Program Comprehension and Software Reengineering

Dagstuhl Seminar, March 9th-13th, 1998

Hausi Müller	Thomas Reps	Gregor Snelting
University of Victoria	*University of Wisconsin*	*Universität Braunschweig*
Canada	*USA*	*Germany*
hausi@csr.uvic.ca	*reps@cs.wisc.edu*	*snelting@ips.cs.tu-bs.de*

Analyzing old software systems has become an important topic in software technology. There are billions of lines of legacy code which constitute substantial corporate assets. Legacy systems have been subject to countless modifications and enhancements and, hence, software entropy has typically increased steadily over the years. If these systems are not refurbished they might die of old age—and the knowledge embodied in these systems will be lost forever.

As a first step in "software geriatrics" one usually tries to understand the old system using program understanding or program comprehension techniques. In a second step, one reconstructs abstract concepts (e.g., the system architecture, business rules) from the source code, the documentation, and corporate knowledge; this is called software reverse engineering. Given an abstract representation of the system, one can then re-implement the system. This forward engineering step ranges from fully automatic approaches to manual reimplementations including restructuring techniques, formal transformations, injecting component technologies, replacing old user interface or database technology. The process of moving from an old legacy system to a new implementation is called software reengineering.

It was the aim of this seminar to bring together researchers who are active in the areas of program comprehension and software reengineering regardless of their particular approaches and research avenues. However, one of the areas of concentration for this seminar was slicing technology which is an important technique for software understanding and maintenance activities. Another topic of increased interest have been empirical studies for software reengineering. Mathematical concept analysis gained some attention as a new framework for program understanding.

Several talks were accompanied by system demonstrations, giving participants first-hand experience of new analysis and reengineering technology. A panel session compiled a list of open problems, both technical and methodological. The traditional Dagstuhl walk offered an opportunity for topological comprehension and map reengineering.

We are grateful to the participants who made the seminar an exciting week. We also acknowledge the financial support provided by the European Union in the scope of the TMR program. As always in Dagstuhl, the staff was doing a terrific job looking after us and everything surrounding the seminar.

Hausi Müller
Thomas Reps
Gregor Snelting

Abstracts

Coping with Software Change Using Information Transparency

William Griswold, University of California, San Diego

Designs are frequently unsuccessful in designing for change using traditional modularity techniques. It is difficult to anticipate exactly how technology will advance, standards will arise, and features from competitor's products will influence future features. Market pressures dictate that most time be invested in timely release of the current product, not in accomodating future changes.

One way to increase the coverage of relevant design decisions is to use a design principle called *information transparency*: Write code such that all uses of an exposed design decision are easily visible to a programmer using available tools. The coding conventions include techniques like variable naming conventions, code formatting, and encoding software architecture into the program source. As a consequence of using such techniques, a programmer can use a searching tool like `grep` to view all the related objects together, creating locality out of similarity.

Flow-Insensitive Pointer Analysis

Susan Horwitz, University of Wisconsin, Madison

Most static analysis rely on knowing what objects are used and defined at each point in the program. In a language with pointers, determining this information can be non-trivial. Lars Andersen defined a flow-insensitive algorithm for computing points-to information that is $O(n^3)$ in the worst case. More recently, Bjarne Steensgard gave another flow-insensitive algorithm that is faster (essentially $O(n)$), but less precise (computes larger points-to sets than Andersen's algorithm).

In the talk, we first define a new points-to analysis algorithm that can be "tuned" to provide resolutions that fall all along the spectrum from Steensgard to Andersen (both in terms of runtime and precision). We then present the results of experiments that measure how the algorithms perform in practice, measuring both the "direct" results (sizes of points-to sets) as well as "transitive" results (the size of the sets computed by GMOD), live, and truly-live dataflow analysis, using the results of the different pointer analyses. We find that (as expected) better points-to analysis leads to better dataflow analysis results, also (surprisingly) that, at least for the harder dataflow problems, the extra time required for the better points-to analysis is not then made up for by a *decrease* in the time required for the subsequent dataflow analysis.

