Call-Graph Based Program Analysis with .Net

Walter Prins, University of Liverpool, Laureate Online Education B.V., Arena Boulevard 61-75, 1011 DL Amsterdam Z.O., The Netherlands; E-mail: prins@ananzi.co.za

Paul Darbyshire, Victoria University, P.O. Box 14428, Melbourne City MC, Victoria 8001, Australia; E-mail: Paul.Darbyshire@vu.edu.au

ABSTRACT

Software development is a complex business, whether maintaining or extending existing legacy systems, or whether developing new systems. Another challenge faced by programmers, is determining whether sufficiently rigorous unit- and integration-testing is employed to give confidence that a system is behaving as intended. One approach to help address such challenges is to use automated program analysis tools and techniques, where the programmer will use a software tool to gain an insight into some aspect of the system they're working on. One particular type of static program analysis technique, call-graph analysis, focuses on the calling relationships that exist in a program. One of the common problems with this and other static analysis techniques is that they tend to be source language based and are therefore often limited in terms of applicability, especially in multi-language/module systems. In this research we investigate call-graph analysis on the .Net platform that sidesteps these common limitations and allows analysis of programs regardless of source language, and regardless of the number of modules/assemblies in the program. We demonstrate the soundness and usefulness of the approach by demonstrating the analysis of a multi-module application that is written in several different source languages from 2 different vendors.

Keywords: Call-graph analysis, .Net, Program analysis, Integration testing

1. INTRODUCTION

Key challenges faced by programmers today include the difficulty of understanding complex codebases while performing maintenance and ensuring that test-suites sufficiently cover the code in question. Software maintenance particularly often involves many difficulties, including gaining an understanding of the system being modified or analyzing an existing system as a whole. Understanding a system is often complicated by documentation which is either out of date, limited or in some cases even non-existent, and analyzing a system is complicated when it spans over several languages, implementation modules, and/or process boundaries. By 1990, the amount of legacy code being maintained was already estimated at 120 billion lines of code (Sommerville 2001, p. 623.) Today, this is estimated to be in excess of 250 billion lines of code! (Losch 2005) With a legacy codebase of this size being maintained, the argument for re-engineering systems instead of outright replacement becomes quite compelling.

These challenges are typically addressed through suitable program comprehension and -coverage analysis tools. A key data structure used by both these types of program analysis tools is the program call-graph. Call-graph analysis focuses on the particular calling relationships that exist in a program, and the results of such a tool are very useful in determining the call relationships used to track errors and design suitable test data for unit and integration testing. However, conventional program call-graph analysis often relies on program source code parsing techniques, which limit it to that particular source language. Thus, if the system consists of multiple modules written in multiple languages then it follows that it would be extremely difficult at best to do a full-program analysis, unless you have analysis tool(s) that can read all the source languages and can inter-operate when generating the call graph. An alternative approach that addresses both these limitations is to perform analysis on .NET Common Intermediate Language (CIL) instead of the program source language. In this research we develop a simple software framework and a prototype call-graph analysis tool using CIL, thus in principle demonstrating the feasibility of this approach in practice.

The contribution of this paper is in the development of the prototype model mentioned which provides a usable foundation from which further work may be conducted, on the .Net platform and CIL. Details of the model are given in Section 3, followed by a short discussion and outline for future work.

2. BRIEF LITERATURE REVIEW

While we've had program development, there seems to be renewed interest in many types of such tools. This interest is driven by several factors, including the increased capacity for analysis on today's machines, the increasing ability for analysis as a result of software platform advances, and the changing focus of the industry with respect to software development and maintenance. Ultimately however all of these things can be largely drawn back to the problem of software maintenance and software change. If a system was not that well designed to begin with, or if it's been modified in less than ideal ways and accumulated a substantial so called "code debt" (Fowler 2000, pg. 66), gaining an understanding of the codebase, clarifying its original design intentions and developing meaningful test suites are crucial (for which such tools are very useful.)

