Andy Zaidman, Abdelwahab Hamou-Lhadj, Orla Greevy *(editors)*

PCODA'05

1st International Workshop on

Program Comprehension through Dynamic Analysis

co-located with the 12th Working Conference on Reverse Engineering (WCRE'05)

Pittsburgh, Pennsylvania, USA
10 November 2005

Technical report 2005-12
Universiteit Antwerpen
Department of Mathematics & Computer Science
Middelheimlaan 1, 2020 Antwerpen, Belgium

UNIVERSITEIT ANTWERPEN

uOttawa
L'Université canadienne
Canada's university

UNIVERSITÄT BERN

Program Chairs

Orla Greevy
Software Composition Group
Institut fur Informatik und angewandte Mathematik
University of Bern
Switzerland
greevy@iam.unibe.ch

Abdelwahab Hamou-Lhadj
School of Information Technology and Engineering
University of Ottawa
Canada
ahamou@site.uottawa.ca

Andy Zaidman
Lab On Re Engineering
Department of Mathematics and Computer Science
University of Antwerp
Belgium
Andy.Zaidman@ua.ac.be

Program Committee

Contents

Using Build Process Intervention To Accommodate Dynamic Instrumentation of Complex Systems

Robert L. Akers

Semantic Designs, Inc.
12636 Research Blvd, C214
Austin, TX 78759

lakers@semanticdesigns.com

Abstract

Complex software is often constructed with the help of complex and sometimes generative build processes, automated with the help of various scripting and incremental development tools. Many dynamic analysis techniques require instrumentation of source code to extract runtime information. Automated transformation systems can be used to insert this instrumentation, but this is typically a separate process that is applied to source files prior to system builds. In cases where the build processes actually construct source files, or determine which source files out of a source base, or in cases where the instrumentation process is aided by configuration information that may not be known until mid-build, the question of what code should be instrumented and how may be difficult or impossible to resolve prior to the build. Moreover, build processes may be rather opaque to quality assurance responsible for instrumentation. Instrumenting source code in mid-build, for example immediately before module compilation, offers a way to deal with these complexities.

This paper discusses the use of automated transformation for dynamic instrumentation. Then it presents a simple technique for intervening in complex build processes that does not require knowledge of the build process itself, but is nevertheless sensitive to build dynamics, so that source code instrumentation can be performed at surgically precise places and times. The technique enables various methods of dynamic analysis to be implemented automatically on demand.

Keywords

Test coverage, instrumentation, dynamic analysis, transformation, scripting, build process

1 Instrumentation for Dynamic Analysis

Instrumentation of source code is a key approach to gathering dyamic analysis information that captures the behavior of a software system during its execution. From the ad hoc insertion of debugging print statements to the rigorous insertion of special-purpose data-gathering instrumentation throughout a software system, the addition of source code that gathers information incidental to the functionality of a software system is a useful technique for improving software quality. Instrumentation can provide a wide variety of information, including fine-grained performance and timing data, records of frequency and sequencing of execution of code segments, data traces, security property monitoing, error tracking, and runtime validation against software metrics and functional specifications..

For many areas of program comprehension, dynamic analysis complements static techniques in a lower bound/upper bound manner. For example, one pillar of information in legacy architecture extraction is the collection of calls/called-by relationships. Static analysis can help determine what functions could possibly call which other functions, establishing an upper bound. Dynamic analysis via instrumentation of call sites can show empirically what functions actually do call which others during a particular execution, establishing a proof-by-example lower bound for the calls/called-by relationship. Another example is control flow. Static analysis can help determine local control flow paths under constraints gleaned from the immediate environment, refining the range of possibilities suggested by the control mechanisms of the source language, for instance by taking into account the possible values of governing conditionals. Dynamic analysis, enabled by pervasive instrumentation of atomic execution path segments, can produce a linear trace of the control path followed by a given execution. In each case, static analysis defines a range of possible behaviors, while the dynamic analysis enabled by instrumentation illustrates cases within the range that actually occur.

2 Instrumentation Via Automatic Transformation

Automated source code transformation systems are particularly useful for performing many types of instrumentation, particularly where some regular but pervasive means of data-gathering is appropriate. Examples include:

- the addition of probes in every atomic control path within a program for the purpose of gathering data illustrating:
  - test coverage
  - frequency of execution
  - order of execution
- the insertion of probes at function or method call points to report caller/called events.[6]
- the automatic insertion of statements to print the values of data objects used in the immediate context, effectively providing a data trace of the program's execution.
- the automatical tagging of data objects with security levels, and use of rules based in control flow semantics to add instrumentation that propagates security level information through a system and traps unauthorized flows at output ports.

The widespread insertion of instrumentation like this in a large system would be both mind-numbing and prohibitively expensive if done manually. However, this kind of task is quite easy for an automated source code transformation system. An example of such a system is the Design Maintenance System (DMS[1]) [2]. DMS has at its heart a software analysis and transformation infrastructure that operates on abstract syntax tree (AST) representations of programs. It also has ming language with a context-free grammar and well-developed front ends for most common languages. Automatically generated parsers transform source to AST, and automatically generated pretty printers convert AST back to source. Sets of rewrite rules, stated in terms of the source languages being manipulated, can be applied by a DMS-based tool to perform the massive regular changes involved in instrumenting a system.

For example, a relatively small collection of rewrite rules can implement test coverage instrumentation. The rules are keyed by the syntax of branching construc-

```
bool fibcached[1000];
int  fibvalue[1000];

int fib(int i)
{ int t;
  switch (i)
  { case 0:
    case 1: return 1;
    default:
      if fibcached(i)
         return fibvalue(i);
      else { t=fib(i-1);
             return t+fib(i-2);
           };
   };
};
```

Original "C" program

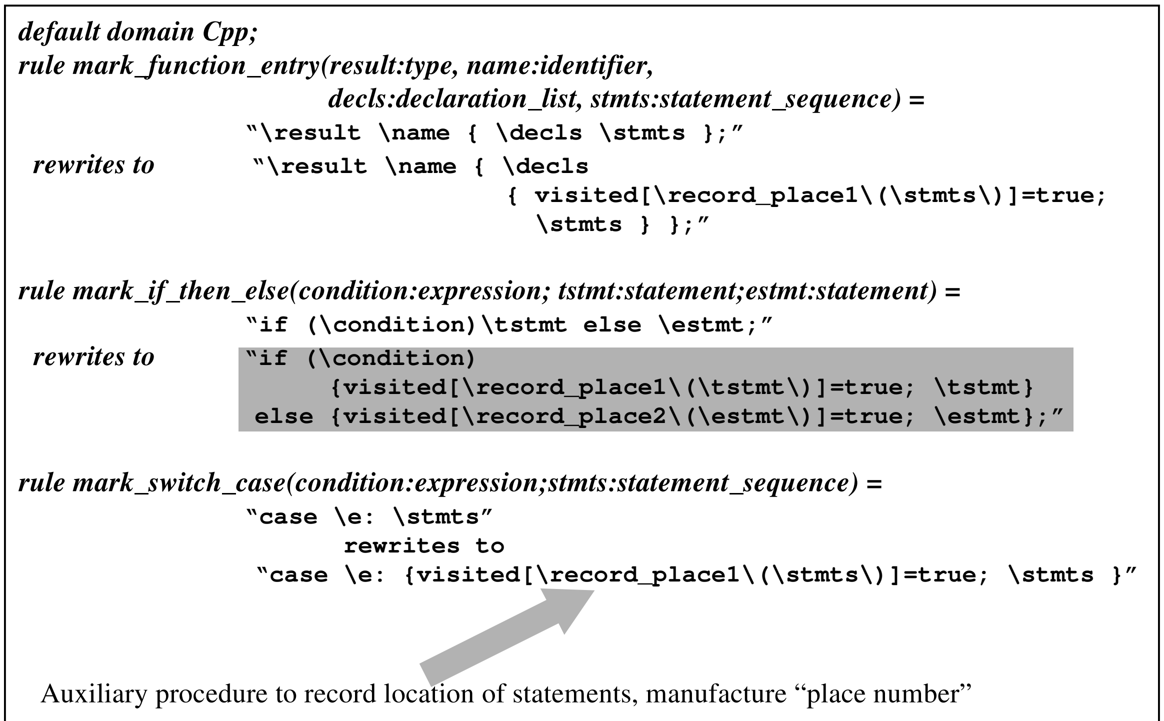
```
bool fibcached[1000];
int  fibvalue[1000];

int fib(int i)
{ int t;
  visited[1]=true;
  switch (i)
  { case 0: visited[2]=true;
    case 1: visited[3]=true;
            return 1;
    default:
      visited[4]=true;
      if fibcached(i)
         { visited[5]=true;
           return fibvalue(i);}
      else { visited[6]=true;
             t=fib(i-1);
             return t+fib(i-2);
           };
    };
};
```

Marked program

Figure 1 -- A simple routine before and after test coverage instrumentation

[1] [1] DMS is a registered trademark of Semantic Designs Inc.



```
default domain Cpp;
rule mark_function_entry(result:type, name:identifier,
                          decls:declaration_list, stmts:statement_sequence) =
                "\result \name { \decls \stmts };"
 rewrites to     "\result \name { \decls
                               { visited[\record_place1\(\stmts\)]=true;
                                 \stmts } };"

rule mark_if_then_else(condition:expression; tstmt:statement;estmt:statement) =
                "if (\condition)\tstmt else \estmt;"
 rewrites to    "if (\condition)
                      {visited[\record_place1\(\tstmt\)]=true; \tstmt}
                 else {visited[\record_place2\(\estmt\)]=true; \estmt};"

rule mark_switch_case(condition:expression;stmts:statement_sequence) =
                "case \e: \stmts"
                       rewrites to
                 "case \e: {visited[\record_place1\(\stmts\)]=true; \stmts }"
```

Auxiliary procedure to record location of statements, manufacture "place number"

Figure 2 -- DMS transformation rules for test coverage instrumentation

tions in the source code. E.g., in the statement sequence of both branches of an if statement, a rule inserts a statement setting a unique element of a test coverage vector to true. (See Figure 1.) Likewise for the beginning of the statement sequence of a function, the statement sequence within a looping construct, and so forth. Figure 2 illustrates some of the transformation rules that implement the instrumentation. Ultimately the size of the test coverage vector equals the number of non-branching blocks in the software system (or the portion of the system that was instrumented). The vector is initialized to false values. After running execution tests, any true value in the vector signifies that the associated code segment was executed. A display tool ties this information back to the source code itself, helping the engineer visualize what portions of his system were executed by his sequence of tests.

A slight variant on this scheme allows the counting of executions of each program segment for purposes of performance analysis, and yet another variant could record traces during execution, either by recording the value of a ticker in the vector or by sending the unique value associated with the instrumented location to a stream. In each case, application of a relatively small number of rewrite rules, which themselves are easily formulated, can result in the pervasive instrumentation of software systems of arbitrary size. The kind of analysis desired dictates what kinds of instrumentation rules are formulated.

Automation allows instrument insertion to be separated from the main line of software development, which has several advantages. Software design and construction is not cluttered with instrumentation concerns. Furthermore, since instrumentation can be done more or less instantaneously, maintaining synchronicity with ongoing system development while the instrumentation is being coded is no longer a problem. As the system changes, it can be re-instrumented as appropriate for testing cycles. Morever, since the instrumentation can be trivially re-generated, it need not be maintained or distributed with software releases, thus imposing no maintenance or end user performance overhead.

In some respects, automatic transformation in this style resembles aspect weaving. The syntactic form of the left hand sides of the rewrite rules and the semantic conditions that can be attached to the rules perform cut point identification appropriate for a given aspect. The right hand sides specify the code to be. In fact, a DMS-based tool has been used to build an aspect weaver [5].

Engineers comfortable with string-hacking tools are sometimes tempted to apply them in situations where they lack adequate power. Instrumenting source code via string recognition and manipulation, for example, comes at very great risk, since string languages have no means for dealing with semantic issues that may come into play. Consider a simple case where a tool is to insert some piece of instrumentation code after any assignment to a certain field 'myfield' of a record of a certain type. Searching for a program text string matching ".myfield = " may identify an assignment to the wrong type record, may not recognize a match because of failure to account for all whitespace possibilities including newlines, and may not recognize whether the identified text is within a comment or not. Intervening manually in the process to resolve these questions may work for small systems but is not viable for large ones. To conduct the operation safely and consistently, semantically aware machinery is required.

3 The Build Process Roadblock

Software quality topics nearly always focus on matters relating to system design and source code, but too rarely deal with build processes. Builds are typically wired together by blends of scripts that navigate between file system manipulations, compiles, links, and all manner of utility calls. The scripting languages do not have the structural regularity of conventional coding languages. Build script processes can be opaque to quality assurance engineers who may be unfamiliar with their operation..

There are usually good reasons for build mechanisms being the way they are, but there are times that build processes can get in the way of software quality measures. Instrumentation for dynamic analysis is a case in point, especially when the source code instrumentation process is automated.

To do accurate identification of insertion points and syntactically accurate insertion of instrumentation requires context and semantic sensitivity, i.e., parsing. But parsing can be confounded by build processes for a variety of reasons. Build processes often set the context for the parsing of a particular file, e.g., by setting environment variables that establish paths to executables, includes, and libraries. Outside the build process, it may not be possible to acquire the environment information necessary to even parse the file in question, much less the semantic content necessary to do accurate instrumentation. Absent environmental context, parse logic like preprocessing conditionals may be unavailable, and macro definitions may be unknowable. Builds can also actually create the source files to be parsed, so that the files that need to be instrumented do not actually come into existence until the build is in progress, making static instrumentation impossible. Moreover, a common characteristic of build processes is that they select only pieces from a large code base for composition into a particular product. Pre-instrumentation of the whole source base may not be appropriate for dealing with products that are assembled from sparsely selected pieces of the base.

Furthermore, modifying home-grown build scripts to do instrumentation at selected times may not be adequate. Build processes sometimes utilize vendor-provided build tools, so that beyond some threshold, in-house mechanisms hand off control and make process modification much more difficult.

4 A Build Process Intervention Technique

These issues all suggest an instrumentation strategy that intervenes in the build process, captures the relevant files, and instruments them just in time for compilation. In general, one cannot count on this opportunity arising, however, until the moment at which the compiler is invoked. However, one knows that for each file of concern, at some point in the build a valid compilation environment is constructed and a compiler is invoked. One can always identify precisely when and how a compiler is invoked and be confident that all the information necessary to do correct parsing and instrumentation is at hand.

We have been successful in taking advantage of this opportunity by using a rather unusual intervention technique. We rename the compiler, replacing it with a command line capture tool that records the relevant environment variable settings and the command line itself. Next it invokes a command script. The script uses the captured information to set up an automated instrumentation tool, invokes the tool, and then invokes the relocated compiler on the instrumented files, directing that it store the resulting object files where the originals would have been stored. On completion of the script, control returns to the capture tool, which catches errors and returns control to the build process for its continuation.

We encompass the entire build process with a script that does the compiler relocation, the placement of the command capture tool in the compiler's original position, the invocation of the original build process, the removal of the capture tool, and the replacement of the compiler in its original home. This entire modified process becomes an atomic action with respect to the host machine, since in its duration the compiler is not available for other purposes. With development and testing typically occurring on single-user desktop machines, this poses no particular problem.

The utility-replacement technique works so neatly that we have applied it also to the vendor-supplied build tool itself, setting aside the build executable to first modify the files that drive the build. Since the system instrumentation includes runtime support, these support files are inserted into the build setup, with modifications specific to the build in progress, so the support files can be compiled as part of the system. One nice aspect of all this is that the technique can be applied without affecting the mainstream development and build process at all, lying dormant unless the developer invokes the wrapper scripts, and staying out of the way of normal system builds. One might imagine other applications of the relocation technique, and we have also considered using it on the linker to introduce other object files.

Mechanizing the relocation of is important, as it minimizes the amount of time the relocated applications are disabled to other uses. Mechanizing also helps ensure care and consistency in the process, important when one is perturbing the development environment itself.

We have successfully applied this technique to complex build processes that assemble code modules from custom code and a large space of USB device driver utilities to build telecommunication device drivers. The source code involved in the build is modified by the build process to do product branding. Components are assembled, compiled, and internally linked, and then devices are assembled from these components. The modules are coded in C [1] and include both custom code and Microsoft DDK modules [4], are compiled with a variety of C compilers, and are built with an elaborate custom process that employs the Microsoft build utility [3] as its bottom layer.

Though our implementation of command capture has process and command line access hooks that are Windows dependent, and the applications in which we intervene (e.g., build and link) are Windows-based, there is no reason why the concepts discussed here would not port readily to other environments. Any conventional platform would support access to command line arguments and offer process control primitives through an API. Any application that is represented by a binary image on an accessible storage device could be relocated and replaced with a command capture tool. We choose scripts to implement our interventions and wrap the entire process, but any other coding mechanisms would suffice. The intervention mechanisms are neither large nor complex and could be coded any way that is convenient for any reasonable platform.

5 Conclusion

Automatic software transformation is a powerful and flexible strategy for instrumenting software to reveal dynamic behavior. A mature transformation infrastructure that parses source code consistently with the language implementation and that performs instrumentation via manipulation of abstract syntax trees rather than strings is highly desirable. Using this technology can be complicated by complex build processes for the target software. One fruitful way of addressing this complication is to insert instrumentation during a build, immediately prior to source file compilation. A technique for doing this reliably is to relocate the compiler, replacing it with a program that captures the compiler invocation and its environment, then invokes a process that performs the instrumentation and re-invokes the relocated compiler on the instrumented code. This trick can be applied for other purposes as well.

This is obviously not deep computer science. We share the strategy with you because, though it is simple, it is not the kind of thing most developers would think of. In a context where it is acceptable to temporarily locate utility software used in software build processes, this redirection strategy provides the benefits of a wrapper strategy while allowing the "wrapping" to occur, in some sense, from the inside, where more context is available. Moreover, it is an illustration of two dynamic instrumentation techniques, one that dynamically modifies a build process, and another that employs source code instrumentation to extract run-time system behavior.

References

[1] *American National Standard for Programming Languages – C*, ANSI/ISO 9899-1990..

[2] I.D. Baxter, C. Pidgeon, and M. Mehlich. "DMS: Program Transformation for Practical Scalable Software Evolution." *International Conference on Software Engineering*, pp. 625-634. IEEE Computer Society, 2004.

[3] "BUILD.EXE command*", http://msdn.microsoft.com.*

[4] "DDKGuide.exe, The Essential Guide to the Windows DDK", *http://msdn.microsoft.com.*

[5] Gray, J., Roychoudhury, S., "A Technique for Constructing Aspect Weavers Using a Program Transformation Engine*", Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, pp 36-45, 2004*

[6] T. Systa, K. Koskimies, "Extracting State Diagrams from Legacy Systems*"*, Technical Report, Department of Computer Science, University of Tampere, Tampere, Finland.

Applying Dynamic Analysis in a Legacy Context: An Industrial Experience Report

Andy Zaidman[1], Bram Adams[2], and Kris De Schutter[2]

[1]LORE, Department of Mathematics and Computer Science, University of Antwerp, Belgium,
Andy.Zaidman@ua.ac.be
[2]SEL, Department of Information Technology (INTEC), University of Ghent, Belgium
{Bram.Adams, Kris.DeSchutter}@ugent.be

Abstract

This paper describes our experiences with applying dynamic analysis solutions with the help of Aspect Orientation (AO) on an industrial legacy application written in C. The purpose of this position paper is two-fold: (1) we want to show that the use of Aspect Orientation to perform dynamic analysis is particularly suited for legacy environments and (2) we want to share our experiences concerning some typical pitfalls when applying any reverse engineering technique on a legacy codebase.

1. Introduction

Legacy software is all-around: software that is still very much useful to an organization – quite often even *indispensable* – but a burden nevertheless. A burden because the adaptation, integration with newer technologies or simply maintenance to keep the software synchronized with the needs of the business, carries a cost that is too great. This burden can even be exaggerated when the original developers, experienced maintainers or up-to-date documentation are not available [10, 5, 8, 6].

Apart from a status-quo scenario, in which the business has to adapt to the software, a number of scenarios are frequently seen:

1. Rewrite the application from scratch, from the legacy environment, to the desired one, using a new set of requirements [4].
2. Reverse engineer the application and rewrite the application from scratch, from the legacy environment, to the desired one [4].
3. Refactor the application. One can refactor the old application, without migrating it, so that change requests can be efficiently implemented; or refactor it to migrate it to a different platform.
4. Often, in an attempt to limit the costs, the old application is ”wrapped” and becomes a component in, or a service for, a new software system. In this scenario, the software still delivers its useful functionality, with the flexibility of a new environment [4]. This works fine and the fact that the old software is still present is slowly forgotten. This leads to a phenomenon which can be called the *black-box syndrome*: the old application, now component or service in the new system, is trusted for what it does, but nobody knows – or wants to know – what goes on internally (white box).
5. A last possibility is a mix of the previous options, in which the old application is seriously changed before being set-up as a component or service in the new environment.

Certainly for scenarios 2, 3, 4 and 5, the software engineer would ideally want to have:

- a good understanding of the application in order to start his/her reengineering operation (or in order to write additional tests before commencing reengineering)[9]
- a well-covering (set of) regression test(s) to check whether the adaptations that are made, are behavior-preserving[6]

However, in practice, legacy applications seldom have up to date documentation available [8], nor do they have a well-covering set of tests.

The actual goals of this experiment are to (1) regain lost knowledge, (2) determine test coverage and (3) identify problematic structures in the source code. For this, we build upon a number of recently developed dynamic analysis techniques that were developed for object-oriented software [12, 11]. The emphasize for this paper however, is more on the pitfalls we encountered along the way when applying the different techniques on a legacy system.

This paper is organized as follows: Section 2 starts with a description of the case study. Section 3 introduces our AOP implementation, while Section 4 briefly discusses the dynamic analysis solutions we used. Section 5 mentions some typical legacy environment pitfalls we stumbled across. Section 6 concludes and points to future work.

2 Case study

The industrial partner that we cooperated within the context of this research experiment is *Koninklijke Apothekersvereniging Van Antwerpen* (KAVA)[1]. Kava is a non-profit organization that groups over a thousand Flemish pharmacists. While originally safeguarding the interests of the pharmaceutical profession, it has evolved into a full fledged service-oriented provider. Among the services they offer is a tarification service – determining the price of medication based on the patient's medical insurance. As such they act as a financial and administrative go-between between the pharmacists and the national healthcare insurance institutions.

Kava was among the first in its industry to realize the need for an automated tarification process, and have taken it on themselves to deliver this service to their members. Some 10 years ago, they developed a suite of applications written in non-ANSI C for this purpose. Due to successive healthcare regulation and technology changes they are very much aware of the necessity to adapt and reengineer this service.

Kava has just finished the process of porting their applications to fully ANSI-C compliant versions, running on Linux. Over the course of this migration effort, it was noted that documentation of these applications was outdated. This provided us with the perfect opportunity to undertake our experiments.

As a scenario for our dynamic analysis, the developers told us that they often use the so-called *TDFS* application as a final check to see whether adaptations in the system have any unforeseen consequences. As such, it should be considered as a functional application, but also as a form of regression test.

[1]http://www.kava.be/

The TDFS-application finally produces a digital and detailed invoice of all prescriptions for the healthcare insurance institutions. This is the end-stage of a monthly control- and tariffing process and acts also as a control-procedure as the results are matched against the aggregate data that is collected earlier in the process.

3. AOP for legacy environments

We recently developed a framework for introducing AOP in legacy languages like Cobol [7] and C [2, 1]. The latter is called *aspicere*[2]. This paper applies aspicere on an industrial case study, provided by one of our partners in the ARRIBA (Architectural Resources for the Restructuring and Integration of Business Applications) research-project[3].

Our industrial partner has a large codebase, mainly written in C, that's why we used aspicere for our experiments.

4. Dynamic analysis solutions

In total we applied 3 dynamic analysis solutions. This section will briefly introduce each of them.

Webmining This solution identifies the most important classes in a system with the help of a heuristic that uses dynamic coupling measures. The idea is based on the fact that tightly coupled classes, can heavily influence the control flow. To add a transitive measure to the binary relation of coupling, webmining principles are used. For a more detailed description of this technique, we refer you to a previous work [11].

Frequency analysis This idea is based on the concept of *Frequency Spectrum Analysis*, first introduced by Thomas Ball [3]. It is centered around the idea that the relative execution frequency of methods or procedures can tell something about which methods or procedures are working together to reach a common goal. For more details we refer to [12].

Test coverage When refactoring or reengineering a system, certain functionality often has to be preserved. Having a well-covering set of tests can be very helpful for determining whether the adaptations to the code are indeed behavior preserving. By establishing the test coverage of modules and procedures, we are able to have a clear view of which parts of the system are tested.

[2]"aspicere" is a Latin verb and means "to look at". Its past participle is "aspectus", so the link with AOP is pretty clear.

[3]Sponsored by the IWT, Flanders. Also see: http://www.iwt.be

```
gcc -c -o file.o file.c
```

Figure 1. Original makefile.

```
gcc -E -o tempfile.c file.c
cp tempfile.c file.c
aspicere -i file.c -o file.c \
     -aspects aspects.lst
gcc -c -o file.o file.c
```

Figure 2. Adapted makefile.

```
.ec.o:
   $(ESQL) -c $*.ec
   rm -f $*.c
```

Figure 3. Original makefile with esql preprocessing.

5. Pitfalls of dynamic analysis in a legacy environment

Applying aspects onto a base program, is intended to happen transparently for the end user. However, while using our experimental legacy AOP tools during our experiments at our industrial partner, we encountered several problems. This section describes some of these.

5.1 Adapting the build process

The Kava application uses `make` to automate the build process. Historically, all 269 makefiles were hand-written by several developers, not always using the same coding-conventions. During a recent migration operation from *UnixWare* to *Linux*, a significant number of makefiles has been automatically generated with the help of *automake*[4]. Despite this, the structure of the makefiles remains heterogeneous, a typical situation in (legacy) systems.

We built a small tool, which parses the makefiles and makes the necessary adaptations. (A typical example is shown in Figures 1 and 2.) However, due to the heterogeneous structure, we weren't able to completely automate the process, so a number of makefile-constructions had to be manually adapted. The situation becomes more difficult when e.g. Informix esql preprocessing needs to be done. This is depicted in Figures 3 and 4.

Using our scripts to alter the makefiles takes a few seconds to run. Detecting where exactly our tool failed and making the necessary manual adaptations took us several hours.

[4]`Automake` is a tool that automatically generates makefiles starting from configuration files. Each generated makefile complies to the GNU Makefile standards and coding style. See `http://sources.redhat.com/automake/`.

```
.ec.o:
   $(ESQL) -e $*.ec
   chmod 777 *
   cp ‘ectoc.sh $*.ec‘ $*.ec
   esql -nup $*.ec $(C_INCLUDE)
   chmod 777 *
   cp ‘ectoicp.sh $*.ec‘ $*.ec
   aspicere -verbose -i $*.ec -o \
      ‘ectoc.sh $*.ec‘ -aspects aspects.lst
   gcc -c ‘ectoc.sh $*.ec‘
   rm -f $*.c
```

Figure 4. Adapted makefile with esql preprocessing.

5.2 Compilation

A typical compile cycle of the application consisting of 407 C modules takes around 15 minutes[5]. We changed the cycle to:

1. Preprocess
2. Weave with aspicere
3. Compile
4. Link

This new cycle took around 17 *hours* to complete. The reason for this substantial increase in time can be attributed to several factors, one of which may be the time needed by the inference engine for matching up advice and join points (still unoptimized).