Class Hierarchy Specialization[1]

Frank Tip, IBM. T.J. Watson Research Center[2]

Class libraries are typically designed with an emphasis on generality and extensibility. An application that uses a library typically exercises only part of the libraries functionality. As a result, objects created by the application may contain unused (user-defined or compiler-generated) members. We present an algorithm for *specializing* a class hierarchy with respect to its usage in a program P. That is, the algorithm analyzes the member access patterns for P's variables, and creates distinct classes for variables that access different members. The algorithm addresses the inheritance mechanisms of C++ in their full generality, including multiple inheritance and virtual (shared) inheritance. Class hierarchy specialization reduces object size, and may be viewed as a space optimization. However, execution time may also be reduced through reduced object creation and destruction time, and caching and paging effects. Class hierarchy specialization may also create new opportunities for existing optimizations. In addition, we believe that specialization may be useful in tools for software maintenance and understanding.

[1] Paper appeared in the proceedings of OOPSLA'97

[2] Joint work with Peter Sweeney

Reengineering Class Hierarchies Using Concept Analysis

Gregor Snelting, Technische Universität Braunschweig[3]

Class hierarchies in legacy code may be imperfect. For example, a member may be located in a class that does not need it, indicating that it may be eliminated or moved to a different class, or different instances of a given class C may access different subsets of C's members, an indication that it might be appropriate to split C into different classes. We present an approach for detecting such design problems based on class hierarchy specialization and concept analysis. Examples demonstrate that our technique can provide remarkable insight into member access patterns of old C++ programs.

Program Analysis via Graph Reachability

Thomas Reps, University of Wisconsin, Madison[4]

This talk describes how a number of program analysis problems are all examples—when viewed in the right way—of a certain kind of generalized graph-reachability problem: context-free language reachability (CFL-reachability). In a CFL-reachability problem, we are given (i) a graph in which the edges are labeled with letters from some alphabet and (ii) a context-free language L (given, say, via a grammar). A path p from node s to node t only counts as a valid connection from s to t when the word formed by concatenating (in order) the letters among the edges of p is a word in L. This generalizes ordinary graph reachability in the sense that L serves to filter out certain paths in a graph. A CFL-reachability problem can be solved in time $O(n^3)$, where n is the number of nodes in the graph.

In the talk, I describe how problems such as interprocedural slicing, interprocedural dataflow analysis, and shape analysis (for a language without destructive update) can all be converted into CFL-reachability problems. I also discuss the relationship between CFL-reachability and a certain class of set-constraint problems.

[3] Joint work with Frank Tip

[4] This represents joint work with Susan Horwitz, Mooly Sagiv, Genevieve Rosay, and David Melski.

Slicing Methods for Large Programs[5]

Tibor Gyimothy, Joszef Attila University of Szeged[6]

A method is presented for the computation of the summary edges representing the interprocedural dependences at call sites. The advantage of this method is that the memory requirement can be reduced for large programs. The reason for it is that the algorithm computes summary information for each strongly connected component of program one at a time. Hence only dependence information for one strongly connected component is stored instead of for the whole program. Moreover, the method reduces the number of nodes of the dependence graphs.

VALSOFT – Validation of Measurement System Software: an Application of Slicing and Constraint Solving

Jens Krinke, Technische Universität Braunschweig[7]

We show how to combine program slicing and constraint solving in order to obtain better slice accuracy. The method is used in the VALSOFT slicing system. One particular application is the validation of computer-controlled measurement systems. VALSOFT will be used by the Physikalisch-Technische Bundesanstalt for verification of legally required calibration standards. We describe the VALSOFT slicing system, its architecture and its application. In particular, we describe our fine-grained version of the underlying system dependence graph. We also describe how to generate and to simplify path conditions based on program slices. The technique can indeed increase slice precision and reveal manipulations of the so-called calibration path.

