Using Program Analysis Infrastructure for Software Maintenance

Jens Krinke, Mirko Streckenbach, Maximilian Störzer, Christian Hammer
Lehrstuhl Softwaresysteme
Universität Passau
Passau, Germany

March 2003

Abstract

Enabling the reuse of available techniques and tools for software maintenance is a major topic. However, research focuses mostly on two topics: parsing and tool interoperability. In the future, more sophisticated approaches to maintenance will be needed and dataflow analysis has to be used. As building dataflow analyzers for real languages is expensive, we must start to provide reusable dataflow analysis infrastructures for software maintenance. This paper first reports on our experience in building program analysis based maintenance tools. From that perspective, we formulate specific requirements for reusable program analysis infrastructures and then take a look at some program analysis infrastructures from compiler optimization research to evaluate if they are (re)usable for software maintenance tools.

1 Introduction

Most software maintenance tasks of today are solvable economically by using text- or syntax-based approaches. Even simple tools like *grep*, *sed* or small *perl* scripts can often be much more helpful than a full-blown parse tree transformation system. As present systems turn into legacy systems, renovators will have to analyze programming languages of today with their highly increased complexity. Also, the maintenance tasks themselves will have increased complexity. Many of those tasks already require complex dataflow analysis, examples are program slicing [Tip95b], class hierarchy restructuring [ST00], construction of the object and object relation diagram [TP02, MRR02], and all kinds of advanced refactorings. When only guaranteed semantic preserving transformations are applied, no expensive regression test are needed.

The rest of this section will focus on four problems of program analysis for software maintenance. The next section presents our experience while building some software maintenance tools. In section three, specific requirements on program analysis infrastructure are presented and some available infrastructures are compared against them in section four. The last section presents drawn conclusions.

1.1 Parsing

For compiler construction the problem of parsing seems solved since the time yacc and lex exist. For software maintenance, parsing is still a challenge. Some problems are:

- The programming languages of systems to be analyzed are dating from pre-lex/yacc times and are complicated to parse anyway, maybe because they are not LALR(1).
- Grammars may be commercial assets and may not be available.
- There are many variants of languages due to different standards or vendor extensions; vendor dialects are usually not well documented.
- Systems under maintenance are never clean, e.g. use preprocessing or embedded languages like COBOL with SQL statements.
- Maintenance tasks sometimes require to analyze systems before preprocessing, making parsing further complicated.
- Systems use various languages and dialects together.

When we cannot parse languages how can we even build simple infrastructures like parse trees? Specific parsing techniques have been developed and parsing is still considered a key issue in software maintenance. See [vdBSV98, vDKV99] for an overview on those aspects.

1.2 Storing and Analyzing the Data

Traditional tools for compiler construction are mostly event driven and seldom contain support even for common data-structures like symbol tables. Besides that collected information cannot be held in core memory for large systems, the results of expensive analyses should be saved to reduce recomputation. Therefore most maintenance tools store the information in databases. Often the maintenance tasks are not adequate for standard databases and specific repositories have been developed, for example GUPRO/GREQL [Kam98], Jupiter [CC01] or GRAS [KSW95].

After the data has been collected, it must be prepared for further use. An often used approach is to just generate cross reference information. This can be presented directly to the engineer for better program understanding or can be used for other analyses like clustering [Lak97] or concept analysis [Sne00].

1.3 Dataflow Analysis

There is already a long history of tools for software maintenance that uses dataflow analysis[1]. Normally, those tools are research prototypes just for a specific language.

[1]We use the term dataflow analysis instead of program analysis [NNH99] to make it clear that we don't consider simple techniques. We also include constraint based analysis, abstract interpretation or type systems.

They use hand-crafted dataflow analyses and are beyond application to real world programs. For example, consider program slicing [Tip95b], which require control- and dataflow analysis. Until now, such tools are only available as research prototypes or for languages which only require simple dataflow analysis. We are only aware of one industrial-strength program slicer: *CodeSurfer* [AT01] for C which is stemming from the Wisconsin Program Slicing Tool [HRB90] and is a standalone tool.

The need of specific dataflow analysis support for software maintenance tools has already been recognized [vdBKV97]. Especially *generic* support is needed, where the language is just the parameter that gets instantiated. Those approaches are still far from application to real world programs.