5.3 Legacy issues

Even though Kava recently migrated from UnixWare to Linux, some remnants of the non-ANSI implementation are still visible in the system. In non-ANSI C, method declarations with empty argument list are allowed. Actual declaration of their arguments is postponed to the corresponding method definitions. As is the case with ellipsis-carrying methods, discovery of the proper argument types must happen from their calling context. Because this type-inferencing is rather complex, it is not fully integrated yet in aspicere. Instead of ignoring the whole base program, we chose to ”skip” (as yet) unsupported join points, introducing some errors in our measurements. To be more precise, we advised 367 files, of which 125 contained skipped join points (one third). Of the 57015 discovered join points, there were only 2362 filtered out, or a minor 4 percent. This is likely due to the fact that in a particular file lots of invocations of the same method have been skipped during weaving, because it was called multiple times with the same or

[5]Timed on a Pentium IV, 2.8GHz running Slackware 10.0

similar variables. This was confirmed by several random screenings of the code.

Another fact to note is that we constantly opened, flushed and closed the tracefile, certainly a non-optimal solution from a performance point of view. Normally, aspicere's weaver transforms aspects into plain compilation modules and advice into ordinary methods of those modules. So, we could get hold of a static file pointer and use this throughout the whole program. However, this would have meant that we had to revise the whole make-hierarchy to link these uniques modules in. Instead, we added a "legacy" mode to our weaver in which advice is transformed to methods of the modules part of the advised base program. This way, the make-architecture remains untouched, but we lose the power of static variables and methods.

5.4 Scaleability issues

Running the program Not only the compilation was influenced by our aspect weaving process. Also the running of the application itself. The scenario we used (see Section 2), normally runs in about 1.5 hours. When adding our tracing advice, it took 7 hours due to the frequent file IO.

Tracefile volume The size of the logfile also proved problematic. The total size is around 90GB, however, the linux 2.4 kernel Kava is using was not compiled with large file support. We also hesitated from doing this afterwards because of the numerous libraries used throughout the various applications and fear for nasty pointer arithmetic waiting to grab us. As a consequence, only files up to 2GB could be produced. So, we had to make sure that we split up the logfiles in smaller files. Furthermore, we compressed these smaller logfiles, to conserve some diskspace.

Effort analysis Table 1 gives an overview of the time-effort of performing each of the analyses. As you can see, even a trouble-free run (i.e. no manual adaptation of makefiles necessary) would at least take 29 hours.

6. Conclusion and future work

This paper describes our experiences with applying dynamic analysis in an industrial legacy C context. We used two dynamic analysis techniques that we had previously developed and validated for Object Oriented software and added a simple test coverage calculation. Furthermore, this paper describes how we used aspicere, our "AspectC" implementation for collecting the traces we needed for performing the dynamic analyses.

Task	Time	Previously
Makefile adaptations	10 s	–
Compilation	17h 38min	15min
Running	7h	1h 30min
Code coverage	5h	–
Frequency analysis	5h	–
Webmining	10h	–
Total	44h 38min 10s	1h 45min

Table 1. Overview of the time-effort of the analyses.

This paper focusses on some common problems we came across when trying to collect an event trace from a legacy C application using aspicere. Some of these problems can be catalogued as being technical, e.g. adapting heterogeneously structured makefiles or overcoming the maximum file size limit of the operating systems.

Some other problems are perhaps more fundamental:

- Performing an effort analysis shows that collecting the trace of the system takes more than 24 hours.
- Subsequently, any dynamic analysis solution, has to cope with analyzing an event trace of around 90 GB. Scaleability of the dynamic analysis solution is thus of the utmost importance.

As such, we can conclude that for what should be considered a medium-scale application, we are already having scaleability issues with our tools. As such, improving the efficiency of our tools is one of our immediate concerns.

7. Acknowledgements

We would like to thank Kava for their cooperation and very generous support.

Kris De Schutter and Andy Zaidman received support within the Belgium research project ARRIBA (*A*rchitectural *R*esources for the *R*estructuring and *I*ntegration of *B*usiness *A*pplications), sponsored by the IWT, Flanders. Bram Adams is supported by a BOF grant from Ghent University.

References

[1] B. Adams, K. De Schutter, and A. Zaidman. AOP for legacy environments, a case study. In *Proceedings of the 2nd European Interactive Workshop on Aspects in Software*, 2005.

[2] B. Adams and T. Tourwé. Aspect Orientation for C: Express yourself. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD*, 2005.

[3] T. Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

[4] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[5] M. Brodie and M. Stonebreaker. *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann, 1995.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[7] R. Lämmel and K. D. Schutter. What does Aspect-Oriented Programming mean to Cobol? In *AOSD '05*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[8] D. L. Moise and K. Wong. An industrial experience in reverse engineering. In *WCRE*, pages 275–284, Washington, DC, USA, 2003. IEEE Computer Society.

[9] H. M. Sneed. Program comprehension for the purpose of testing. In *IWPC*, pages 162–171. IEEE Computer Society, 2004.

[10] H. M. Sneed. An incremental approach to system replacement and integration. In *CSMR*, pages 196–206. IEEE Computer Society, 2005.

[11] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, pages 134–142. IEEE Computer Society, 2005.

[12] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR*, pages 329–338. IEEE Computer Society, 2004.

Crystallizing Application Configurations to Aid Program Comprehension

Ken Zhang
kzhang@swag.uwaterloo.ca

Richard C. Holt
holt@uwaterloo.ca

Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Ontario., Canada, N2L 3G1

Abstract

Software applications have both static and dynamic dependencies. Static dependencies are those derived from the source code and dynamic dependencies are established at runtime and maybe based on information external to the source code, such as configuration. Flexible applications commonly rely on configuration to adapt to diverse environments. An application's configuration encodes runtime dependencies between the various parts of the application. Reverse engineering tools have traditionally been based solely on static dependencies extracted from the source code. Neglecting dynamic dependencies encoded in an application's configuration can result in incorrect or incomplete program comprehension. Unfortunately, many applications store their configuration in an ad hoc, unstructured format from which it is not feasible to extract runtime dependencies by traditional reverse engineering. Our work takes advantage of well structured, published configuration formats, such as that of J2EE applications. By understanding the format we are able to extend reverse engineering to analyse this previously neglected information. We introduce a technique called ***crystallization****, which extracts configuration facts that encode dynamic dependencies. We use these recovered facts to predict and validate dynamic dependencies. Crystallizing configurations has the potential to increase developer productivity by providing better program comprehension.*

1 Introduction

To help developers understand applications, the reverse engineering community commonly uses analysis techniques [6, 10, 11, 12, 13] to extract dependencies from the application's source code and then uses these dependencies to help developers understand the relationships between its components. Unfortunately, dependencies derived from an application's source code may be insufficient to reveal key relationships between its components. This is due to external information such as its configuration adding or modifying relationships between the components. The information encoded in an application's configuration can be essential to the comprehension of a program. But, in many cases, the ad hoc, unstructured format of this configuration information makes it difficult to understand. This is especially difficult in that each application could store its configuration in its own proprietary manner. Integrated Development Environments (IDEs) typically do not understand an application's configuration information and are thus unable to help developers ensure that the configuration is configuration. Fortunately, application frameworks such as J2EE have a well structured, published format to store configuration information. This makes developing program comprehension tools, including IDEs that leverage configuration information possible.

Our technique, which we call *crystallization*, enhances *static* dependencies from source code, with *dynamic* dependencies [10, 11, 12. 13] encoded in the configuration. This more complete extracted information allows us to deduce and validate dynamic dependencies. While our discussion and implementation is based on J2EE, our technique can be applied to other frameworks which use structured formats to configure runtime application behaviour.

We will use an example to illustrate one of the ways that the configuration of a J2EE application determines dynamic dependencies. In Object Oriented (OO) software, a reference to an object instance is used to invoke methods of the object.

J2EE generalizes this approach in that the name of a component is used to invoke predefined methods in particular components.

As illustrated by Figure 1, in traditional OO programming, Java class *Foo* references object *BarImpl* which implements the *Bar* interface. The ***new*** construct creates the object instance. The reference to the new object instance is stored in *ref*, which is used to invoke a method, for example, doPost().

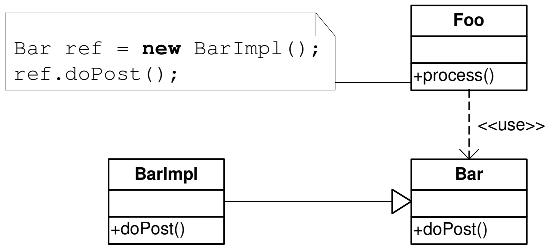


Figure 1: Static Dependency in Traditional OO

J2EE stores configuration information in deployment descriptors (DDs), which are XML files. Deployment descriptors contain definitions of J2EE components including their names, implementing Java classes and other runtime attributes. Each component is defined in a component type tag containing a name and an implementing class element. As illustrated in the pseudo snippet below, a component named *BAR* with implementation class *BarImpl* is declared.

```
…
<component>
  <name>BAR</name>
  <class>BarImpl</class>
</component>
…
```

In J2EE, it is recommended to reference components by name instead of by object reference to acquire the services they provide. A use of the name of a component indicates a dynamic dependency on that component, and hence on the implementing class of the component. For example, at runtime, this statement `HttpServletResponse.sendRedirect("BAR")`, which references component *BAR* by name, triggers J2EE to create an instance of the *BarImpl*, the implementing class of component *BAR* and to invoke *BarImpl*'s predefined method, *doPost()*. If the configuration is changed so that the implementing class of *BAR* becomes *BarImpl2*, an instance of *BarImpl2* would be created and its *doPost()* would be invoked. This is an example of the flexibility J2EE provides to switch implementations without recompilation. In this example, there is neither an object reference nor a function call involved and thus static analysis techniques would be unable to capture the dependency from component *BAR* to the implementing class *BarImpl.*

These dynamic configuration dependencies are neither captured nor indicated by tools such as compilers and IDEs when mis-configured. This increases the likelihood that new members of a development team who are not familiar with the code base will make mistakes. The newcomer, in the process of refactoring the source code, may change the name of a class without knowing the dependencies on the name of the class. This would break dynamic dependencies without introducing any compilation errors. The common approach is to manually inspect the code base which is time consuming and tedious.

In order to tackle the difficulties brought with dynamic dependencies that are native to J2EE, our crystallization processes first extracts component definitions from deployment descriptors and component name references from the source code, and then matches these component name references and component definitions to predict dynamic dependencies. The Crystallization process notifies developers to correct any erroneous dependencies such as references to components that do not exist.

The rest of this paper is organized as follows: Section 2 introduces J2EE components and services. We explain the crystallization process in section 3 and its implementation in section 4. We demonstrate crystallization with an example in section 5 followed by possible enhancements in the future in section 6. Section 7 concludes the paper.

2 J2EE Configuration

Our experiment work is based on the J2EE framework. We will explain J2EE configuration and how it introduces dynamic dependencies between components. This is not meant as a J2EE tutorial but as an introduction to components that are difficult to understand or maintain due to dynamic configuration dependencies.

J2EE provides an architecture framework for enterprises to build multi-tier distributed applications. As shown in Figure 2, J2EE provides JavaServer Page (JSP) [7, 9] and Servlet technology to implement web tier components. It also provides Enterprise JavaBean (EJB) [8, 9] technology to implement business tier components. J2EE supports component communication and interaction using Java Messaging Service (JMS) technology, which supports application modularity, scalability and flexibility.

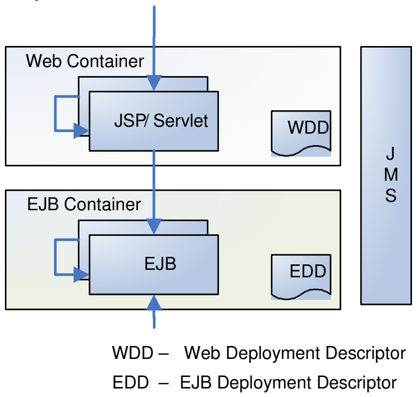

Web Container
JSP/ Servlet
WDD
EJB Container
EJB
EDD
J M S
WDD – Web Deployment Descriptor
EDD – EJB Deployment Descriptor

Figure 2: J2EE Components and Services

2.1 Web Tier Components and Configuration

JavaServer Pages (JSPs) and Servlets are technologies used to implement web tier components of J2EE applications. These components run in a Web Container as shown in Figure 2 and greatly simplify user interface development. JSPs are comprised of HTML intermingled with scriptlets of Java code which dynamically construct HTML pages. Servlets are written in pure Java. While they can be used for the same purpose as a JSP, their intended purpose is to provide business workflow control. This allows JSP developers to take HTML pages that are designed by graphic designers and add business logic such as retrieving data dynamically without dramatically altering the page and worrying about the page layout. User input collected from HTML forms, either created statically or dynamically by JSPs, is normally submitted to Servlets. These Servlets collate the input and invoke business logic components such as EJBs to process the input.

```
<servlet>
    <servlet-name>BARServlet</servlet-name>
    <servlet-class>example.web.Bar</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>BARServlet</servlet-name>
   <url-pattern>BAR.DO</url-pattern>
</servlet-mapping>
```

Figure 3: Snippet from Web Deployment Descriptor

Each JSPs or Servlets is assigned a name in the web deployment descriptor. As illustrated in Figure 3, servlets are defined in a servlet tag, *<servlet>*, which contains a servlet name element, *<servlet-name>*, and a servlet class element, *<servlet-class>*. The servlet name is an internal name, e.g. *BARServlet*, which is used to define the external name that the servlet will be referenced by. The servlet class defines the implementing class. A servlet definition is followed by a servlet mapping tag, *<servlet-mapping>*, containing a mapping from the internal name to the external name, e.g. from *BARServlet* to *BAR.DO*. A component's name, optionally prefixed with the name of the server hosting the J2EE application forms a Unified Resource Locator (URL), which is used to reference the component.

As elaborated, a reference, possibly in URL format, to a web component name defined in the web deployment descriptor indicates a dynamic dependency on the implementing class of the component. This dependency is not visible to static analysis techniques which do not analyze this configuration.

2.2 Business Tier Components and Configuration

An EJB is a business tier component and runs in an EJB Container as depicted in Figure 2. EJBs can be deployed on multiple servers and the J2EE framework provides load balancing and fail-over protection to improve service performance and availability. In addition to these services, J2EE also provides transaction management facilities to applications.

At runtime, EJB service requesters ask the J2EE framework for an EJB instance by *name*. Figure 4 shows a snippet from an EJB deployment descriptor that defines an EJB named *BAREJB* as specified in the *<ejb-name>* tag. The *BAREJB* provides services defined in the remote interface, *example.ejb.BarImpl* as specified in the *<remote>*

tag. The services *BAREJB* provides are implemented in *example.ejb.BarImpl* as specified in the <*ejb-class*> tag.

```
<enterprise-beans>
<session>
        <display-name>BAREJB</display-name>
        <ejb-name>BAREJB</ejb-name>
        <remote>example.ejb.BarRemote</remote>
        <ejb-class>exampl.ejb.BarImpl</ejb-class>
<session-type>Stateful</session-type>
```

Figure 4: Snippet from EJB Deployment Descriptor

Each EJB service requester has an object reference to the EJB remote interface, *example.ejb.BarRemote*, while the implementation logically "implements" the remote interface. By logically implementing the class, we mean the implementing Java class does not have to inherit the remote interface using the Java keyword "**implements**" although it does contain implementation of the methods defined in *example.ejb.BarRemote*.

J2EE discourages EJB implementing classes from implementing EJBs by inheriting the remote interface. Instead, it is the configuration, namely the EJB Deployment Descriptor, which glues the parts of EJBs together. A reference to the remote interface indicates a dynamic dependency on the EJB implementation class. Unfortunately, static analysis techniques are unable to capture these dependencies because the service requester references the remote interface which is not inherited by the implementing class.

2.3 Java Messaging Service (JMS)

JMS allows J2EE components to communicate by exchanging synchronous or asynchronous messages using message queues. There are types of JMS communication mechanisms other than message queue. While we focus on queue in our research, a similar approach can be applied to other types of JMS communication mechanism. In the queue based communication model, messages are stored in queues which can receive messages from multiple senders. JMS queues can also have multiple receivers. When JMS delivers a message, it picks a receiver and delivers the message to that receiver.

The message based communication model decouples message senders from message receivers. At compile time, each message sender knows the name of queue to which it is sending and each receiver knows the name of queue from which it is receiving. However a sender does not in general know which receiver will receive a given message, nor does a receiver know which receiver sent a message. This allows different developers, possibly different vendors to work on senders and receivers separately as long as a common message format is agreed upon. The ability to ensure the persistence of messages allows the message sender and receiver to run asynchronously. These persisted messages are delivered when receivers become available. Adding more receivers increases the throughput of message processing leading to better performance.

As illustrated in Figure 5, *Foo* and *Bar* are not statically dependent on each other. At runtime, *Foo* and *Bar* ask the J2EE framework for a reference to a common queue by invoking a predefined method. This queue object is used to send or receive messages. It is clear that if *Foo*, the sender and *Bar*, the receiver are referencing the same queue, *Foo* has a dynamic dependency on *Bar*.

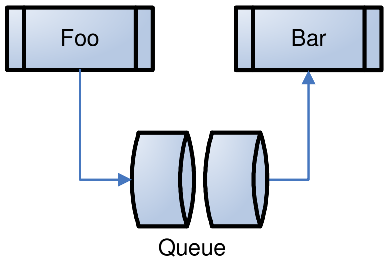


Figure 5: JMS Communication

Although there are no configuration files required for components using JMS, we consider the name of the JMS queue to be configuration information since this can be changed without affecting expected behaviour. In fact, the name of the JMS queue used by components is normally stored in a configuration file as key value pairs such as: *Order.Queue.Name=orderQueue*, although this is not specified by J2EE as a standard configuration file.

2.4 Development Challenges

Although J2EE provides great flexibility to configure application behaviour to meet changing business requirements without recompilation, it also introduces additional complexity to application development. Because components are statically decoupled and no longer visible to regular IDEs, dynamic configuration dependencies that are established at runtime are often not apparent to developers.

For example, large, evolving projects with web interfaces may have many, possibly thousands of ever changing JSPs. And due to changing requirements, JSPs can be obsolete very quickly without immediately being removed from the code base, which results in many unused JSPs. Without assistance from tools that crystallize dynamic configuration dependencies, it is difficult to locate and remove these unused JSPs due to these unapparent dynamic dependencies.

EJBs also pose problems, since the implementing class is not required to inherit the remote interface. Missing implementations of exposed methods would not immediately result in compilation errors. This delays detection of these errors and consequently negatively impacts programming productivity.

The complexity of JMS technology makes it difficult to dependencies between components. Without a tool that understands J2EE configuration, problem determination requires time consuming error prone manual inspection.

3 Crystallizing Dynamic Configuration Dependencies

Understanding dynamic configuration dependencies is a challenge facing J2EE application developers. Existing Java compilers and IDEs do not notify developers of erroneous dependencies resulting from mis-configuration. To assist developers in overcoming these challenges we crystallize the configuration information into an understandable form.

We accomplish this using our crystallization process as follows. First, we analyze the J2EE technology and its configuration to identify configuration and coding patterns that could result in runtime dependencies. Second, we search for the identified patterns in the source code and configuration to predict dynamic dependencies. Finally, developers are presented with the recovered dependencies in an easily consumed form using color highlighting.

3.1 Crystallization Process

In order to crystallize dynamic configuration information, we need to understand *what* and *how* J2EE components are invoked. Different types of J2EE components are invoked in different ways. J2EE provides APIs to invoke J2EE components. The invoked J2EE components are normally identified by name which is passed to the API methods as parameters. The name of J2EE components has to be further resolved into Java classes by investigating J2EE configuration files, e.g. Deployment Descriptors.

Figure 6 shows the type of documents that are included in our process. Java Server and HTML Pages, Web deployment descriptors, and EJB Deployment descriptors are analyzed using our crystallization parsers. The Java source code is analyzed using traditional reverse engineering methods to extract static dependencies. Further, our crystallization parser is also applied to capture parameters that are used to invoke special APIs. These parameters are use later in the process to determine the target of the invocation.

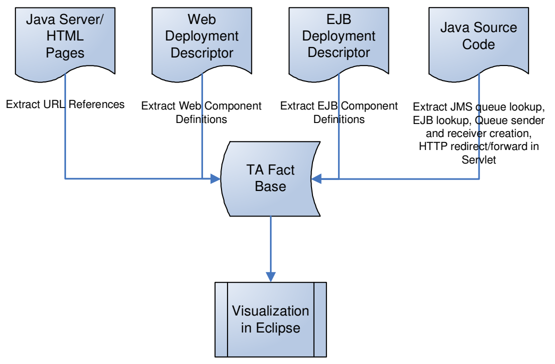

Java Server/ HTML Pages
Web Deployment Descriptor
EJB Deployment Descriptor
Java Source Code
Extract URL References
Extract Web Component Definitions
Extract EJB Component Definitions
Extract JMS queue lookup, EJB lookup, Queue sender and receiver creation, HTTP redirect/forward in Servlet
TA Fact Base
Visualization in Eclipse

Figure 6: Crystallization Process

3.2 Dependency Notation

The extracted dependencies including method invocations, which are relationships between components, are stored in Tuple Attribute[2] format as follows:

```
dependency-type origin destination
```

A reference to URL, *bar.do* in *foo.java* is stored as

```
url-reference foo.java bar.do.
```

A web component named *bar.do* with implementing class, *example.web.Bar* is stored as

```
web-define bar.do example.web.Bar.
```

The composition of these two relations reveals the real relation between *foo.java* and *example.web.Bar*. The employment of TA allows us to apply relational calculus operations such as union, subset, and composition [2] on the set of extracted references to obtain higher level, meaningful relations.

3.3 **Important J2EE APIs**

Method invocations to particular J2EE APIs such as Servlet request dispatches and JMS queue sender and receiver creation indicates runtime dependencies on Sevlet and JMS queue, respectively. The invocation and the parameters to the invocation, together with information gathered from deployment descriptors, enable us to locate target J2EE components to be invoked at runtime and hence allowing us to predict possible dynamic dependencies in the source code.

It is normal for Servlets to forward requests to another servlet for further process. In order to do so, the Servlet creates a *RequestDispatcher* object with the URL of the target servlet and calls its *forward()* method. These invocations indicate dependencies from the Servlet to the web component represented by the URL.

Senders and receivers in JMS communication are distinguished by the API method invoked. Senders and receivers are created by invoking *javax.jms.QueueSession.createSender()* and *javax.jms.QueueSession.createReceiver()* respectively. Based on this difference, we are able to determine the direction of the communication and hence the dependency.

Table 1 summarizes the types of reference that can be found in a type of components. E.g. we can find component-url type references in web components defined in HTML and JSP pages

Component Type	**References Captured**	**Sources Included**
Web Component	"component-url"	HTML, JSP

	sendRedirect()
	forward()
		Servlet

EJB	ejb.RemoteInterface	Java

JMS	Queue queue = …
	createSender();
	createReceiver();
		Java

Table 1: Component References and Sources

3.4 Crystallizing Web Dependencies

Web components are referenced by their URLs in JSPs, Servlets and HTML files. There are various places these URLs may be used as enumerated by the following list:

- HTML links, e.g.,
  `<a href = "bar.do">bar</a>`
- HTML form action targets
  `<form action="bar.do">`
- JSP forward tags, a special tag used by JSP to forward HTTP requests
- Servlet request redirects and forwards, invocations to *sendRedirect()* and *forward()* introduced in section 3.3

Extracting the URL references in these files is the first step to crystallizing web dependencies. For this step, We have built three parsers. The HTML/JSP parser extracts references to URLs. The Servlet parser captures method invocations to *sendRedirect()* and *forward()*. The DD parser extracts web component definitions.

The following source code snippet shows how a HTTP request is dispatched to "*bar.do*" by invoking the *RequestDispatcher.forward()* method.

```
ServletContext context =
getServletConfig().getServletContext();
RequestDispatcher dispatcher =
context.getRequestDispatcher("bar.do");
dispatcher.forward(req, resp);
```

In the deployment descriptor shown in

```
<servlet>
    <servlet-name>BARServlet</servlet-name>
    <servlet-class>example.web.Bar</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>BARServlet</servlet-name>
   <url-pattern>BAR.DO</url-pattern>
</servlet-mapping>
```

Figure 3, "*bar.do*" is implemented by *example.web.Bar*. An HTTP request to "*bar.do*" results in the invocation of the predefined method, *example.web.Bar.doPost()*.

The extracted URL reference in TA notation, `url-reference foo.java bar.do`, is composed with the web component definition, `web-define bar.do example.web.Bar`, which reveals a dynamic dependency from *foo.java* to *example.web.Bar* as shown in Figure 7.

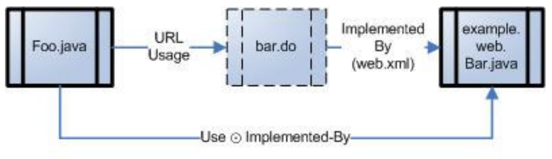


Figure 7: Crystallized Web Dependency

3.5 Crystallizing EJB Dependencies

EJB callers possess references to the remote interface defined in the EJB deployment descriptor. The remote interface defines the business methods that are exposed by the EJB.

As shown in the following code snippet, the EJB caller has a reference *bean* to the BarRemote interface as depicted in Figure 4. The reference *bean* is then used to invoke business methods.

```
BarRemote bean = getInstance();
bean.method1();
bean.method2();
```

The EJB parser captures references to remote interfaces of EJBs. As shown in Figure 8, the remote interface reference relation is composed with EJB definitions found in EJB deployment descriptors to reveal the real dependency between the referencing component and the EJB implementing class.

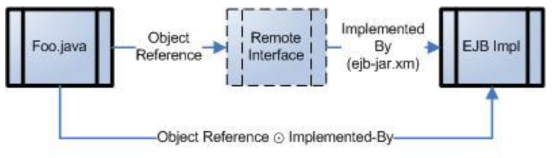

Foo.java
Object Reference
Remote Interface
Implemented By (ejb-jar.xm)
EJB Impl
Object Reference ⊙ Implemented-By

Figure 8: EJB Interaction

3.6 Crystallizing JMS Dependencies

In order to communicate through a JMS queue, senders and receivers must have a reference to the JMS queue. Capturing JMS interactions starts with capturing JMS queue references. JMS queue references reveal all components; consisting of senders and receivers. However, we are not only interested in the participants of communication but also their relationships. JMS participants invoke the *createSender()* and *createReciever()* methods, capturing these invocations allows us to separate them into senders and receivers.

The following code snippet shows how *QueueSender* and *QueueReciever* objects are created. JMS participants ask J2EE for a reference to a queue instance. To initiate communication a connection must be established followed by opening a session.

```
Queue queue = (Queue)  context.lookup("OrderQueue");
QueueConnection conn = createConnection();
QueueSession session = createSession(conn);
QueueSender qSender = session.createSender(queue);
QueueReceiver qReceiver = session.createReceiver(queue);
```

JMS queue communication involves sender(s) and receiver(s); we need to determine which sender(s) is associated with which receiver(s). This is achieved by matching sender(s) and receiver(s) that communicate through the same queue. Since a reference to a JMS queue is obtained from J2EE by queue name as follows:

Queue queue = (Queue) context.lookup("OrderQueue");

In TA notation, we have the following:

```
queue-send Foo.java OrderQueue
queue-recv Bar OrderQueue
```

The composition of these relations yields the following dependency:

```
dependency foo.java bar.java
```

This is illustrated in Figure 9.