Call Graph Analysis

Olin Shivers' provides a succinct description of the call graph analysis problem: "*For each call site c in program P, what is the set F(c), that c could be a call to?*" (Shivers 1988):

Call Graph analysis is the process of generating a program call multi-graph (call-graph) for a program. It's a directed graph where nodes represent procedure, function or method names, and edges represent calling relationships (Aiken 2005). A program call graph is therefore a control flow representational construct at the inter-procedural level (therefore not showing detail inside procedures/methods.) A call graph may also be described in textual form by enumerating all the nodes) together with the set of edges between them. Note however that this description actually encompasses several possible meanings, for example:

method m_i *invokes method* m_j

call site c_k *inside mi invokes* m_j

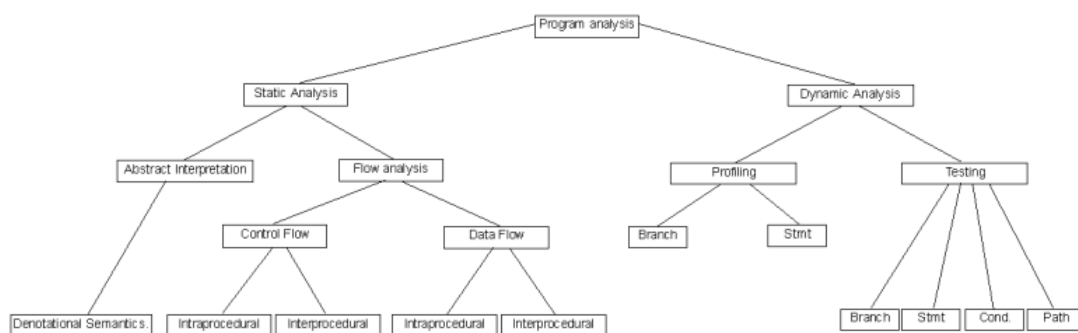
call site c_k *inside mi invokes* m_j *on an instance of X* (Rountev, Kagan & Gibas 2004)

Figure 1 below shows a taxonomic breakdown of program analysis in general, serving to place call-graph analysis into the broader program analysis context:

Call graphs are central to various types of compiler optimizations, including both inter-procedural optimization (where the effects of callers and callee's are summarized into the call graph) as well as intra-procedural optimization (for example where the included receiver class sets may allow method invocation to be bound statically instead of dynamically). Call graphs are also central to several other types of analysis such as call-chain analysis and call-tree coverage analysis, and is also useful in various types of developer tools, such as test tools, debug tools and program understanding tools.

Copyright © 2007, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 1. Taxonomic breakdown of program analysis (Losch 2005)



Call graph generation in first order languages such as FORTRAN is very easy: It can be performed by firstly generating the nodes by finding all functions in the program, followed by insertion of edges for each function call that exists in the program. That is, for each call to function *b()* in arbitrary method *a()*, you insert an edge (a,b) (Lakhotia 1993, pg. 273).

In higher order languages things are unfortunately substantially less simple, due to the requirement to estimate the receiver classes at call-sites prior to call-graph construction. The problem is more or less as follows: In order to perform Inter-procedural data flow analysis (a process whereby you compute summaries of the effects of callers and callee's at function/procedure entry points and call-sites respectively, which may be consulted during optimization (Grove 2001, pg. 686)), you need to have already constructed a call graph that may be traversed during this analysis. As mentioned, in first-order languages there is no problem as the the target function is directly and unambiguously evident from the call site. However, with object-oriented languages with dynamic dispatch mechanisms, the actual target of a call site is usually dependent on the data flow(s) to that point in the program (in particular, the actual class type of the object variable on which the method call is performed), which implies that you need to have already performed some form of data flow analysis in order to discover the actual receiver classes for a call site (ibid)! Thus there occurs a seemingly paradoxical situation.