[5] Paper appeared in the proceedings of SEKE'97

[6] Joint work with Istvan Forgacs

[7] Joint work with Gregor Snelting

Towards Dataflow Minimal Slicing

Mark Harman, Goldsmiths College, University of London[8]

Consider this program

```
while i < 3 do
   begin
     if c = 2 then
       begin x := 17, c := 25 end;
     i := i+1
   end
```

what primitive statements and predicates affect the final value of x ? (that is, what is the end slice on $\{x\}$?) Most slicing algorithms (all?) will leave in the assignment to c. This is not a special case; it applies to all data flow equivalent programs. So its true of the program schema

```
while {i} do
  begin
    if {c} then
      begin x := {}   ; c := {}
      end;
    i :=  {i}
  end
```

where the sets denote the variables upon which an expression depends. The question we ask is "at this dataflow level of abstraction is minimal slicing computable?" Weiser asked this question in his PhD thesis, having observed that his algorithm was not minimal in this sense. We believe the answer is "yes". We have an algorithm (`http://www.unl.ac.uk/~11danicics/`) and are working on a proof. The algorithm uses an unconventional approach to data and control flow analysis based upon repeated (but finite) instances of predicate nodes together with their "dependence history".

[8]Joint work with Sebastian Danicic

Experience Building an Industrial-Strenght Program Understanding Tool

John Field, IBM Research[9]

In early 1996, G. Ramalingam and I became involved with designing and implementing a program understanding tool for use in IBM's Cobol tool suite. The functional requirements for the tool were modest (centered around control-flow rationalization and a restricted form of slicing), and our original intent was to use well-understood algorithms in its design. However, we soon discovered that despite the limited goals, Cobol posed technical challenges that to our knowledge have not been previously addressed. In addition to a large number of endearing, but technically inconsequential design quirks, Cobol possesses several distinctly peculiar control-flow and data manipulation constructs. Typical Cobol programs also use certain otherwise unremarked constructs in atypical ways; For example, programs tend to contain vast quantities of global data and frequent use of non-disjoint unions. In this talk, I discuss the original design goals for the tool and describe certain technical challenges posed by Cobol and our approach to solving them. In particular, I describe an efficient algorithm to transform instances Cobol's PERFORM construct to semantically equivalent procedural representations used for program slicing. Further, I describe special techniques for computing interprocedural reaching definitions and constructing pseudo-parameters for the procedural representations computed during PERFORM analysis; these techniques are necessitated by the unusual properties of Cobol data manipulations.

A Model of Change Propagation

Vaclav Rajlich, Wayne State University

Change in programs starts with programmer a specific component. After the change, the component may no longer fit with the next, because it may no longer provide what the other components require, or it may now require different services from the components it depends on. The dependencies that no

[9]Joint work with G. Ramalingam

longer satisfy the require–provide relationships are called inconsistent dependencies, and they may arise whenever a change is made in software. When these inconsistencies are fixed, they may introduce additional inconsistencies, etc. The paper describes a formal model of change propagation, and two specific examples of it: change-and-fix scenario, and top-down scenario.

Software Migration

Hausi Müller, University of Victoria

Software migration is a subset of software reengineering and involves moving existing system to a new platform. Important problems are migrating to object technology from an imperative language, to GUI technology from a text-based user interface; to a network-centric environment from a standalone application, to Year 2000 compliant software. Automation is a key requirement for these processes and wrapping seems to be a promising technology.

CESR is a Canadian Centre for Excellence for Software Engineering Research. The IBM CSER project which involves John Mylopoulos, University of Toronto, Ric Holt and Kostas Kontogiannis, University of Waterloo and my research group, currently investigates how PL/I programs can be automatically and incrementally migrated to C++. The target application is about 300 kloc written in a PL/I derivative. As a pilot project we converted a subsystem of 3000 lines to C++. The resulting subsystem was integrated into the existing PL/I application. Early performance tests revealed that the new subsystem was 50% slower. Simple C++ optimizations were performed resulting in significant speedup and a subsystem that is 5-20% faster than the original code. Using Refine from Reasoning Systems we then built an automated solution to convert the entire system implementing the optimizations.

