Estimating the Run-Time Progress of a Call Graph Construction Algorithm*

Jason Sawin
Ohio State University
sawin@cse.ohio-state.edu

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

Abstract

This work considers static analysis algorithms that are integrated with a development environment. In this context, IDE users can benefit from continuously-updated information about the run-time progress of the analysis algorithm (i.e., what portion of the analysis work is completed). IDEs can provide the means to convey this information back to the user — for example, the Java IDE in Eclipse achieves this by employing GUI elements such as progress bars.

Precise tracking of the run-time progress of an analysis algorithm requires a priori knowledge of the total running time of the analysis. Such knowledge is typically not available, and analysis builders need to employ various heuristics to estimate run-time progress. In this paper we describe our initial work on defining and evaluating such heuristics for a whole-program analysis in Eclipse. The analysis, based on the well-known Rapid Type Analysis (RTA) approach, builds a call graph for a Java program, for subsequent use in various software tools. We propose multiple heuristics to estimate run-time analysis progress; these heuristics have been implemented in a publicly available Eclipse plug-in. We report the results of evaluating each heuristic on an large set of Java programs.

1 Introduction

Many modern integrated development environments (IDEs) allow tool builders to define extension modules or "plug-ins" to augment the IDE's base functionality. This trend can be observed in mainstream IDEs such as Microsoft Visual Studio [4] and Eclipse [2]. Such extensions have become a valuable tool for researchers exploring various static analyses of program code. IDEs provide an off-the-shelf infrastructure that supplies researchers with common low-level services that are often needed to perform static analyses — for example, parsers, intermediate representations of code, editors, GUI elements, platform-specific resources, etc. More than just facilitating the building and refinement of an analysis, plug-ins provide a vehicle to widely distribute implementations of analyses that have hither-to been primarily confined to academia.

The Eclipse tool platform is a prime example of an IDE that encourages various extensions. There are dozens of publicly-available Eclipse plug-ins for source code analysis.[1] These plug-ins have the potential to contribute significantly to programmer productivity and software quality, by providing sophisticated tool support for program understanding, evolution, testing, verification, and optimization.

In order to build production-quality implementations of static analyses for use by the general public, static analysis researchers have to face numerous challenges. In addition to the traditional issues of scalability and precision, analysis designers have to pay close attention to the integration with the rest of the IDE, and to the usability issues from the point of view of an end-user (i.e., a programmer using the IDE). A significant challenge for plug-in builders is creating front-end user interfaces that are both meaningful and user friendly. Often the IDE being extended provides a UI framework to which the implementation of the analysis must conform. One typical component of such frameworks is a *progress bar*; an example of a progress bar is shown in Figure 1. Progress bars provide valuable feedback to the users, assure them that the desired task is making progress, and provide an estimate of the remaining time to complete the running task. Even though progress bars are an important part of

*This work was supported, in part, by an IBM Eclipse Innovation Grant.

[1] For example, the Eclipse web site lists more than 30 plug-ins that are related to code analysis. As another example, the proceedings of the Eclipse Technology Exchange Workshop show a large number of research plug-ins for code analysis.

UIs (as a mechanisms for *user responsiveness*), they are often overlooked or implemented poorly [3]. For a static analysis designer, an interesting challenge is to provide the IDE with enough information to allow a meaningful progress bar to be displayed while the analysis algorithm is running.

Precise tracking of an application's progress requires a priori knowledge of the *total amount of work* performed by the application. For example, an application that downloads a file may use file size as a priori measure of the total amount of work that will be performed. This application may indicate *a unit of progress* has been completed, for example, for every 1KB downloaded. Unfortunately, it is typically impossible to know ahead of time the amount of work that will be performed by a static analysis. Thus, heuristics must be employed to estimate the total running time and the progress made during the analysis execution. Analysis designers are faced with the following questions: *What are possible heuristics for producing a priori estimates of analysis running time? How should the analysis estimate the progress it is making? What are appropriate metrics of precision and cost for the estimation techniques?*

This paper presents our initial work on answering these questions for a specific whole-program analysis for Java, implemented as an Eclipse plug-in. The analysis, based on the well-known Rapid Type Analysis (RTA) approach [1], builds a call graph for a Java program in an Eclipse project. We consider this work to be a first step in a longer-term research investigation of accurate and efficient heuristics for tracking the runtime progress of various static analyses. The specific contributions of this work are as follows:

- We define two cases for tracking the progress of a static analysis: *initial analysis* without any history, when the analysis is executed on a particular program for the first time, and *repeated analysis*, when the analysis is executed repeatedly on different versions of the same program. We propose a history-based approach for repeated analysis that could potentially be used by various static analyses to produce more accurate progress estimates.