For languages that contain preprocessor directives like C, dataflow analysis is usually done on the preprocessed source code. For maintenance this may not be acceptable: e.g. refactoring can only be done on the original, non-preprocessed code. Dataflow analysis for non-preprocessed code is still an open question.

Another key problem in using dataflow analysis is *Pointer Analysis*: for languages like C, C++ or Java an analysis must first decide which objects a pointer in the program may point to. There is a broad range on pointer analysis techniques for different languages and some issues are still to be solved. A generic pointer analysis is not yet in sight: pointer analysis for C is very different from pointer analysis for Java. A discussion of pointer analysis is beyond the scope of this paper, a good overview can be found in [HP00, Hin01].

Until generic dataflow analysis systems for software maintenance have much improved, we have to pursue other approaches. One possibility is to use the expertise from compiler optimization, where sophisticated dataflow analysis infrastructures are already available.

2 Experience Reports

Our group has build three maintenance prototypes that use program analysis. One is a program slicing infrastructure for C-programs, one is a tool to analyze the usage of class members in Java-programs and restructure the class hierarchy according to the actual use of members, and the third does impact analysis for aspect oriented programs. For all three system, we will present some of the design decisions and architecture aspects which causes certain requirement on the used program analysis. In all three projects, we tried to find program analysis infrastructure complying to our requirements. Based on our experience, we infer more general requirements and evaluate the infrastructures we found in sections three and four.

2.1 VALSOFT

The VALSOFT Slicing System [KS98] is a set of tools which are used together to analyze C source code to help the engineer to understand and validate the code. The main application is the calculation and visualization of slices for ANSI-C.

Our frontend (scanner/parser etc.) reads all preprocessed sources, constructs an attributed abstract syntax tree and a symbol table, and “links” all corresponding symbols of different sources together. The next step is a traversal of the AST which does a simple, flow insensitive dataflow analysis. It calculates the **gmod** and **gref** sets, the call graph, the points-to set and the frame of the PDGs, which consists of control flow and control dependence edges. A finer, flow sensitive analysis follows, which traverses the frame and calculates the data dependencies. The resulting PDGs are linked together to the SDG [HRB90] at last.

The created SDG is persistently saved to disk and all other tools are working with the saved SDG. The system consists of:

- The analyzer, which generates the SDG for a set of C sources.
- The slicer, which can do backward and forward slices and chops. It is also used to compare different slicing techniques.
- The solver, which can calculate *path conditions*: precise conditions under which a dataflow along SDG chops or paths can happen.
- A constant propagation and common subexpression elimination module, which tries to simplify the SDG (eliminating edges and nodes).
- A call graph simplifier, which eliminates redundant call edges in presence of function pointers.
- A GUI, which visualizes the PDGs and the corresponding sources and controls the execution of the slicer and the solver.
- A clone detector [Kri01].

The requirements of the system have been the following from the beginning:

- The frontend must be flexible enough to support variants of ANSI-C.
- The frontend must not only build abstract syntax trees and symbol tables for single object files but for complete programs. It therefore must also fulfill the tasks of the linker to provide program-wide valid identifiers.
- The internal program representation (the SDGs) should be kept very similar to the abstract syntax tree, because we wanted to provide a comprehensible graphical presentation of it.
- The expressions used in the program had to be fully recoverable to enable the generation of path conditions.
- The internal dataflow analysis infrastructure had to be flexible enough to enable a wide range of different dataflow analyses in multiple stages.

At the time we started the project, almost no program analysis framework was available; the only available infrastructure was SUIF1 which we considered to be a dead project at that time. We also decided not to use any transformation to a low level intermediate format due to the above requirements. The frontend has been build externally by the LINEAS company. The company evaluated different possibilities and ended up building a complete new frontend only using lex/yacc because of the following reasons:

- Our specific requirements of the frontend were not fulfilled by most of the available infrastructure.
- Freely available infrastructures could not be used due to licensing restrictions (the company holds the commercial rights on parts of our system).
- The cost of adaption of available infrastructures was estimated to be higher than a complete new implementation.

