Building a Whole-Program Type Analysis in Eclipse[*]

Mariana Sharp
Ohio State University

Jason Sawin
Ohio State University

Atanas Rountev
Ohio State University

ABSTRACT

Eclipse has the potential to become a widely-used platform for implementation and dissemination of various static analyses for Java. In order to realize this potential, it is important to understand the challenges for building high-quality static analyses in Eclipse. This paper discusses some of these challenges in the context of the TACLE plug-in for whole-program type analysis and call graph construction. In particular, we argue that the treatment of the standard Java libraries should be an important concern for static analysis builders. Our experiments indicate that it may be necessary to use pre-computed summary information for the libraries, in order to improve the scalability of whole-program analyses for Eclipse. The experience described in this paper could be beneficial for static analysis researchers who use Eclipse as the infrastructure for their analysis implementations.

1. INTRODUCTION

In the last few years, Eclipse has become an attractive platform for research and distribution of static program analyses for Java. This is largely due to Eclipse's ability to provide solutions to several recurring problems that analysis researchers have to address in their projects:

- **Off-the-shelf infrastructure:** The builders of a static analysis are typically faced with the challenge of finding a suitable infrastructure which provides the necessary services (e.g., parsing, intermediate representation, etc.) for implementing a program analysis. The infrastructure choice can have a significant impact on the progress and success of a research project. Eclipse provides a mature, free, open-source platform for implementing various static analyses.
- **Analysis comparison and integration:** Researchers often cannot perform experimental comparison of two or more competing algorithms, because analysis implementations are not portable due to their different underlying infrastructures. Similarly, it is very common for a research group X to have an in-house implementation of some analysis which could be beneficial for the analysis developed by another research group Y; however, since X's implementation is not portable to Y's environment, the potential synergy between the two research projects remains unrealized. If Eclipse is used as a common implementation environment for static analyses for Java, widespread analysis comparison and integration will become much easier.
- **Realistic analyses:** It is not uncommon for analysis creators to "cut corners" when building prototype implementations of their algorithms, by ignoring various conceptual and engineering issues. This is often justifiable in the early stages of the work, or when the ignored issues are trivial. However, such simplifications may (and often do) abstract away critical details that have significant effect on the real-world practicality and usefulness of the resulting analyses. Eclipse provides a natural proving ground for demonstrating that a proposed analysis can operate in the context of a realistic development environment, in which the analysis is only one component of a large system with numerous complicated constraints and interactions.
- **Technology transfer:** The ultimate goal of static analysis research is to improve programmer productivity and software quality. The transfer of technology from research to practice is an important and challenging problem for this research community. By implementing and distributing their work in Eclipse, analysis builders have the exciting opportunity to reach a wide audience of Eclipse users, and to achieve real-world impact for their research.

In order to realize the potential of Eclipse to become *the* platform of choice for static analysis research for Java, it is essential to investigate the challenges for building high-quality analysis implementations in Eclipse, and to find effective solutions for these challenges. This paper is a step in this direction. We outline our experience with building a whole-program analysis for Java, some of the issues we had to address, and an investigation of possible solutions to these issues. This experience will be useful for other analysis researches that are currently building (or will be building in the future) whole-program analyses for Java in Eclipse.

The analysis described in this paper was implemented as part of the ongoing work on the TACLE Eclipse plug-in (`presto.cse.ohio-state.edu/tacle`). The goal of the

[*]This work was supported by an IBM Eclipse Innovation Grant.

eclipse'05, October 16-17, 2005, San Diego, CA
Copyright 2005 IBM 1-59593-342-5/05/0010…$5.00

Tacle project is to build a public implementation of several algorithms for type analysis and call graph construction for Eclipse. We describe our experience with one of these algorithms: *rapid type analysis* (RTA) [1]. Even though this analysis is conceptually simple, it illustrates some of the fundamental issues that need to be addressed by the builders of whole-program analyses in Eclipse. In particular, our experiments show that the treatment of the standard Java libraries is an important engineering decision that should be a central concern for analysis builders. We describe three approaches for handling of the standard libraries during RTA, and present a preliminary empirical comparison between these approaches. These results provide useful guidance for implementing more complicated whole-program analyses, where the handling of the standard Java libraries will be even more important than it is for RTA.