- We define metrics that evaluate the quality of progress-tracking heuristics. These metrics consider both the precision of the generated progress information, as well as certain aspects of its user-friendliness (Section 4).

- We propose and implement multiple heuristics for estimating the progress of a whole-program RTA call graph construction algorithm for Java, both for an initial analysis and for a repeated analysis (Section 3). These heuristics will serve as the starting point for generalizing our approach to other static analyses such as points-to analysis and side-effect analysis.

- We present the results from an experimental study of the heuristics on a large set of Java programs (Section 5). This experimental evaluation provides insights about the strengths and weaknesses of the proposed techniques. The results of the study will be used to decide which heuristics to incorporate in the next public release of our analysis plug-in.

2 Background

This section provides brief descriptions of our analysis plug-in, the analysis algorithm it currently uses, and the progress monitor interface in Eclipse.

2.1 TACLE

The problem considered in this paper was motivated, in part, by our work on the TACLE project[2] [7]. TACLE stands for type analysis and call graph construction for Eclipse. The project produced an open-source Eclipse plug-in that performs a whole-program type analysis and call graph construction for a Java program in the Java IDE in Eclipse.

Call graphs are essential for any form of interprocedural static analysis. Instead of "rediscovering the wheel", implementers of such analyses could save valuable time and resources by interfacing with TACLE and utilizing the call graph it produces.

In the current release of TACLE, the analysis algorithm executes in the foreground and provides no feedback to the user until the processing has completed. The inability to understand how the analysis makes progress, and if such progress is made at all, could be rather frustrating for Eclipse users, especially when the analysis time is non-trivial (e.g., more that a few seconds). There have been requests from TACLE users to make the analysis more user-friendly, with the help of the progress monitor interface defined by Eclipse (this interface will be described shortly). After the addition of progress monitoring, it becomes possible to provide useful GUI representations as shown in Figure 1.

2.2 Rapid Type Analysis

TACLE implements a version of Rapid Type Analysis (RTA) [1]. RTA takes a complete program as input

[2]Available at `presto.cse.ohio-state.edu/tacle`

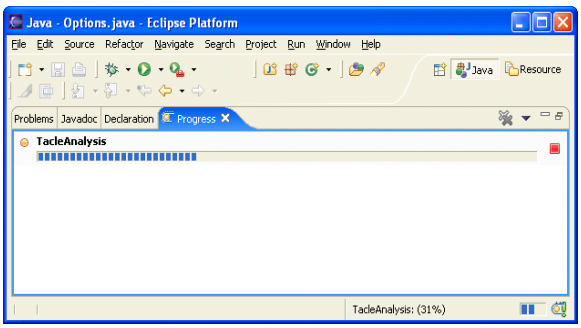

Figure 1. Progress bar for TACLE in Eclipse

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*.

To implement RTA, TACLE uses a worklist algorithm. Initially, the only elements of the worklist are the main method and the library methods executed at JVM startup. When the algorithm removes a method from the worklist, it considers two cases. First, if the method is a user-defined method (i.e., not a library method), the analysis builds and processes the method's abstract syntax tree (AST) using Eclipse. To reduce analysis time and memory, TACLE utilizes summary information about the standard Java libraries. The library summary contains the information needed to perform RTA — namely, library classes, methods, call sites, and object creation sites. Thus, if the method removed from the worklist is a library method, TACLE examines the summary for this method and takes the appropriate actions.

2.3 Run-time Progress in Eclipse

To improve the user-friendliness of TACLE, we utilized the Eclipse Job API which allows clients to execute tasks in separate threads. This service of the Eclipse platform provides a rich functionality which, for brevity, is not discussed in detail in this paper. The basic features of the service are as follows. Runnable work is wrapped in subclasses of class `Job`. Jobs have properties similar to Java threads, in that they have `sleep`, `wakeup` (similar to `resume`), `join` and `run` methods. Jobs can be scheduled for immediate or delayed execution, and various rules can be associated with them. An invocation of a method `run` declared in some subclass of `Job` results in the execution of the corresponding job. A `run` method takes as its only parameter an `IProgressMonitor` object. This method is invoked by the scheduler when the job is at the head of the schedule queue; in this call, the scheduler passes to `run` an implementation of a progress monitor.