The result was a well supported frontend until the company left the project. However, we are still able to adapt and fix the frontend as we have the right to use and modify the sources together with their documentation. For example, we have successfully changed the frontend to support Cilk, a parallel variant of C.

To fulfill the other requirements, we started to build dataflow analyses directly on top of the abstract syntax tree without transformation to a control flow graph first. For flow sensitive analyses this turned out to be impractical, as the cost to support unstructured constructs exploded (an experience that we share with others) and we use control flow graph based analyses now.

Recently we started to integrate an analysis for Java. Our prototype implementation is based on a frontend of the MIT, the FLEX compiler infrastructure [Ana99] which supports several intermediate representations. In our analysis we used one that is already in SSA form [CFR$^+$91].

The main challenges in dataflow analysis for Java are the data dependences for objects which are represented as a tree [LH98] to make the subobjects visible, and the points-to analysis which is crucial for dynamic binding and resolution of aliasing situations.

At the beginning of the project no all-purpose points-to analysis for Java was available, the one included in the FLEX library turned out to be too specialized for their purposes and too slow for whole-program analysis because it is flow-sensitive. Recently the SOOT infrastructure (which will be presented later) introduced SPARK [LH03], an infrastructure which allows to compare several different approaches of points-to analysis.

2.2 KABA

KABA is a tool for analyzing and reengineering class hierarchies. It performs a fine-grained analysis of member accesses from objects and creates a new class hierarchy using concept analysis [ST00]. In this hierarchy, all redundant accesses are removed and all objects only contain the minimum amount of members required. Leaving aside Java's "runtime" features like reflection, the transformation guarantees preservation of program semantics.

The new class hierarchy is valuable for reengineering, because it gives an impression on how classes are used in the analyzed programs, no matter how they are declared. It can serve as a useful proposal how the original hierarchy can be restructured.

KABA currently consists of three tools:

- An analyzer for Java bytecode, including pointer analysis.

- The GUI, which creates the concept lattice, displays the new class hierarchy and allows interactive manipulation.
- A rewriter which creates bytecode for the transformed classes. It tries to preserve debug information so Java decompilers can be used to produce human-readable source code from its output.

The transformation done by KABA was originally designed for C++. For a prototype implementation Java was chosen because the language is much simpler than C++. For a source to source transformation it seemed natural to build a Java source front-end to our analysis. However there was no infrastructure available to parse Java sources. So we decided to have a look at the bytecode. When the work on KABA started in 1999, no toolkit which fullfilled all our needs was available. So we wrote our own bytecode analyzer which was accomplished within a reasonable amount of time.

Nowadays KABA could propably easily reimplemented with most toolkits which have a sufficient internal representation of bytecode. But up to now, SOOT is the only known toolkit to contain the necessary pointer analysis.

2.3 Impact Analysis for Java and AspectJ

Our group is currently working on change impact analysis for Java and AspectJ [SK03]. This analysis is especially interesting in the field of aspect oriented programming (AOP) [KLM$^+$97], as an Aspect should be applied to a given system without breaking its functionality. AOP is used for software maintenance purposes recently [GM03, GO03].

The basic idea behind our work is to provide an analysis tool allowing programmers to get an impression of how the Aspect might change system behavior before actually applying it. With this information, the programmer can decide better whether the Aspect will work as intended than from Aspect documentation alone.

Up to now, a parser and some simple analysis based on hierarchy information have been implemented. However, this project is far from maturity. For the planned functionality a set of standard data structures to represent programs like call graphs, control flow graphs, and program dependence graphs are needed—including points-to analysis for Java/AspectJ.

Obviously it will reduce the neccessary effort dramatically if an off-the-shelf infrastructure to generate necessary data structures could be used. So we now basically face the same problem again as when we started with VALSOFT or KABA. Possibly an analysis on byte code level can be applied, although aspects no longer exist there. If the mapping of aspect effects in the byte code back to the source code can be done, the infrastructure provided e.g. by SOOT will be an adequate means to implement the impact analysis. If this approach fails, no (source) infrastructure is available which meets our requirements. Building such infrastructure is on one hand conceptually not very challenging as all concepts are basically known, but on the other hand results in a huge amount of work.