Task-aware Program Understanding Techniques

Gail C. Murphy, University of British Columbia[10]

Many software engineering tools attempt to fully automate tasks that software engineers must perform on software systems. Even more tools exist to analyze code without any notion of how the analyzed information will be used. In this talk, I argue that there is a useful set of tools and approaches that fall in the middle of this spectrum. These tools are task-aware. Task-aware tools may be more amenable to exploiting partial and approximate information about source. This type of information can help engineers more effectively perform software engineering tasks on large systems within the time constraints placed on the task.

I describe two task-aware techniques we have developed to investigate if an approach of overlaying logical structure on existing source can aid a software engineer in quickly and easily assessing appropriate source information for a task at hand. The software reflexion model technique helps an engineer gain an overall 'gestalt' of the source by using a high-level structural model as a lens through which to summarize the code for a system. This technique has been used to drive an experimental reengineering of the million lines-of-code Microsoft Excel spreadsheet product. The conceptual module technique provides direct support for performing a reengineering task by enabling source-level queries about a desired, rather than the existing, source structure. This technique has been used to help reengineer an over fifty thousand lines-of-code binary decision diagram package prior to the source being parallelized.

The AST Toolkit and ASTLOG

Roger F. Crew, Microsoft Research

The AST Toolkit provides the developer/tester with a C++ application programmer interface to data structures used by the front end of Microsoft's

[10]The reflexion model work is joint with David Notkin (U.Washington) and Kevin Sullivan (U.Virginia). Conceptual modules are joint with Elisa Baniassad (U. of British Colombia)

C/C++ product compiler, specifically the abstract syntax trees (ASTs), symbols (including symbol tables and scopes), and the type descriptions. Given the variations in possible AST structures and compiler idiosyncracies that product groups often take advantage of, it is often important that one have access to the actual structures of the compiler one is using. The toolkit has already been used to solve a variety of elementary program-comprehension, problems and meta-programming (e.g., automatic generation of stub function and thunks) tasks.

In addition to providing a direct C++ API, the toolkit also provides access via a query language, ASTLOG. ASTLOG was inspired by grep/awk-style tools that allow the programmer to locate program artifacts without incurring the overhead of writing an entire C++ application. In contrast with prior such tools, our goal is to provide a pattern language with sufficiently general abstraction/composition facilities so that programmers can write queries/abstractions tailored to specific code bases and re-use them for later works. The language itself is a Prolog variant for which we have written a small, fast interpreter. The execution model, in which terms are treated as patterns to be matched against an implicit current object rather than as simple predicates leads to a "reverse functional" programming style distinct from both the usual relational Prolog style and the usual "forward" style found in Algol-Family languages, one that is well-suited to the particular application of querying ASTs and related structures.

GUPRO - Generic Unit for Program Understanding

Andreas Winter, University of Koblenz

The aim behind GUPRO is to develop an adaptable tool to support program understanding even in a multiple language environment. This adaptability is based on an user-defined conceptual model which defines the internal data-structure of the tool (which could be viewed as an instance of a MetaCARE-tool) and the parsing process into the repository structure. Analysis is done by a source code independent Query-mechanism.

The formal foundation of GUPRO is given by the EER/GRAL-approach on graph based, conceptual modeling.

A presentation of GUPRO including queries to a coarse-grained multi-language conceptual model, a fine grained C-model and the according meta model was given after the talk.

A Range Equivalence Algorithm and its Application to Type Inference

G. Ramalingam, IBM T.J.Watson Research Center[11]

A common program maintenance activity is that of changing the representation/implementation of an abstract type. A well-known example is that of making programs "year 2000 compliant", which requires ensuring that the implementation of the abstract type "YEAR" can adequately distinguish between years belonging to different centuries.

Ideally, appropriate use of "abstract data types" would make such changes easy, requiring appropriate modifications only to the (single) implementation of the abstract daty type. In practice, such changes turn out to be very expensive and time-consuming because of inadequate use of abstractions. In fact, much of the existing legacy code is written in languages such as Cobol that do not provide adequate abstraction facilities.