2. THE TACLE PROJECT

Type analysis for object-oriented software answers the following question: what types of objects could an expression e refer to? A simple answer could be obtained by considering e's compile-time type and all of its subtypes in the program. However, extensive empirical evidence shows that the "all-subtypes" answer is often too conservative. There is a large body of work on type analyses for object-oriented languages; most of it is summarized in [2, 8]. The main motivation for this work is the key role of type information as an "enabler" of other static analyses for object-oriented languages.

A typical use of type analysis is the generation of a program's call graph, where type information is needed to determine the target methods at virtual calls. The call graph is essential for any form of interprocedural static analysis, and plays an important role in program understanding tools, compiler optimizations, verification techniques, and test coverage tools. Type analysis (especially its more precise versions based on points-to analysis) is also useful for many other commonly-used techniques—for example, analysis of side effects, def-use analysis, and escape analysis.

Despite the large number of research papers and projects dedicated to type analysis, the use of such analyses still remains confined to a small community of experts. The goal of the Tacle project is to build a type analysis for Java that is seamlessly integrated with Eclipse and is carefully engineered for real-world use by non-experts. The project will provide other researchers (that are not experts in type analysis) with an off-the-shelf implementation of a practical and easy-to-use type analysis and call graph construction. In the best case, this will increase the attractiveness of Eclipse as a research platform for static analysis for Java, and will enable a variety of other analyses to be implemented and ultimately made available to the Eclipse community.

The long-term goal of the Tacle project is to develop type analyses with the properties described below. We strongly believe that these properties are also very desirable for other static analyses implemented in Eclipse, and that analysis builders should consider them as high-priority goals.

- **Incremental analysis:** Existing type analyses are typically designed to work in "batch mode": they are initiated by a client, analyze the whole program from scratch, and at the end produce a solution. However, in Eclipse the analysis should work in "incremental mode", similarly to the auto-build feature of JDT. This feature rebuilds a project automatically every time a file is saved; this is done in the background without user intervention. The type analysis should be integrated with this model: it should run continuously in the background, and should maintain a solution for the latest build that was saved on disk. The analysis will recompute its solution incrementally every time a file is saved. This recomputation should reuse as much as possible from the previous solution.
- **Pre-analysis of libraries:** Among researchers on type analysis for Java, it is common knowledge that a large portion of the analysis running time is spent processing the relevant parts of the standard libraries. However, most existing approaches still analyze these libraries from scratch for every analyzed program. To make our plug-in practical, it is critical to reduce this cost by pre-analyzing the libraries. This could be done once when the plug-in is deployed in the user's Eclipse installation. The computed information can be stored on disk and read when Tacle is started at run time.
- **Partial programs:** Existing type analyses are typically designed as whole-program analyses: they take as input a main method and all code transitively reachable from that method. However, in Eclipse it should be expected that users may be developing partial programs (e.g., class libraries). The analysis in Tacle will be designed to handle such partial programs. Our previous work [6] defined a solution to this problem, by creating an artificial main method that "simulates" the possible flow of object references between the analyzed code and unknown future code. We will incorporate this solution in Tacle.
- **Source code analysis:** Type analyses for Java are typically performed on bytecode or on some representation that is constructed from bytecode. Such low-level representations are appropriate for use in optimizing compilers. However, in Eclipse the analysis should work at the level of the source code, by taking as input the abstract syntax tree (AST) for that source code. Our goal is to produce an analysis that can be used by software productivity tools that analyze or transform the source code. For example, if another researcher were to use our analysis as part of some algorithm for program slicing or for code refactoring, he/she would require information about expressions in the source code. Eclipse ASTs provide a common infrastructure for our analysis and for other static analyses for Java.