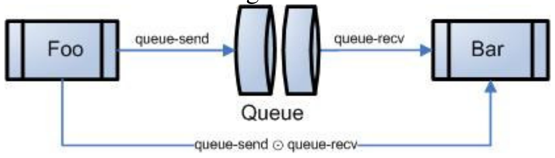


Figure 9: Crystallized JMS Interaction

4 Implementation

The parsers we have implemented extract dependencies from HTML files, JSPs, Java sources and deployment descriptors. Moreover, we integrated these parsers into the Eclipse Java Development Tool [1], a popular Java IDE, to improve programming efficiency and comprehension by providing easy navigation. By easy navigation we mean the ability to follow dynamic dependencies and present their endpoint implementation such as a Servlet or EJB implementing class. The integration of these parsers within an IDE ensures the instant accessibility of the Crystallization technique without switching to another tool.

Reverse engineering is traditionally a slow process because it extracts and calculates dependencies from the complete code base. As we are integrating our crystallization technique into Eclipse, it is unacceptable for this integration to incur a perceivable impact on its responsiveness. In order to achieve this, we employ a "lazy" approach, which extracts and processes only those dependencies in the source code currently being edited. These extracted dependencies are cached into an in-memory database for reuse.

4.1 Crystallization Framework

The crystallization framework is responsible for presenting dynamic dependencies in a manner that does not interfere with daily programming activities. Our integration of Crystallization into Eclipse does not clutter the user interface with syntax highlighting. Dependencies are presented to developers as HTML like links that become visible only when the "control" key is pressed while the curser is over the origin of a dynamic dependency. By clicking on the link, the IDE will unveil the implementing source code in an editor. This allows the developer to easily rationalize and comprehend dynamic dependencies. The framework displays erroneous dependencies by placing problem markers, shown as red crosses, beside their origins in the source code editor.

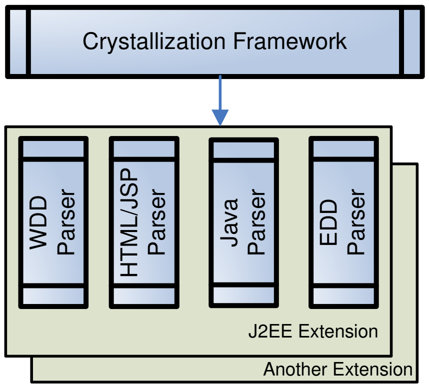


Figure 10: Crystallization Framework

As shown in Figure 10, the Crystallization Framework relies on extensions to detect and validate dynamic dependencies. The crystallization framework is implemented as an Eclipse extension point to leverage the automatic discovery of extensions. Crystallization extensions must implement an extractor which extracts a specific type of dependency and a validator which validates the extracted dependency.

Although we have only implemented a J2EE extension to assist developers in comprehending J2EE applications, crystallization is extensible to encompass other application frameworks.

4.2 Crystallization Extensions

A crystallization extension consists of an extractor and validator for dynamic dependencies in a specific domain.

An extension is added to the Crystallization Framework simply by adding its compiled binary code to a predefined directory known to Eclipse. When Eclipse is started, it examines the directory and notifies the Crystallization Framework of available extensions.

Detected extensions are activated automatically and applied to the source code and thus provides the ability to detect and validate applicable dynamic dependencies.

4.3 Dependency Visualization

Besides revealing valid and erroneous dynamic dependencies through the use of HTML like links and problem markers, we have also implemented a view which presents all dynamic dependencies of the currently edited source file in a graph. Valid and erroneous dynamic dependencies are differentiated through the use color. This view provides developers with a high level overview of all dynamic dependencies in the source file currently being edited.

4.4 Performance and Scalability

In order to achieve acceptable performance and still allow for quick dynamic dependency detection, we employ an in-memory database to store information about extracted dependencies. For traditional reverse engineering processes which extract all dependencies from all of the source files at once, storing these dependencies in memory may present problems and inhibit scalability and performance. Since Crystallization extracts only dependencies from files that are currently being edited, the number of dependencies is reduced. In one test, Eclipse demonstrated acceptable responsiveness with one million dependencies in the in-memory database.

Our extraction strategy ensures the ability to scale to large projects because a developer is only capable of working on a handful of files at any given moment. This means the number of dependencies stored in the database is not proportional to the size of the project, instead; it is proportional to the number of scrutinized files.

5 Case Study

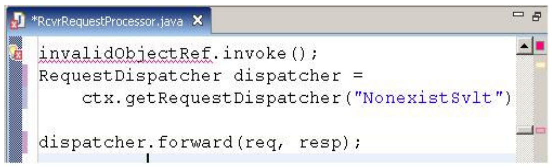
The Pet Store application is a sample J2EE application [3] from Sun Microsystems used to evangelize J2EE technologies. The Pet Store application demonstrates the capabilities these technologies provide to develop robust, scalable, portable and maintainable distributed e-business enterprise applications. We use this application to demonstrate how the crystallization technique increases the visibility of erroneous dependencies and hence improves programming efficiency and quality. The following table enumerates the artifacts found in the Pet Store application.

Number of Java Classes	283
Number of JSPs	75
Lines of Java Source Code	45261
EJB Components	23
Lines of Configuration (WDD + EDD)	14710
Other Configuration Files	20

We imported the source code of the Pet Store into an Eclipse project; the unmodified version of Eclipse is unable to detect any dynamic dependencies in the project. No error messages are displayed when we change the code base by removing needed JSPs or referencing nonexistent JSPs. These errors would only manifest themselves at runtime.

5.1 Dependency Manifestation

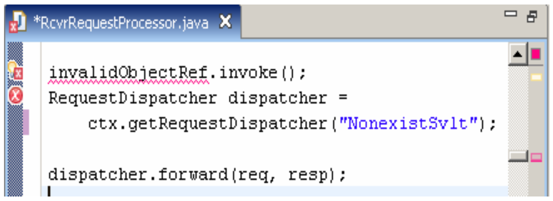
The editor contained in the unmodified Eclipse provides real-time dependency checking and syntax highlighting. Eclipse pinpoints erroneous static dependencies though the use of problem markers and underlining. Within the Crystallization Framework, invalid dynamic dependencies are shown in the same manner.



```
invalidObjectRef.invoke();
RequestDispatcher dispatcher =
    ctx.getRequestDispatcher("NonexistSvlt")

dispatcher.forward(req, resp);
```

Figure 11: Highlighting Invalid Dependency in Unmodified Eclipse



```
invalidObjectRef.invoke();
RequestDispatcher dispatcher =
    ctx.getRequestDispatcher("NonexistSvlt");

dispatcher.forward(req, resp);
```

Figure 12: Highlighting Invalid Dependency in "Crystallized" Eclipse

Figure 11 is a screen shot of the unmodified Eclipse showing an invalid dependency, "`invalidObjectRef`". It is unaware of the other invalid dependency, "`NonexistSvlt`". However, the "crystallized" Eclipse is aware of both as shown in Figure 12. The invalid dependency is shown on the left hand side of the editor pane as a red cross. Invalid Servlet references are also shown in the same manner as invalid object references. The attention grabbing red color enables developers to quickly identify problems. On the right hand side is a bookmark, which if clicked will take the developer immediately to the origin of the invalid dependency. This is especially beneficial when working with large source files.

The "lazy" approach we employ allows us to focus on a specific source file without incurring a noticeable impact on responsiveness. Dependencies are extracted from the source code and deployment descriptors and validated on the fly as the developer is typing. Changes to source files trigger the extraction and validation process to ensure the up-to-date analysis of dependencies.

Since extracted dependencies are not discarded when developers switch to another source file, we do not perceive any negative impact in responsiveness after many files are scrutinized. This ensures the stability of the enhanced Eclipse.

5.2 Source File Visualization

Although the erroneous dependencies are indicated clearly in the source editor, this presentation is insufficient to provide a concise overview of large source files with many dependencies. We have thus introduced a "Class View" which presents dynamic dependencies that are extracted by our crystallization process.

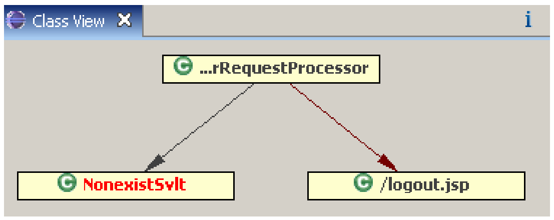


Figure 13: Class View

Figure 13 shows the dynamic dependencies extracted from the "`RcvrRequestProcessor`" servlet. It shows a dependency against the "`NonexistSvlt`" in red which indicates an erroneous dependency. This view is refreshed whenever there is a change to the source code.

6 Future Work

Our implementation covers the case where string literals are used to represent component names. However, it is unable to detect dependencies on components referenced using constant string variables even though their values are known at compile time.

Although our research covers several important areas of J2EE, we are not crystallizing tag libraries. Tag libraries allow developers to define custom tags similar to HTML tags. These tags are mapped to Java class executed on the J2EE server. An HTML or JSP page using a custom tag has a dependency on the tag library's implementing Java class.

We have not used extracted dynamic dependencies to help derive application architectures. With an integrated application architecture viewer, the crystallization process can assist developers, especially new comers to quickly grasp the intricacies of application components and their interrelations and thus gain program comprehension.

7 Conclusion

Typically developers use a "trial and error" strategy to gain program comprehension about applications. The feedback from IDEs assists developers in learning the dependencies between components. Without prompt feedback on dynamic dependencies, the "trial and error" learning cycle is prolonged and hence results in longer learning cycle. Developers must either wait until runtime or manually inspect the configuration and source code to observe dynamic dependencies. This resulting long turn around time complicates the program comprehension process.

Crystallization allows IDEs to detect and validate dynamic dependencies in applications. The results of crystallization can potentially improve developer efficiency, coding productivity and code quality.

References:

[1] The Eclipse Project, http://www.eclipse.org

[2] R. C. Holt. An Introduction to TA: the Tuple-Attribute Language, March 1997

[3] Java Pet Store. http://java.sun.com/developer/releases/petstore/, Sun Microsystems Inc.

[4] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf, *IBM Systems Journal,* Vol. 36, No. 4, pp. 564-593, November 1997.

[5] Inversion of Control Containers and the Dependency Injection Pattern. Martin Fowler, http://www.martinfowler.com/articles/injection.html

[6] Ahmed E. Hassan. Architecture Recovery of Web Applications, *Master's Thesis. Department of Computer Science, Faculty of Mathematics, University of Waterloo, Ontario, Canada. 2001*

[7] JavaServer Pages Technology - Documentation. http://java.sun.com/products/jsp/docs.html

[8] Enterprise JavaBeans Fundamentals: Introduction. http://java.sun.com/developer/onlineTraining/EJBIntro/

[9] J2EE introduction. http://java.sun.com/developer/technicalArticles/J2EE/Intro/

[10] Lei Wu, Houari Sahraoui, Petki Valtchev. Program comprehension with dynamic recovery of code collaboration patterns and roles. *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research. 2004*

[11] Kenny Wong. Software Understanding through integrated structural and run-time analysis. *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research.* 1994.

[12] Carlo Bellettini, Alessandro Marchetoo, Andrea Trentini. WebUml: Reverse Engineering of Web Applications. *Proceedings of the 2004 ACM symposium on Applied computing. 2004.*

[13] Eleni Stroulia, Tarja Systä. Dynamic Analysis for Reverse Engineering and Program Understanding. *ACM SIGAPP Applied Computing Review. 2002.*

DDgraph: a Tool to Visualize Dynamic Dependences

Françoise Balmas Harald Wertz Rim Chaabane
Laboratoire Intelligence Artificielle
Université Paris 8
93526 Saint-Denis (France)
{fb,hw,lysop}@ai.univ-paris8.fr

Abstract

Following previous work on displaying static *data dependences and experience with large sets of dependence displaying strategies, we developed a tool for visualizing* dynamic *data dependences.*

Our prototype is based on a modified Lisp interpreter and this paper presents our evaluation of its application to a highly complex AI program. This permitted us to build efficient visualizations and to evaluate the benefits of using dynamic dependences for program understanding, debugging and correctness checking.

In this paper, we present our prototype, detailing especially the different visualizations we introduced to allow users to deal with hard to understand programs, and we discuss our findings working with dynamic dependencies.

1. Introduction

In this paper, we report on our research using dynamic data dependences during program maintenance.

In previous work on static data dependences [3], where we developed displaying strategies for very large sets of dependences, we discovered that visualizing sample values for a well chosen execution could be of great help to understand what a program computes and how it works [2]. This pushed us to explore dynamic dependences – dynamic analysis is recognized to bring *precise* information for a given execution [1] – and to evaluate the benefits of visualizing them for those activities where knowledge about given executions is crucial, that is program understanding, debugging and correctness checking.

For the sake of evaluation, we developed a prototype around the Lisp language; actually, modifying an interpreter is much easier than modifying a compiler, and hard to understand Lisp programs are still small enough to prevent algorithmic and optimization problems which arise when manipulating huge amounts of data. We thus modified a Lisp interpreter to let it, in addition to normal execution of programs, extract dependences at runtime. These dependences are sent to a Lisp program that acts as a database, storing the dependences and producing on demand the corresponding graph – in *dot* [5] format. Finally, a Tcl/Tk GUI displays the graph, using strategies to reduce its size, and allows users to interact with it to tune several kinds of visualizations.

To evaluate our approach, we applied our tool to a version of the classical AI Blocks World program [6]. In our version, the world is a table and the blocks – different possible shapes of objects – are manipulated by a one-handed robot. Basically, the program presents itself as an interpreter the user interacts with in order to create objects, let the robot move them to other places or ask for information about the current state of the world.

The program is around 1200 LOC long[1] and includes more than 125 functions and macros, many global variables modified through pointers, indirect recursive calls, thus long circularities, and escapes (i.e. non standard return controls). It evolved over time, since first developed for an AI programming class and then modified several times to add further reasonning capabilities. All these features make this program rather complex, hard to understand for newcomers to the program and difficult to maintain for the one of us who developed it.

In this paper, we present our tool (Section 3), the different kinds of visualizations we defined (Section 4) and then we discuss the benefits we got for the maintenance of a hard to understand program (Section 4 and 5).

2. Tool

Our tool relies on three modules: a modified Lisp interpreter (a C version is under construction [4]), a database (currently a Lisp program) and a GUI (implemented in

[1]Note that LOC in Lisp is very different from LOC in more usual programming languages such as C, because of the compactness of code and the powerfull functional primitives it offers.

```
(de square (a)
   (* a a))

(de som2 (x y)
   (+ (square x) (square y)))

? (som2 3 5)
= 34
```

Figure 1. Sample code

Tcl/Tk). We modified a Lisp interpreter to make it, in addition to normal execution of programs, extract dependences at runtime. These dependences are sent to a Lisp program that acts as a database, storing the dependences and producing, on demand, the corresponding graph – in *dot* [5] format. Finally, a Tcl/Tk GUI displays the graph, using mechanisms to reduce its size, and allows users to interact with it to tune several kinds of visualizations.

The full set of dependences for a given call is unlikely to be displayed as is, since it is usually to large to be readable. For this reason, following our past experience with displaying strategies to deal with large sets of dependences [3], we integrated *aggregation* and *filtering* mechanisms in our tool.

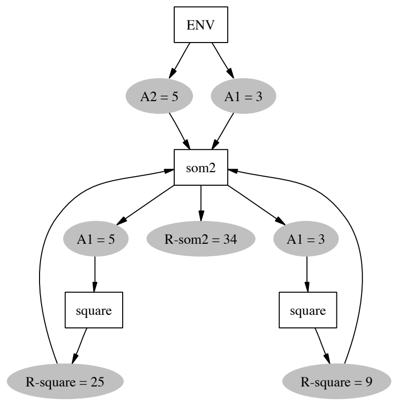


Figure 2. Data dependence graph with all calls visible

Aggregation is done by *grouping* together nodes (that is pieces of code) belonging to the same function call. For example, in the sample code of Fig. 1, wich computes the sum of the square of two numbers, we have nodes belonging to the two calls to function square and we aggregate them to form two groups. These two groups, as well as other nodes, belong to function som2 and are aggregated to form the main group. We can then display dependences showing only these groups, thus only the calls, and the dependences between them. Fig. 2 gives the corresponding graph for the call (som2 3 5) and shows how values are transmitted

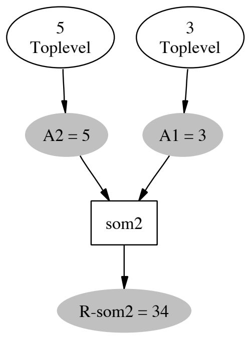


Figure 3. Data dependence graph with only the toplevel call visible

between calls. Alternatively, we can also get a graph with only the toplevel call visible (see Fig. 3), showing just input and output of the whole program. Such views are very helpful when global variables are used and modified by the program (see Section 4).

For a large program, the number of function calls may become too large to get readable graphs. For this reason, our filtering mechanism classifies functions into control structures (they are functions in Lisp), primitives (those standard functions that are implemented in Lisp itself), routines (small reusable functions related to the program at hand) and user functions (all the remaining functions). The next Section will show different visualizations that depend on this classification to filter out given set of calls.

3. Visualizations

Our basic navigational functionalities – going down/up one level while opening/closing groups – becomes tedious as soon as the call tree exceeds more than a dozen levels. Actually, a typical call to the robot instruction for moving an object produces a call tree of more than 3600 groups (calls), distributed in a maximum depth of 90 levels and 45 in the mean. That's why we propose different visualizations of the call graph to use as an help either to understand the program or to navigate in the dependences graphs. This Section introduces the different possible visualization of both call graphs and data dependence graphs.

Call Graph This view is based on the group hierarchy created to handle aggregation and shows the different calls performed during the program execution. It is displayed in another window than the data dependence graph.

Such a visualization offers a global overview of the functions the program evaluated and the way they are organized (see the Section 4). It also permits the user to ask for a given

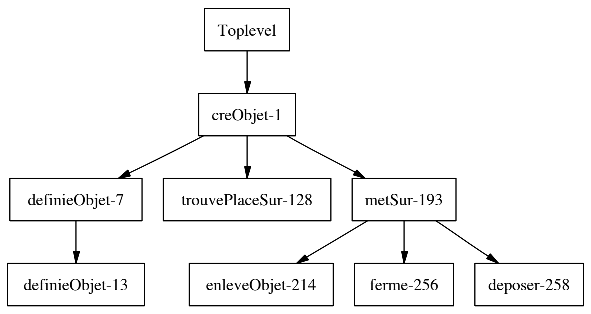


Figure 4. User call graph

data dependence graph by interactively selecting a call. This group becomes then the focus of the displayed data dependence graph (see below).

Note that such call graphs may be very large, thus restricted versions are also available (see below).

User call graph This is a restricted version of the call graph just described where only user functions are shown. This not only permits to get a graph with much fewer groups – from more than 3600 groups in the whole call graph for a 'move-object' instruction we could get down to about 30 groups –, thus more easily readable, but also to get a global overview of the main function calls from a programmer's conceptual perspective. In Fig. 4, we see the user call graph for the creation of an object: from the initial 159 groups, only 9 are displayed.

One level user call graph This view is a mix of the two previous. Actually, in many cases, once the programmer found the function s/he is interested in investigating further, s/he might be willing to know more about *all* the calls performed by this function, and not only the user function calls. For this, we provide a call graph beginning at a given user function and ending at the next user function call, that is when traversing the call tree, we stop drawing the graph whenever we reach leaves or we encounter user functions.

Return graph The Blocks World program uses intensively the 'escape' mechanism of Lisp[2] that allows the program control to directly return to a calling function up in the call tree, restoring the local environment of the place where the 'escape' was set. If this clearly eases coding and speeds up execution time – less tests are to be written and evaluated – it also seriousely complicates maintenance and debugging: as soon as several 'escapes' are embedded, because in recursive calls, it becomes hard to conceptually follow where the control is supposed to get back and how the program is supposed to continue after the activation of the 'escape'.

[2]Sometimes called 'catch-and-throw', this mechanism is similar to the 'setjmp-longjmp' mechanism of C.

That's why we integrated the possibility to extend the *call* graphs with the *return* graph: whenever control gets back to another function than the one that called the current one, the return arrow is displayed in red.

Data dependence graph This visualization provides the standard data dependence graph as we introduced in Section 2, with either only the top level call, or all calls visible. It may focus on a given call, this way considering only the sub calltree beginning at this call.

The construction of several different views is possible. When all groups are visible, the visualization gives a global overview of the different calls of a program execution, showing more specifically how arguments and returned values are transmitted between calls. When only the main group is visible, one can clearly see the effect of the call on global variables. When one or more groups are open, examination of the detail of the evaluated code is possible.

Examples are given in Sections 2 and 4 (Fig. 2, 3, 5 and 6). The next Section will further discuss this visualization.

Filtered data dependence graphs This visualization is obtained whenever classes of functions are flagged to be filtered out. It is especially useful with data dependence graphs where all groups are to be displayed, since it permits to hide functions of lesser interest for the task at hand. For example, it is often useful to filter out primitives – very often recursive functions called a huge number of times – that fill a graph with irrelevant information. Displaying control structures is also often useless when the programmer is more interested in focusing on *what* the program computes than on *how* it does it. To the contrary, s/he might be interested in examing the overall control of the program execution without considering how it is encoded in functions.

With this mechanism, one has just to tune the settings for each class of functions and then to select a group – in a call graph for example – and the tool automatically builds the corresponding view.

First level graphs The two basic possibilities to examine groups – only the top group visible, or any group visible – proved to be insufficient in several cases, since giving either too few or too many details. We extended our tool functionalities with a view where the focus group is visible along with each first level group. This allows the user to examine how a given action – implemented by a function call – is decomposed into smaller actions without the need to examine the actual code of the call, which is always visible through the group nodes. One can then navigate up/down one level for further examination.

Note that the filtering out of given function classes is also active in this view.

Sets of groups Sometimes, the automatically built views we just described are not satisfying because centered on *one* function, while we might need the ability to see a *set* of specific calls, especially to examine the values of global variables before and after these different calls (see discussion in Section 4). For this reason, selecting a few groups on a call graph results in a data dependence view where only these groups are shown while all others are hidden.

The different visualizations presented in this section were inspired by the needs we encountered during the process of trying to understand a rather large and complex program. They proved to be very useful for interactive goal-directed exploration. In the next Section we will discuss the use of dynamic data dependences during program maintenance.

4. Dynamic data dependences for program maintenance

In this section, we report on different programming activities around the Blocks World program where we used dynamic data dependences and we discuss our findings.

4.1. Program discovery

The first context where our visualizations proved to be useful is program discovery, that is the task a programmer faces when s/he has to get aquainted with a program s/he didn't implement her/himself. Two of the authors were in this situation with the Blocks World program and had to work hard to understand the program. Even if interacting with the robot, on the Lisp terminal, was easy to grasp, trying to understand how the program works in order to handle object creation, placement and moving was another question!

The first view we used for this is the data dependence graph which focused on the called function. Fig. 5 shows this view for the call (creObjet 'a 'boite 'taille '(2 2 3)) that asks for the creation of an object, named 'a', that is a box – *boite* in french – of size 2x2x3. The view shows the input/output of the call, highly uninformative, since the result of the call is just printing out 'c'est fait' (or 'done') and that doesn't say anything about how the program did this. However, this view also shows the global variables (filled in dark gray) that were used and/or modified by the call, information not easily accessible in the interpreter itself. Here we see that the table before the call was empty[3], as was the object list (variables on the top), while after the call, it has been filled with 'a' that also appears in the object list and has properties (variables below the call). With this view, we could discover the real effect of the call.

[3]nil stands for empty in Lisp.

To better understand how the program functions, we used the user-call-graph, as it gives a first global overview of the actions performed by the program. Of course, this relies on the fact that our program is well decomposed into well named functions: looking at the user-call-graph given in Fig. 4, one can easily grasp that creating an object means first to define the object (definieObject) – this function is recursively called once –, then to find a place where to put it (trouvePlaceSur) and then to actually put this object at this place (metSur); this last action is again decomposed into three steps, namely grasping the object (enleveObjet), closing it whenever it is a box (ferme) and putting it down on the table (deposer).

We then got back to the data dependence graph to get more information about how these different steps affect the global variables. We filtered out everything but user functions and built a first level graph focused on the call to creObjet.

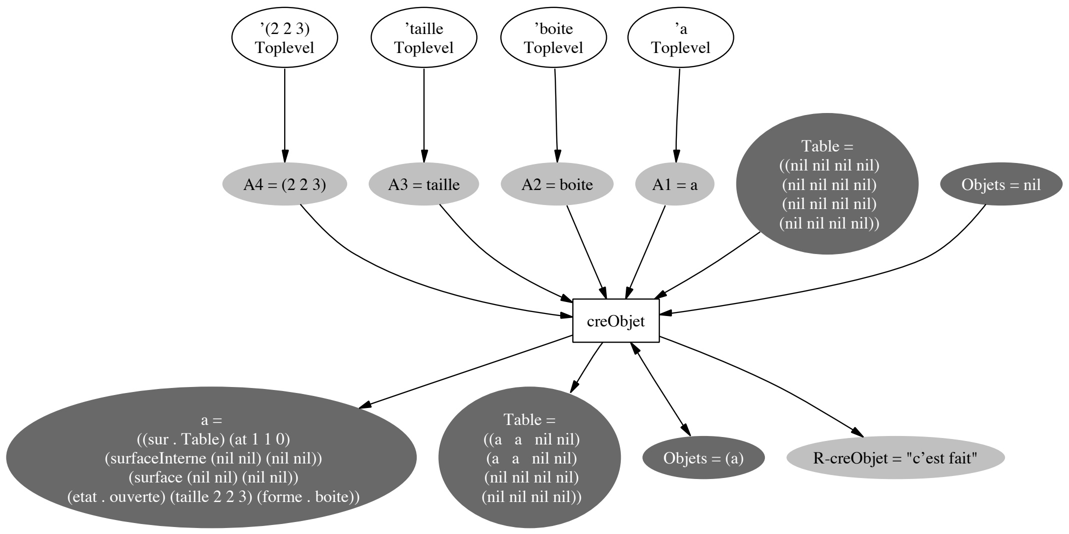
In the case of an object creation, we could verify that the performed actions are always the same. In some other cases, on the contrary, different calls to the same function resulted in really different sets of actions; this was immediately visible in the user call graph and pointed us towards other possible 'traversals' of the program we had to analyze.

4.2. Finding bugs

While working on the data dependence graphs we mentioned in the previous section, we examined in detail how finding a place where to put a object was done. Here, to put an initial 2 by 2 box on an empty table, the program checked whether positions 1-1, 1-2 and 2-1 were free and decided that this was a good place where to put the box. Check of 2-2 was not performed and this didn't produce any error since the table was empty, but of course this also showed a buggy behavior that caused errors in other cases where many objects were already on the table. Incorrectly nested loops were responsible for this error that could be quickly corrected.

Detecting this bug would have been very difficult looking only at the input/output of the program on the Lisp terminal, even when using the built-in inspecting features, while it was straightforward with our data dependence graphs. Fig. 6 shows the corresponding graph, where only three calls, instead of four, to quoiA? are performed, and the argument values indicate which positions were checked.