An implementation of `IProgressMonitor` is an object that monitors the work being performed by the job. The key methods defined in this interface are as follows:

- `void beginTask(String taskName, int totalWork)`: this method notifies the progress monitor that a task has started. Parameter `totalWork` is the total number of work units the task is projected to complete.
- `void worked(int work)`: this method notifies the monitor that task has completed the number of work units indicated by `work`.
- `void done()`: alerts the monitor that the task has completed its work.

These methods are used to drive the GUI progress bar provided by Eclipse, as illustrated in Figure 1.[3] With each invocation of method `worked`, the monitor updates the progress bar. This continues until the incremental installments of `work` equal `totalWork`, or until `done` is invoked. Either of these events result in the progress bar showing that 100% of the work has been completed. It is left to the client of the API to ensure that the progress monitor is initialized with the correct `totalWork` value, and that methods `worked` and `done` are invoked at the appropriate times with the correct values.

In TACLE, the call graph construction algorithm can be executed as a task. As a result, the user has the ability to browse code in Eclipse even while that code is being analyzed by our plug-in. In this context, two obvious questions must be answered. First, *what is the total amount of work to be used in the call to* `beginTask`? Second, *when and with what work increments should* `worked` *be called to alert the monitor that*

[3]Eclipse provides multiple styles of progress bars, falling into two main categories: *measured* and *indeterminate*. A measured bar displays the portion of work that has been completed. An indeterminate bar indicates work is being conducted without providing information about the percent of effort completed. In this work we consider only measured progress bars.

progress has been made? The next section presents our initial work on answering these questions. While the proposed techniques are specific to this particular analysis algorithm, the two questions from above will have to be answered by *any* static analysis designer that intends to provide a measured progress bar in Eclipse. Furthermore, these questions are relevant not only for Eclipse-based analyses, but also for any progress reporting inside a static analysis tool.

3 Estimating Run-time Progress

Of course, it is impossible to know in advance the precise amount of work RTA will perform on a given program. Thus, the key challenge is to define some heuristic estimate of the total amount of work, and to provide it as a parameter of the call to `beginTask`.

The heuristics outlined in this paper consider two separate cases. First, we focus on the case of an *initial analysis*, which is executed for the first time on a given program. Next, we consider the case of a *repeated* analysis, which is performed on a new version of a program what had already been analyzed in the past. In this case, some high-level historical information from the last analysis execution is used to create estimates for the current analysis run.

3.1 Initial Analysis

The following are the heuristic that we consider for the case when no historical information is available from previous runs of the analysis on the analyzed program (or from runs on earlier versions of that program).

3.1.1 Naive Estimate

The bulk of the analysis processing time is spent in a loop, where each loop iteration processes one worklist element. Thus, the total amount of work for the analysis is (roughly) proportional to the *number of methods reachable from the main method*. We use an estimate of this number as a parameter to the call to `beginTask`. Subsequently, whenever a method is taken off the worklist and processed, we make a call `monitor.worked(1)` immediately after the processing, indicating that one unit of work was performed.

The key question, of course, is how to obtain the estimate of the total number of reachable methods *before* we run the RTA algorithm. In this simple heuristic, we use a hard-coded, predefined estimate of 8101 reachable methods. This number was obtained as the average number of reachable methods across 15 Java programs. In our experiments, this heuristic (and all other ones) was evaluated on a set of Java programs that were *different* from the 15 programs used to obtain this hard-coded estimate. The number is rather large because of the significant number of library methods that are transitively reachable from the main method.

In the case when the actual number of reachable methods in the analyzed program is greater than 8101, the progress bar will reach 100%, and will be "stuck" at that point until the algorithm completes. On the other hand, if the number of reachable methods is less than 8101, the progress bar will jump from some smaller value (e.g., 70%) directly to 100% at the end of the analysis. Clearly, both of these behaviors are undesirable. Later in the paper we discuss metrics that quantify the imprecision of this and other heuristics.