At present, we have focused on the second objective: *effective pre-analysis of the standard Java libraries.* As of now, we are still considering whole-program analysis which is not incremental and which does work on partial programs; the handling of these issues is left for our future work.

3. WHOLE-PROGRAM RTA

Rapid Type Analysis (RTA) [1] takes as input a complete program and produces type information and a call graph for that program. The analysis constructs a set *Reachable* of methods that are reachable from the main method of the program, and a set *Instantiated* of class types that are instantiated in reachable methods. Initially, *Instantiated* is empty and *Reachable* contains the main method. Whenever a method m is added to *Reachable*, the body of the method is processed to (1) update set *Instantiated*, due to expressions `new X`, and (2) update set *Reachable*, based on the call sites inside m. The virtual call sites in m are resolved based on the current set of types in *Instantiated*. If later some class type is added to *Instantiated* and this type implies additional target methods at already-processed call sites, these new target methods are added to *Reachable*.

A possible implementation of RTA for Eclipse is the following. Start with the main method of the program. When a method is added to *Reachable*, the AST of the surrounding class is built and processed by the analysis. Of course, the AST can be reused the next time a method from the same class becomes reachable. In addition to the main method, set *Reachable* is also initialized with all methods that are executed at JVM startup; in our case, this list was obtained using the `hprof` JVM command line option. Similarly, the set of all classes instantiated during JVM startup is used to initialize *Instantiated*; again, the `hprof` JVM option was used to obtain this list. Note that the set of executed methods and instantiated classes at JVM startup could differ between JVM implementations (e.g., across different JVM vendors, and across different platforms). If a JVM implements the JVMPI profiling interface, a simple profiling agent can be executed at TACLE-installation time to obtain this information.

There are several other analysis issues that are not discussed in this paper for the sake of brevity. These include the handling of static initializers; initializers of static fields; initializers of instance fields; default constructors; methods `start` and `run` of threads; dynamic class loading; reflection; native methods.

3.1 Version 1: AST-Based Analysis

The outline of RTA from above describes one possible implementation for RTA in TACLE. A key issue for this implementation is the need to build and repeatedly use ASTs for all classes that contain reachable methods, *including library classes*. As a result, the analysis has to construct hundreds of ASTs, and has to keep them in memory until the entire RTA solution is completed. For example, even for a simple hello-world program, RTA reports 4353 reachable methods in 692 classes.[1] These methods are considered reachable by the analysis due to the effects of JVM startup code, and the implementation has to build the corresponding ASTs.

Not all of the library classes have publicly available source code (in Sun's J2SE 1.4.2 for Windows, which we used for experimentation). Thus, AST-based RTA is not possible even for a trivial program. Nevertheless, in order to get better understanding of the cost of AST building and storage, we considered all 692 classes from the hello-world program, and constructed the ASTs for all top-level classes (i.e., nonnested classes) for which source code was available. (The AST for a top-level class contains as subtrees the ASTs for its nested classes.) A total of 308 ASTs were built. The running time for AST construction was around 10 seconds.[2] The total memory needed to store the ASTs was in the order of 140 MB. Note that these memory requirements are in addition to the memory needed for the internal data structures of the JVM and Eclipse.

These results indicate that the cost of AST building and storing may be substantial (or even prohibitive) for large programs that make extensive use of the Java libraries. If a few thousand library classes need to be processed by RTA, the time for AST construction could be in the order of minutes, and the memory usage could be in the order of gigabytes. In the next section we present experimental results that provide additional insights into this issue.

These AST-related issues are not unique to RTA; for the other type analyses we plan to implement in TACLE (e.g., points-to analysis based on [5, 3]), AST construction time and memory usage could become limiting factors for analysis scalability. Of course, this is also true for any whole-program static analysis in Eclipse which requires access to the fine-grain information available in ASTs. In particular, for more complicated analyses with long running times and significant memory requirements, it will be highly desirable to avoid AST building and traversal for library classes.