3 Infrastructure Requirements

Before we start to use dataflow analysis in software maintenance, we should know what specific requirements are stemming from such an application and what implications are caused. Without defining the requirements and doing a full evaluation first, the use of a specific program analysis infrastructure (PAI) can cause a grinding halt on research. For example, research on a program slicer [AG01a] had to replace the used infrastructure in the middle of the project from SUIF to SOOT.

In this section, we identify some general requirements. Most of the requirements are stemming from our own experience presented in the preceding section.

3.1 Adequate Representation

The (abstract) representation of the program elements is a major factor in usability of program analysis infrastructure. Representations based on abstract syntax trees are well known for tool builders, but control flow graph based representations are better suited for dataflow analyses. On the other hand, compiler-like representations like three-address code are sometimes problematic if they have to be use for maintenance tasks (see next requirement). The representation must be easily accessible (and manipulatable) via a well-designed API. Ideally, a PAI would support a multitude of representations.

Requirement 1 *PAI should provide an adequate representation.*

3.2 Source Tracking

Many maintenance tasks are based on source code representation of the results. For example, program slices must be shown in source code or transformations can change the source code. Especially for transformation purposes the changes to the source code should be as small as possible—Generating complete new source code through pretty printing is not acceptable, because the source may look completely different afterward.

Requirement 2 *PAI should be able to map any element of the used representation back onto the source code.*

3.3 Full Language Support

Research on software maintenance tools that use program analysis is normally using real programs as tests. If a PAI only supports a subset of the language, a massage of the input programs are normally necessary—an unusable approach for large systems.

A tool builder (customer) will probably only use a subset of the features and may even want to restrict them according to coding guidelines.

Requirement 3 *PAI must support all features of the supported programming languages.*

3.4 Multi-Language Support

Program analysis infrastructures that only support one single language may be valuable for software maintenance research, but will be of restricted use. Infrastructure is needed that support many languages (different languages and different variants):

Requirement 4 *PAI should support more than one programming language.*

This is a very general requirement and as better PAI will become available, it must be replaced by more specific requirement, like

Parameterization: The program analysis infrastructure is generated from a formal specification of the input language. However, nowadays it is almost impossible to formally define a current programming language.

Unified Representation: The program analysis infrastructure is independent of its language frontends, because it uses one common intermediate representation on which the dataflow analysis is performed. Until now, all "one size fits all" approaches more or less failed.

Transferability: An implemented (or specified) dataflow analysis for one of the input programming languages should also be applicable to other supported input languages of the PAI.

Linkability: The PAI support analysis of input programs that consists of modules in more than one (supported) language. This includes the analysis of meta-information describing the relationship between the different modules.

Unified representation is different than transferability: A unified representation can be achieved from building a superset of all language specific representations. A dataflow analysis for one input language may just support the specific subset and may not be transferable to a different input language.

Most of the more specific requirements are not yet achievable, until research in that areas went further.

3.5 Scalability

Targeting real programming languages implies targeting real-world-sized programs. Infrastructures that only scales up to sizes of 100LOC are as unusable as infrastructures that only support languages without function calls.

Requirement 5 *PAI should scale.*

This doesn't imply that the dataflow analyses themselves should scale—many analyses are of non-polynomial complexity. The infrastructure itself should be able to handle large systems and can leave the problem of scalability of the analyses to be solved by the user.

3.6 Library of Analyses

An infrastructure, that supports dataflow analyses but doesn't provide even the simplest ones, is of reduced usability for software maintenance, as nobody wants to reinvent the wheel. The dream of a renovator would probably be a PAI that comes with a set of off-the-shelf pointer analyses which he is able to use without modification. However, most of the provided analyses are as simple as intra-procedural defs-use chains.

Requirement 6 *PAI should provide a library of reusable analyses.*

3.7 Support

Program analysis infrastructures from research are mostly *proof-of-concept* only: after finishing the research project, the prototype is abandoned unfinished. This is acceptable for research projects in program analysis infrastructure, but ignores communities that could benefit from it. The wheel has to be reinvented again and again.

Requirement 7 *A PAI should be supported.*

It is clear that academia cannot support their prototypes longer than their research project. However, the need for infrastructures has already been acknowledged and some projects are targeted at reusable program analysis infrastructures. Some of those infrastructures are available and will be evaluated in the next section.