Consequently, a programmer facing the problem of making such a change needs to find all variables in a program that belong to some abstract type. This talk describes a type inference algorithm that partitions the variables in a program into equivalence classes, where all variables in an equivalence class are likely to have the same abstract type. Our algorithm is particularly suited for languages such as Cobol and PL/I. The primary technical problem that our algorithm solves is that in a Cobol or PL/I program the set of all "logical" variables in a program may not be apparent from the declarative section of the program. For example, what was declared to be a single (unstructured scalar) variable may in fact be a record consisting of a sequence of fields, each with its own abstract type... and this fact has to be inferred from how the variable is used in the program.

Our algorithm is based on an extension of the well-known UNION-FIND data-structure/algorithm that enables us to efficiently create equivalences between "sub-ranges" and map-valued variables.

---

[11] Joint work with J. Field and F. Tip

Software Reengineering: Finding Leverage for Existing Technologies

Dennis Smith, Carnegie Mellon University, Software Engineering Institute

Based on experiences with the analysis and reengineering of large system, we have been focusing on large grain issues of strategy reuse. One particular focus has been the use of legacy system as core assets for the development of product lines, or families of systems. Although significant technical problems exist in this type of migration, a number of successful examples exist, including such companies as Celsina Tech, HP and Motorola. The reengineering issues of most relevance for product line technologies include:

1. Identification of the enterprise wide issues of relevance, such as the organizational goals, project, legacy and target systems, technology and SW engineering.
2. System understanding, including program understanding and architectural extraction.
3. Distributed object technology and wrapping approaches
4. Net centric approaches and levereraging of web technologies.

Our work is reengineering that addresses these issues was described.

Evaluating Software Maintenance Tools for their Support of Program Comprehension

Anneliese von Mayrhauser, Colorado State University

The talk presented deliberations, possibilities and limitations of various approaches to evaluate tool technologies with respect to their support for software understanding. It used two types of maintenance tasks as examples, debugging and enhancement. We compared two static analysis environments and showed that an environment that includes even limited data flow analysis and slicing capabilities has the potential of decreasing necessary comprehension activities by between 19-50% depending on the type of maintenance tasks. Related publication can be found in Proceedings of IEEE Aerospace Conference, March 21-28, 1998, Snowmass, CO.

Approaches to detect abstract data types and abstract state encapsulations

Rainer Koschke, University of Stuttgart

One of the first activities in software architecture recovery is to detect atomic components in the source code. Examples of these are abstract data types and abstract state encapsulations (global state variable or objects). They are atomic in the sense that they consist of routines, variables, and types respectively. They do not have any further subcomponents other than these programming entities. These atomic components are building blocks for larger components and so must be understood first. They are candidates for re-use and in the case of a migration to an object-oriented system they have to be detected before one can take care of the inheritance relationship. Older programming languages do not let the programmer specify them. So, in order to detect them in legacy code certain other relationships have to be considered. Several heuristics were proposed in the literature to detect them. We implemented six of them, namely *Same Module* (Koschke, Girard 1997), *Part Type* (Ogando, Yan, Wilde 1994), *Internal Access* (Yan, Harris, Reubenstein 1994), *Delta IC* (Canfora, Cimible, Munro, 1993), and *Similarity Clustering* (Girard, Koschke, Schied 1997; our enhancement of Schwanke's approach to detect subsystems). In order to compare them quantitatively we asked five software engineers to compile a list of atomic components manually from three C systems (altogether 100,000 LOC). These references were compared with the candidate components by a metric for the detection quality. The results show that *Part Type* and *Similarity Clustering* recover most ADT's and *Same Module* and *Similarity Clustering* most abstract state encapsulations. However, the overall result is that none of the heuristics is sufficient. To improve the results the techniques should be combined, in a post analysis many false positives can be removed, and also dataflow information can be taken into account.

Program Tucking

Arun Lakhotia, Univ. of Southwestern Louisiana[12]

To tuck a set of program statements is to "gather and fold" these statements into a function without changing the external behavior of the system. We present a transformation to tuck non-contiguous program fragments. Tuck has three steps: wedge, split and fold. One first drives a wedge in the code, then splits the wedged code, and then folds the split code. Folding replaces the split code, a single-entry single-exit subgraph with certain constraints, into a function. That tuck does not alter the behavior of the original function follows from the semantics preserving property of the other transformations.