Note that another possible estimate of work could be based on analysis time. For example, we could have computed the average running time over the 15 programs, and used that value (e.g., as an integer value in milliseconds) in the call to `beginTask`. During the subsequent analysis of some program, we could have called `worked`, at regular time intervals, with the appropriate integer value representing the duration of the last interval. We believe that such a time-based approach is not desirable for the initial analysis. We plan to compute the hard-coded estimate on a local machine in our department, and to include it in the public distribution of TACLE. Since TACLE users may use computers that are significantly different from ours, the machine-dependent running time is not a "portable" estimate. Later we discuss the use of a time-based estimate for the case of a repeated analysis, where the analysis is executed multiple times on different versions of the same program, on the same physical machine.

3.1.2 Number of User-Defined Methods

This technique uses estimates based on the program that is about to be analyzed with RTA. The estimate of total work in the call to `beginTask` is *the number of user-defined methods* in the Java project (inside the Eclipse Java IDE) for the program being analyzed. In Eclipse, it it easy and relatively cheap to find this number before RTA starts. During the analysis, `monitor.worked(1)` is called immediately after a user-defined method is taken off the worklist and processed.

Note that there are two sources of imprecision for this technique. First, there may be user-defined methods that are not reachable from the main method. Typical examples are methods that contain code related to testing, as well as deprecated methods. More importantly, the user-defined methods are only a small por-

tion of the total number of reachable methods — most of the methods in the call graph are methods from the standard Java libraries.

3.1.3 Number of User-Defined Methods and Library Entry Methods

This approach refines the user-methods-based technique presented above. The estimate of total work is the number of user-defined methods in the Java project plus the number of library methods called by these user-defined methods. For all user-defined classes in the Java project, we build the corresponding ASTs and count the number of library methods that are static targets (i.e. compile-time targets) of calls inside the ASTs. During RTA, `monitor.worked(1)` is called whenever the analysis processes (1) a user-defined method, or (2) a library method that was counted in the definition of the estimate. Since the processing of library methods contributes significantly to the analysis cost, taking into account some of the reachable library methods may produce better progress estimates, compared to just using the number of user-defined methods.

3.2 Repeated Analysis

The estimation techniques presented above are severely constrained in terms of the information they have about the analyzed program. However, once the analysis has been executed once on a program, certain *historical information* can be saved and used subsequently when modified versions of this program are analyzed in the future. In Eclipse, this information can be saved in the workspace of the Java project being analyzed, and can be used later when the analysis is executed again on the same project. Of course, we assume that some changes have occurred in the code of the project between successive invocations of the analysis; if no changes have been made, progress estimation based on historical data is trivial.

This approach assumes that the analysis is likely to be applied on multiple versions of the same program. We believe that this is a legitimate assumption. For example, we plan to use TACLE as part of our tool for reverse engineering of UML sequence diagrams [5, 6]. Such reverse-engineered diagrams can be used to ensure that successive modifications of the analyzed program still adhere to the original design. In this case, the analysis would be run multiple times on slightly different versions of the same application. More generally, it is quite common for the same static analysis to be applied multiple times to an evolving program.

3.2.1 Number of Reachable Methods

Similarly to the naive estimate for the initial analysis, this technique uses the number of reachable methods to estimate the total amount of work. However, there is no hard-coded predicted number of reachable methods. Instead, this approach uses the number of reachable methods from the last execution of the analysis on this program — more precisely, on the earlier version of the program at the time of that last run. During the worklist algorithm, every processed method (in user code or in library code) corresponds to a unit of work.

3.2.2 Methods Weighted by Relative Time

Not all methods contribute equally to the cost of the analysis. In order to refine the naive approach from above, each execution of the analysis saves (1) the list of reachable methods, and (2) the percent of analysis time that was spent processing the body of each method. These method weights are subsequently used in runs of the analysis on a modified version of the program. The total amount of work is computed as the sum of the weights, and the parameter of the call to `worked` is the method's weight. If the analysis processes a method that was not reachable in the previous analysis run (e.g., because this method was added to the program after the last run), the analysis does not make a call to `worked`.

3.2.3 Elapsed Time

A very simple technique for estimating progress is the following. Each execution of the analysis saves its total running time. This time is used as an estimate for the total amount of work the next time the analysis is executed. At regular intervals, the analysis reports the appropriate unit of work to `worked`. If the total running time of the analysis is exactly the same as in the previous run, this approach will produce perfect progress information.