3.2 Version 2: Summary-Based Analysis

One technique for avoiding (some of) the cost of AST construction is to pre-analyze the standard Java libraries, compute *summary information* about them, store this summary on disk, and later use it when performing AST-based analysis of client code (i.e., non-library code). The one-time summary construction cost could be incurred at the time TACLE is installed in the user's Eclipse environment. The summary can be reused multiple times, once for each execution of RTA on a client program. Each time a client is analyzed, the ASTs for non-library code are built as in version 1, but the information for library classes is obtained by reading the summary information from disk. It is unnecessary to read the entire summary; rather, the summary-based analysis of the client code can read on-demand the relevant subsets of the summary.

The library summary should contain all and only information that affects the subsequent analysis of client code. In the case of RTA, the summary contains:

- the name of each library class and interface, together with the names of its immediate supertypes
- the signature of each library method, together with the name of the corresponding enclosing class
- the details for each call site in the library, including the compile-time target method, the compile-time receiver type (for virtual calls), and the enclosing method
- the instantiated class and the enclosing method for each allocation expression `new X` in the library

The summary can be constructed by building and traversing the ASTs of library classes. Alternatively, if the source code for the library is not available (as is the case for our experiments), bytecode analysis can be used to create the summary. Our implementation uses Soot [9] to process the bytecode in the JAR files for the standard Java libraries.

The format of the summary information is an engineering decision that should consider the need for a compact representation, as well as the need for easy and fast reading during the analysis of the client code. We currently employ a rather simplistic representation, and summary size is 17.2 MB on disk, corresponding to about 10000 classes and 80000

[1]These numbers will be even larger in our final implementation of RTA; the current implementation does not consider all possible effects of dynamic class loading, reflection, and native methods.

[2]On a PC with 3 GB memory and a 2.8 GHz Pentium 4 processor.

methods. Of course, we plan to investigate alternative representations (e.g., compression using `java.util.zip`), and to measure their effect on the running time of the analysis.

It is important to note that more complicated (and expensive) static analyses can benefit significantly from performing some of the analysis work in advance, at summary creation time. For example, for the implementation of points-to analysis in TACLE, we plan to compute partial analysis results for the library and to include them as part of the summary. The one-time work of library summary creation can reduce the cost of all subsequent executions of the analysis on many client programs. A general approach for summary creation is outlined in [7, 4]. This approach is applicable to a wide range of program analyses, and will be used in our future work in TACLE. We believe that other static analysis researchers that implement their analyses in Eclipse will also benefit from using this or a similar technique.

3.3 Version 3: Incomplete Solution

Depending on the intended uses of the static analysis solution, it may be acceptable (or even desirable) to compute an *incomplete solution*. Suppose, for example, that the RTA-computed call graph will be used for program understanding and visualization, and a tool user is interested only in calling relationships that involve at least one non-library method (i.e., an incomplete call graph). Based on this intended use of the RTA solution, the summary can be optimized in various ways, in order to reduce summary size and RTA running time. One particularly simple example is the following: suppose that a library call site c is definitely monomorphic (e.g., because the compile-time receiver type for c is a final class). Furthermore, suppose that the target method of c does not make any calls and does not instantiate any classes. It is clear that information about c can be removed from the summary. As a result of this optimization, call graph edges between library methods will be missing from the solution computed by RTA; however, edges involving non-library methods will be preserved. This idea can be generalized in several directions. For example, repeated applications of this technique could remove additional call sites, since an earlier removal could "empty" the body of a method, thus making it eligible for the optimization. As another example, the removal could happen even for polymorphic call sites, as long as all possible target methods are definitely in the library.