4 Available Infrastructures

It is well known that using dataflow analysis is expensive. First, we have the *computational* cost: for maintenance, we need very precise information, resulting in algorithms with a non polynomial complexity. Second, we have the *infrastructure* costs: Building dataflow analyzers is not trivial and needs a lot of man power. The infrastructure cost is the main reason for the nonexistence of dataflow aware infrastructures in software maintenance. For example, DMS, an industrial-strength transformation system (without dataflow analyses), needed over 50 person-years of engineering until now [BM01].

Without maintenance specific infrastructure, we have to reuse available infrastructures from compiler optimization. As early as 1973 Killdall presented a theoretic framework [Kil73] for dataflow analyses. It is a lattice-theoretic model based approach and is used by most analysis frameworks. Other infrastructures are based on abstract interpretation [CC77].

In the following, we evaluate some *available* infrastructures from compiler optimization under the requirements of software maintenance. We are not claiming that the following list is complete or precise, as there are too many infrastructures out there. Worth a look are also OPTIMIX [Aßm98], PAF [RLS$^+$01], ICARIA/PONDER [AG01b] and BLOAT [Nys98].

We are not evaluating the scalability of the infrastructures, as this is beyond the scope of this paper. However, all presented infrastructures have shown the ability to work on programs larger than 10kLOC.

4.1 Vortex

Vortex is a typical language-independent optimizing compiler infrastructure for object-oriented languages. It does whole-program-analysis based on the Vortex RTL intermediate language. Its primary purpose is research of aggressive optimization. It provides a reusable framework with generic support facilities [CDG96]. We are not aware of any maintenance project that uses this infrastructure.

Representation, Source Tracking

The Vortex RTL intermediate language is the only representation without a mapping back to source code positions.

Multi-Language Support, Full-Language Support

Vortex has frontends for Cecil, Java, C++ and Smalltalk, which transforms the input languages into Vortex RTL. The Java frontend only supports a subset of Java without

- threads and synchronization
- reflection and dynamic class loading
- finalization.

Library of Analyses

Vortex comes with a set of intra- and inter-procedural analyses, for example:

- several traditional optimizations (constant propagation and folding, common subexpression elimination, dead assignment elimination)
- must-alias and side-effect analysis
- class hierarchy analysis
- wide range of inter-procedural class analysis algorithms

Support

The status of Vortex is unknown.

4.2 SOOT

SOOT [PQVR$^+$01] is an infrastructure to optimize Java bytecode. Other than usual infrastructures, which are transforming the (high-level) input languages to (low-level) intermediate representations, SOOT works bottom-up: From a low-level language (Java bytecode) it builds high-level intermediate representations. This may sound awkward for software maintenance, but SOOT has already been used e.g. for a program slicer [AG01a].

Representation

SOOT provides three abstraction levels:

BAF is similar to the bytecode,

JIMPLE is three-address code,

GRIMP is similar to Java.

Each representation can be transformed into the next higher level and compilation from GRIMP into BAF closes the circle. GRIMP is similar to an AST, but the main representation is JIMPLE.

Source Tracking

As SOOT is using bytecode as input language, it is bound to the restrictions of bytecode. Therefore it is not only loosing source code positions (in the newer versions the line number tables created by the compiler can be parsed), but also names of identifiers. As SOOT has no knowledge about the transformation/compilation of the source languages into bytecode and the applied optimizations, it may be impossible to recover even the smallest hints to origins.

Full Language Support

We are not aware of any restrictions on the bytecode.

Multi-Language Support

As the the bytecode may come from multiple source languages like Java, SML, Scheme or Eiffel, it may be considered as a *multiple-language* framework. However, as high-level representations are generated bottom up from bytecode, all programs will look like Java programs in GRIMP.

Library of Analyses

Besides the analyses that are need for transformations between the different representations, SOOT contains some simple analyses like intra-procedural def-use chains. All analyses are based on the JIMPLE representation because the analyses are optimization-centric. Lately a pointer analysis kit (SPARK) has been added, which supports scaling points-to analysis for Java [LH03]. Its purpose is to integrate several well-known approaches to points-to analysis in one framework to make them comparable. Furthermore, SPARK provides multiple implementations of points-to sets which yield different analysis times and two major propagation algorithms affecting the speed of the analysis as well.