4 Evaluation of Progress Reports

There are several properties that determine the quality of a progress monitor. The first is the ability to *accurately* represent the progress of a running application. For example, a progress monitor that indicates an application is 75% complete when in reality it is only 25% complete is very misleading to the user. To evaluate the accuracy of our heuristics, every time a call to `worked` was made, we recorded (1) the heuristic's current estimate of progress, and (2) the actual system time for which the analysis has been running

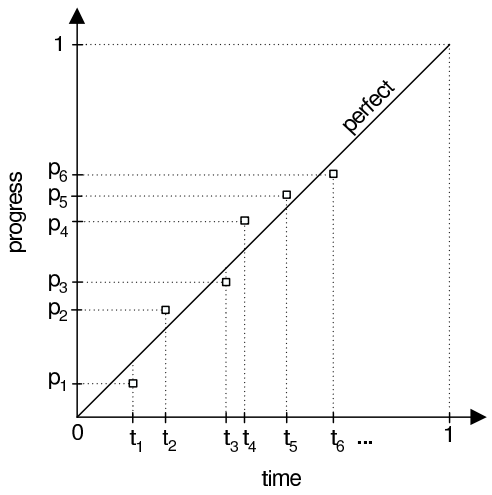

Figure 2. Measurements for progress reports

Comp	.java	.class	Meths	ReachMeths
`javacup_h`	39	41	382	5665
`javacup_i`	39	41	382	5665
`javacup_j`	40	42	391	5673
`jflex1.3.3`	48	60	446	7903
`jflex1.3.4`	48	60	447	7904
`jflex1.3.5`	48	60	447	7904
`jgraph5.7.4.6`	58	137	1467	11505
`jgraph5.7.4.7`	58	137	1467	11505
`jgraph5.8`	59	137	1475	11518
`jpws.2.0`	89	141	1130	12246
`jpws.3.0`	103	187	1459	12712
`jpws.3.1`	104	193	1491	12771
`sablecc.2.8`	210	249	2108	7327
`sablecc.3.1`	198	267	2138	7528
`sablecc.3.2`	198	267	2137	7517
`verbos.1.4`	50	57	479	10947
`verbos.1.5`	52	58	513	10988
`verbos.1.7`	54	60	545	11117
`vietpad.1.2`	79	197	577	11128
`vietpad.1.2.1`	79	197	578	11135
`vietpad.1.3`	97	215	596	11186

Table 1. Subject Programs

up to this point. Upon completion of the analysis, the total running time was used to evaluate the actual percentage of running time that the application had completed for each estimation point. These measurements are illustrated in Figure 2. Each estimation time point t_i corresponds to a call to `worked`. The values of t_i are in the interval $(0, 1]$ — that is, they are normalized by the total running time of the analysis. The last estimation point is $t_{last} = 1$, corresponding to the end of the analysis.

The y axis in Figure 2 corresponds to the cumulative progress made up to this estimation point. These values are also normalized to belong to the interval $(0, 1]$. More precisely, we have

$$p_i = (\Sigma_{k \leq i} \texttt{worked}_k)/\texttt{totalWork}$$

where $\texttt{worked}_k$ is the amount of work reported by the call to `worked` at time t_k. For an estimation point at time t_i, we compute

$$\Delta_i = |p_i - perfect_i|$$

where $perfect_i$ is the progress report that would have been made by a "perfect" estimation technique.[4] The average value of Δ_i over all estimation points t_i can be used as an overall indicator of the accuracy of an estimation technique. Note that in some cases, the estimates may reach 1 before the actual analysis has completed; in this case, the values of p_i are equal to 1 for the rest of the analysis running time. Ideally, the average value of Δ_i should be close to 0.

The second desirable property of a progress monitor is the *smoothness* at which it progresses. A monitor that only indicates progress at 25%, 75%, and 100% is not particularly useful. The smoothness depends on (1) the number of times when progress is indicated through a call to `worked`, (2) the length of time between the calls to `worked` and (3) the amount of work indicated at each call to `worked`. Since `worked` is called relatively frequently, factors (2) and (3) are the important ones.

Intuitively, the smoothness can be estimated by considering the *slope* of the "perfect" and "real" functions in Figure 2.[5]. For each estimation point t_i we compute the difference between the slopes, as follows

$$\epsilon_i = |1 - (p_i - p_{i-1})/(t_i - t_{i-1})|$$

The average value of ϵ_i over all estimation points t_i is an indication of the smoothness of the progress reports. Ideally, this average value should be close to 0.