The tuck transformation was developed to aid in program restructuring. The first prototype developed used the transformation to split non-cohesive functions into cohesive functions. We are now developing an interactive environment that provides this transformation as a primitive accessible to a programmer through mouse clicks.

Finding Objects in COBOL Code

Peter Reichelt, GMD (German National Research Center for Computer Science)

We often find a split of paradigms in companies: Here are the COBOL guys, there are the OO-guys programming in C++ or Smalltalk etc. Our aim is to bring together these two worlds. The new COBOL standard including OO stuff will help. In the project ROCOCO (Reengineering for Object-Orientation and Reuse for COBOL Code) (carried out with partners IBM and CAI, funded by BMBF) we develop a tool to find objects in COBOL Code. The existing code can be objectified, but our main goal is to just extract the found objects, and store the resulting classes in a class library. Our hope is that we can do some work of generalization and standardization on these classes, so that they can be offered for reuse in the company. To support that reuse we do not only store the COBOL class, but also data about the structure of the class, so we call it a repository. Our tool is highly user-oriented. The user can do all the needed transformations by clicking around with the

[12]Joint work with Jean-Christophe Deprez

mouse in the program text. But the user will want to use our proposal generator which will create proposals for what part of the program the user may want to select.

Analysis of Software Variants

Christian Lindig, TU Braunschweig

Software comes in variants because computer platforms are so diverse. When this diversity can not be encapsulated into modules it gets into the actual source files. Source file preprocessing then creates a variant for each platform. Under some simplifying assumptions over the C preprocessor (CPP) all variants that can be generated from a specific source file using the CPP can be efficiently computed. The technique for this is formal concept analysis. Formal concept analysis is an algebraic theory for binary relations. Its main theorem states that there exists a lattice of so-called concepts for every binary relation. The idea to use it for the analysis is to record the dependencies of source code segments on CPP expressions in a binary relation. Then concept analysis can be used to analyze this relation. Each concept of the resulting concept lattice describes a variant and thus this lattice is called the variant lattice of the original source. Besides that a concept describes a canonical way how to generate the actual variant from the actual source. The original expressions that describe all variants may contain redundancies. These redundancies also show up in the concept lattice. After they are detected there they can re removed from the original source. It is guaranteed that this will not lead to a loss of variants.

Rewriting "poor" (design) patterns by "good" (design) patterns

A. Zündorf, U. Paderborn[13]

Gamma and his "Gang of 4" proposed a number of "good" solutions to frequently recurring problems. Along with the "good" solutions they describe "poor", i.e. naive solutions to these problems and why these are faulty. Our

[13]Joint work with J. Jahnke and W. Sch"afer

goal is to employ program analysis techniques for detecting poor solutions of a problem and rewriting them to good solutions. We consider this as an interaction engineering task and propose to support this with an 'CARE' environment, driven by a high level (reengineering) process description, i.e. Generic Fuzzy Reasoning, Nets + Petri Nets + Programmed Crash Rewriting Rules.

Classification and Retrieval of Software Components using Semantic Nets

Hans-Jürgen Steffens, FH Kaiserslautern - Standort Zweibrücken

The possibilities of a semantic net for classifying SM entities in a repository are discussed. In comparison to the complete KL-ONE language only a small number of features are used to construct the net: starting with a fixed set of undefined concepts and a fixed set of binary relations new concepts are defined recursively by mapping a given concept C and relation n to a new intermediate concept $op(r, C)$. Thus starting with two concepts C_1 and C_2 a new concept "C_1 and $op(r, C_2)$" finally is defined which is a subconcept of C_1 and a "side concept" of C_2. Thus we have introduced an explicit ISA-link between "(C_1 and $op(r, C_2)$)" and "C_1". During construction of the net implicit ISA-links emerge in addition and should be detected by a "classifier". When we restrict to the above rules the classifier is computable but may be np-hard, when we choose $op = csome$ instead of $op = all$. Using *csome* we have more expressive power, but our special application may justify using "all", thus having a classifier of lower complexity.