Support

SOOT is stable and maintained (it is still a research project itself). Commercial support is not available.

4.3 SUIF

The SUIF system is an infrastructure for research in compiler optimization techniques, based upon the *Stanford University Intermediate Format (SUIF)* [WFW+94]. It started with a first version (SUIF1) which had C and Fortran as primary languages. It was the base for two "spin-offs": OSUIF for object oriented languages (higher level) and MachSUIF for assembler like languages (lower level). As those and other extensions were complicated in SUIF1, a complete redesign and reimplementation was needed and resulted in SUIF2, which had extensibility and modularity as primary design goals.

Representation

SUIF has multiple levels of abstraction: three different main-levels are based on the used SUIF version (core, OSUIF or MachSUIF). The program representation itself supports multiple levels of abstraction, where the highest level is similar to ASTs. SUIF has *dismantlers* that break down higher level constructs to lower levels.

Source Tracking

The core SUIF has source code position annotations. However, the use of them is dependent on the used frontend. The C/C++ frontend is based on the EDG frontend, which supports the annotations. The Java frontend is using Java bytecode without support for source code positions.

Full Language Support

SUIF comes with frontends for Fortran, C, C++ and Java. The support of Fortran and C is basically complete. OSUIF, which includes the frontends for C++ and Java, is less complete and doesn't support exceptions or garbage collection for example. The Java (bytecode) frontend doesn't support exceptions, threads, synchronization and dynamic loading [KH98], partly due to missing support in OSUIF.

Multi-Language Support

SUIF basically only has support for the four mentioned languages and new languages need their own frontends. We have no knowledge if the Java bytecode frontend also supports bytecode generated from other languages. SUIF is a unified representation that supports analyses on different levels of abstraction. Through the use of dismantlers a pipe of multiple analyses is possible (going from high-level to low-level representations).

Library of Analyses

SUIF already contains a set of intra- and inter-procedural dataflow analyses like copy propagation, call graph construction or (simple) pointer analysis. It also contains helpful utilities ranging from presburger arithmetic or gaussian elimination to graph visualization.

Support

The base infrastructure is solid and in active use by different groups, but active development and maintenance seems to be stopped. OSUIF is less stable and less used. Commercial support is not available.

4.4 BANE

BANE is a constraint-based toolkit for constructing dataflow and type inference systems. Analyses are formulated as systems of constraints, which are generated from the program text. The desired information is computed by Constraint resolution (solving the constraints) [AFFS98]. This might sound inadequate for software maintenance purposes—however, BANE has been used for implementation of a system that finds Y2K problems in C programs (implemented in only one month).

Representation, Source Tracking

Due to its nature, BANE has no support to access an underlying infrastructure. However, BANE has implicit support for source code positions (at least source code positions can be reported).

Full Language Support, Multi-Language Support

There are two frontends available for BANE: SML and C.

Library of Analyses

BANE includes a flow-insensitive pointer analysis for C. Compared to other systems, it is one of the most scalable pointer analyses: it was able to do an pointer analysis on a 500kLOC C program.

Support

BANE is only available in an old version from 1998.

4.5 PAG

All the previous presented infrastructures were *frameworks* that help *implementing* dataflow analyses. PAG [Mar99] is a dataflow analysis *generator*. From a specification (in a high level functional language) a complete analysis in ANSI-C is generated. The benefit are *provably correct and terminating* analyses. PAG is targeted at program analysis for optimizing compilers.

Representation, Source Tracking

Due to the nature of the generated analyses, there is no accessible representation.

Full Language Support, Multi-Language Support

Languages are supported through different frontends. The only available frontend for a real language is a C frontend, which, in principle, supports full C.

Library of Analyses

Various analyses have been specified for PAG with the C frontend, e.g. constant propagation or shape analysis.

Support

PAG has changed into a commercial product which is supported.

4.6 DMS Software Reengineering Toolkit

The DMS Software Reengineering Toolkit [BM01] is "a set of tools for automating customized source program analysis and modification of large scale software systems, containing arbitrary mixtures of languages". Unlike the other infrastructures which are built for compiler optimization purposes, it is dedicated to software maintenance. Despite that it does not contain support for dataflow analysis yet, it is included in this survey because such support will be in the next release. On the other hand it is a good example for an industrial-scale infrastructure.