Our experiments, presented in the next section, use these techniques to create an optimized summary. The results from these experiments illustrate that even these simple optimizations can have positive effects on summary size and analysis running time. In particular, summary size was reduced from 17.2 MB to 16.7 MB, and the number of call sites was reduced from about 200 thousand to about 158 thousand. The time to optimize the summary was about 40 seconds, including the time to read the summary from disk. This one-time cost is insignificant since it is incurred once, at plug-in deployment time.

Of course, the above example is just one optimization from a large set of possibilities. In the general case, the anticipated uses of the analysis solution determine what constitutes "acceptable incompleteness" in the solution. This, of course, is true for any whole-program static analysis, not just RTA. The incomplete solution can be thought of as a projected version of the "real" solution. This intuition can be formalized theoretically by defining a *projection function* which maps the complete solution to an incomplete one. Different applications will require different projection functions. We are currently investigating projection functions that are appropriate for the points-to analysis in TACLE. Other static analyses that will be implemented in Eclipse (by other researchers or by us) should also be designed with particular projection functions in mind. This will remove redundancies from the library summary and will reduce the running time of the summary-based analysis of client code.

Program	Cls	Meth	ASTNodes	ReachMeth
`proxy`	13	102	5821	4671
`fractal`	25	178	8538	6753
`echo`	17	177	9828	7024
`jtar-1.21`	54	198	16687	6850
`javacup-0.10j`	32	329	16784	4731
`jlex-1.2.6`	24	149	20375	4585
`jflex-1.4.1`	73	484	33444	7154
`mindterm-1.1.5`	67	589	44872	7487
`muffin-0.9.3a`	125	797	44998	4541
`sablecc-2.18.2`	244	1758	64561	6270

Table 1: Analyzed programs.

4. EXPERIMENTAL STUDY

This section presents the results from a small-scale experimental study of the three versions of RTA described earlier. The experiments were executed on a PC with a 2.8 GHz Pentium 4 processor and 3 GB memory, using J2SE 1.4.2 for Windows and Eclipse 3.0. The JVM heap (JVM option `Xmx`) was set to the highest possible value of 1.5 GB. The measurements are the median values out of three runs. We used a partial implementation of RTA which does not consider all possible effects of dynamic class loading, reflection, and native methods. Nevertheless, even for this implementation, the comparison between the different versions of RTA provides useful insights.

The programs used in the experiments are shown in Table 1. The source code for each program is publicly available; this code was used to create an Eclipse project, which was then analyzed with TACLE. The program were obtained from various on-line Java projects. The number of classes and methods are shown in the first two columns of Table 1. Column "ASTNodes" shows the total number of AST nodes in the ASTs constructed from all classes in the program (i.e., the classes from the first column).

The last column in the table shows the number of methods that were reported by RTA as reachable. This number includes both library and non-library methods. Clearly, for these data programs, most of the analysis work will be done in the libraries, and the running time will be dominated by the cost of processing the reachable library methods.

Table 2 shows the results of running three versions of RTA. The first version ("w/ summary") uses the library summary for all library classes—that is, no ASTs are ever built for library code. The second version ("w/ optimiz") uses an optimized summary from which some call sites are removed, as described in Section 3.3. The third version ("w/ AST") provides estimates of the cost of AST construction and storage. Since we could not construct ASTs for all library classes (due to unavailable source code), we ran the summary-based version and "simulated" the actions of an AST-based analysis: every time the summary-based analysis determined that

Program	w/ summary		w/ optimiz		w/ AST	
	Time	Mem	Time	Mem	Time	Mem
`proxy`	8.2	26.1	7.2	25.1	102.2	233.3
`fractal`	20.2	38.5	18.9	36.9	127.1	302.4
`echo`	15.7	39.3	14.3	37.7	128.0	323.8
`jtar`	21.1	53.3	19.5	51.7	129.4	322.0
`javacup`	9.9	34.6	10.6	33.5	102.9	223.0
`jlex`	8.8	30.0	8.4	28.9	105.5	218.8
`jflex`	23.6	56.3	23.5	54.9	132.8	321.5
`mindterm`	22.8	59.9	21.4	58.4	132.1	337.8
`muffin`	8.8	27.8	8.0	26.7	103.3	214.7
`sablecc`	69.7	89.0	66.1	86.1	158.4	276.2