In other cases, we noticed unexpected behaviors of the program and used different possible views to find why it was behaving this way. Our main strategy was to examine the values of the global variables, in iteratively refined



'(2 2 3) Toplevel
'taille Toplevel
'boite Toplevel
'a Toplevel
A4 = (2 2 3)
A3 = taille
A2 = boite
A1 = a
Table = ((nil nil nil nil) (nil nil nil nil) (nil nil nil nil) (nil nil nil nil))
Objets = nil
creObjet
a = ((sur . Table) (at 1 1 0) (surfaceInterne (nil nil) (nil nil)) (surface (nil nil) (nil nil)) (etat . ouverte) (taille 2 2 3) (forme . boite))
Table = ((a a nil nil) (a a nil nil) (nil nil nil nil) (nil nil nil nil))
Objets = (a)
R-creObjet = "c'est fait"

Figure 5. Overview of computation performed

views on the different actions performed by the program, to point to a function call working incorrectly. Then we navigated backward and forward to see whether this call was receiving a bad argument – in this case, we again refined views to see what happend *before* this call – or whether it was effectively performing incorrectly. As soon as we had detected the buggy function, we could analyze in more depth what was performed during its call to find the problem. For example, in one such case, where an object had to be moved but was actually not moved, we could detect that a generic sorting function was called with an incorrect function pointer as argument, resulting in not sorting at all. Simply modifiying the call solved the problem!

4.3. Correctness checking

As an extension of the two former points, we also used our views to verify that the program was behaving properly. For instance, after correction of the bug in the 'finding a place' action, we built several views of calls where this actions was performed and carefully verified that it was, now, correctly implemented.

We also used our views to verify that the program was behaving the way we expected it to do. Remind that it is an AI program, relying on the key concept that most general problems can be recursively solved through a divide and conquer method. That's why, in many contexts, large parts of the program are reused and reused again, resulting in deep and broad call trees, extremely difficult to capture.

For example, the user instruction pose-sur, that is put-on in english, intended to let the robot move objects in the world, is reused whenever objects are on the object to move – the robot must first *put* these objects *on* the table –, reused (again) whenever the necessary place on, say, the table is not available – the robot must first *put* other objects *on* another place, and then finally it can *put* the initial object *on* some place on the table. This way, a single call to function pose-sur may result in it being recursively called several times, each one driving calls to a huge number of other functions each one possibly including non-local returns.

In order to check that this process was correctly implemented, we looked both at user call graphs to check whether function pose-sur was recursively called the correct number of times and at a data dependence graph where we rendered visible only calls to function pose-sur. With such a view, we were able to examine the values of the global variables at the different steps of the program execution in order to verify that they were modified the way we expected.

5. Discussion

From our experience working with the Blocks World program, as well as several other small to medium sized Lisp programs, we can affirm that the major benefit given by the dynamic dependences our tool handles is that precise information about a program execution is recorded: details about how execution was driven from one expression to another, as well as about which values variables had at any point of the program and how these values are transmitted from point to point.

The different visualizations we propose were designed to minimize the conceptual overload in order to allow users to find the exact information they need, otherwise barely accessible in the database. Different variants of call graphs

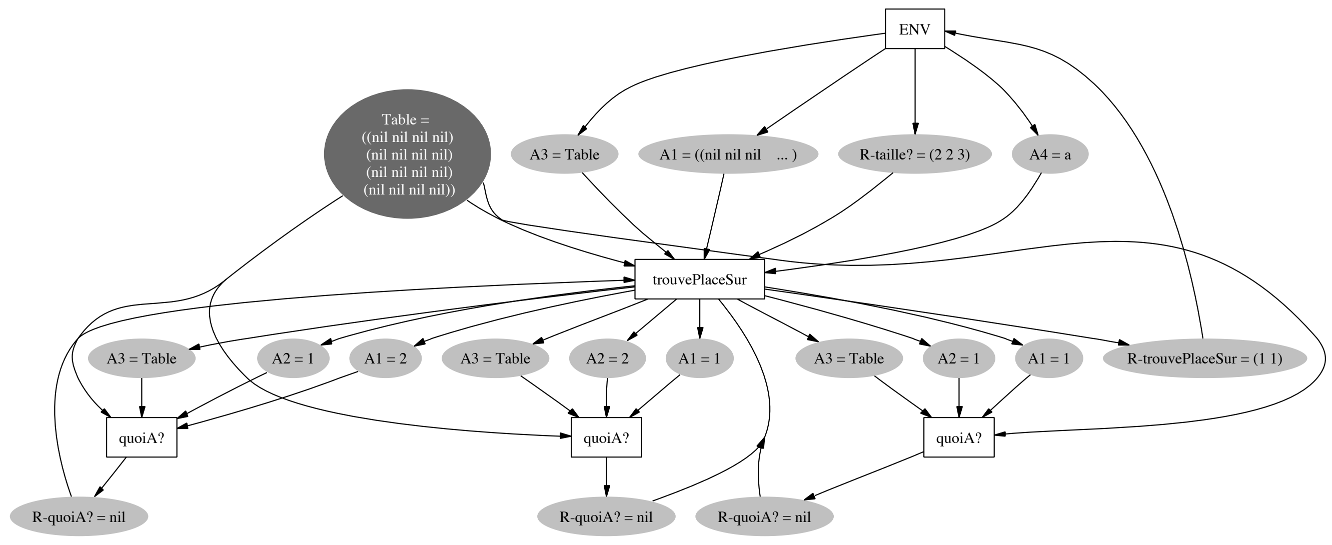


Figure 6. A buggy search for place

respond to questions about the control of the program, and data dependence graphs about the data flow. Clearly, this dynamic information is of great help when working on problems like debugging, verifying that a program works properly, or even optimizing, since it gives information only for *one* given execution, when static dependences would give too much information.

On the other hand, the weakness of this approach is that it requires enough knowledge from the user on the possible paths in the programs: verifying that a program behaves properly means checking *many* possible executions, and the user has to find which ones are necessary. However, our approach also makes possible to discover some unforseen execution paths, sometimes impossible to detect through static analysis. Combining static information with dynamic dependences is a possible extension we plan to investigate.

The second problem we encountered with our tool is that even if the set of dependences is restricted to one execution of interest, it's still sometimes hard to find the right information: either too many nodes and groups are displayed at the same time, or too much navigation is required in the graphs before one finds the place to examine more in depth. For this, we plan to enhance our filtering mechanism with the ability to filter out global variables, since they are not all of interest at the same time, and to implement a query language that will permit to find, thus to jump to, parts of the execution corresponding to given criteria.

Besides enhancements of our visualizations we just mentioned, our main perspective is now to development further a similar tool for the C language [4], where we will be able to integrate it with a debugger. This way, the user not only will examine the dynamic dependence graph *after* the execution of the program, but s/he will have the possibility to execute the program step by step, or from breakpoint to breakpoint, while looking at the corresponding graph. We expect this functionality to greatly enhance maintenance of long to execute and hard to understand programs.

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, Toulouse (France), 1999.

[2] F. Balmas. Using dependence graphs as a support to document programs. In *Proceedings of the Workshop on Source Code Analysis and Manipulation*, Montreal, Canada, 2002.

[3] F. Balmas. Displaying dependence graphs: a hierarchical approach. *Journal on Software Maintenance and Evolution: Research and Practice*, 16(3):151 – 185, May/June 2004.

[4] R. Chaabane. Analyse Dynamique de Programmes C. Mémoire de DEA, Université Paris 8, Saint-Denis, France, 2005.

[5] E. Koutsofios and S. North. *Drawing graphs with* dot. AT&T Labs – Research, Murray Hill, NJ, March 1999.

[6] T. Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.

Dynamic Estimation of Data-Level Parallelism in Nested Loop Structures: A Preliminary Report

Lewis B. Baumstark, Jr.
Department of Computer Science
University of West Georgia
Carrollton, GA
lewisb@westga.edu

Linda M. Wills
Department of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA
linda.wills@ece.gatech.edu

Abstract

Retargeting sequential code to data parallel execution is difficult, but can provide significant increases in efficiency. Since data parallel execution depends on performing highly regular operations, typically on a multi-dimensional data set, retargeting requires understanding the regular data access patterns within an application and the homogeneous operations performed across the multidimensional space. These are often obscured by their implementation in sequential code. This paper describes a dynamic approach to understanding data dependences in multi-dimensional iteration spaces in order to estimate the amount of data-level parallelism that could be exploited in sequential program loops. For tractability, it uses a dynamic technique to derive an estimate based on a "representative corner" of the iteration space. The technique is implemented in a prototype tool, called DLPEST3, which estimates data parallelism in sequential loops, regardless of the depth of nesting, i.e., it is capable of measuring along all axes in the iteration space.

1. Introduction

Few automatic tools exist for reverse engineering and retargeting sequential code assets to data parallel execution mechanisms. This often manual retargeting process is time consuming and expensive. Potential performance improvements are difficult to estimate until lengthy, often undocumented programs are reverse engineered.

We have previously explored automated techniques for retargeting sequential code to data parallel execution mechanisms [1] . These techniques are applied selectively to programs or code blocks with high potential for data-level parallelism (DLP). In order to facilitate the overall retargeting process, low cost techniques are needed to identify portions of code that have a high potential for data parallelism. Such techniques will not only aid the reverse engineering process, but could also be used to focus more traditional vectorization-related analyses, such as Fourier-Motzkin elimination or the Omega test [2].

Data parallel execution depends on uniform data access. The key to retargeting is to understand the regular data access patterns that can be extracted from sequential code and exploited in a data parallel execution environment. This paper presents a dynamic approach to understanding data dependences in a multi-dimensional iteration space in order to estimate the amount of DLP that could be exploited in sequential program loops. For tractability, it uses a dynamic technique to derive an estimate on a "representative corner" of the iteration space. The technique is implemented in a prototype tool, called DLPEST3 (DLP ESTimator, 3rd generation). A significant feature is that it can do this for loop nests of arbitrary depth, i.e., it is capable of measuring DLP along all axes in the iteration space. The resulting information can guide a retargeting tool and/or human developer in selecting the best parallelization strategy with respect to, for example, specific nesting levels to parallelize, number and dimensionality of processor arrays, and data distribution.

2. Related Work

Larus [3] presents a system targeted at recognizing loop-level parallelism. Using an idealized parallel machine model (no limit on processors, etc.) it seeks to find an upper bound on the available program parallelism. The system scans through an externally generated program trace. When it identifies a loop, it tracks register- and memory-based data dependencies for the loop. The parallelism estimate is computed from the number of loop iterations without loop-carried dependencies. The system also offers the option of ignoring loop-carried data-dependencies, which provide insight into the amount of parallelism available if a compiler can recognize "false" loop-carried dependencies and remove them. We incorporate their tracking of loop-carried dependencies, but at the source-code level instead of the assembly-level, allowing better mapping of dependences to their location in code.

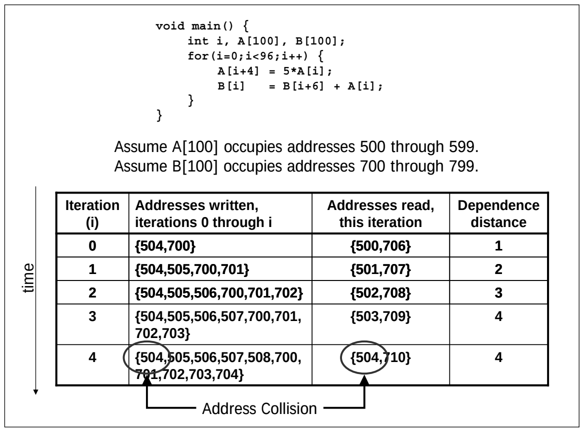
Kumar [4] offers a source-code instrumentation scheme for Fortran programs. The added code allows the program to self-compute the earliest time slot for each source code statement that would complete execution on a parallel machine with unlimited resources, based on satisfying data dependence constraints (i.e., the statement's variable and/or memory reads) and control constraints (e.g., a statement inside an if-then body cannot execute before the conditional is resolved). Under this model, potentially many statements can fall into the same time slot, yielding a parallelism metric. We perform source-code instrumentation as well, but focus on calculating the dependence distance which requires only counting the number of iterations for which a dependency does not exist. This avoids the complexity of tracking every memory access and when its values would be ready for consumption in an ideal machine.

Wills, et al. [5] developed a technique to estimate measures of three different types of parallelism – thread-level, instruction-level, and data-level – using a modified version of the SimpleScalar simulator [6]. After executing each machine instruction, the modified simulator places the instruction into a schedule grid (where rows are time slots and columns are instructions scheduled in parallel into the time slots) based on its dynamic data-dependency constraints. This schedule modeled a processor with theoretically unlimited (but practically limited) hardware resources (functional units, registers, memory, etc.). Each instruction placed into the schedule was scheduled as early as possible, i.e., one time slot after the producers of its operand registers were scheduled (all instructions were assumed to have a latency of a single time slot). Value prediction and perfect branch prediction were used to measure the maximum theoretical DLP possible. A parallelism metric was formed by dividing the total number of instructions exhibiting parallelism (i.e., those that occur more than once in a given time slot) by the total number of instructions in the program. While this technique provided validated parallelism measures, it did not provide some information of interest. It measured parallelism for the entire program and thus was unable to map the results back to certain sections of code. Similarly, it was unable to report on parallelism on a per-loop-nest basis. It was also a very time-consuming process; we are interested in lightweight techniques that can be applied to an agile development process and/or used within a profiling compiler.

3. Proposed Technique

DLPEST3 estimates the minimum dependence distance for each nesting level of a loop. Dependence distance [7] is the number of loop iterations between the source of a memory-based dependence and its target. It is also a useful measure of DLP, as the minimum distance of all dependence pairs represents the number of iterations of a loop that can be executed in parallel.

In Baumstark [8], dependence distance was dynamically measured for the simplest case: an inner loop. (Here, an inner loop is defined as one that contains no other loops, even transitively via a function call.) Fig. 1 illustrates this technique. This approach compares addresses read in the current iteration with those written in all previous iterations of the loop. Addresses are recorded for data types reported as arrays or pointers by the original C code. If the intersection of the read set and the write set is empty, no dependence has been found between the current iteration and all past iterations, so the measured dependence distance is incremented and the next iteration is considered. Otherwise, the last dependence distance is returned. Note that if multiple loop-carried dependencies exist, this technique will detect the one with the shortest depend-



```
void main() {
    int i, A[100], B[100];
    for(i=0;i<96;i++) {
        A[i+4] = 5*A[i];
        B[i]   = B[i+6] + A[i];
    }
}
```

Assume A[100] occupies addresses 500 through 599.
Assume B[100] occupies addresses 700 through 799.

Iteration (i)	Addresses written, iterations 0 through i	Addresses read, this iteration	Dependence distance
0	{504,700}	{500,706}	1
1	{504,505,700,701}	{501,707}	2
2	{504,505,506,700,701,702}	{502,708}	3
3	{504,505,506,507,700,701, 702,703}	{503,709}	4
4	{504,505,506,507,508,700, 701,702,703,704}	{504,710}	4

Fig. 1. Original single-loop DLP estimation technique.

ence-distance, as this is the dependence constraining parallelization.

This approach works well for estimation of inner loops. The amount of memory and number of comparisons required is proportional to $I \times W$ where I is the number of loop iterations and W (a constant) is the number of static write operations. In practice, we limited I to a threshold T_L to improve the time performance (the tool was programmed to cease measurement once a dependence was found), reasoning that real hardware would place practical limits on the amount of parallelism that could be exploited. This threshold, then, could be set based on available hardware configurations.

Such an approach does not scale, however, to measuring DLP in non-inner loops. Since this approach records all past memory addresses written and compares them with current memory reads, memory write addresses must be recorded for the entire multi-dimensional iteration space. This increases the memory requirements and the number of comparisons to

$$W \times \prod_N I_N \qquad (1)$$

where I_N refers to the number of iterations at loop nesting level N. Clearly, such an approach would quickly fall short of our goal of a lightweight estimation technique.

In our previous one-dimensional approach, dependence distance measurement ceases when a certain threshold is reached. Part of the reasoning for using the threshold was that after “a long time,” i.e., after a sufficiently large dependence distance had been measured, we could assume the loop was fully parallelizable or, at the least, far more parallelizable than practical hardware could exploit. Our goal was only to estimate DLP, not measure it precisely.

The question now is can such a threshold be reasonably employed in a multi-dimensional iteration space? The key to keeping the amount of memory and number of computations tractable is to build the estimate on a “representative corner” of the iteration space and ignore all other iterations. Fig. 2 illustrates. In Fig. 2 (a), the representative corner of a one-dimensional iteration space (a single loop with no enclosing loop) is simply the first T_L iterations of that loop. In Fig. 2 (b), the corner of a two-dimensional iteration space is the rectangular region covering the first T_L iterations along each axis. Similarly, for an N-dimensional space, the corner is a small N-cube covering the first T_L iterations along each axis. By using a threshold, we constrain the I_N-terms of Eq. (1).

To understand why this is acceptable, consider multimedia instruction set extensions, such as Intel's SSE [9] or the Complex Streaming Instructions project [10], which are capable of exploiting DLP. These typically operate on array-based data with constant memory strides between elements. The analogous sequential loops operate on array-based data where the index of iteration (proportional to the data stride) increases linearly. Thus, within this representative corner of the iteration space, if we detect iterations that can be executed in parallel, those patterns can be assumed to extend to the entire iteration space. For example, if we found a dependence distance of T_L (i.e., no dependence detected within the threshold number of iterations), we can assume, as with the earlier technique, this dimension of the loop nest can be fully parallelized. Similarly, if we were to find a minimum dependence distance of eight by the time T_L iterations occurred, we assume, based on the linearity described above, that this holds for the entire dimension (and not that an irregular pattern of dependence distances would occur).

4. Preliminary Results

The proposed technique is being prototyped in DLPEST3. We have been able to run some simple tests on its current functionality as proof-of-concept.

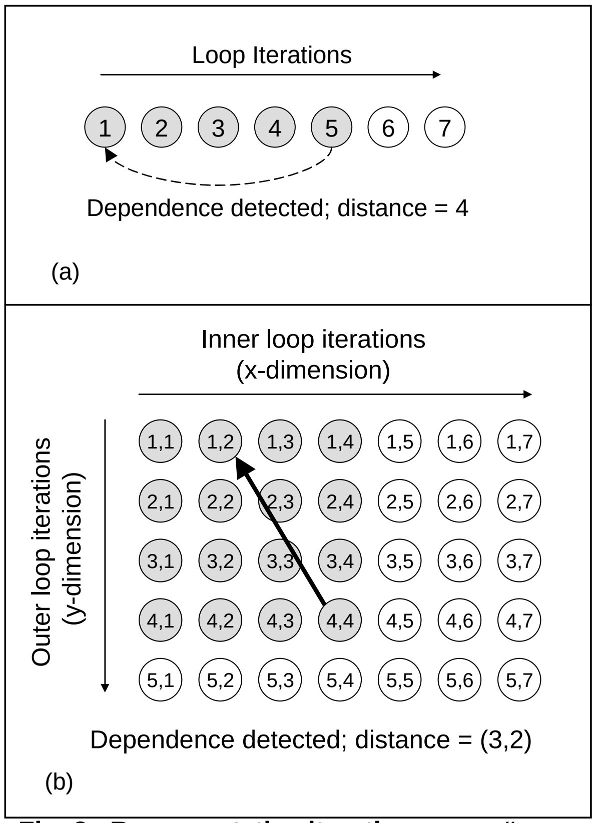


Fig. 2. Representative iteration space “corners” for (a) one-deep loop nest and (b) two-deep loop nest.

One test is shown in Fig. 3. This code contains two synthesized loop nests, a two-deep nest and three-deep nest, both with some non-trivial dependence distances. A second example comes from the Mediabench suite [11], which contains several programs covering a wide range of multimedia applications. We test DLPEST3 on Mediabench's ADPCM program, which encodes and decodes waveform data into a quantized digital format.

The results from the two test programs are summarized in Table 1. The expected results were gained by a hand examination of the code. The preliminary results are encouraging, matching well the expected estimates.

Table 1
Measured vs. Expected Results

Loop nest	Expected dependence distance	Measured dependence distance
Simple 2D nest	(5,8)	(5,8)
Simple 3D nest	(4,-14,-14)	(4,-14,-14)
ADPCM (encode)	(1,0)	(1,0)
ADPCM (decode)	(1,0)	(1,0)

5. Discussion

Beyond the core DLP-estimation functionality of DLPEST3, other useful information could be recorded. For example, many data-parallel hardware architectures are limited in the stride they allow between elements. DLPEST3 could be extended to report on the stride between iterations at any depth of nesting, providing more information for a compiler or developer wishing to parallelize an application.

6. References

[1] Lewis Baumstark, Jr., and Linda M. Wills, "Retargeting Sequential Image-Processing Programs for Data-Parallel Execution," *IEEE Trans. on Software Engineering*, Vol. 31, No. 2, pp. 116-136, Feb. 2005.

[2] Randy Allen and Ken Kennedy, *Optimizing Compilers for Modern Architectures*, San Francisco:Morgan Kaufmann Publishers, pp. 35-121, 2002.

[3] J. R. Larus, "Loop-Level Parallelism in Numerica and Symbolic Programs," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 7, pp. 812-826, July 1993.

[4] A. M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Trans. on Computers*, Vol. 37, No. 9, pp. 1088-1098, September 1988.

[5] L. Wills, T. Taha, L. Baumstark, and S. Wills, "Estimating Potential Parallelism for Platform Retargeting," In *Proc. of the 9th Working Conference on Reverse Engineering (WCRE '02)*, Richmond, VA, pp. 55-64, October 2002.

[6] Doug Burger and Todd Austin, "The SimpleScalar Tool Set, Version 2.0," Tech. Report TR #1342, Univ. of Wisconsin-Madison Computer Sciences Dept., Madison, WI, June 1997.

[7] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua, "Automatic Program Parallelization," *Proc. of the IEEE*, vol. 81, no. 2, pp. 211-243, 1993.

[8] Lewis Baumstark, "Extracting data-level parallelism from sequential programs for SIMD execution," doctoral dissertation, Georgia Institute of Technology, 2004, UMI Catalog No. AAT 3154911.

[9] Srinivas K. Raman, Vladimir Pentkovski, Jagannath Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro*, Vol. 20, No. 4, pp. 47-57, July/August 2000.

[10] Ben Juurlink, Dmitri Tcheressiz, and Stamatis Vassiliadis, "Implementation and Evaluation of the Complex Streamed Instruction Set," In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques* (PACT '01), Barcelona, Spain, pp. 73-82, September 2001.

[11] Chunho Lee, Miodrag Potkonjak, William H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *Proc. of the 30th annual ACM/IEEE Int'l Symposium on Microarchitecture*, Research Triangle Park, NC, pp. 330-335, 1997.

```
int x[50][50], y[50][50], z[50][50];
char* a, *b, *c;
int main() {
  int i, j, k;
  a=(char*)malloc(50*50*50);
  b=(char*)malloc(50*50*50);
  c=(char*)malloc(50*50*50);

  /* 2-deep loop nest */
  for( i=0; i<50; i++ ) {
     for( j=0; j<50; j++ ) {
        z[i][j] = z[i-5][j-8]
                + x[i][j] + y[j][i];
     }
  }

  /* 3-deep loop nest */
  for(i=0;i<15;i++) {
     for(j=0;j<15;j++) {
        for(k=0; k<15; k++) {
           *(a+i) += *(a+i-4) + *(b+50*j+k)
                   + *(c+j+50*k);
        }
     }
  }
  return 0;
}
```

Fig. 3 Test code for DLPEST3

Selective Tracing for Dynamic Analyses*

Matthias Meyer, Lothar Wendehals
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
[mm|lowende]@uni-paderborn.de

Abstract

Reverse engineering based on dynamic analyses often uses method traces of the program under analysis. Recording all method traces during a program's execution produces too much data, though for most analyses, a "slice" of all method traces is sufficient.

In this paper, we present an approach to collect runtime information by selectively recording method calls during a program's execution. Only relevant classes and methods are monitored to reduce the amount of information. We developed the JAVATRACER *which we use for the recording of method calls in Java programs.*

1. Introduction

In the last years, we developed a tool-supported semiautomatic approach to design recovery [5]. Our approach facilitates the recognition of design pattern [3] instances in the source code of a system. We recently extended this approach by combining the existing static analysis with a dynamic analysis [7]. The static analysis identifies pattern instance candidates based on their structural properties. The subsequent dynamic analysis confirms or rejects the candidates by checking their behavior.

The behavior of a design pattern is specified by UML 2.0 sequence diagrams [8]. In our approach, these specifications are called behavioral patterns. Behavioral patterns describe typical sequences of method calls between objects of classes that participate in a design pattern instance. To check the conformance of a given design pattern instance to the behavioral pattern, method traces have to be gathered during the execution of the program under analysis.

Recording all method traces during a program's execution not only produces too much information, but also reduces the runtime performance of the program significantly. Consequently, the tracing should be restricted to those method calls that are really needed in the dynamic analysis. In our approach, only specific methods of pattern instance candidates have to be monitored, which means only to record a "slice" of method calls of the whole program.

For this purpose, we developed a selective tracer which takes a list of classes and methods to be monitored as input. The tracer executes the program to be analyzed and records only calls to the given methods. The gathered information is saved to a file which can be used by post-mortem analyses.

*This work is part of the FINITE project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-2.

In the next section we present the application scenario for our selective tracer in more detail by means of a concrete example. We will refer to this example throughout the rest of the paper. In Section 3 we report about related work. Our approach to selective tracing is described in detail in Section 4 whereas its good performance is shown in Section 5. The paper is concluded with future work in Section 6.

2. Application Scenario

In a case study of our design recovery approach, we analyzed the ECLIPSE platform [2]. Among others, our static analysis identified several candidates of the *Strategy* design pattern in the source code.

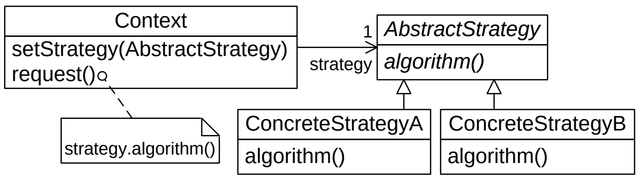


Figure 1: The *Strategy* Design Pattern

A *Strategy* design pattern (Figure 1) lets an algorithm vary independently from the client that uses it. An abstract class defines the algorithm interface, which is implemented by different concrete classes (the strategies). A context class references a strategy and delegates requests received from its clients to the strategy. Usually, the clients configure a context object with the appropriate concrete strategy.

Classes	Methods
org.eclipse.swt.widgets.Composite	setLayout WM_SIZE
org.eclipse.swt.widgets.Layout	*layout*
org.eclipse.jface.viewers.StructuredViewer	addFilter filter getSortedChildren setSorter
org.eclipse.jface.viewers.ViewerSorter	sort
org.eclipse.jface.viewers.ViewerFilter	select

Table 1: Classes and Methods Identified as Parts of Pattern Candidates.

Table 1 shows the classes and methods[1] that have been identified as parts of three *Strategy* pattern candidates. The first candidate consists of the classes Composite and Layout (cf. Table 1) which were recognized as context and abstract

[1]Abstract classes and methods are written in italic.

strategy, respectively. The method setLayout was identified as the method to configure the context with a strategy and WM_SIZE is called by clients to place a request. The method layout of class Layout was recognized as the method implementing the actual algorithm. The other classes and methods listed in the table belong to other candidates.