Representation, Source Tracking

The main representation are abstract syntax trees which contain links back to the source code. The trees can be pretty-printed according to specified layout rules or in a way that comments, spacing and lexical formatting information of unchanged code is preserved.

Full Language Support, Multi-Language Support

Parsing is handled via GLR based parser generation; a long list of programming languages are supported. Sources in different languages can be represented at the same time.

Library of Analyses

No dataflow analysis is included.

Support

The DMS Toolkit is fully supported.

4.7 Other Approaches

There is also work (but no available infrastructure) on dataflow analysis infrastructure not coming from compiler optimization but directly from software maintenance. There, the dataflow analysis together with the tool that needs the results of the analysis gets generated from a formal, algebraic specification [Tip95a]. A similar approach specifically targeted to reverse engineering is [Moo97]: the analyzed program is first transformed into a Dhal program. Dhal is a specifically for maintenance designed language, which enables an automatic transformation into a well-structured program without gotos etc. The dataflow analysis is also generated from a specification, which is then performed on the transformed program.

The CodeSurfer slicing tool for C [AT01] has a strong underlying dataflow analysis infrastructure. It is planned to open the infrastructure via an API and to include support for C++ and Java.

The Eclipse Platform, designed for building integrated development environments, contains a Java development tool (JDT) which provides an API to the Java element tree[2]. This element tree represents elements of a project (.java, .class and .jar files) together with abstract syntax trees for the files. This platform is already used for various program analysis and software maintenance research projects, despite the lack of support for dataflow analysis. However, work on support for dataflow analysis has started, e.g. SOOT will be integrated into the Eclipse Platform.

5 Conclusions

As of today, there are no program analysis infrastructures available that are specifically targeted at maintenance. Therefore, one has to reuse infrastructure from compiler optimization. However, the specific requirements of maintenance are not completely fulfilled by any of the available infrastructures. Some lessons we learned are:

- Building infrastructure on top of Java bytecode instead of Java source code is often inadequate for maintenance. However, none of the available infrastructures supports Java source code.
- Building dataflow analysis based on abstract syntax trees (omitting other intermediate representations) may not be a good idea as it might not provide the right abstraction.
- Infrastructure is expensive—even learning to use it. Also, choosing the right infrastructure is hard: evaluation means learning to use it.
- Academic infrastructure is *fragile*. There is basically no support and as research funding terminates, the development of infrastructure normally stops in an unfinished state. Professional infrastructure is almost unavailable, despite that demand is rising.

[2]A similar development tool exists for C++.

- Frameworks like SUIF are better suited than generators like PAG: A framework for compiler optimizations is more flexible and can be reused for software maintenance. Generators for compiler optimizations are more restricted and may not support basic requirements for software maintenance. This will change as soon as there are generators targeted at software maintenance.

The implications of multi language support are not fully understood yet. There are basically two approaches: one is to break down the input language to a simple low-level representation, which delegates the work into the language frontend. The other is to have a unified high-level representation, which basically provides a superset of all supported languages. Both approaches have disadvantages: consider a pointer analysis for C++ and Java, which differ significant. With a low-level representation one looses the ability to use specific properties of the languages and with a high-level representation one has obey every detail of the (elaborate) representation.

Off-the-shelf program analysis infrastructure does not exist and available program analysis frameworks are clearly not in the state that they can be used for production quality maintenance tools, but they provide a good base for research in software maintenance.

Acknowledgments. Michael Mehlich and Holger Cleve provided valuable comments on earlier versions of this paper.

References

[AFFS98] A. Aiken, M. Faehndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 76–96, 1998.

[AG01a] Gagan Agrawal and Liang Guo. Evaluating explicitly context-sensitive program slicing. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 6–12, 2001.

[AG01b] Darren C. Atkinson and William G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, 2001.

[Ana99] C. Scott Ananian. The static single information form. Master's thesis, MIT Dept. of Electrical Engineering and Computer Science, September 1999. Technical Report MIT-LCS-TR-801.

