available (as of August 2002) at `http://www.fzi.de/ecoop97ws8/`.

[4] O. Ciupke. Automatic detection of design problems in object-oriented engineering. In *Technology of Object-Oriented Languages and Systems (TOOLS 30)*, pages 18–32. IEEE Pr., 1999.

[5] L. de Alfaro. Model-checking the world wide web. In *Computer Aided Verification, 13th International Conference (CAV 2001)*, LNCS 2102. Springer, 2001.

[6] H. Fahmy and R. Holt. Using graph rewriting to specify software architectural transformations. In *15th Conference on Automated Aoftware Engineering (ASE 2000)*, pages 187–196. IEEE Pr., 2000.

[7] H. Fahmy, R. Holt, and J. Cordy. Wins and losses of algebraic transformations of software architectures. In *16th Conference on Automated Aoftware Engineering (ASE 2001)*, 2001.

[8] C. Fecht and H. Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2–3):137–161, 1999.

[9] L. Feijs, R. Krikhaar, and R. V. Ommering. A relational approach to support software architecture analysis. *Science of Computer Programming*, 28(4):371–400, 1998.

[10] A. Geser, J. Knoop, G. Lüttgen, O. Rüthing, and B. Steffen. Non-monotone fixpoint iterations to resolve second order effects. In *6th International Conference on Compiler Construction (CC '96)*, LNCS 1060, pages 106–120. Springer, 1996.

[11] R. C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *5th Working Conference on Reverse Engineering (WCRE'98)*, pages 210–219. IEEE Pr., 1998.

[12] R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *7th Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171. IEEE Pr., 2000.

[13] G. A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages (POPL 73)*, pages 194–206. ACM Pr., 1973.

[14] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *1994 Conf. on Programming Language Design and Implemention (PLDI '94)*, pages 147–158. ACM Pr., 1994.

[15] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[16] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A two-phase process for software architecture improvement. In *International Conference on Software Maintenance, Oxford UK (ICSM '99)*, pages 371–380. IEEE Pr., 1999.

[17] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. Technical Report 22/98, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1998. `http://www.uni-koblenz.de/fb4/`.

[18] Portable Bookshelf (PBS) Tools. Available (as of August 2002) at `http://swag.uwaterloo.ca/pbs/`.

[19] A. Postma and R. Krikhaar. Applying relation partition algebra for reverse architecting. In *1999 Workshop on Software Reengineering, Bad Honnef, Germany*. `http://www.uni-koblenz.de/~ist/RWS99/`.

[20] PROGRES. Available (as of August 2002) at `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/`.

[21] D. Rayside and K. Kontogiannis. A generic worklist algorithm for graph reachability problems in program analysis. In *Software Maintenance and Re-engineering, Proc. of the 6th International Conference (CSMR '02)*, pages 67–76. IEEE Pr., 2002.

[22] G. Schmidt and T. Ströhlein. *Relations and Graphs*. Springer-Verlag, 1993.

[23] A. Schürr, A. Winter, and A. Zündorf. PROGRES: Language and environment. In G. Rozenberg, editor, *Handbook on Graph Grammars: Applications, Vol. 2*, pages 487–550. World Scientific, 1999.

[24] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *Concurrency Theory, 6th International Conference (CONCUR '95)*, LNCS 962. Springer, 1995.

[25] SUIF. Available (as of August 2002) at `http://suif.stanford.edu/`.