Table 2: Running time (sec) and memory (MB).

some method m is reachable, the AST for m's class was constructed (or reused, if it had been constructed in the past), and the method's AST was traversed in a manner similar to that of an AST-based analysis. Of course, this was done only for classes with available source code; thus, the results represent a lower bound on the cost of an AST-based analysis. In all three versions, the ASTs were kept alive until the completion of the analysis, and were not released for garbage collection. This was done to obtain better understanding of the amount of heap memory required for AST storage.

The columns labeled "Time" show the running time of the analysis in seconds, and columns "Mem" show the memory usage of the analysis in MB. The results clearly indicate that AST building for library classes results in increased running time and memory usage. The experiments also show that the simple optimizations from Section 3.3 can reduce running time. While this is a small-scale preliminary study, it confirms the motivation for summary-based analysis. For more expensive analyses, in which the summary contains partial analysis results, it is likely that the savings for a summary-based analysis will be even more significant.

5. RELATED WORK

The problems related to practical analysis implementations are not unique to analyses built in Eclipse. However, since the number of static analysis researchers using the Eclipse infrastructure continues to grow steadily, it is critical to have in-depth understanding of the challenges for implementing high-quality analyses. The work described in this paper is a step in this direction.

The closest related work to TACLE is the publicly available Eclipse plug-in that is based on the popular Soot framework for Java [9]. The Spark component of Soot [3] performs points-to analysis, which is a particular form of type analysis. Spark uses state-of-the-art algorithms for points-to analysis, but unfortunately has some drawbacks. First, it is designed to operate in batch mode: an analysis client starts the analysis from scratch, and the analysis algorithms are not incremental. We believe that incremental analyses running in the background are essential in Eclipse, in order to provide a tool user with immediate access to up-to-date analysis information. Second, Spark re-analyzes the relevant library classes every time it is invoked. This repeated work is a major contributor to the overall running time of the analysis; we strongly believe that for this and similar analyses, using library summary information is a necessity in order to be able to analyze large program. Finally, the analysis in Spark takes as input Jimple, which is a low-level intermediate representation constructed from bytecode. One of our design goals for TACLE is to have type analyses that are tightly integrated with the AST representation of the source code. This is especially important for incremental analyses, where AST modifications should trigger updates of the analysis solution.

6. CONCLUSIONS AND FUTURE WORK

Practical implementations of static analyses in Eclipse will require solutions to various challenging problems. One such problem is the handling of the standard Java libraries, as well as other class libraries used by an application. We have investigated an approach based on library summary information. For RTA this approach produces significant savings due to the reduced cost of AST construction and storage. We expect that for more complicated type analyses (e.g., points-to analyses based on [5, 3]), the savings will be even more substantial.

The summary-based implementation of RTA is available at `presto.cse.ohio-state.edu/tacle`. In the near future, we will add to TACLE (1) an implementation of a whole-program summary-based points-to analysis, and (2) a whole-program summary-based side-effect analysis. The long-term goals of the project are to make these analyses incremental, to enable summary-based analysis in the presence of multiple libraries (not just the standard libraries), and to allow analysis of partial programs with unknown clients (as opposed to analysis starting from `main`).

7. REFERENCES

[1] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[2] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[3] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[4] A. Rountev. Component-level dataflow analysis. In *International SIGSOFT Symposium on Component-Based Software Engineering*, LNCS 3489, pages 82–89, 2005.

[5] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, 2001.

[6] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, June 2004.

[7] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 235–252, 1999.

[8] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, LNCS 2622, pages 126–137, 2003.

[9] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.