An important issue to consider is the run-time cost for computing the estimates. While in some cases this cost is trivial (e.g., for the naive approach from Section 3.1.1), for other techniques the cost could have an impact on the user. This is especially important for the pre-RTA computation that estimates the total amount of work. Until this estimation is completed, a progress monitor cannot be initialized and displayed. A lengthy pre-computing step may lead the user to believe that the task has stalled.

[4]Since both t_i and p_i are normalized, $\Delta_i = |p_i - t_i|$.

[5]If these were continuous functions, we would compare their first derivatives; the "perfect" function has a derivative $f'(x) = 1$

	(1)		(2)			(3)		
Program	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i	PreProc	Avg Δ_i	Avg ϵ_i	PreProc
javacup_h	.13	.70	.09	3.30	5.6%	.12	2.91	33.9 %
javacup_i	.13	.72	.10	3.07	3.9%	.12	2.81	33.1%
javacup_j	.12	.74	.10	3.17	5.4%	.12	2.92	34.1%
jflex1.3.3	.04	1.06	.12	3.69	1.8%	.14	3.28	28.4%
jflex1.3.4	.03	1.07	.12	3.84	1.6%	.14	3.44	24.6%
jflex1.3.5	.04	1.06	.12	3.77	3.1%	.14	3.36	25.7%
jgraph5.7.4.6	.15	4.55	.17	6.27	.5%	.15	5.48	17.9%
jgraph5.7.4.7	.15	4.55	.17	6.05	.7%	.16	5.48	17.9%
jgraph5.8	.15	4.71	.18	6.18	.7%	.17	5.60	17.6%
jpws.2.0	.21	2.71	.04	4.23	.5%	.05	3.40	22.7%
jpws.3.0	.27	5.18	.05	6.35	.6%	.05	5.29	20.7%
jpws.3.1	.27	5.55	.04	6.45	.6%	.04	5.60	19.5%
sablecc.2.8	.18	10.19	.09	8.57	.3%	.09	9.76	9.9%
sablecc.3.1	.07	12.91	.10	10.62	.1%	.11	11.32	11.0%
sablecc.3.2	.07	13.03	.10	10.89	.1%	.11	11.03	10.8%
verbos.1.4	.20	1.32	.11	4.75	.6%	.09	2.83	16.9%
verbos.1.5	.20	1.31	.10	4.43	.6%	.09	2.63	16.8%
verbos.1.7	.20	1.30	.12	4.21	.6%	.09	2.55	17.7%
vietpad.1.2	.26	12.07	.27	16.30	.1%	.33	13.00	3.9%
vietpad.1.2.1	.26	12.07	.27	18.49	.1%	.33	12.95	3.9%
vietpad.1.3	.26	11.18	.25	18.59	.1%	.33	13.17	4.2%

Table 2. Initial analysis: (1) naive, (2) number of user-defined methods, (3) number of user-defined methods and library entry methods

5 Experimental Study

This section describes an experimental study of the estimation techniques defined earlier. For the experiments we used several versions of open-source Java applications, as shown in Table 1. Column "Meths" shows the number of user-defined methods in the program, while column "ReachMeth" shows the number of reachable methods reported by RTA, including all reachable library methods.

5.1 Initial Analysis

Table 2 contains the evaluation results for the three estimation techniques for the initial analysis (Section 3.1). Columns "Avg Δ_i" show the average difference between p_i and $perfect_i$ for all estimation points, as described in Section 4. Recall that smaller average values for Δ_i indicate better accuracy; ideally, the value should be close to 0. Columns "Avg ϵ_i" shows the average difference between the slopes of the estimated curve and that of the perfect slope (Section 4). Smaller values mean better smoothness, with ideal values being close to 0. Finally, columns "PreProc" contain the preprocessing time needed to obtain the initial estimate (e.g., to count the number of user-defined methods for technique 2), as percent of the total running time of the analysis.

Naive estimate. The results for the naive approach show that the accuracy of the estimates depends the number of reachable methods in the call graph. Applications with a total number of reachable methods close to the 8101 method estimate, such as `sablecc` and `jflex`, have low Δ_is. However, applications with a much larger or lower number of total reachable methods have much greater values of Δ_i, as much as .27 in some cases. This indicates that the estimated progress is 27 percentage points off on average. The variance seen in the smoothness results can be attributed to the differing composition of the applications. Since this approach reports that one unit of work is completed every time a method is processed, to achieve the ideal slope, every method must take the exact same time to process. This of course is not the case — not only do methods often vary greatly in size and complexity, but the analysis itself processes library methods differently then user defined methods (Section 2.2). In some cases, such as `sablecc`, series of small methods may be processed very quickly producing sections of very steep slope. Other methods in the same application may take a very long time to process and will produce a very flat slope. This combination of very steep and very flat slopes leads to a large discrepancy between the average slope of the estimates and the ideal slope. This, of course, indicates that the heuristic does not generate a very smooth result.