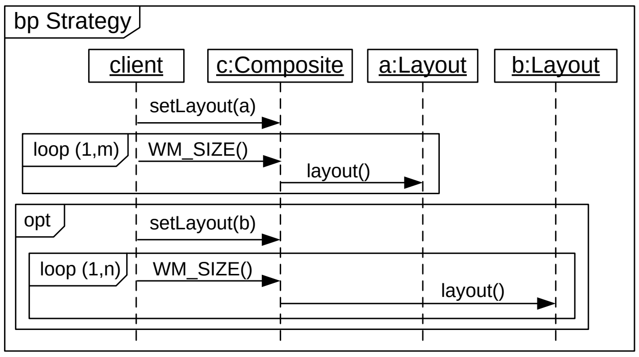


Figure 2: Behavioral Pattern for a Concrete *Strategy* Candidate.

The dynamic analysis now has to check whether the interaction of instances of the candidate's classes conforms to the behavioral pattern of a *Strategy* design pattern, i.e. the identified methods are called in the specified sequence.

Figure 2 shows the behavioral pattern of *Strategy* in which the methods and object types have been replaced by the classes and methods of the first pattern candidate. The behavioral pattern requires that a context object c of type Composite is configured with a strategy object a of type Layout by calling setLayout. Afterwards, a client has to place at least one request which has to be delegated to the strategy, i.e. WM_SIZE and layout have to be called consecutively an arbitrary number of times (indicated by the loop fragment)[2]. Furthermore, after several requests have been handled, the concrete strategy may be changed by another call to setLayout with a different b:Layout object. After that, requests on the context c:Composite have to be delegated to the new strategy object by calling layout on b:Layout. However, the change of the strategy is not required and is thus enclosed by an optional fragment.

In order to check if the pattern candidate behaves as specified by the concrete behavioral pattern shown in Figure 2, we need to record method call traces at runtime. However, a behavioral pattern does not define a complete trace. Only significant method calls are specified. Other calls of methods that are not mentioned in the behavioral pattern may interleave the given sequence. Consequently, we do not need to record a complete program trace but only calls to those methods explicitly mentioned by the pattern.

Furthermore, since some of the classes and methods identified in the source code are abstract, e.g. Layout and its layout method, they cannot be monitored directly during runtime. Instead, classes and methods that implement the abstract classes and methods must be monitored. Due to polymorphism and dynamic method binding, the same holds for methods which override methods to be monitored. The concrete classes and methods could be determined by static analysis easily. In our approach, however, this is done by our selective tracer as well.

3. Related Work

The Java Debug Interface (JDI) [6] offers debuggers a native technique to receive *MethodEntry-* and *MethodExitEvents*. The debugger has to provide a filter which specifies the classes to be monitored. This approach can not be used to monitor specific methods. Instead, all methods of classes given in the filter are monitored during the execution of the program under analysis. For each method call, *MethodEntry-* and *MethodExitEvents* are sent to the debugger. This technique is not practicable, since it slows down the analyzed program significantly (cf. Section 5).

The Omniscient Debugger [4] records method calls and variable state changes of Java programs. It instruments the source code on the byte code level, i.e. additional code is inserted into the original source code of the program to be analyzed. The code is used to inform the debugger about method calls. The instrumentation is also done in a non-selective way. The author reports about 100MB/sec of data produced during the execution.

The Instrumentation, Execution, and Coverage Tool INSECT [1] allows for collecting different kinds of dynamic information including method traces by instrumenting and executing the program under analysis. Instrumentation tasks are used to specify which entities of the program are to be instrumented and which kind of information is to be collected. Monitors can be implemented to process the collected information. In [1] it is shown that INSECT is efficient.

However, a problem of instrumentation is that it strongly depends on the programming language and the runtime environment used. This approach is difficult to transfer to other languages, especially those that do not use intermediate code such as C or C++. Instrumentation may also affect the synchronization of concurrent threads, since instrumented code directly influences the runtime of threads. This may cause for example time outs in the synchronization, thus resulting in a completely different behavior of the analyzed program.

4. Selective Tracing

We developed the JAVATRACER [9] for selective tracing of Java programs. As input, it gets a list of classes and interfaces as well as methods that have to be monitored during the execution of the program under analysis. The JAVATRACER acts as a debugger and executes the program, called the debuggee. JDI is used for connecting to the debuggee's virtual machine.

The principle idea of selective tracing is rather simple. The JAVATRACER is informed by the virtual machine each time a class is loaded. If this class belongs to the classes in the input, it adds a breakpoint at the beginning and the end of the body[3] of each method given in the input, indicating when a method is called and when it returns.

Abstract methods declared by interfaces or abstract classes can also be monitored, even though they don't have a method body. The JAVATRACER determines each time a

[2]Since no methods are called on the client object, its class needs not to be determined and can be ignored during analysis.

[3]The Java VM creates a virtual code line at the end of each method body that will be passed regardless of the actual executed return statement.

class is loaded if it is a sub class of the classes given as input. If the loaded class is a sub class, it adds breakpoints to methods which implement or override one of the given methods, thus supporting analyses that include polymorphism and dynamic method binding.

The advantage of this simple idea is that the approach is not bound to Java even though the JavaTracer is implemented for Java programs only. Breakpoints are a common feature of debuggers for nearly all languages. The JavaTracer just needs another implementation for the interface that is used to set breakpoints and receive breakpoint events to adapt to another debugger.

The JavaTracer will be informed when a breakpoint is reached during the program's execution. It then halts the debuggee. This guarantees that all threads of the program are halted, not only the thread that is currently running. Thus, concurrent threads depending on the current thread are not affected by halting just the current thread, since they are halted, too.

In the case of a breakpoint event at the beginning of a method call, the JavaTracer asks the debuggee's virtual machine for additional information about the method call. This includes information about the method name, the time stamp for the method call, the names and unique identifiers of the caller and callee objects, the identifiers and values of objects passed as arguments as well as the current thread. Then the debuggee's execution is continued. This information is recorded as a method entry event. Breakpoint events at the end of a method call are recorded as method exit events. Events about loaded classes are recorded as well.

The debuggee is controlled either manually by the reengineer or by automated tests. The output consists of a list of class loading events as well as method entry and method exit events in the order of their occurrence. The output can then be further analyzed, e.g. by our dynamic analysis of design pattern behavior.

Input for Tracing

The JavaTracer is started with a trace definition document describing the classes and methods that have to be monitored during the program's execution. Figure 3 shows an excerpt of this document using the example of Table 1.

```
<TraceDefinition>
  <ConsiderTrace>
    <Class name="org.eclipse.swt.widgets.Composite">
      <Method name="setLayout"/>
      <Method name="WM_SIZE">
        <Parameter type="int"/>
        <Parameter type="int"/>
      </Method>
    </Class>
    <Class name="org.eclipse.swt.widgets.Layout">
      <Method name="layout">
        <Parameter
          type="org.eclipse.swt.widgets.Composite"/>
        <Parameter type="boolean"/>
      </Method>
    </Class>
    ...
  </ConsiderTrace>
  <CriticalTrace>
  ...
  </CriticalTrace>
</TraceDefinition>
```

Figure 3: Example of the JavaTracer's Input

The trace definition has two sections. Within the ConsiderTrace section, classes are listed for which only selected methods are monitored. That means, only the given methods and overriding methods are considered in the tracing, calls of other methods are ignored.

The JavaTracer also provides a tracing on the class level, the so-called *critical* monitoring of classes. Using critical tracing, all methods of a class are monitored. This facilitates analyses where all method calls on objects of specific classes have to be recorded. These classes are specified within the CriticalTrace section of the input.

Output of Tracing

Figure 4 shows an excerpt of the JavaTracer's output. The output consists of a list of class loading events as well as method entry and exit events in the order of their occurrence.

```
<TraceResult>
  <ProcessStart name="main" time="1127705886787"/>

  <ClassLoaded name="org.eclipse.swt.widgets.Composite">
  </ClassLoaded>

  <ClassLoaded name="org.eclipse.swt.widgets.Shell">
    <SuperType name="org.eclipse.swt.widgets.
                     Composite"/>
  </ClassLoaded>

  <ClassLoaded name="org.eclipse.swt.widgets.Layout">
  </ClassLoaded>

  <ClassLoaded name="org.eclipse.swt.layout.GridLayout">
    <SuperType name="org.eclipse.swt.widgets.Layout"/>
  </ClassLoaded>
  ...
  <MethodEntry id="22" name="WM_SIZE" thread="main"
               time="1127705893547">
    <Caller id="1515"
            type="org.eclipse.swt.widgets.Shell"/>
    <Callee id="1515"
            type="org.eclipse.swt.widgets.Shell"/>
    <Argument value="0" type="int"/>
    <Argument value="3473906" type="int"/>
  </MethodEntry>

  <MethodEntry id="23" name="layout" thread="main"
               time="1127705893557">
    <Caller id="1515"
            type="org.eclipse.swt.widgets.Shell"/>
    <Callee id="1516"
            type="org.eclipse.swt.layout.GridLayout"/>
    <Argument id="1515"
              type="org.eclipse.swt.widgets.Composite"/>
    <Argument value="false" type="boolean"/>
  </MethodEntry>
  ...

  <MethodExit id="23" time="1127705893617"/>
  <MethodExit id="22" time="1127705893627"/>
  ...
  <ProcessEnd time="1127705926565"/>
</TraceResult>
```

Figure 4: Example of the JavaTracer's Output

The class loaded events comprise not only the class that was actually loaded, but also its super class, if the super class was given in the input. This information is needed in dynamic analysis to identify where polymorphism and dynamic method binding was used.

The two pairs of method entry and exit events describe two method calls. The first method call WM_SIZE (id 22) was called by an object of org.eclipse.swt.widgets.Shell on itself. The second method call with id 23 is nested in the first one which means that the method layout is called within the first method WM_SIZE on an object of type org.eclipse.swt.layout.GridLayout.

The output of the JavaTracer can be optimized for the analysis it is used for. Some information can be omitted such as the time stamps or even method exit events if information about method stack traces are not needed. Since tracing can produce huge amounts of information, it is vital to cut down the recording to a minimum.

The JavaTracer

Figure 5 depicts a screen shot of the JavaTracer Eclipse plug-in. We made this screen shot during the monitoring of Eclipse in the application scenario. On the right hand, the currently used trace definition document is displayed. In the upper left corner, the *Execution Monitor* view shows a tree of classes and methods that are monitored. For each method, the number of executions is given and an icon indicates if the method was executed at all. In the lower left corner, the *JavaTracer* view displays events occurred during the monitoring, whereas the *Console* view displays the output of the monitored program.

5. Performance

We measured the performance of our approach by comparing the startup times of Eclipse with and without tracing. Without tracing or instrumentation, it is very difficult to measure the startup time due to the lack of well-defined measuring points. Since we only want to make a qualitative statement of the performance, we decided to measure the time manually. The time was stopped when the CPU-load of the Eclipse process dropped to 0%. We run the scenarios ten times and calculated the average duration.

The performance was measured on a Pentium 4-M machine with 1.8 GHz and 1024 MB RAM. The system was running Windows XP Professional SP2 and Java 2 Standard Edition 5.0 Update 4. All other processes were stopped as far as possible. The workspace of the Eclipse platform consisted of one Java project, which was initially loaded during the startup of Eclipse.

Scenario	#c	#m	$\#act_c$	$\#act_m$	#mc
1	5	9	59	107	2945
2	8	13	204	336	12314

Table 2: Performance Measuring Scenarios

Table 2 shows two different scenarios. In the first scenario, we monitored the 5 classes (#c) and 9 methods (#m) given in the example. The actual number of monitored subjects were 59 classes ($\#act_c$) and 107 methods ($\#act_m$) due to implementations of abstract classes and methods as well as polymorphism. During the startup of Eclipse, there were 2945 method calls (#mc) of the 107 methods recorded.

The second scenario comprised 8 classes/interfaces and 13 methods to be monitored. All classes of the first scenario plus additional classes and interfaces that play a central role in the Eclipse environment are monitored. The additional classes are org.eclipse.core.runtime.Plugin, org.eclipse.core.runtime.IAdaptable and org.eclipse.core.runtime.IAdapterFactory. These classes and interfaces are extended or implemented by multiple other classes. This resulted in a scenario where 204 classes and 336 methods were actually monitored. We used this second scenario to show the scalability of our approach.

Scenario	$t_{w/o}$	t_{break}	t_{events}
1	16 sec.	41 sec.	36 min.
2	16 sec.	65 sec.	?

Table 3: Duration of Program Tracings

In Table 3, we present the average startup time for each scenario. First, the program was executed without any tracing ($t_{w/o}$). Then, the program was monitored using our breakpoint events (t_{break}) and at last (t_{events}) by using the native tracing technique offered by the Java Debug Interface (JDI) [6]. This technique is limited to monitor all methods of a class. To compare the native tracing of JDI to our approach, we recorded only entry and exit events of those methods given in the input.

The startup times without any tracing are of course equal for both scenarios. The performance results show that our approach to selectively trace method calls is feasible. Even though the number of monitored methods is three times higher than in the first scenario and the number of method calls is four times higher, the startup time rises by less than 60%.

In comparison to our approach, the event based approach offered by JDI is not practicable. We abandoned the performance analysis of the event based approach for the second scenario, since it took too much time.

Although the XML output format may seem too verbose, it has only a very slight influence on the performance of the JavaTracer. We analyzed the JavaTracer with a profiler discovering that more than 90% of the time spent in tracing is consumed by the JDI interface.

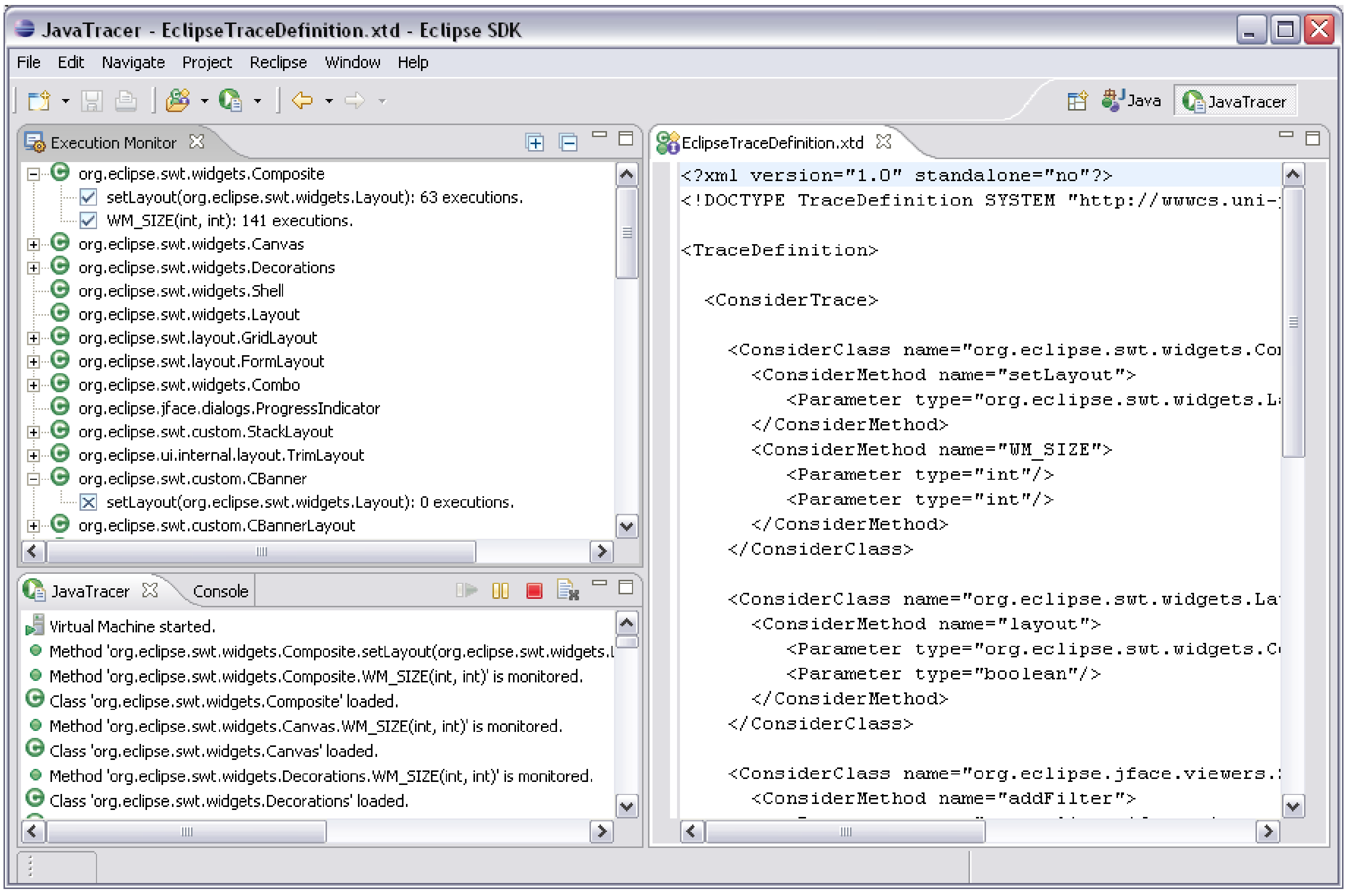
6. Future Work

We are planning to use our behavioral pattern analysis for conformance checking. When designing components, behavioral patterns can be used to describe protocols on how to use the interface of the component. In an ideal *Model Driven Development* process, the source code is completely generated from the model. In practice, a hybrid development process is often used, where parts of a system are generated and parts are implemented manually. During the implementation and testing of the components, our dynamic analysis can check if the actual behavior of the components conforms to the behavior defined by the behavioral patterns.

References

[1] A. Chawla and A. Orso. A Generic Instrumentation Framework for Collecting Dynamic Information. *SIGSOFT Software Engineering Notes, Section: Workshop on Empirical Research in Software Testing. ACM Press, New York, NY, USA*, 29(5):1–4, September 2004.

[2] Eclipse Foundation. *The Eclipse Platform. Online at http://www.eclipse.org.* Last visited: September 2005.


JavaTracer - EclipseTraceDefinition.xtd - Eclipse SDK
File Edit Navigate Project Reclipse Window Help
Java
JavaTracer
Execution Monitor
EclipseTraceDefinition.xtd
JavaTracer
Console

```
org.eclipse.swt.widgets.Composite
    setLayout(org.eclipse.swt.widgets.Layout): 63 executions.
    WM_SIZE(int, int): 141 executions.
org.eclipse.swt.widgets.Canvas
org.eclipse.swt.widgets.Decorations
org.eclipse.swt.widgets.Shell
org.eclipse.swt.widgets.Layout
org.eclipse.swt.layout.GridLayout
org.eclipse.swt.layout.FormLayout
org.eclipse.swt.widgets.Combo
org.eclipse.jface.dialogs.ProgressIndicator
org.eclipse.swt.custom.StackLayout
org.eclipse.ui.internal.layout.TrimLayout
org.eclipse.swt.custom.CBanner
    setLayout(org.eclipse.swt.widgets.Layout): 0 executions.
org.eclipse.swt.custom.CBannerLayout
```

```
Virtual Machine started.
Method 'org.eclipse.swt.widgets.Composite.setLayout(org.eclipse.swt.widgets.L
Method 'org.eclipse.swt.widgets.Composite.WM_SIZE(int, int)' is monitored.
Class 'org.eclipse.swt.widgets.Composite' loaded.
Method 'org.eclipse.swt.widgets.Canvas.WM_SIZE(int, int)' is monitored.
Class 'org.eclipse.swt.widgets.Canvas' loaded.
Method 'org.eclipse.swt.widgets.Decorations.WM_SIZE(int, int)' is monitored.
Class 'org.eclipse.swt.widgets.Decorations' loaded.
```

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE TraceDefinition SYSTEM "http://wwwcs.uni-

<TraceDefinition>

  <ConsiderTrace>

    <ConsiderClass name="org.eclipse.swt.widgets.Co
      <ConsiderMethod name="setLayout">
         <Parameter type="org.eclipse.swt.widgets.L
      </ConsiderMethod>
      <ConsiderMethod name="WM_SIZE">
         <Parameter type="int"/>
         <Parameter type="int"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="org.eclipse.swt.widgets.La
      <ConsiderMethod name="layout">
         <Parameter type="org.eclipse.swt.widgets.C
         <Parameter type="boolean"/>
      </ConsiderMethod>
    </ConsiderClass>