Analysis and Conversion Tools for Application Software Reengineering to EMU

Rainer Gimnich, IBM Scientific Center, Heidelberg and IBM EMU Transition Services, Stuttgart

The European Economic and Monetary Union (EMU) will be effective from 1st January 1999 and probably include 11 member states at the beginning. Over a transition period of 3 years both Euro and each national currency

(NC) may be used and need to be dealt with, also by business partners outside Europe. Probably on January 1st 2002, the Euro bills and coins will be introduced to replace the NC money physically. This time table, along with strict EU regulations to guide the transition, lead to wide-ranging and technically challenging software reengineering tasks which are comparable to the year 2000 (Y2K) transition. Though EMU entails only financial processing and data, the analysis and conversion tasks are harder to solve than in Y2K projects.

We approach the EMU problem area by a dedicated methodology called IBM EuroPath, which is supported in IBM's project management tool world-wide. EuroPath accounts for both the business aspects and the IT aspects of the transition. From this methodology, we derive the tool requirements for each phase and consider existing reengineering technologies to meet these requirements for instance a "memory-level" dataflow analyzer for COBOL and PL/I, with built-in heuristics of amount field propagation, will be used during detailed analysis. The EMU tools portfolio currently consists of some 25 tools, subset of these are chosen to best meet individual project needs.

The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem

Thomas Reps, University of Wisconsin[14]

A path profile is a finite, easily obtainable characterization of a program's execution on a dataset, and provides a behavior signature—a kind of spectrum—for a run of the program. When different runs of a program produce different path spectra, the spectral differences can be used to identify paths in the program along which control diverges in the different runs. By choosing input datasets to hold all factors constant except one, any such divergence can be attributed to this factor. The point of divergence itself may not be the cause of the underlying problem, but provides a starting place for a programmer to begin his exploration. In the talk, I describe how this idea can be applied to the Year 2000 problem: In this case, the input datasets should be chosen to keep all factors constant except the (ranges of) dates that appear. Applications to other software-maintenance problems are also described.

[14]Joint work with Manuvir Das, Tom Ball and Jim Larus

Demos

Demos supporting talks were presented by R. Crew, J. Krinke, P. Reichelt, A. Winter/J. Ebert, A. Zündorf/W. Schäfer.

In addition, the following independent demos were given:

Demonstration of a Prototype Commercial Slicing Tool

Tim Teitelbaum, GrammaTech, Inc. (and Cornell University)

GrammaTech is commercializing the University of Wisconsin precise interprocedural slicing technology developed over the past decade by Reps and Horwitz. It will offer this technology in two forms: an end-user understanding tool, and a collection of components to be integrated into the tools of others. The prototype end-user tool provides forward and backward slicing, chopping, and immediate predecessor/successor information by suitable highlighting on program text and various summary information thereof.

Demonstration of Serving ASTs with the Synthesizer Generation

Tim Teitelbaum, Cornell University and GammaTech, Inc.

Each edition/interface generated by the Synthesizer Generator (SG) represents edit buffers as pretty printed attributed abstract syntax trees (ASTs). These ASTs are available to end users in Scheme, the SG's scripting language. Constructions and destructions on these terms are dynamically type checked, and attributes on these terms are updated incrementally upon mutations (unless disabled). End users of generated tools, e.g., Ada-ASSURED, use computations on ASTs in lieu of the text-oriented manipulations of languages such as PERC.

IBM Visual Age for Cobol Professional Redeveloper (I) Menagerie: A Prototype slicing, symbolic analysis, and debugging tool (II)

John Field, IBM Research[15]

In (I), I demonstrate IBM's Cobol program understanding tool. The tool provides efficient data dependence slicing, a code browser with various navigation facilities, and a graphical view of rationalized control flow.

In (II), I demonstrate the facilities of a prototype tool developed at IBM's Watson Research Center. The tool operates by translating the program source (written in a subset of the C language) to an intermediate representation called PIM. PIM has an accompanying equational logic, a subset of which provides an operational semantics. By normalizing PIM graphs using term graph rewriting, the graph may be simplified to a canonical form. Using a technique called dynamic dependence tracking, a slice can be computed by traversing the canonical form graph, which has been annotated with origin information during rewriting. The normalized graph can also be displayed in a form that simplifies the semantics of the original source to aid program understanding.