Number of user-defined methods. For the estimation technique based on the number of user-defined

	(1)		(2)			(3)	
Program	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i	PreProc	Avg Δ_i	Avg ϵ_i
javacup_i	.07	1.01	.02	.53	.7%	.01	.09
javacup_j	.07	1.04	.02	.51	1.0%	.01	.11
jflex1.3.4	.05	1.11	.01	.63	.7%	.02	.04
jflex1.3.5	.05	1.16	.02	.63	0.9%	.03	.01
jgraph5.7.4.7	.16	3.00	.07	1.71	.1%	.009	.009
jgraph5.8	.17	3.12	.08	1.73	.1%	.01	.02
jpws.3.0	.03	3.24	.03	2.00	.1%	.05	.03
jpws.3.1	.04	3.53	.02	1.98	.1%	.02	.02
sablecc.3.1	.08	14.79	.08	7.35	.02%	.001	.006
sablecc.3.2	.08	14.90	.03	6.59	.02%	.05	.57
verbos.1.5	.03	.96	.02	.62	.2%	.009	.06
verbos.1.7	.03	.95	.01	.62	.5%	.03	.01
vietpad.1.2.1	.32	8.53	.16	4.42	.03%	.004	.01
vietpad.1.3	.32	8.49	.16	4.47	.03%	.02	.07

Table 3. Repeated analysis: (1) number of reachable methods, (2) methods weighted by relative time, (3) elapsed time

methods, the overhead of counting all user-defined methods is not an issue (column "PreProc"), and the user will not be affected by this delay. The comparison with the naive approach shows that the second technique is slightly more accurate, but the progress is less smooth. The reduction in smoothness is not surprising, since the user-defined methods are a small subset of all reachable methods, and their processing is not necessarily uniformly distributed throughout the duration of the analysis.

Number of user-defined methods and library entry methods. The last portion of the table shows the results for the technique that takes into account the number of library methods called by user-defined methods (Section 3.1.3). Compared with techniques (1) and (2), (3) does not provide any significant increase in accuracy. This is due to the fact that the number of data points added — by including the library entry methods — is relatively small. For example, in `javacup_h` only 42 unique library methods were identified and added to the estimate. The highest number of library methods, 409, was added for `vietpad.1.3`. However, this number is still low when compared to the total number of reachable methods. The increased number of data points did provide a slightly smoother result than that of technique (2). However, any improvement seen in technique (3) is overshadowed by the prohibitive preprocessing time it requires. In smaller applications the cost of pre-building the ASTs (which is how the calls to the library methods were found) can increase the running time as much as 34%, making the heuristic impractical for use.

Conclusions. Of the techniques we investigated for the initial analysis, it appears that basing the estimation on the number of user-defined methods is the most practical approach for TACLE. It provided the most consistently accurate estimation for an acceptable amount of pre-processing time. It has a distinct advantage of not appearing to stall at 100% for long periods of time, as is the case for technique (1) if the application contains significantly more reachable methods than the hard-coded estimate.

5.2 Repeated Analysis

To evaluate the estimation techniques for repeated analysis, we considered each pair of consecutive program versions (e.g., `jflex1.3.4` and `jflex1.3.5`). The earlier version was executed to obtain historical information, as described in Section 3.2. This information was then used to create estimates for the analysis of the later version.

Table 3 shows the results of these experiments. There are several interesting observations that can be made about the results. First, comparing techniques (1) and (2), it is clear that the finer-grain information about relative method weights is beneficial for improving the precision of the estimates. Since the methods that take longer to process are weighted to show more work, (2) produces a dramatically smoother result. It is also clear that the cost of pre-processing for technique (2) is negligible: basically, this is the cost of reading from disk a list of reachable methods together with their weights.