    <ConsiderClass name="org.eclipse.jface.viewers.
      <ConsiderMethod name="addFilter">
```

Figure 5: The JavaTracer implemented as an Eclipse Plug-In

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[4] B. Lewis. Recording Events to Analyze Programs. In *Object-Oriented Technology. ECOOP 2003 Workshop Reader*. Lecture notes on computer science (LNCS 3013), Springer, July 2003.

[5] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.

[6] Sun Microsystems. *Java Platform Debugger Architecture(JPDA). Online at http://java.sun.com/products/jpda/index.jsp*. Last visited: September 2005.

[7] L. Wendehals. Improving Design Pattern Instance Recognition by Dynamic Analysis. In J. Cook and M. Ernst, editors, *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, pages 29–32, May 2003.

[8] L. Wendehals. Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams. In E.-E. Doberkat and U. Kelter, editors, *Proc. of the 6th Workshop Software Reengineering (WSR), Bad Honnef, Germany, Softwaretechnik-Trends*, volume 24/2, pages 63–64, May 2004.

[9] L. Wendehals. Tool Demonstration: Selective Tracer for Java Programs. In *Proc. of the 12th Working Conference on Reverse Engineering, Pittsburgh, Pennsylvania, USA*, November 2005. to appear.

Dynamic Fan-in and Fan-out Metrics for Program Comprehension

Wang Yuying, Li Qingshan, Chen Ping, Ren Chunde
Software Engineering Institute, Xidian Univ.,Xi'an,710071,China
E-mail:xawyy@hotmail.com

Abstract

This paper presents ongoing work on using run-time information to discover knowledge about software systems thus facilitating program comprehension. Some dynamic metrics based on traces of the subject system execution are proposed. An approach to get these dynamic metrics is introduced, in which instrumentation implemented by using reflective mechanism based on an open compiler. The system run-time information is captured with the instrumented system running. From the information, we obtain the dynamic metrics. Some cases study is given to illustrate the use of these dynamic metrics, i.e. identifying critical components of the subject system. These critical components should be focus of user attentions in order to understand the subject system well.

Keywords: *program comprehension, dynamic metrics, critical components, functionality*

1. Introduction

A well documented problem faced by maintainers when understanding a software system is the lack of familiarity with it, combined with the lack of accurate documentation. Several techniques and methods have been proposed in order to facilitate this time consuming activity [1] [2] [3]. The work presented in this paper is part of a wider research effort investigating the applicability and suitability of using dynamic information to facilitate program comprehension. This effort aims at development a methodology for semi automated program comprehension using dynamic metrics. A fundamental underlying assumption is that the maintainer may have little or no knowledge of the examined program .The work presented here aims to help maintainers to recognize critical parts of the subject system and to infer the tasks of this system, i.e. facilitating program understanding. This work focuses on definitions of some dynamic metrics with traces of the system execution. After obtaining these metrics, we can concentrate on system components that have high metric values. Careful analyses on it result in system main functionality inferred.

The remaining sections of this paper are organized as follows. First, we introduce related works that could possibly appeal to be considered metrics. Section 3 summarizes the terminology and definitions used to express the proposed metrics. In Section 4 we give out an approach to get dynamic fan-in and fan-out metrics we proposed in detail. In Section 5, we use a case study of a highway application, as well as a simple Client/Server system, to demonstrate the use of dynamic metrics defined here. We conclude the paper and discuss possible future works finally.

2. Related work

2.1. Definitions fan-in and fan-out metrics

Fan-in and fan-out metrics are structural metrics which measure inter-module complexities.

The fan-in and fan-out metrics of modules were first defined by Henry and Kafura [4]. They defined the fan-in of a module as “the number of local flows that terminate at a module, plus the number of data structures from which information is retrieved”; and the fan-out as “the number of local flows that emanate from a module, plus the number of data structures that are updated by that module”.

For Objected Oriented Programs, modules can be considered in method-level and in class-level respectively. (Because an object is a class instance only existing in run-time, it is not necessary to consider fan-in and fan-out in object-level.)

In[5], author extended the original fan-in and fan-out metrics in class-level, he used the following definition for fan-in. Let C be a class and S the set of classes calling methods from C. Then $\mathrm{FI} = |S|$.

As the fan-out metric can be used synonymously with CBO (coupling between objects [6]), the author only use the latter term and use the following definition for CBO:

Let C be a class and $M = \{ m_1, m_2, ..., m_n \}$ the set of methods of C, R_i the set of methods called by m_i and A_i the set of attributes accessed by m_i. Let

$W(\ ) : Set(Feature) \rightarrow Set(Class)$

determine from a set of features (i.e. attributes or methods) the set of their owners (i.e. classes). Then

$$\mathrm{CBO}(C) = |\bigcup_{v=1}^{n} W(R_v) \cup \bigcup_{w=1}^{n} W(A_w)|$$

In [7], author defined fan-in and fan-out metrics in method-level. They defined the fan-in of a method *m* as the number of distinct methods that can invoke m, the fan-out of a method *m* as the number of distinct methods that can be invoked by *m*.

2.2. Usage of fan-in and fan-out metrics

In [4], Henry and Kafura used fan-in and fan-out metrics to define complexity of a module. The complexity of each module is then defined as:

Module length * (fan-in * fan-out) ^{2}

In [9], authors gave out a simplification of the original complexity of each module metric using fan-in and fan-out, they considered that it is better to measure interface complexity than measuring the complexity of the modules, something that can be achieved by excluding length from the Henry-Kafura formula. Therefore, the formula used to calculate IF(information f low) would be the following:

IF= (Fan-in * Fan-out) ^{2}

In class-lever, the fan-in metric can be used to find classes which services are used by many others, i.e. those that have a high reusability. Fan-out(i.e.CBO) measures the number of classes to which a class is coupled. High CBO value indicates large numbers of interconnections between classes[5].

In[7],authors described a technique to identify Aspects using fan-in metrics. Methods with higher fan-in values are candidate aspects in a number of open-source Java systems. Case studies demonstrate the high fan-in method is a key element of the aspect implementation, such as the output method for logging, tracing or debugging functionalities, and some design patterns with a crosscutting structure can lead to high fan-in values when they are given a central role in the project design. So fan-in metric can be used to identify above aspects in Aspect Oriented Program and find design patterns.

2.3. Limitations of static fan-in and fan-out metrics

Fan-in and fan-out metrics reflect structure dependency. They are defined based on a static analysis of source code and the ability of these metrics to accurately predict the actual amount of coupling between modules(or class, or method) is as yet unproven. As a static metric, they cannot capture all the dimensions of object-level coupling, as features of object-oriented programming such as polymorphism, dynamic binding and inheritance render them imprecise in evaluating the run-time behavior of an application. The behavior of a program is going to be a function of its operational environment as well as the complexity of the source code. Therefore static metrics may fall short when determining the run-time properties of a program.

For a example[7], fan-in metric is derived from parsing design models, the systems run-time information are not under consideration.

During the process of calculate the fan-in value, many times method *m* invoke method *n* only contribute one to the value of *n*’s fan-in metrics. This reveals the structural dependence.

Because of polymorphism, one method call can affect the fan-in of several other methods. It is difficult to determine which one is called from source code, so a trade-off approach is given out. A call to method *m* contributes to the fan-in of all methods refined by *m* as well as to all methods that are refining *m*. detailed description is presented in [8].

Static fan-in and fan-out metrics contribute less to the behavior of system during the process of program understanding.

We are convinced that the static (syntactical) situation of a software program reflects only inaccurately the situation of the dynamic behavior of the system, like actual number and type of procedure calls, size of the actual transferred information etc. Only dynamic characteristics present us a real picture about the coupling in software system [10].

Currently, to the best of our knowledge, most researches on OO metrics are on class-level, and these metrics are static. Many researches aim at evaluation of the quality of OO software as well as system complexity. Quality of software systems can be characterized by the presence of a certain number of external attributes like functionality, reliability, usability, efficiency, maintainability and portability [11].Using appropriate metrics and evaluation techniques, they give out a quantitatively description on software quality and complexity. How to use the result to understand a software? Less literature is available. In this paper, we propose an approach to understand a subject system use fan-in and fan-out metrics.

In following sections, we identify a set of new dynamic metrics and discuss their uses in reverse engineering.

3. Dynamic fan-in and fan-out metrics

The dynamic metrics are less frequently discussed in OO metrics literature as compared to static metrics.

In this section, we define four dynamic metrics based on dynamic behavior of applications. The dynamic behavior of a system is obtained from run-time information rather than inferred from design models.

Before the definition of fan-in and fan-out metrics, we give out some terminology.

m_i : is a method of a class

C_i : is a class of a system.

Definition 1*:* scenario s . A scenario s is a sequence of user inputs triggering actions of a system that yields an observable result to an actor. In other words, s is a sequence of interactions between objects stimulated by input data or events.

Formally we can define the function:

$$Number-call(s, m_i, m_j) =$$

$$\begin{cases} n. & \text{iff } m_i \text{ has been invoked directly by } m_j \\ & n \text{ times in the execution of scenario } s, \\ & m_i \neq m_j \\ 0. & \text{iff } m_i \text{ has not been invoked directly by} \\ & m_j \; n \text{ times in the execution of scenario } s, \\ & m_i \neq m_j \end{cases}$$

$Number-call(s, m_i, m_j)$ indicates the number of the method m_i have been invoked directly by method m_j in the execution of scenario s , where methods m_i and m_j can be defined in same class or not.

Using this formula, we define some fan-in and fan-out metrics in method-level and class-level separately.

Definition 2 :

$$Fan\_in(s, m_i) = \sum_{j=1}^{TMS} number\_call(s, m_i, m_j)$$

$$Fan\_out(s, m_i) = \sum_{j=1}^{TMS} number\_call(s, m_j, m_i)$$

$$Fan\_in(s, c_i) = \sum_{i=1}^{TMC} fan\_in(s, m_i)$$

$$Fan\_out(s, c_i) = \sum_{i=1}^{TMC} fan\_out(s, m_i)$$

Where TMS is the total number of methods defined and implemented in all classes under consideration system. TMC is the total number of methods defined and implemented in a class under consideration.

$Fan\_in(s, m_i)$ indicates the times of method m_i be invoked by other methods in the execution of scenario s while $Fan\_out(s, m_i)$ the times m_i invoke other methods. $Fan\_in(s, C_i)$ indicates the total number of methods defined in class C_i be invoked in the execution of scenario s while $Fan\_out(s, C_i)$ the total number methods defined in class C_i invoked other methods.

A method m_c is called a client of method m_s, and m_s a supplier of method m_c , whenever m_c call at least one times method m_s. A high $Fan\_in(s, m_i)$ value shows high activity of m_i as a client and a high $Fan\_out(s, m_i)$ value shows high activity of m_i as a supplier. In our experience those methods with high fan-in or fan-out values play an important role in the system performance, as well as classes with high fan-in or fan-out values. These will be demonstrated in section 6.

4. Approach to get dynamic fan-in and fan-out metrics

The tools XDRE we developed can be used to get metrics we defined above section.

The approach adopted in XDRE is divided into 5 steps.

Step 1. Duplicate the source code. It is necessary because we will change the source code during the followed steps.

Step 2. Creat a software trigger for the source code.

We developed a class named **_FunctionTracer**. It can be used to instrument C++ program. It has no member function but constructor and destructor. Using object-lifecycle it traces system execution. When a function start a call, an object of class **_FunctionTracer** is instanced and saved in system stack. This object will be destroyed automatically when the call returns. The traced information is handled in the object's constructor and destructor. The advantage of this instrumentation is that it is not need to concern when the call occurs or ends.

Step 3. Instrument the subject system.

A set of objects of class _FunctionTracer are used as software trigger. Utilizing Open C++, we instrument these objects into the subject system.

After studying Open C++, we find it has many limits to instrument a system. For example it has no interface to handle global functions, etc. So we improved it. The improvement is out of scope for this paper, so we don't discuss it in detail. The improved Open C++ is facility to instrument the subject system.

We develop a set of meta objects to implement software triggers, dynamic information protocol, and instrumentation mechanism. These meta objects act on the compiler by MOP(Metaobject Protocol.). During the base level program compiling, these objects inject the software trigger into base-level program automatically. After being compiled, the source codes have been instrumented and then are delivered to a regular compiler, linked to supports needed in run-time (i.e. the implement of dynamic information protocol.). Thus by employ the reflective mechanism of Open C++, the instrumented codes and system source codes are placed at different two levels. The process of instrumentation is completed in the code analysis process of open compiler.

For example, after instrumentation, the method Draw(pDC) is wrapped into a method named void _occ_Draw (__InteractionInfo* pIInfo,CDC * pDC) (we call it wrapper method). In the wrapper method, an object of class **_FunctionTracer** is declared.When Draw(pDC) is invoked, the wrapper method is then invoked. So the object is created while its constructor runs, the call information is recorded. After Draw(pDC) returns , its destructor runs, the return information is recorded. In this way, we obtain traces of the subject system execution.

Step 4. Run the instrumented system, and collect the dynamic information .

Software triggers which are injected into the subject system don't affect the system behavior but generate information about method calls and returns while the system running. In XDRE, the dynamic information is sent to a block of shared memory, and then filtered and collected by a process responsible for information handling, written into a XML file.

Step 5. Calculate metrics from dynamic information file.

A class name shows the class functionality in many cases, as well as a method name. When parsing the dynamic information file to calculate metrics, we force on the class name and method name regardless of the object name and method parameters. Based on this, several overloading methods of a method are regarded as a same method.

5. Usage of dynamic fan-in and fan-out metrics

In a system, there are some components that implement the main functions or main structure. We call these components as critical components. In our experience, methods that invoke others frequently or invoked by others frequently play an important role in the system performance. So we propose a hypothesis.

Hypothesis: Methods with high fan-in or fan-out values implement the system main functions in all probability, and can be used to infer the subject system functionality. So do the Classes that with high fan-in or fan-out values.

This will be demonstrated in next section by a case study.

6. Case study

We have selected a case study of a highway simulation system to discuss the applicability of the proposed dynamic metrics, fan-in and fan-out., and demonstrate the hypothesis we proposed.

6.1. The experiment

The highway simulation system is developed by a programmer who studied in our Research Center several years ago. The system simulates buses states running in a highway linking two cities.

It is required to give out the number of passengers in each bus and the location of each buses running on the highway. It is also needed to show passenger ports information including the number of passengers who want a bus and the number of buses which will start.

Main classes of this system are CHiwaySystem, CPassengerPort, CHighWay, and CPassengerList etc.

Using the tool **XDRE**, the approach we discussed in section 4 has been applied on the highway simulate system. Its source code is duplicated firstly, and then software triggers are developed and injected into it. The run of instrumented source code results that its run-time information, i.e. dynamic information, stream into a block of shared memory. A process responsible for information collection fetches the information from the shared memory and filters it, and writes its useful part into a XML file.

Examination on the XML file, metrics we defined are calculated.

6.2. Result and Analysis

The result we obtained suggests that those metrics is effective to identify critical methods, critical classes, and can be used to infer the system functionality. The methods with higher $Fan\_in(s,m_i)$ values are CPassengerPort::getYQ(), CPassengerPort::getWQ( ), CHighWay::getpYQE( ), CHighWay::getpYQW( ), CMyTime::getMinute( ), which $Fan\_in(s,m_i)$ values higher than the average value. After inspected the source codes, we find those methods just deal with the critical entities of the system. They return information about buses that locate in passenger ports or run in the high way.

The methods with higher $Fan\_out(s,m_i)$ values are CHiwaySystem::savePortInfo (CString a [] , int & aSize , CString b [] , int & bSize) and CHiwaySystem::run ( ). CHiwaySystem::savePortInfo (CString a [] , int & aSize , CString b [] , int & bSize) formalizes and saves running buses states , CHiwaySystem::run ( ) realizes all simulate, i.e. passengers actions ,buses actions (start, run ,stop and enter stations). The system main function is even implemented using them.

The class with the highest $Fan\_in(s,C_i)$ is CPassengerPort in which passengers ports are dealt with. The class with the highest $Fan\_out(s,C_i)$ is CHiwaySystem in which all actions of the highway system are simulated. Both of these classes are important in this system.

Analysis of the system document and source code suggests that methods and classes we identified using fun-in and fan-out metrics just are critical.

This experiment demonstrates dynamic fan-in and fan-out metrics are clues which assist us to identify critical components and capture the main functionalities of the subject system for program understanding.

Experiment on a simple Client/Serve system also validates our hypothesis. In this system, methods with high fan-in and fan-out values are Receive (char * buf , int len , int flags), and Send (const char * buf , int len , int flags). They are the critical methods and from them we can infer that the system main business is communication.

7. Further works

In this paper, some dynamic metrics have been defined. These metrics depend on a scenario, a sequence of user inputs triggering actions of the subject system. For a system, different user input will generate different scenario, and a scenario is related to a part of functionality in many case. In order to detect te critical component of a system and infer its functionality fully, we should run the system in different inputs as much as possible and get their scenarios, and obtain those metrics we proposed under each scenario considered. An open question is how to merge those aspects to realize our purpose.

This is the further work we will study.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China under Grant Nos. 60473063, the Defence Pre-Research Project of the ‘Tenth Five-Year-Plan’ of China under contract number 413060601 and the National Research Foundation for the Doctoral Program of Higher Education of China under Grant No. 20030701009.

References

[1] G. Canfora, L. Mancini, and M. Tortorella. A Workbench for Program Comprehension during Software Maintenance. Proc. 4th Int’l Workshop on Program Comprehension (IWP96), IEEE Comp. Soc. Press, 1996, pp. 30-39.
[2] P. Linos, Z. Chen, S. Berrier, and B. O'Rourke.A Tool For Understanding Multi-Language Program Dependencies. Proc. IEEE 11th Int’l Workshop Program Comprehension (IWPC 03), IEEE Comp. Soc. Press, 2003, pp. 64-72.
[3]Von Mayrhauser and A.M. Vans. Program Understanding Behavior During Adaptation of Large Scale Software. Proc. 6th Int'l Workshop Program Comprehension (IWPC 98), IEEE Comp. Soc. Press, 1998,pp.164-172.
[4] S Henry and K Kafura. Software structure metrics based on information flow. IEEE Transactions on Software Engineering, 1981, 7(5):510– 518.
[5] R. Kollman, M. Gogolla, Metric-Based Selective Representation of UML Diagrams, Proc. 6th European Conf. Software Maintenance and Reengineering (CSMR 2002). IEEE, Los Alamitos, 2002.
[6] B. Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. Prentice Hall, 1996.
[7] Marius Marin, Arie van Deursen, Leon Moonen. Identifying Aspects using Fan-In Analysis. http://csdl2.computer.org/dl/proceedings/wcre/2004/2243/00/22430132.pdf
[8] Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson. Dynamic Metrics for Object Oriented Designs. Proceedings of the 6th International Symposium on Software Metrics table of contents. pp50 Year of Publication: 1999 ISBN:0-7695-0403-5
[9] Shepperd, M. J. “Software Engineering Metrics Volume I: Measures and Validations”. McGraw-Hill International, 1983.
[10] Erich Schikuta, Dynamic Software Metrics, http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93361.pdf
[12] ISO/IEC 9126, Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their use, 1991.

The Concept of Trace Summarization[*]

Abdelwahab Hamou-Lhadj
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
ahamou@site.uottawa.ca

Abstract

Recently, trace analysis techniques have gained a lot of attention due to the important role they play in understanding the system behavioral aspects. However, manipulating execution traces is still a tedious task despite the numerous techniques implemented in existing trace analysis tools. The problem is that traces are extraordinary large and abstracting out their main content calls for more advanced solutions. In this paper, I introduce the concept of trace summarization as the process of taking a trace as input and returning a summary of the main invoked events as output. A discussion on how text summarization techniques can be applied to summarizing the content of traces is presented.

Keywords:

Analysis of program execution, Program analysis for program understanding, Dynamic Analysis, Reverse Engineering.

1. Introduction

Dynamic analysis is crucial for understanding the behavior of a software system. Understanding an object-oriented (OO) system, for example, is not easy if one relies only on static analysis of the source code [15]. Polymorphism and dynamic binding, in particular, tend to obscure the relationships among the system artifacts.

Run-time information is typically represented using execution traces. Although, there are different kinds of traces, this paper focuses on traces of routine calls. I use the term routine to refer to a function, a procedure, or a method in a class.

Many studies such as the ones presented by Systä [14], Zayour [17], Lange et al. [8], and Jerding et al. [6] have shown that, if done effectively, trace analysis can help with various reengineering tasks such redocumenting the system behavior, maintaining the system, or simply understanding the implementation of software features.

However, the large size of traces poses serious limitations to applying dynamic analysis. To address this issue, most existing solutions provide a set of fine-grained operations embedded into tools that software engineers can use to go from a raw sequence of events to a more understandable trace content [6, 8, 14, 17]. But due to the size and complexity of typical and most interesting traces, this bottom-up approach can be difficult to perform.

In addition, software engineers who have some knowledge of the system and the domain will most likely want to have the possibility to perform a top-down analysis of the trace – They want to have the ability to look at the 'big picture' first and then dig into the details. Many research studies in program comprehension have shown that an adequate understanding of the system artifacts require usually both approaches (i.e. bottom-up and top-down) [12].

In this paper, I discuss the concept of trace summarization, which is a process of taking an execution trace as input and return a summary of its main content as output. This is similar to text summarization where abstracts can be extracted from large documents. Using an abstract, the reader can learn about the main facts of the document without having to read entirely its content.

Trace summaries can be used in various ways:

- Enable top-down analysis of execution traces, something that is not supported by most existing trace analysis tools.
- Recover the documentation of the dynamics of a software system that suffers from poor to non-existent documentation.
- Uncover inconsistencies that may exist between the way the system is designed and its implementation. This can be achieved by

[*] This research is supported by Natural Sciences and Engineering Research Council of Canada (NSERC)

comparing the extracted models to the models created during the design phase [6, 11]. The analysis of these inconsistencies can help determine areas of the system that need reengineering.

The rest of this paper is organized as follows: In the next section, I discuss trace summarization from the perspective of text summarization techniques and show the similarity between the two fields. In Section 3, I discuss how a summary can be validated.

Most of the concepts presented in this paper are still fresh ideas that constitute an ongoing research. They will need to be validated in the future.

2. What is Trace Summarization?

In general, a text summary refers to an abstract representing the main points of a document while removing the details.

Jones [7] defines a summary of a text as "a derivative of a source text condensed by selection and/or generalization on important content". Similarly, I define a summary of a trace as an abstract representation of the trace that results from selecting the main content by both selection and generalization.

Although, this definition is too specific to be used to define a summary of a trace, it points towards several interesting questions that deserve further investigation. These are: what would be a suitable size for the summary? And how should the selection and generalization of important content be done?

2.1 Adequate Size of a Summary

While it is obvious that the size of a summary should be considerably smaller than the size of the source document, it seems unreasonable to fix the summary's size in advance.

In fact, a suitable size of a summary of a trace will depend in part upon the knowledge the software engineer has of the functionality under study, the nature of the function being traced and the type of problem the trace is being used to solve (debugging, understanding, etc.). This suggests that any tool should allow the summary to be dynamically expanded or contracted until it is right for the purpose at hand. I suggest that no matter how large the original trace, there will be situations when a summary of less than a page will be ideal, and there will be situations where a summary of several thousand of lines may be better.

2.2 Content Selection

In text summarization, the selection of important content from a document is usually performed by ranking the document phrases according to their importance. Importance is measured using various techniques. In what follows, I present the most classical techniques and discuss their applicability to trace summarization.

Perhaps, the most popular technique for building text summaries is the word distribution method [4, 9]. This method is based on the assumption that the most frequent words of a document represent also its most important concepts. Once the word frequencies are computed, the document phrases are ranked according to the number of the most frequent words they contain. Similarly, one possible way of selecting the most important events from a trace is to examine their frequency distribution.

In fact, frequency analysis has also been used in various contexts of dynamic analysis. Profiles, for example, use the number of times specific events are executed to enable software maintainers prevent performance bottlenecks. In [1], Ball introduces the concept of Frequency Spectrum Analysis which is a technique that aims to cluster the trace components according to whether they have similar frequencies or not. This can help recover the system architecture.

However, the application of frequency analysis to select important events from execution traces raises several issues. First, the fact that traces contain several repetitions due to the presence of loops and recursion in the source code might render the results of frequency analysis inaccurate. For example, there is no evidence that something that is called ten times due to a loop would be more or less important than a routine that is called once or twice just because it did not happen to be in a repetitive code. Second, something that is repeated several times in one trace might not have the same behavior in another trace. Finally, our experience with using traces has shown that even if we remove the most frequent event from traces, traces will still be very large for humans to understand, which might make this technique useful but far from sufficient.

Another text summarization technique is the cue phrases method, which is based on the idea that most texts contain phrases that can lead to the identification of important content (e.g., "in conclusion", "the paper describes", etc) [4]. Similarly, the routine names can be used to extract important routines assuming that the system follows strict naming conventions. For example, during the exploration of a trace generated from a system that implements the C4.5 classification algorithm [16], my colleagues and I found that many routines are actually named according to the various steps of the algorithm such as buildClassifier, buildTree, etc. The ‘cue routines (or events)’ technique is certainly a powerful approach for building summaries from traces. However, in order to be successful, it requires having a system that follows some sort of naming conventions. In addition to this, there is a need to deal with the various naming matching issues that might occur. For example, some routine names might use acronyms or short names which might complicate the matching process.

The third text summarization technique discussed in this paper is the location of phrases in the document [2]. The idea is that the position of sentences in a document can be an indicator of how important they are. In text summarization, the first and last phrases of a paragraph are usually the ones that convey the most relevant content.

When applied to traces, we need to investigate whether the location of routines in the call tree (i.e. trace) can play a relevant role in determining their importance. There are certainly situations where this can be valid. For example, if the system is designed according to a layered architecture then the bottom layers are perhaps the ones that are the least important since they implement the system low-level details. These usually appear in the call tree as leaf nodes.

Some thoughts: a trace can be viewed according to two dimensions: vertical and horizontal dimensions as shown in Figure 1. The vertical dimension reflects the sequential nature of the execution of the system. One possible scenario for applying the location technique is based on the ability to partition the trace into smaller sequences that depict different behavioral aspects of the system, and then select the first calls of each sequence and add them to the summary. This is like having a text composed of many sequential paragraphs and that the summarizer needs to visit each of them. It is obvious that in practice this might not be easy to perform. Indeed, the partitioning of a trace might be challenging. And even if it is done successfully, we might end up having a considerably large number of partitions where some of them do not necessarily convey the most important content.

The horizontal dimension focuses on the fact that a trace is viewed as a tree structure containing many levels of calls. The idea is to develop a level analysis technique in order to detect the levels that introduce trace components used as mere implementation details. For example, the routines that appear always in the first levels of the tree might represent the system high-level concepts whereas the ones that appear at all levels might be utilities (because they are called by many other routines).

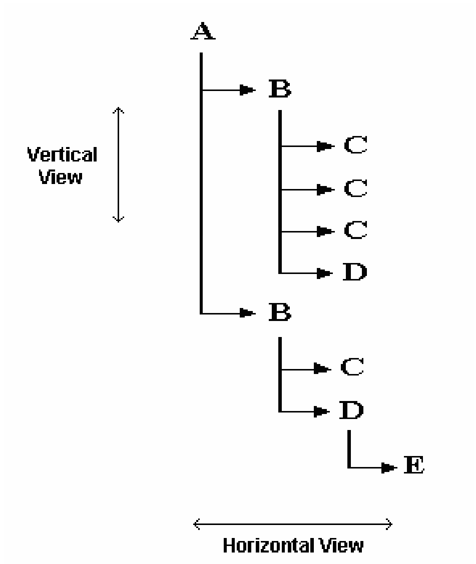


Figure 1. The vertical and horizontal views of a call tree

2.3 Content Generalization

Content generalization consists of generalization of specific content with more general abstract information [7]. When applied to execution traces, generalization can be performed in two ways:

The first approach to generalization involves assigning a high-level description to selected sequences of events. For example, many trace analysis tools provide the users with the ability to select a sequence of calls and replace it with a description expressed in a natural language. However, this approach relies on user input and would be very hard to automate.

A second approach to generalization relies on treating similar sequences of execution patterns as if they were the same. This approach can be automated by varying the similarity function. For example, in the simplest case all sequences with the same elements, ignoring order, could be treated the same. Or, all subtrees that differ by only a certain edit distance could be treated the same. All trace summarization approaches will need to use this technique to some extent.

For example, the call tree of Figure 1 can be summarized into the tree shown in Figure 2 by ignoring the number of contiguous repetitions of the node labeled ‘C’ and by comparing subtrees up to level 2 (this will ignoring the node ‘E’). A discussion on how matching criteria can be used to reduce the size of a trace is presented by De Pauw et al. [3].

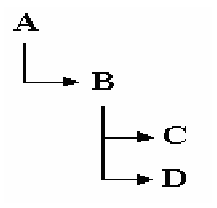


Figure 2. A summary extracted from the tree of Figure 1 by applying generalization

However, it might be hard to determine how the matching criteria should be combined in order to extract the most meaningful content. Different combinations will most likely result in different summaries. Tools that support the generation of summaries will need to allow enough flexibility to apply the matching criteria in several ways.

3. Validating a Trace Summary

Perhaps, one of the most difficult questions when evaluating a summary is to agree about what constitutes a good summary. In other words, what distinguishes good summaries from bad summaries (assuming that there are bad summaries)?

In text summarization, there are two techniques for evaluating summaries: extrinsic and intrinsic evaluation. The extrinsic evaluation is based on evaluating the quality of the summary based on how it affects the completion of some other tasks [5]. The intrinsic evaluation consists of assessing the quality of the summary by analyzing its content [10]. Using this approach, a summary is judged according to whether it conveys the main ideas of the text or not, how close it is to an ideal summary that would have been written by the author of the document, etc.

Extrinsic evaluation of a trace summary will typically involve using summaries to help with various software maintenance tasks such as adding new features, fixing defects, etc.

The intrinsic evaluation technique can be used to assess whether the extracted summary reflect a high-level representation of the traced scenario that would be similar to the one that a software engineer would design. In practice, I suspect that both types of evaluations are needed.

4. Conclusions and Future Directions

The objective of this paper is to present a technique for analyzing traces based on summarizing their main content. This technique is referred to as Trace Summarization, which the process of taking a trace as input and generating an abstract of its main content as output. I argued that summaries can be very useful to software engineers who want to perform top-down analysis of a trace, understand the system behavior, or uncover inconsistencies between the system design and its actual implementation.

In the paper, a discussion on how text summarization techniques can be applied to extracting summaries from trace is presented.

Future directions should focus on examining the techniques presented in this paper in more detail including experimenting with several traces. The experiments should take into account systems of different domains, the expertise software engineers have of the system, and the type of software maintenance performed.

References

[1] T. Ball, “The Concept of Dynamic Analysis”, *ACM Conference on Foundations of Software Engineering (FSE)*, September 1999

[2] P. Baxendale, “Machine-made index for technical literature – an experiment”, *IBM. Journal of Research and Development* 2:354-361, 1958

[3] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, “Execution Patterns in Object-Oriented Visualization”, In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, NM, 1998, pp. 219-234

[4] H. Edmundson, “New methods in automatic extracting”, *Journal of the ACM* 16(2): 264-285, 1969

[5] H. R. Jing, K. McKeown, and M. Elhadad, “Summarization evaluation methods: Experiments and analysis”, *In Working Notes of the AAAI Spring Symposium on Intelligent Text Summarization*, 1998, pp. 60-68

[6] D. Jerding, S. Rugaber. “Using Visualization for Architecture Localization and Extraction”, In *Proc. of the 4th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, October 1997

[7] S. K. Jones, “Automatic summarising: factors and directions”, In *Advances in Automatic Text Summarization, MIT Press*, 1998, pp. 1-14

[8] D. B. Lange, Y. Nakamura, “Object-Oriented Program Tracing and Visualization”, *IEEE Computer*, 30(5), 1997, pp. 63-70

[9] H. Lunh, “The Automatic Creation of Literature Abstracts”, *IBM Journal of Research and Development 2(2):* 159-165, 1958

[10] C. Paice, and P. Jones, “The identification of Important Concepts in Highly Structured Technical Papers”, In Proc. of the 16th Annual International ACM SIGR Conference on research and Development in Information retrieval, 1993, pp. 69-78

[11] Reiss S. P., Renieris M., “Encoding program executions”, In *Proc. of the 23rd international conference on Software Engineering*, Toronto, Canada, 2001, pp. 221-230

[12] M.A. Storey, K. Wong, H.A. Müller, “How do Program Understanding Tools Affect how Programmers Understand Programs?”, In *Proc. of the 4th Working Conference on Reverse Engineering*, 1997, pp. 183 - 207

[13] T. Strzalkowski, G. Stein, J. Wang, B. Wise, “Robust Practical Text Summarization”, In *Advances in Automatic Text Summarization*, MIT Press, 1999

[14] T. Systä, “Understanding the Behaviour of Java Programs”, In *Proc. of the 7th Working Conference on Reverse Engineering (WCRE)*, Brisbane, QL, 2000, pp. 214-223

[15] N. Wilde, R. Huitt, “Maintenance Support for Object-Oriented Programs”, *Transactions on Software Engineering*, 18(12):1038–1044, Dec. 1992

[16] Witten I. H., Frank E. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 1999

[17] I. Zayour, “Reverse Engineering: A Cognitive Approach, a Case Study and a Tool”, Ph.D. dissertation, University of Ottawa, 2002

Applying Semantic Analysis to Feature Execution Traces

Adrian Kuhn, Orla Greevy and Tudor Gîrba
Software Composition Group
University of Bern, Switzerland
{akuhn, greevy, girba}@iam.unibe.ch

Abstract

Recently there has been a revival of interest in feature analysis of software systems. Approaches to feature location have used a wide range of techniques such as dynamic analysis, static analysis, information retrieval and formal concept analysis. In this paper we introduce a novel approach to analyze the execution traces of features using Latent Semantic Indexing (LSI). Our goal is twofold. On the one hand we detect similarities between features based on the content of their traces, and on the other hand we categorize classes based on the frequency of the outgoing invocations involved in the traces. We apply our approach on two case studies and we discuss its benefits and drawbacks.

Keywords: reverse engineering, dynamic analysis, semantic analysis, features, feature-traces, static analysis.

1. Introduction

Many reverse engineering approaches to software analysis focus on static source code entities of a system, such as classes and methods [5, 16]. A static perspective considers only the structure and implementation details of a system. Using static analysis alone we are unable to easily determine the roles of software entities play in the features of a system and how these features interact. Without explicit relationships between features and the entities that implement their functionality, it is difficult for software developers to determine if their maintenance changes cause undesirable side effects in other parts of the system.

Several works have shown that exercising the features of a system is a reliable means of correlating features and code [7, 24]. In previous works [9, 10], we described a feature-driven approach based on dynamic analysis, in which we extract execution traces to achieve an explicit mapping between features and software entities like classes and methods. Our definition of a feature is a unit of behavior of a system.

Dynamic analysis implies a vast amount of information, which makes interpretation difficult. We introduce a novel approach that uses an information retrieval technique, namely Latent Semantic Indexing (LSI) [4], to analyze the traces and their relationship to the source code entities. LSI takes as an input a set of *documents* and the *terms* used, and returns a similarity space from which similarities between the documents are ascertained.

In a previous work, we built a reverse engineering approach to cluster the source code entities based on their semantic similarities [13]. In this paper we apply our approach on dynamic information. In other words we use the traces of features as the *text corpus* and we sample this corpus in two different ways to show the generality of our approach.

1. To identify similar features, we use as a document the trace and the method calls involved in the trace as the terms of the document.
2. To identify similarities between classes, we use the classes that participate in feature execution as documents, and all method calls found in the traces outgoing from a class as the terms of the document.

Structure of the paper. We start by introducing the terminology we use to describe and interpret dynamic information. In Section 3 we give an overview of LSI. In Section 4 we describe the details of our approach. In Section 5 we report on the two case studies conducted. We summarize related work in Section 7. Section 8 outlines our conclusions and future work.

2. Feature Terminology

In this section we briefly outline the feature terminology we use. The terms here are based on our previous work [9].

We establish the relationship between the features and software entites by exercising the features or *usage scenarios* and capturing their execution traces, which we refer to as *feature-traces*. A *feature-trace* is a sequence of runtime events (*e.g.,* object creation/deletion, method invocation) that describes the dynamic behavior of a feature.

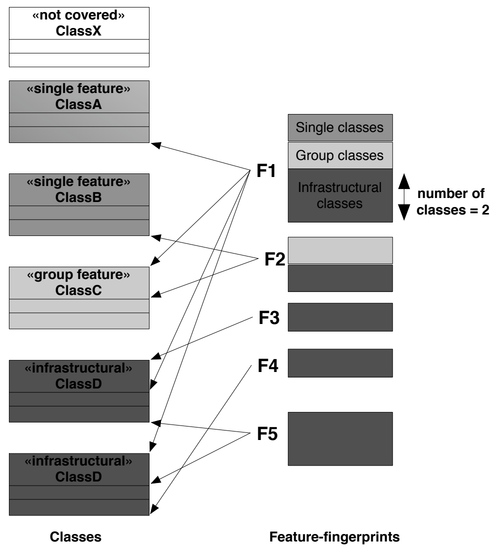


Figure 1. Feature-Fingerprints and Classes Relationships

We define the measurements $NOFC$ to compute the number of feature-traces that reference a class and FC to compute a characterization of a class in terms of how many features reference it and how many features are currently modeled. NOF refers to the number of feature-traces under analysis.

- *Not Covered (NC)* is a class that does not participate to any of the features-traces of our current feature model.
$$(NOFC = 0) \rightarrow FC = 0$$
- *Single-Feature (SF)* is a class that participates in only one feature-trace.
$$(NOFC = 1) \rightarrow FC = 1$$
- *Group-Feature (GF)* is a class that participates in less than half of the features of a feature model. In other words, group-feature classes/methods provide functionality to a group of features, but not to all features.
$$(NOFC > 1) \wedge (NOFC < NOF/2) \rightarrow FC = 2$$
- *Infrastructural (I)* is a class that participates in more than half of the features of a feature model.
$$(NOFC >= NOF/2) \rightarrow FC = 3$$

Feature characterizations of classes attach semantic significance to a class in terms of its role in a feature. Our feature characterization approach reduces the large feature-traces to consider only the relationships between features and software entities. Information about the frequency of references to a method or class in a feature-trace is not taken into consideration.

3. Semantic Driven Software Analysis

Common software analysis approaches focus on structural information and ignore the semantics of the problem and solution domain semantics. But this information is essential in getting a full interpretation of a software system and its meaning. As an example: the class structure of a text processor, a physical simulation or a computer game might all look the same; but the naming of the source code will differ, since each project uses its own domain specific vocabulary. *Semantic driven software analysis* gathers this information from the comments, documentation, and identifier names associated with the source code using information retrieval methods.

Our semantic analysis tool Hapax [13] uses *latent semantic indexing*, a state of the art technique in information retrieval to index, retrieve and analyze textual information [4]. LSI treats the software system as a set of text documents and analyzes how terms are spread over the documents. Principal components analysis is used to detect conceptual correlations and provides a similarity measurement between both documents and terms.

As most text categorization systems, LSI is based on the Vector Space Model (VSM) approach. This approach models the text corpus as a term-document matrix, which is a tabular listing of mere term frequencies. Originally LSI was developed to overcome problems with synonymy and polysemy that occurred in prior vectorial approaches. It solves this problem by replacing the full term-document matrix with an approximation. The downsizing is achieved with Singular Value Decomposition (SVD), a kind of Principal Components Analysis originally used in Signal Processing to reduce noise. The assumption is that the original term-document matrix is noisy (the aforementioned synonymy and polysemy) and the approximation is then interpreted as a noise reduced – and thus better – model of the text corpus.

Even though search engines [2] are the most common usage of LSI, there is a wide range of applications, such as: automatic essay grading [8], automatic assignment of reviewers to submitted conference papers [6], cross-language search engines [15], thesauri, spell checkers and many more. As a model, LSI has been used to simulate language processing of the human brain, such as the language acquisition of children [14] and high-level comprehension phenomena like metaphor understanding, causal inferences and judgments of similarity.

3.1. Semantic Clustering at Work

To get a semantic model of the software system, we implemented the following four steps:

1. First, we split the software system into text documents. While static approaches work with the source code of classes or methods, in this paper we use the textual representation of feature-traces as documents.
2. The second step counts the frequencies of term occurrences in the documents. A term is any word found in the source code or comments, except keywords of the programming language. Identifiers are separated based on standard naming conventions (*e.g.*, camel-case).
3. Then singular value decomposition, a principal components analysis technique, is applied on the term occurrence data. This yields an index with conceptual correlations and similarities between both documents and terms. More in-depth information on using LSI is given in [4, 2].
4. To understand this semantic correlations, we group the documents using a hierarchical clustering algorithm. We visualize the clusters on a shaded correlation matrix. A shaded correlation matrix is a square matrix showing the similarity between documents in gray colors. The color at $m_{i,j}$ shows the similarity between the i-th and the j-th document: the darker the color, the more similar these two documents. The visualization algorithm itself is detailed in [13].

4. Our Approach

The novelty of our approach is the combination between dynamic analysis and semantic analysis. Our paper has two goals: to detect similarities between traces, and to detect similarities between classes based on their involvement into the traces.

We outline how we apply our technique to obtain a semantical analysis on top of feature-traces from a software system.

1. We instrument the code of the the application under analysis and execute a set of its features as described in Section 6. Our dynamic analysis tool *TraceScraper* extracts execution traces and models them as a tree of method invocation calls. We treat feature-traces as first class entities and incorporate them into the static model of the source code. By doing so we establish the relationships between the methods calls of the feature traces and the static model class and method entities. We compute the feature characterizations of the classes as described in Section 2.
2. Our semantic analysis tool *Hapax* is applied on the feature-traces. To use the feature-traces as text corpus, we create *ad-hoc* text documents with the method names found in the feature traces. Hapax applies LSI on the documents, clusters them and finally delivers a visualization of document clusters and their similarities. For more detail refer to Section 3.1.

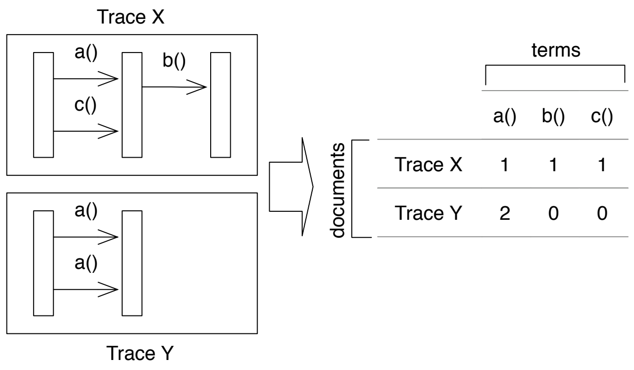


Figure 2. Example of how traces form documents and the method calls form the terms.

Both tools are built on top of our reverse engineering framework Moose [19], that provides a generic mechanism which allows for an easy composition of different tools. Because of that, we could easily integrate the two tools to perform the semantic analysis on the traces.

5. Validation: Ejp-Presenter and Smallwiki

In this section we present the results of applying our approach to the *Ejp-presenter* and the *SmallWiki* case studies.

Ejp-presenter [22] is an open source tool written in java which provides a graphical user interface for viewing execution traces of java programs. It consists of 166 classes. To obtain feature traces we instrumented 13 unit tests provided with the application. Our assumption was that each unit test exercised a distinct feature.

SmallWiki [20] is a collaborative content management system used to create, edit and manage hypertext pages on the web. It is implemented in Smaltalk and consists of 464 classes. To identify features of *SmalWiki* we associate features with the links and entry forms of the *SmallWiki* pages. We assume that each link or button on a page triggers a distinct feature of the application. For this experiment we executed 6 features.

As mentioned in the introduction we tackle the case studies at two levels of abstraction, once using features and once using classes as granularity.

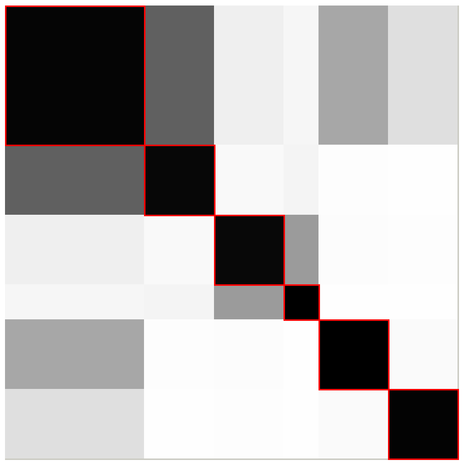


Figure 3. Correlation matrix with the features of Ejp-Presenter, showing well distributed concepts.

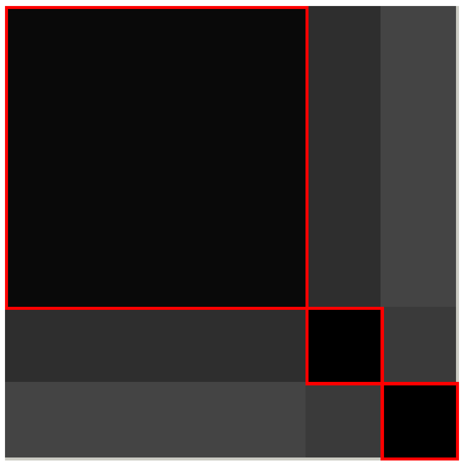


Figure 4. Correlation matrix with the features of Smallwiki, showing one concept only.

5.1. Identifying Similar Features

To identify similar features, we use the feature-traces as documents and the method calls involved in a trace as terms. Similar features are clustered together, and the clusters visualized on a shaded correlation matrix. The visualization reveals the semantic similarity between the features, showing how they are related to each other.

Figure 3 shows the *Ejp-presenter* case study. Its features are well distributed: there are 6 clusters of different sizes, and – as indicated by the gray blocks in the off-diagonal – different correlations among them.

This is a list with the features in each cluster, starting from top left to bottom right:

1. boolean parameter, string list parameter, radio parameter, and remove non significant.
2. loaded method and loaded class.
3. configuration and mainframe.
4. dom.
5. highlight hotspot and color parameter.
6. file chooser dialog and color chooser.

The names shown in the above listing are of a descriptive nature, and not part of the vocabulary used by the Information Retrieval engine itself. Thus we can judge, based on them, that the analysis revealed meaningful correlations.

Figure 4 shows the *SmallWiki* case study. Because its features use similar methods, they belong to the same semantical concept. A closer look at the feature-traces reveals that *SmallWiki* has a very generic structure, and the traces are not discriminated by their method usage but by the parameters passed to their methods. Taking only the method names into account, our approach fails discriminating these features.

5.2. Identifying Similar Classes

In Section 2 we give a characterization of classes based on their structural relationship to features. In Section 3.1 we show how we retrieve a characterization of classes based on their semantic correlation.

To identify the semantic correlation between classes, we use the classes as documents, and all method invocations originating from a class as terms. Thus classes with similar outgoing method invocations are clustered together, that is classes that are based on the same functionality belong to the same cluster. We expect these clusters to match with the ‘feature terminology’ characterization, since *single feature* classes are based on *group feature* classes with in turn are based on *infrastructure* classes.

Figure 5 reveals seven semantical clusters of different shape. In Table 1 we compare theses clusters – numbered from top left to bottom right – with the ‘feature terminology’ characterization.

And in fact, the two characterizations – once based on semantical analysis, once based on structural analysis – match pretty well.

6. Discussion

The large volume of information and complexity of dynamic information makes it hard to infer higher level of information about the system.

cluster	#1	#2	#3	#4	#5	#6	#7
single	19	4	–	–	1	3	1
group	4	6	2	–	–	–	–
infra.	–	–	–	1	–	–	–
size	23	10	2	1	1	3	1

Table 1. The clusters from Figure 5 and the types of classes contained.

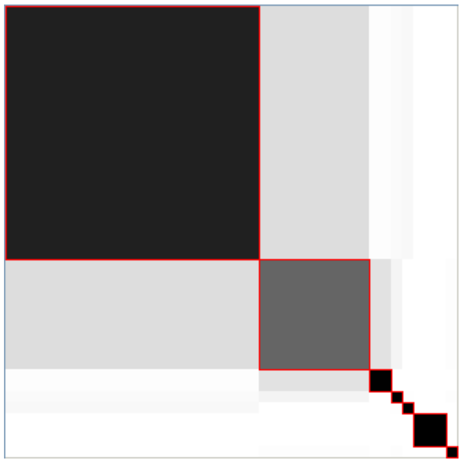


Figure 5. Correlation matrix with Ejp-Presenter classes, based on their usage in features-traces.

Coverage. We limit the scope of our investigation to focus on a set of features. Our feature model does not achieve 100% coverage of the system. For the purpose of feature location, complete coverage is not necessary. However, LSI analysis yields better results on a large text corpus. Therefore to improve our results, we need to increase the coverage of the application by exercising more of its features.

Trace as Text Corpus. In this paper, we build the text corpus from the names of the methods that get called from the studied traces. When applying the approach to *SmallWiki*, the result was not very relevant because *SmallWiki* has a generic structure and the difference between traces is not given by the method names, but by the actual parameters passed to the methods. Hence, a variation of the approach would be to take the actual parameter names into consideration when building the text corpus.

Language Independence. Obtaining the traces from the running application requires code instrumentation. The means of instrumenting the application is language dependent. *Ejp-presenter* is implemented in java. To instrument it we used the *Ejp (Extensible Java Profiler)* [22] based on the Java Virtual Machine Profiler Interface (JVMPI). *SmallWiki* is implemented in Smalltalk. Our Smalltalk instrumentation is based on method wrappers [3].

We abstract a feature model from the traces we obtain by exercising the features of the instrumented system. Our analysis is performed on the feature model and is therefore language independent.

7. Related Work

Many researchers have identified the potential of feature-centric approaches in software engineering and in particular as a basis for reverse-engineering [7, 23, 24]. Our work is directly related to the field of dynamic analysis [1, 11, 25] and user-driven approaches [12].

Feature location techniques such as *Software Reconnaissence* described by Wilde and Scully [23] , and that of Eisenbarth et al. [7] are closely related to our feature location approach. In contrast, our main focus is applying feature-driven analysis to object-oriented applications.

LSI has been recently proposed in static software analysis for various tasks, such as: identification of high-level conceptual clones [18], recovering links between external documentation and source code [17], automatic categorization of software projects in open-source repositories [21] and visualization of conceptual correlations among software artifacts [13].

8. Conclusions and Future Work

Reverse engineering approaches that focus only on the implementation details and static structure of a system overlook the dynamic relationships between the different parts that only appear at runtime. Our approach is to complement the static and dynamic analysis by building a model in which features are related to the structural entities.

Dynamic analysis offers a wealth of information, but it is exactly the wealth of information that poses the problem in the analysis. To deal with it, we employed Latent Semantic Indexing, an information retrieval technique that works with documents and terms . Our goals were to identify related features and to identify related classes that participate in features. We use the method calls from the traces as the text corpus and then we use two mappings to documents: (1) traces as documents, and (2) classes as documents. We clustered the documents based on the terms used to find relationships between them.

The results obtained on two case studies are promising, yet we did encounter problems with only considering the method names as text corpus. From our findings we conclude that more work is needed to assess the different variations of the approach. Furthermore, LSI yields better results on large text corpus, hence we also need to apply our approach on larger case studies or to achieve a higher coverage of the system by our feature-traces.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077).

References

[1] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[2] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. Technical Report UT-CS-94-270, 1994.

[3] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[4] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000.

[6] S. T. Dumais and J. Nielsen. Automating the assignment of submitted manuscripts to reviewers. In *Research and Development in Information Retrieval*, pages 233–244, 1992.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[8] P. W. Foltz, D. Laham, and T. K. Landauer. Automated essay scoring: Applications to educational technology. In *Proceedings of EdMedia '99*, 1999.

[9] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society Press, 2005.

[10] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, pages 347–356. IEEE Computer Society Press, Sept. 2005.

[11] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

[12] I. Jacobson. Use cases and aspects—working seamlessly together. *Journal of Object Technology*, 2(4):7–28, July 2003.

[13] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference On Reverse Engineering (WCRE 2005)*, Nov. 2005. to appear.

[14] T. Landauer and S. Dumais. The latent semantic analysis theory of acquisition, induction, and representation of knowledge. In *Psychological Review*, volume 104/2, pages 211–240, 1991.

[15] T. Landauer and M. Littmann. Fully automatic cross-language document retrieval using latent semantic indexing. In *In Proceedings of the 6th Conference of the UW Centre for the New Oxford English Dictionary and Text Research*, pages 31–38, 1990.

[16] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

[17] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 125–135, May 2003.

[18] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.

[19] O. Nierstrasz, S. Ducasse, and T. Girba. The story of Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 1–10. ACM, 2005. Invited paper.

[20] L. Renggli. Smallwiki: Collaborative content management. Informatikprojekt, University of Bern, 2003. http://smallwiki.unibe.ch/smallwiki.

[21] M. M. Shinji Kawaguchi, Pankaj K. Garg and K. Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC.04)*, 2004.

[22] S. Vauclair. Extensible java profiler. Masters thesis, EPF Lausanne, 2003. http://ejp.sourceforge.net.

[23] N. Wilde and M. C. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[24] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *J. Syst. Softw.*, 54(2):87–98, 2000.

[25] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

Enhancing Static Architectural Design Recovery by Lightweight Dynamic Analysis (Position Paper)

Andrew Malton
Atousa Pahelvan
{ajmalton,apahleva}@uwaterloo.ca
Software Architecture Group
University of Waterloo

Abstract
Architectural views of software systems recovered from static analysis of source code often mask what is really going on, because the dependencies which are visible by static analysis cannot reveal views which are part of the dynamic mental model of the developers. Nevertheless, the static view is always a starting point. In this work we attempt a minimal dynamic analysis, performed using simple available tools and without invasion of the software system, aimed specifically at resolving the ambiguities of a purely static architecture.

1 Introduction

The subjects of our analyses are software systems available as source code and related artefacts. By means of static analysis, essentially *fact extraction* [Lin], *reflexion* [Murphy], and *visualization* [Finnigan], we try to capture the mental model [Holt,Fowler] of the system's developers. In Kruchten's terminology [Kruchten] we are reconstructing a *development view* from the available artefacts, and producing a model in a disciplined style (that is, the landscape view [Finnigan]) which then has meta-properties which we can predict: browsable as a landscape (see, *e.g.*, [Bowman]), explorable using a catalogue of abstraction patterns [Bull], exchangable using standard exchange languages [GXL].

Static fact extraction provides the kind of information which a compiler can obtain about a program, and about a whole system when the "build time architecture" [Tu] is considered as a whole. A *fact base* is really a representation of the symbol table of the compiler (or of some other related build-time tool such as a pre-processor, or linker) as a simple relational database. Such "facts" are the ground of static reverse engineering, so that a static view basically "only knows what the compiler knows".

With reflexion, which is a disciplined way of drawing on the knowledge of domain experts, a view of the software architecture can be built which goes past the pure static analysis. However, when the goal is a well structured development view, basically still a static view, the information drawn from the expert still only tells us about modules, subsystems, and dependencies which are "always true" because they are the static view. Even though the expert may have a mental model of the dynamics of the system, that will not be revealed, at least not recovered and represented, by a static reflexion model.

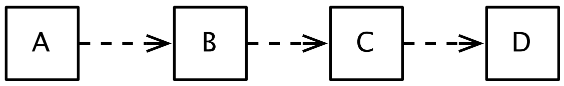


Figure 1

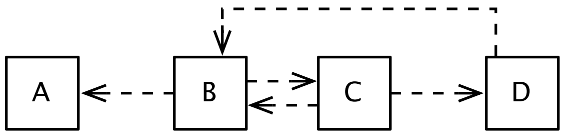


Figure 2

2 The Limitations of the Landscape View

The landscape view of a large software system is a presentation which allows the huge amount of detailed information to be viewed at many levels of detail: the metaphor is that of viewing a landscape "from above", as it were from an airplane (or from Google maps!), zooming in to see detail, or out to see the big picture. The basic outline is static: containment structures and relationships visible from static analysis of the code.

A simple example shows the sort of problem which arises when recovering the static architecture of an imagined software system from its code base. In Figure 1, there appears the expected architecture: probably the expert view. It is a pipeline of four processes. Typically process A would be obtaining an event stream from an external source (*e.g.* a user) and translating it to an internal form: processes B and C would be performing operations on the model based on the events; and process D would be translating model changes back into external events.

In Figure 2 we see the result of static architectural recovery on a code base for such a system. The static structure does not resemble the pipeline in the least! It appears that the subsystem B is the control centre, pulling data from A and maintaining the domain repository. Subsystem C does its work under B's control and using B's representation: so the dependence is mutual. And D receives control from C and data from B.

Of course, certainly, the view in Figure 2 is "true", and an explorer must see that landscape in order to understand the design. But the pipeline is the mental

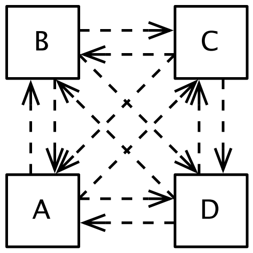


Figure 3

model of the designers, as stated above. In order to recover and visualize the architecture in keeping with that mental model, we need information about the *actual* behaviour, such as would be drawn as a "communication diagram" of some sort in the UML.

In large systems the mismatch between the static view and the process view [Kruchten] can be particularly wide. Because large systems tend to evolve into layers, with lower layers providing services to upper layers, the desired "official picture" of the system is obscured by universal dependency on utility libraries, or the fact that key interactions (data flow, signaling, event handling) are mediated by other parts of the system. In the worst case a dependency clique (Figure 3) appears at the top level of the landscape, ultimately telling us nothing except the names of the top-level subsystems.

3 Lightweight Dynamic Analysis

Above we discuss the need for actual behavioural information. This need is quite specific, *viz.*, to annotate or enrich a basically static view of the architecture, or to display that static view in a manner more in keeping with the mental model of the designers, who know the subsystem interaction patterns. Thus we attempt a focused dynamic analysis, executing the system and collecting call sequences which can be visualized simultaneously with the static architecture. We call this Lightweight Dynamic Analysis because we mean to gather a minimum of such information with a minimum intrusion into the system's structure.

Our process presupposes a static architecture (landscape) has been constructed, and consists of the following steps, some of which have been automated.

3.1 Identifying Key Scenarios

Using an analogue of the reflexion method, we identify key scenarios of the system usage by interaction with domain experts and from informal artefacts (written documentation).

3.2 Identifying Pivotal Functions

Based on the static landscape, which combines subsystem structure at the highest level of abstraction with module, method, and function dependencies at lower levels, we identify *pivotal functions*, which are those externally linked entities which most seem to break the visual knot of static dependencies. For example (see Figure 3) those functions which contribute most to the top-level clique are the ones which we identify as pivotal.

Pivotal functions are similar in spirit to Walkinshaw's *landmark functions* [Walkinshaw], because both classes of function are chosen to play a role in a scenario and have the role of reducing the space which is afterwards covered by dynamic analysis. In our work, we identify pivotal functions by examining the static architecture: those functions are chosen as pivotal which have the most links across subsystem boundaries, and so which contribute the most to the confusion of a purely static view. Thus, this step is an automatic analysis of the landscape.

3.3 Key-Pivotal Interactions

For each key scenario we execute the system in the debugger (**gdb**, because we have studied open source C and C++ software) preceded by a script that requests breakpoints on entry and exit to each pivotal function.

This step is automated dynamic fact extraction. The resulting debugger log is reduced to a well-nested sequence of events of the form

call Pi
return Pi

where Pi is a pivotal function. The well-nesting is simply the fact that it is a call history, so that each **call**

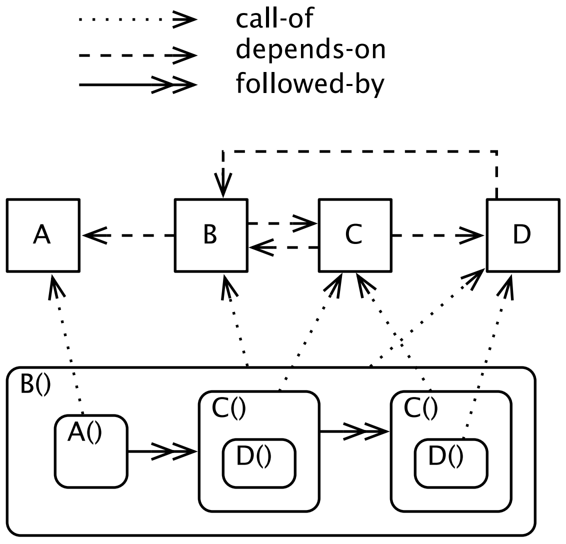


Figure 4

P is followed by a well-nested subsequence and then by **return *P***. It's convenient to write a call history in a nested way, like in LISP, as (Pi (Pj ...) (Pk ....)).

Figure 4 displays a call history for (B (A) (C (D)) (C (D))).

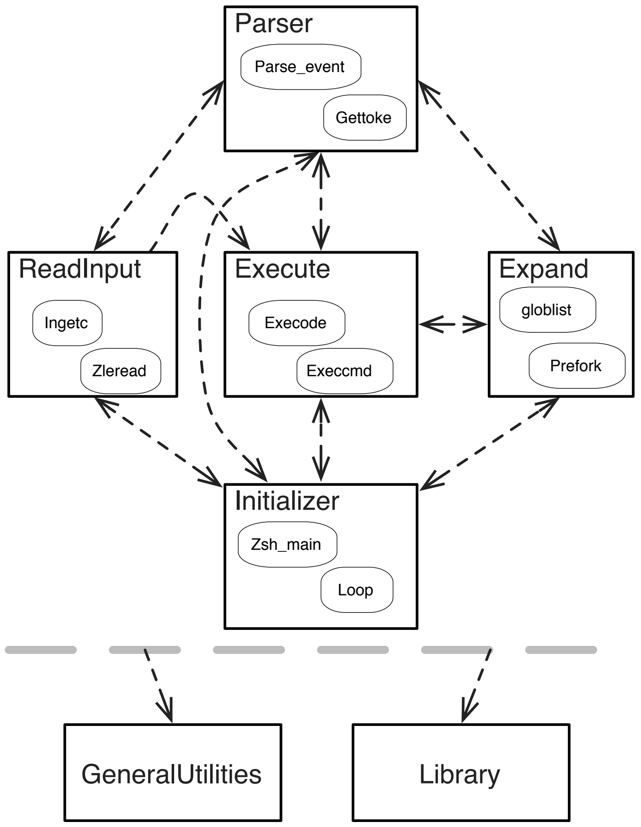


Figure 5

3.4 Static / Lightweight Dynamic Visualizations

The landscape visualization technique we use is based on nested boxes and arrows [Malton]. Although designed for visualizing static structures, it can be used for any well-nested relationship, to be designated as "containment". Since key-pivotal interactions are well-nested call histories, they can be visualized in the same way and even in the same diagram as the static architecture which we presuppose.

We visualize a call history of a function *Pi* as a box, labeled with *Pi*, which "contains" visualized well-nested subsequences. We draw a "next" arrow from each to the next in sequence, as exemplified in Figure 4. We link visualized interactions with their static structure by drawing a "from" arrow from a call history of *Pi* to the box which represents *Pi* in the static architecture.

4 In Practice

We extracted the architecture of three Unix shell programs (zsd, bash, and (t)csh) following the process presented in previous section. Each is available in open source. As we sought a common 'reference architecure' which might cover all of them, we studied four releases of each of the three systems.

The derived static architecture is shown in Figure 5, in layered fashion. It consists of seven subsystems, which are divided into three tiers.

The highest tier, **Initializer**, is a repository for storing central data (state and shell variables) for a shell session.

The **Process** tier is a collection of interdependent components that carry out command processing: the **Parser** parses an input command and selects a suitable process to execute it. The **ReadInput** subsystem handles input from the various sources (files for a non-interactive shell, keyboard for an interactive user, or from the command line string), and also handles job control and forking. The **Expand** subsystem performs expansion and substitution on different parts of the parsed command (pathname expansion, parameter expansion, variable substitution, and command substitution). The **Execution** subsystem actually executes the commands, *e.g.* by executing subprocesses.

The **GeneralUtility** subsystem provides different facilities such as pattern matching, string libraries, and signal handling for other subsystems. It also provides interface to operating-system-dependent built-in functions,storage allocator, and error printing routines. Moreover it has many commonly used functions, which provide support for the basic functionality needed by various subsystems.

For this example we choose a simple use case of entering a direct command with a global variable

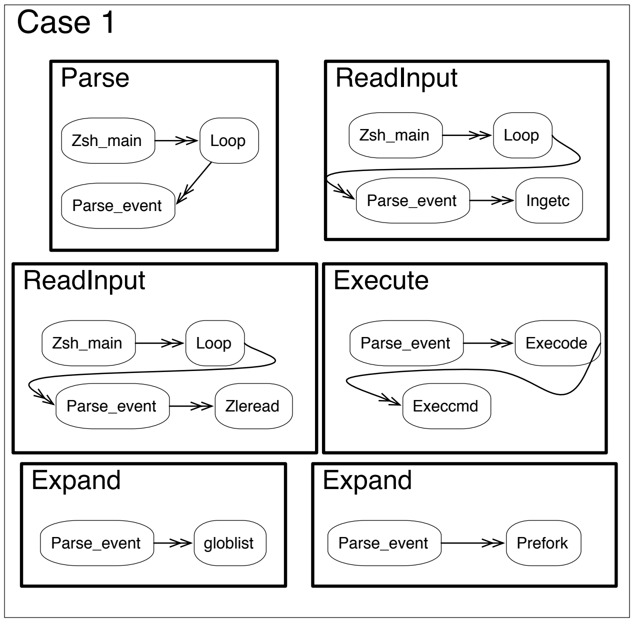


Figure 6

subsitution, as for example, **cat $FILE1**.

Pivotal functions are chosen by analysing the static dependency graph. The pivotal functions found for the zsd shell are shown in Figure 5, embedded in their subsystems. Figure 6 shows the pivotal functions executed in series for the use case which was chosen. In this case, all the pivotal functions have unique names, and so the edges, which normally would be displayed to link dynamic events (calls and returns) in the call history to static sources (functions) in the static architecture, need not be drawn.

Two aspects of browsing these structures are invisible in our diagrams. We use the **lsedit** tool to explore landscapes, including our lightweight-dynamic landscapes. With **lsedit**, one may (a) suppress the contents of a box, causing incident edges on its contents to be "lifted" and appear attached to the containing box; and (b) suppress edges by class or origin. Both of these browsing techniques allow the analyst to adjust his level of "abstraction" depending on the complexity of the case.

5 Conclusion

Static architectures are usually what results from fact-based and source-based architectural design recovery. When the source code clearly reflects the data and control flow in the design, this is ideal, but for large systems whose multi-tier architecture both implements and uses control and data flow, a static design recovery is not sufficient.

We have suggested a means for clarifying the static architecture by adding 'just enough' dynamically-recovered information. By basing the dynamic data recovery (instrumentation) upon a set of pivotal functions derived *a priori* from the static architecture, we have had some success in limiting the large amounts of data often produced by dynamic analyses.

References

[Holt] R. Holt, "Software architecture as a shared mental model", *Proc. ASERC Workshop on Software Architecture*, Edmonton, 2001.

[Bowman] T. Bowman *et al*, "Linux as a case study: its extracted architecture". *Proc. ICSE* (1999).

[Bull] I. Bull. *Abstraction Patterns for Reverse Engineering*. MSc thesis. Dep. Comp. Sci., U. Waterloo. (2002).

[Finnigan] P. J. Finnigan *et alii multi*, "The Portable Bookshelf". *IBM Systems J. 36(4)*. (1997).

[Fowler] M. Fowler, "Design – who needs an architect?", *IEEE Software 20:11-13* (2003).

[GXL] R. Holt *et al*, "GXL: A graph-based standard exchange format for reengineering". *J. Sci. Comp. Prog.*, accepted (2006).

[Kruchten] P. Kruchten. "The 4+1 view model of architecture", IEEE Software, Nov 1995.

[Kruchten] P. Kruchten. "The 4+1 view model of architecture", IEEE Software, Nov 1995.

[Lin] Y. Lin, R. C. Holt, Andrew Malton. "Completeness of a fact extractor". *Proc. 10th WCRE* (2003).

[Malton] A. J. Malton and R. C. Holt. "Boxology of NBA and TA: A basis for understanding software architecture", Proc. 12th WCRE (2005).

[Malton] A. J. Malton and R. C. Holt. "Boxology of NBA and TA: A basis for understanding software architecture", Proc. 12th WCRE (2005).

[Murphy] G. C. Murphy and D. Notkin, "Reengineering with reflexion models: a case study", IEEE Computer 17(2), (1997).

[Tu] Q. Tu and M. Godfrey. "The build-time architectural view". *Proc. ICSM*, Florence, 2001.

[Walkinshaw] N. Walkinshaw, M. Roper, M. Wood. "Understanding object-oriented source code from the behavioural perspective". In *Proc. 13th International Workshop on Program Comprehension.* (2005)

An Approach to Program Comprehension through Reverse Engineering of Complementary Software Views

Aline Vasconcelos[1,2]
[2]*Computing Department - CEFET Campos Brazil*
apires@cefetcampos.br

Rafael Cepêda[1]
[1]*Systems Engineering and Computer Science Program COPPE/UFR J, Brazil*
rcepeda@cos.ufrj.br

Cláudia Werner[1]
[1]*Systems Engineering and Computer Science Program COPPE/UFR J, Brazil*
werner@cos.ufrj.br

Abstract

This paper presents an approach to the reverse engineering of application dynamic models from Java programs. These models are represented through UML sequence diagrams reconstructed at varying levels of abstraction, i.e. class level and architectural level. The diagrams are associated with application use-case scenarios and are extracted in a reuse based software development environment, complementing the views already existent for an application. The main goal is to support program comprehension through complementary application views. In order to support the proposed approach, a set of tools is being developed.

1. Introduction

In order to comprehend how the functionalities are implemented in a system and to localize the impacts of a maintenance in the code, its behavioral models are essential. In object oriented systems these models are of particular interest, due to specific characteristics of that paradigm such as late binding and polymorphism that make it difficult to comprehend the behavior of the system by means of a code analysis. Moreover, due to these particularities, dynamic models for an object oriented system must be extracted by means of dynamic analysis.

Many approaches to the extraction of dynamic models for object oriented systems based on dynamic analysis have already been proposed [2] [3] [4] [7]. One problem faced when dealing with dynamic analysis of object oriented applications is the volume of information generated in the execution traces. The existent approaches apply a set of techniques in order to reduce this volume of information, such as filtering, pattern matching, sampling, slicing, information hiding etc. In [5], a summary and a discussion of such techniques are presented.

In this work, we employ three techniques to deal with the trace explosion problem: filtering, slicing, and information hiding. Filtering allows the selection of the desired packages or classes to be monitored for traces collection, being optional for the user. Traces are sliced by use-case scenarios and message depth level. Information hiding is achieved through varying levels of abstraction: traces can be extracted at the class level or at the architectural level, hiding the messages exchanged among classes of the same subsystem.

The main goal of the proposed approach is to support program comprehension for maintenance and reuse purposes. We argue that in order to effectively support program comprehension, an approach needs to integrate different software views. Some approaches, such as [3], integrate static and dynamic views in the recovery process, allowing the user to get a broader understanding of the application. In this work we propose the extraction of some software architectural views of the "4+1 View" model [1]. Architectural views are essential to allow the comprehension of large-scale software. Following the "4+1 View" model to architecture description, the proposed approach is able to recover the process and scenario views of the architecture. This last one is achieved through the association of sequence diagrams to use-case scenarios of the application.

In order to detail the architectural models, the dynamic diagrams can also be extracted at the class level. The approach is integrated into a reuse based software development environment, named Odyssey [8], and the extracted views complement the static view of the application that can be reconstructed by a tool set already integrated to the environment [6].

The rest of the paper is organized as follows: section 2 presents the proposed approach to the reconstruction of dynamic software views; section 3 presents a usage example; section 4 presents related work; and, section 5 presents some conclusions and future work.

2. Extracting dynamic software views

The proposed approach to the extraction of dynamic software views is divided into two steps: execution traces collection and reconstruction of sequence diagrams in the Odyssey environment. Two tools were developed to support these activities: the Tracer tool and the Phoenix tool, each one being responsible for one part of the process. Figure 1 depicts the integration schema between the tools and the Odyssey environment.

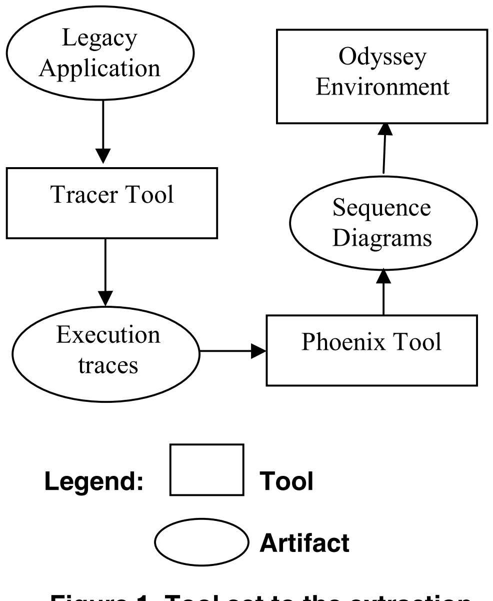
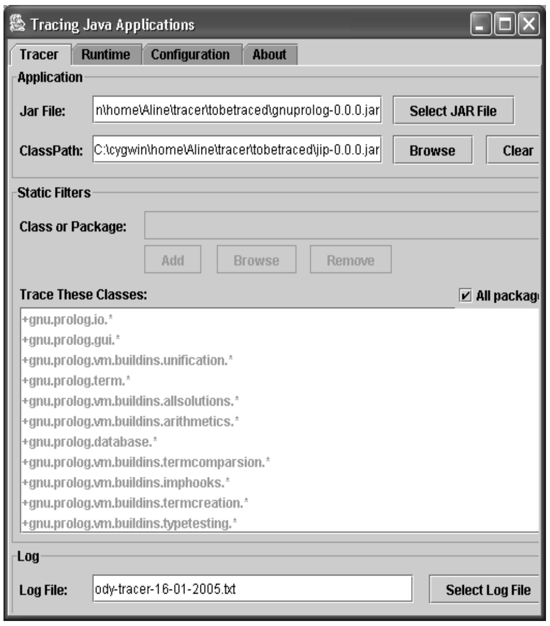


Figure 1. Tool set to the extraction of dynamic software views.

2.1. Execution traces collection

The Tracer tool, shown in figure 2, uses aspect technology in order to instrument the bytecode of Java applications. It uses AspectJ [9], an extension of the Java language to support aspects. In fact, many techniques can be applied to the collection of event traces from application programs, such as: instrumentation of the source code, instrumentation of the object code, instrumentation of the running environment, or running the system under the control of a debugger or profiler. Our use of aspects is motivated by the fact that they are not intrusive in the source code and because they allow parametrization, such as the selection of the classes to be monitored. Moreover, this approach is general to any Java application and the user just needs to inform the jar file and classpath of the application (see figure 2).



Application

Jar File: n\home\Aline\tracer\tobetraced\gnuprolog-0.0.0.jar

ClassPath: C:\cygwin\home\Aline\tracer\tobetraced\jip-0.0.0.jar

Static Filters

Class or Package:

Trace These Classes: ✓ All packag