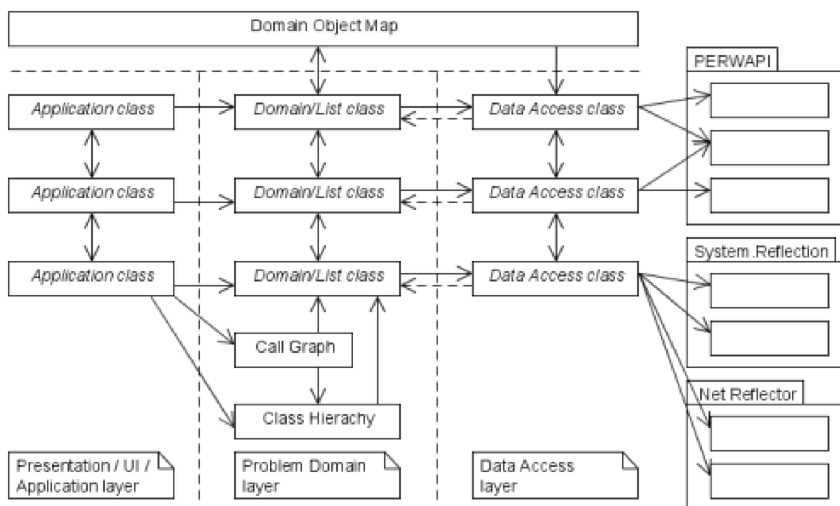
Typical solutions include performing the two steps in parallel (to be precise to interleave them), or to make suitable assumptions (whether optimistic or pessimistic) for one of the three entities involved (call-graph, receiver class sets or inter-procedural analysis) in order to break the deadlock and then iterating the solution to a suitable solution point (ibid). Dean compares and contrasts several techniques for the construction of call-graphs in the presence of "higher-order" functions and goes on to evaluate them in terms of three properties, namely: Precision; Time complexity; Supported Language features (Dean 1997, pg. 2).

Grove & Chambers present a fairly comprehensive evaluation of existing call-graph construction algorithms in (Grove 2001). They implement the evaluation via a generic parameterized call-graph construction algorithm implemented in the context of an optimizing compiler infrastructure. This approach allows them to validly compare different call-graph algorithms on a "level playing field."

Call Graph Analysis at Intermediate Code Level

While some research has been done with respect to program analysis in general and call-graph construction in particular using VM's and intermediate languages, most of this work is Java based. For example, Lance presents work where the unmodified bytecode output by a Java compiler was analyzed to produce "end-product program analysis information" and utilized a prototype to prove the concept (Lance 1999). Another work by Maggi & Sisto demonstrates the feasibility of performing data flow analysis against Java bytecode to infer type information (Maggi 2001). On the other hand, Zhao demonstrates the viability of performing intra-procedural dependence analysis using Java bytecodes in their work (Zhao 2000). Other work such as that by Arnold (2005) examines the idea of using dynamically collected profiling information, collected via the virtual machine itself to generate high-accuracy call-graphs.

Figure 2. Analyzer architecture layout



Copyright © 2007, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Finally, Searle presents a tool called "DUCT" (for Define Use Chain Tool) which focuses on "relative debugging" and the following of "define-use" chains in program code. DUCT, unlike most other research focusing at the intermediate level (and very much like this research) leverages .Net CIL to allow it to operate on "a wide variety of languages without modification." (Searle 2003) Additionally, it should be noted that DUCT uses an incremental approach and mostly avoids traditional global program analysis (although like other incremental algorithms it also does require an initial full analysis to start with). DUCT's implementation uses essentially 3 analysis data structures, namely a Control Flow Graph, a Class Hierarchy Graph, and a Call Graph.

3. CALL GRAPH ANALYZER IMPLEMENTATION

A layered architecture, typical of many systems and particularly 3-tier business systems was used, as initially there was some uncertainty with respect to exactly how the .Net files would be interacted with (i.e. it was uncertain which back-end library or combination of libraries would be used to read the IL assemblies.) The analysis framework along with all algorithms and logic it contained was therefore to be well separated from the mechanics of actually retrieving the information, thus making it possible to easily change the data-access aspect without affecting the rest of the analyzer too much (hence the similarity to how a tiered business app might use an object persistence framework to insulate itself from database platform changes.) The particular architecture was inspired by an example object-persistence framework by Philip Brown (2000). The original architectural layout is shown in Figure 2 below. The idea was to contain the "analysis domain" logic in the middle layer, keeping separate all assembly data access concerns in data access classes in the right-had layer. These in turn delegate to one or more underlying reflection libraries to get their work done. Lastly, user interface and other application logic are kept in the application layer on the left.

Since most domain classes would have a common need/requirement in terms of having to be populated/loaded from the back-end API's by the data access classes, having this functionality common to a base domain class seemed sensible. Similarly, common data access behavior could be put in a base class for all data access classes.