Open Problems

An evening session was devoted to open problems. The participants collected the following list of open problems, which was edited by John Field:

Conceptual

- How to push the world to use languages for which it is easy to obtain useful analysis? (J. Krinke)
- How can we package analysis tools so as to be useful to non-experts (queries and results must be intuitive)? (J. Krinke)

[15] Joint work with F.Tip and G.Ramalingam

- What is the role of domain-specific type information in re-engineering? (A. Goldberg)
- Infrastructure/packaging issue: - exchange formats (both textual and in-core) - scripting/query languages for computing on intermediate program representations (H. Mueller)
- Replacement of `#ifdef`s for version control in the C language (S. Horwitz)
- Design a "sane", analyzable preprocessor for the C language (M. Ernst)
- Can design information or user assertions be used to feed into and improve program analysis (E. Ploedereder)
- Define a taxonomy or vision of tasks, scenarios and corresponding information-gathering needs (D. Notkin)
- Can dynamically-gathered profile information be used for program understanding (C. Lindig)

Experimental

- Do we recoup costs of early analysis phases (e.g., pointer analysis) when analysis phases are solved in a demand-driven fashion (T. Reps)
- Why are slices large? Is it pointers, array usage, infeasible paths? (A. Goldberg)
- How useful are data-dependence slices? (J. Field)
- How useful are non-conservative analyses? (G. Ramalingam)
- To what extent does static analysis really reflect what goes on at runtime? (T. Reps/J. Krinke)
- Can we find better experimental benchmarks or testbeds for programming tools (e.g., Netscape, Emacs) (H. Mueller)

Algorithms

- Can we have an algorithm/framework for program analysis that exploits adaptive granularity (e.g., that matches "effort" to program region (ála multi-grid finite-element analysis) (G. Murphy/T. Reps)?
- Exploit synergistic integration of analysis results (from different analyses) (G. Snelting)
- Cheap ways of obtaining analysis results when the program artifact changes, e.g., eagerly/lazily (E. Ploedereder)
- How can "anticipatory" or "speculative" analyses be used in programming tools? (F. Tip)
- How can probabilistic algorithms be used in programming tools? (T. Reps)
- How can heuristic algorithms be used in programming tools? (M. Harman)
- Are there useful algorithms for recovering design information from legacy code? (J. Ebert)
- Distilling slices by recognizing cliches or design patterns within them (E. Ploedereder)
- Can CFL-reachability be solved in less than cubic time? If so, by a practical algorithm? (T. Reps)
- Can practical demand-driven pointer analysis algorithms be developed? (S. Horwitz)
- Can flow-sensitive analysis be engineered to apply to greater than 1M line programs? (J. Field)
- How can we do whole-program analysis on programs when the information does not fit in core? (F. Tip)
- What analysis techniques are useful in supporting program design (e.g., can concept analysis be used for object-oriented design)? (T. Teitelbaum/ D. Notkin)

CALL FOR PROBLEMS: nPPPA – n Pathological Problems in Program Analysis[16]

Jens Krinke, University of Braunschweig

During the “Open Problems” session of the workshop the need for benchmarks or testbeds for Program analysis tools became obvious. On the other side, some interesting examples were presented during the workshop which represents unsolved or hard-to-solve problems. There are many more of those problems—both users and developers of program analysis tools discover 'interesting' examples from time to time. These examples often get lost again, as nobody collects them. Therefore we call the developers and users of program analysis to submit their problems to `krinke@ips.cs.tu-bs.de`.

Requests for the actual list of problems may use the same address.

[16] At this time, $n \geq 5$

Taxonomy of participants and their interests

The following concept lattice was generated from a boolean table which encoded the interests of the participants. The lattice reveals a hierarchical structure of both participants and interests. A person x is interested in topic y, iff x appears below y in the lattice. Suprema factor out common interests, infima display multi-interested participants.