```
+gnu.prolog.io.*
+gnu.prolog.gui.*
+gnu.prolog.vm.buildins.unification.*
+gnu.prolog.term.*
+gnu.prolog.vm.buildins.allsolutions.*
+gnu.prolog.vm.buildins.arithmetics.*
+gnu.prolog.database.*
+gnu.prolog.vm.buildins.termcomparsion.*
+gnu.prolog.vm.buildins.imphooks.*
+gnu.prolog.vm.buildins.termcreation.*
+gnu.prolog.vm.buildins.typetesting.*
```

Log

Log File: ody-tracer-16-01-2005.txt

Figure 2. The Tracer tool.

The Tracer tool allows the user to select the classes to be monitored. The advantage of this filter is to eliminate from the execution traces messages to the libraries used by the application, such as the Java API. The tool generates in its output an XML file containing the methods invoked at runtime along with the executing thread, class and instance, ordered by method execution, as shown in figure 3. Methods are indented according to their calling hierarchy. In the trace in figure 3, the method m-1 from class package1.A in thread T-1 calls the method m-2 from class package2.B of thread T-1. Therefore, method m-2 is indented in relation to method m-1.

The tag "label" in the trace file indicates the use-case scenario that is being executed and is informed by the user at runtime. The user can select different use-case scenarios to execute according to his maintenance requirements. The same execution trace can contain many use-case scenarios delimited by tags "label". In order to delimit these use-case scenarios, the Tracer tool allows the user to enable and disable the data collection at runtime.