Technique (3) provides the most accurate and smoothest results. The inaccuracies seen are due to the slightly different running times seen between versions. The smoothness of this approach is very nearly ideal, as it is designed to report progress in increments that are exactly proportional to the amount of time that passes

	(1)		(2)		(3)	
Program	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i	Avg Δ_i	Avg ϵ_i
javacup_i	.07	1.05	.05	.73	.35	.82
javacup_j	.07	.98	.03	.84	.39	.90
jflex1.3.4	.05	1.23	.01	.84	.46	1.01
jflex1.3.5	.05	1.08	.02	1.02	.42	.52
jgraph5.7.4.7	.15	3.49	.08	2.12	.64	1.27
jgraph5.8	.18	3.42	.08	2.07	.64	1.35
jpws.3.0	.04	3.86	.03	2.55	.67	1.05
jpws.3.1	.04	3.45	.02	2.89	.65	.98
sablecc.3.1	.08	19.96	.11	9.47	.71	1.52
sablecc.3.2	.08	18.71	.05	8.96	.70	1.44
verbos.1.5	.03	1.14	.02	.82	.41	.86
verbos.1.7	.03	.94	.01	1.16	.39	.47
vietpad.1.2.1	.33	13.27	.16	6.20	.53	.46
vietpad.1.3	.32	13.45	.16	7.01	.55	.87

Table 4. Repeated analysis under load: (1) number of reachable methods, (2) methods weighted by relative time, (3) elapsed time

between reports (i.e. $1 = (p_i - p_{i-1})/(t_i - t_{i-1})$). The cost of pre-processing for this technique is the same as technique (1), a read of one line from a file, which is negligible.

Unfortunately, the quality of estimates produced by technique (3) depends on the execution environment in which the analysis is operating. The results in Table 3 were gathered on a pristine environment, meaning the analysis was the only user-level application executing in the operating system. Table 4 shows the same experiments (including the same history summaries) conducted on a system with load. The load was generated by concurrently running `Google Earth`[6], a CPU and memory intensive program. This represents a more "real world" situation for TACLE, where the amount of load can easily vary between executions. In the case of the experiments under load, the total running time of the analysis was significantly longer then the estimates gained by technique (3) from the no-load summary information. Thus, the progress monitor was "overly optimistic" — it reported significant progress percentages based on the elapsed time, while in reality the analysis had made very little progress, being off as much as 71 percentage points on average for `sablecc`. Note that the other two techniques for repeated analysis were not as susceptible to environmental changes. The approach from Section 3.2.1 — technique (1) — is clearly independent of the machine load. The approach from Section 3.2.2 — technique (2) — considers *relative* times, not absolute ones. Therefore, if the machine load does not change significantly during the execution of the analysis, the weights computed at one load level should also be valid at another load level. The slight differences in accuracy for (1) and (2) between the loaded and the unloaded environments are likely due to the fact that the load generated by `Google Earth` was not completely constant.

Considering technique (3)'s reliance on a consistent environment, we feel that technique (2) is best suited for use in TACLE, because it provides an accurate low-cost estimation and is relatively robust.

[6] http://earth.google.com/

6 Conclusions and Future Work

This paper presents an initial investigation into the difficulties of estimating the run-time progress of static analyses. For the specific analysis considered in our work, we propose and evaluate several techniques for progress estimation. In the case of an initial analysis without any historical information, the counting of user-defined methods is a simple and reasonable solution. For a repeated analysis, methods weighted with relative times from past executions appear to be a good choice for TACLE.

Clearly, the problem we consider has much broader applicability than the specific RTA-based analysis we discuss. We believe that many static analysis designers will have to face the problem of estimating analysis progress, especially when the analysis is part of a modern software development environment or tool. In this paper we provide such designers with quantifiable metrics that can be used to evaluate their efforts. Our experience also provides evidence that utilizing summary information in consecutive runs of the analysis produces much more accurate progress estimates.

In the future we plan to implement other whole-program analyses in TACLE (e.g., side-effect analy-

sis and dependence analysis), and to investigate techniques for estimating their run-time progress. We also plan to consider usability testing to establish user-acceptable thresholds for our metrics.

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] Eclipse: An Open and Extensible IDE. http://www.eclipse.org.

[3] J. Johnson, editor. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[4] Microsoft Visual Studio. http://msdn.microsoft.com/vstudio.

[5] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *International Conference on Software Engineering*, pages 254–263, 2005.

[6] A. Rountev, O. Volgin, and M. Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102, 2005.

[7] M. Sharp, J. Sawin, and A. Rountev. Building a whole-program type analysis in Eclipse. In *Eclipse Technology Exchange Workshop*, pages 6–10, 2005.