[Aßm98] Uwe Aßmann. *OPTIMIX, A Tool for Rewriting and Optimizing Programs*. Chapman-Hall, 1998.

[AT01] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering (CAV 2001)*, 2001.

[BM01] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 281–290, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CC01] A. Cox and C. Clarke. Representing and accessing extracted information. In *International Conference on Software Maintenance (ICSM)*, 2001.

[CDG96] C. Chambers, J. Dean, and D. Grove. Frameworks for intra- and inter-procedural dataflow analysis. Technical report, Department of Computer Science and Engineering, University of Washington, 1996.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[GM03] Tanton Gibbs and Brian Malloy. Weaving aspects into C++ applications for validation of temporal invariants. In *European Conference on Software Maintenance and Reengineering*, 2003.

[GO03] Thomas Gschwind and Johann Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *European Conference on Software Maintenance and Reengineering*, 2003.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, June 2001.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, 2000.

[HRB90] Susan B. Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[Kam98] M. Kamp. a multi-file, multi-language software repository for program comprehension tools: A generic approach. In *Sixth International Workshop on Program Comprehension*, pages 64–71, 1998.

[KH98] Holger Kienle and Urs Hölzle. j2s: A SUIF java compiler. Technical Report TRCS98-18, University of California, Santa Barbara. Computer Science., August 26, 1998.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM SIGACT and SIGPLAN, ACM Press, 1973.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[KS98] Jens Krinke and Gregor Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11-12), 1998.

[KSW95] N. Kiesel, A. Schuerr, and B. Westfechtel. GRAS, A graph-oriented (Software) engineering database system. *Information Systems*, 20(1):21–52, 1995.

[Lak97] A. Lakhotia. A unified framework for expressing software subsystem classication techniques. *Journal of Systems and Software*, 36(3):211–231, 1997.

[LH98] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *12th International Conference on Compiler Construction*, 2003.

[Mar99] Florian Martin. *Generating Program Analyzers*. PhD thesis, Universität des Saarlandes, Germany, 1999.

[Moo97] L. Moonen. A generic architecture for data flow analysis to support reverse engineering. In A. Sellink, editor, *Proc. 2nd Int. Workshop on the Theory and Practice of Algebraic Specifications*, Amsterdam, 1997. Springer-Verlag.

[MRR02] Ana Milanova, Atanas Routev, and Barbara G. Ryder. Constructing precise object relation diagrams. In *International Conference and Software Maintenance*, pages 586–595, 2002.

[NNH99] F. Nielson, H. Nielson, and C. Hankin. *Principles of program analysis*. Springer Verlag, 1999.

[Nys98] N. Nystrom. Bytecode-level analysis and optimization of java class files. Master's thesis, Purdue University, West Lafayette, IN, August 1998.

[PQVR$^+$01] Patrice Pominville, Feng Qian, Raja Vallee-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *Proceedings of Compiler Construction, 2001*, pages 334–554, 2001.

[RLS$^+$01] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(1):105–186, 2001.

[SK03] Maximilian Störzer and Jens Krinke. Interference analysis for aspectj. In *Foundations of Aspect-Oriented Languages (FOAL)*, Boston, 2003.

[Sne00] Gregor Snelting. Software reengineering based on concept lattices. In *4th European Conference on Software Maintenance and Reengineering*, pages 3–10, 2000.

[ST00] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, 2000.

[Tip95a] Frank Tip. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995.

[Tip95b] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[TP02] Paolo Tonella and Alessandra Potrich. Static and dynamic C++ code analysis for the recovery of the object diagram. In *International Conference and Software Maintenance*, pages 54–63, 2002.

[vdBKV97] M. van den Brand, P. Klint, and C. Verhoef. Reengineering needs generic programming language technology. *SIGPLAN Notices*, 32(2):54–61, 1997.

[vdBSV98] M. van den Brand, M. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proc. Sixth International Workshop on Program Comprehension*, pages 108–117, 1998.

[vDKV99] A. van Deursen, P. Klint, and C. Verhoef. Research issues in the renovation of legacy systems. In *Fundamental Approaches to Software Engineering*, volume 1577 of *LNCS*, 1999.

[WFW$^+$94] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, Shih-Wei Liao, Chau-Wen, Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.