```
<?xml version="1.0" encoding="UTF-8" ?>
<trace>
<Label name="Use case 1 – Scenario 1 ">
    <Method  class="package1.A" instance="@a7552"
                  method="m1"thread="T-1"timestamp=
                  ”01/04/200512:00:01">
        <Method class="package2.B"
                    instance="@14db8d"  method="m2 “
                    thread="T-1"
                    timestamp=”01/04/200512:00:01”/>
    </Method>
……………..
```

Figure 3. A sample of an execution trace.

2.2. Sequence diagrams extraction

The Phoenix tool reads the execution traces in XML and generates the corresponding sequence diagrams in the Odyssey environment. The diagrams are extracted and associated with use-cases. During the extraction, if the use-case defined by the tag label doesn't exist yet in the application model of the environment, it is created by the Phoenix tool. However, the object types, i.e. classes, must already exist in the static model of Odyssey in order to allow the extraction. It is important to state that at this moment each use-case scenario is being represented by a distinct use-case in the Odyssey environment.

During the extraction process, the user can select the thread from which to read the message calls, if he doesn't want to represent all threads in the same diagram, and the method in which to start the diagram. If a method isn't selected, the extraction will start in the first method of the selected thread or in the first method of the first thread.

The user can also select the depth level of the message call in the diagram, as shown in figures 5 and 6. Supposing we have a method call sequence as shown in figure 4, the diagram in figure 5 depicts the correspondent sequence diagram extracted until the message level 3, and figure 6 depicts another diagram representing the detailing of the message mE.

The Phoenix tool is also able to extract behavioral diagrams at different levels of abstraction (i.e. class level and architectural level). At the class level, instances of the same class are replaced by only one representation of that class. This is done in order to reduce the volume of information in the sequence diagrams and is useful to comprehend which classes implement which application functionalities. The architectural level, on the other hand, allows a considerable reduction of the diagrams size, since messages exchanged between classes of the same subsystem are encapsulated in this subsystem and not shown in the diagram. Therefore, the developer can get the mapping from system functionalities to architectural elements, that must already exist in the static structural view of the Odyssey environment to allow the extraction.

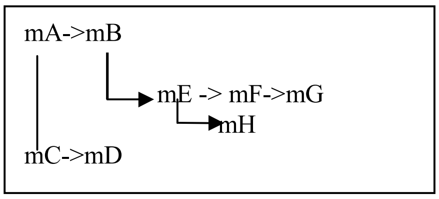


Figure 4. A sample call sequence.

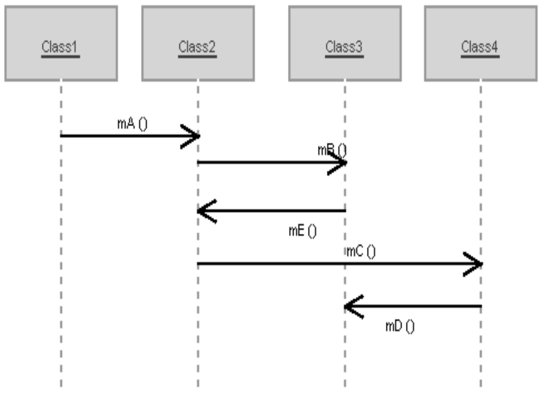


Figure 5. A sequence diagram until level 3.

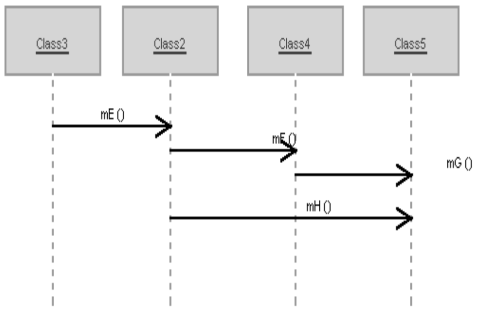


Figure 6. Detailing of message mE.

3. A usage example

We have been testing the approach in the extraction of dynamic diagrams of the Odyssey environment itself and from some other applications. Here, due to space limitations, we present a usage example for a use-case of a MDA (model driven architecture) transformation tool.

Using the Tracer tool, we generated the trace file for the "Export Model" use-case. Then, with the Phoenix tool, a sequence diagram until level 3 was extracted in the Odyssey environment, as shown in figure 8.

automatically found out that: MDAGui and MDAFacade classes belong to the same architectural element, called mda; RepositoryManager class belongs to repository; and FileUtils belongs to utils. Thus, when extracting the same sequence diagram at the architectural level, the messages between MDAGui and MDAFacade were automatically encapsulated, generating a higher-level diagram, as presented in figure 9.

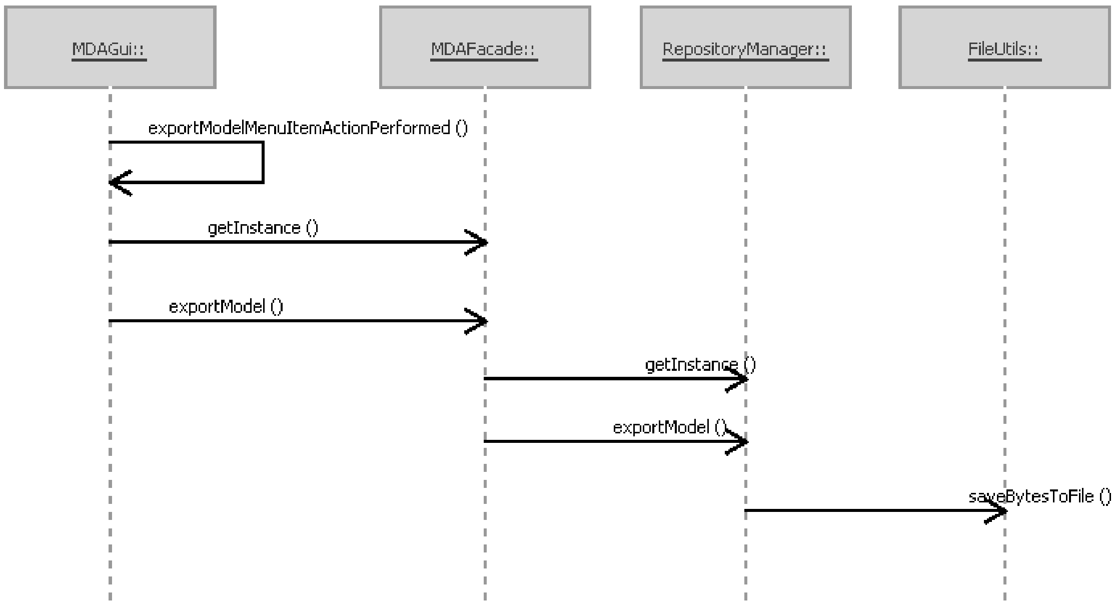


Figure 8. Diagram extracted for the MDA-Tool at the class level.

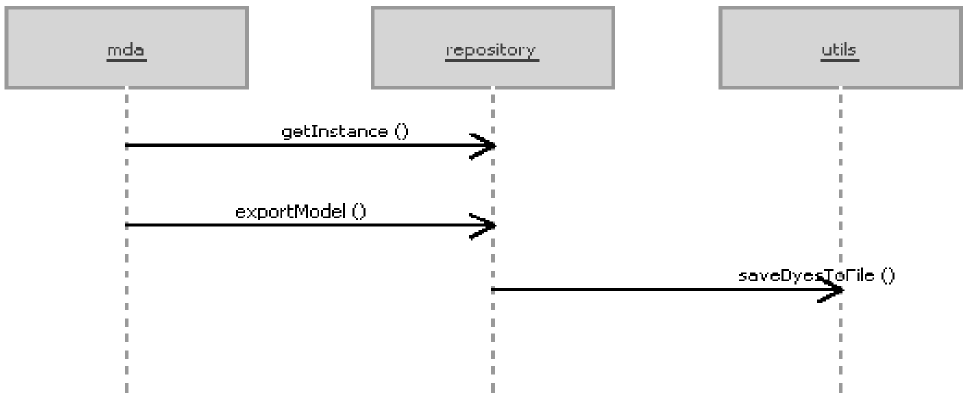


Figure 9. Diagram extracted for the MDA-tool at the architectural level.

Based on the architectural elements extracted for the MDA-Tool using the software architecture approach integrated to Odyssey [6], the Phoenix tool

Therefore, it can be realized that in the Odyssey environment the developer has the opportunity to navigate through different application views (i.e. static and dynamic) at different levels of abstraction (i.e. architectural and low-level design).

4. Related work

In [2], traces are reduced through the detection of interaction patterns, which are abstracted to high-level scenarios. Moreover, to facilitate the visualization they use an information mural, which allows the visualization of a whole trace in a compact form, where areas of repeated sequences of events are emphasized. In [3], Riva and Rodriguez synchronize static and dynamic views in the architecture reconstruction. Manipulations made in one view are reflected in the other. On the other hand, in [4], only a dynamic view is recovered, but they convey a more rich set of technical information in the diagrams, such as conditions and iterations since data collection is performed through the instrumentation of the source code. In [7], a method for the extraction of compact sequence diagrams is proposed, through the identification and reduction of repetitive call sequences and recursive call sequences.

In this work, the goal is to support program comprehension through the extraction of complementary application views in a software development environment.

5. Conclusions and Future Work

The main contribution of our work is the generation of integrated software views in a software development environment. Different from other approaches, that are more concerned with dealing with trace volume and visualization, we are concerned with supporting system comprehension through complementary views, i.e. logical, process and scenario views, at different levels of abstraction. The scenario view allows the mapping from system functionalities to source code and architectural entities. The logical view is extracted by a tool set already existent in the Odyssey environment, complementing the extracted dynamic views. Therefore, in our approach a richer set of views for an application is extracted.

Since Odyssey is a reuse environment, the generated artifacts can later be reused in a domain engineering process.

As future work we intend to synchronize the manipulations made to the static and dynamic views, to identify loops in the execution traces and to generate diagrams that can reflect the behavior of individual objects of a class (i.e. statecharts). At this moment, the sequence diagrams depict message exchange at the class level. This prevents analyzing the behavior of individual objects of a class. Moreover, each use-case scenario must be represented as a case description of a use-case and not as a distinct use-case in the Odyssey.

6. References

[1] P.B. Kruchten, *The 4+1 View Model of Architecture*, IEEE Software, Vol. 12, Number 6, November, 1995, pp. 42-50.

[2] D. Jerding, and S. Rugaber, "Using Visualization for Architecture Localization and Extraction", *In Proc.4th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, October, 1997, pp. 56-65.

[3] C. Riva, and J.V. Rodriguez, "Combining Static and Dynamic Views for Architecture Reconstruction", *Sixth European Conference on Software Maintenance and Reengineering*, Budapest, Hungary, March, 2002, pp. 47-56.

[4] L.C. Briand, Y. Labiche, and Y. Miao, "Towards the Reverse Engineering of UML Sequence Diagrams", *In Proc 10th IEEE Working Conference on Reverse Engineering,* Victoria, Canada, November, 2003, pp. 57-66.

[5] A. Hamou-Lhadj, and T.C. Lethbridge, "A Survey of Trace Exploration Tools and Tehcniques", *In Proc. of the 2004 Conference of the Centre for Advanced Studies and Collaborative Research,* Markham, Ontario, Canada, October, 2004, pp. 42-55.

[6] A.P.V. Vasconcelos, and C.M.L. Werner, "Software Architecture Recovery based on Dynamic Analysis", *XVIII Brazilian Symposium on Software Engineering, Workshop on Modern Software Maintenance*, 2, Brasilia, DF, Brazil, October, 2004.

[7] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting Sequence Diagram from Execution Trace of Java Program", *In Proc. International Workshop on Principles of Software Evolution (IWPSE'05),* Lisbon, Portugal, September, 2005, pp. 148-151.

[8] Odyssey Project, in: http://reuse.cos.ufrj.br/odyssey.

[9] Eclipse org., in: http://eclipse.org/aspectj/, AspectJ 1.5.0.