The implementation used C# as primary implementation language, but also used all of the other languages available in Visual Studio 2005 (Beta 2) namely C++, VB.Net and J#, as well as Borland Delphi, as an alternative vendor's language. Small test libraries or assemblies were constructed in each of these languages, some with multiple links, as test cases for the analysis. For example, there's a C# test application that calls on a J# library, that in turn calls on a C++ library.

Other tools used include the Nunit unit testing framework for .Net as well as TestDriven.Net (a plug-in for Visual Studio that make Nunit testing available from within the VS IDE.) Note that Visual Studio 2005 (however, only the high end Team Suite edition) now has built-in Unit testing support (which is clearly closely modeled on Nunit's approach). It also includes other code analysis support functions, such as unit-test coverage analysis. For version control Subversion was used, together with the TortoiseSVN plug-in for Windows Explorer.

4. PRELIMINARY RESULTS

We now demonstrate and evaluates the .Net CIL based analyzer in actual use, firstly using an "Animal Taxonomy" example inspired by a somewhat similar example using the usual Shape/ Square/ Rectangle hierarchy in the Java language, by Rayside (2001). The main assembly listing is shown in Table 1 below. The components were compiled into .Net executables files and then run through the analyzer. Then the analyzer is evaluated and demonstrated to operate on a multi-assembly application where one assembly was constructed using a compiler from another vendor.

Firstly, the analyzer was run without Class Hierarchy Analysis, and as would be expected, this was processed extremely quickly, but also incorrectly includes an edge between AnimalInheritance.doATrick(Mamal) and Mamal.rollOverAndPlayDead(). Class Hierarchy Analysis (CHA) can be described as the process of the calculation of a program's inheritance hierarchy (Dean et al. 1995, pg. 1). Performing Class Hierarchy Analysis produces some form of Class Hierarchy Graph (whether explicit or implicit). This structure describes all the inheritance relationships between the classes in the program, as well as the methods that each class contains, particularly virtual and overridden methods and which ones are abstract (Bairagi 1997, pg. 2). In this case, the runtime was on the order of 47ms

Table 1. Animal taxonomy main listing (written in C#)

```
 1: namespace AnimalInheritance {
 2:   abstract class Mamal { public abstract void rollOverAndPlayDead (); }
 3:
 4:   class Cat     : Mamal { public override void rollOverAndPlayDead() { } }
 5:   class Hamster : Mamal { public override void rollOverAndPlayDead () { } }
 6:   class Dog     : Mamal { public override void rollOverAndPlayDead () { } }
 7:   class Terrier : Dog { }
 8:
 9:   class AnimalInheritance {
10:     static void doATrick ( Mamal m ) {
11:       m.rollOverAndPlayDead();
12:     }
13:
14:     static void Main ( string[] args ) {
15:       doATrick( new Terrier() );
16:     }
17:   }
18: }
```

in this case. This value appeared quite stable and repeated runs did not alter this value appreciably. The results of a particular run can be viewed in Figure 3.

Following these experiments, the analyzer was run with the aid of Class Hierarchy Analysis, but using a conventional full-program scan to build the class hierarchy. The results for this can be seen in Figure 4. As expected this was several orders

Figure 3. Execution of call graph analyzer without class hierarchy analysis

```
(Classes seen during traversal/in object map: No = 5)
(Methods seen during traversal/in object map: No = 6)

Call Graph:
Nodes: (No = 7)
    [AnimalInheritance.Main(String[]) : Void]
    [Terrier..ctor() : Void]
    [Dog..ctor() : Void]
    [Mamal..ctor() : Void]
    [Object..ctor() : Void]
    [AnimalInheritance.doATrick(Mamal) : Void]
    [Mamal.rollOverAndPlayDead() : Void]

Edges: (No = 6)
    [AnimalInheritance.Main(String[]) : Void] ------->[Terrier..ctor() : Void]
    [Terrier..ctor() : Void] ------->[Dog..ctor() : Void]
    [Dog..ctor() : Void] ------->[Mamal..ctor() : Void]
    [Mamal..ctor() : Void] ------->[Object..ctor() : Void]
    [AnimalInheritance.Main(String[]) : Void] ------->[AnimalInheritance.doATrick(Mamal) : Void]
    [AnimalInheritance.doATrick(Mamal) : Void] ------->[Mamal.rollOverAndPlayDead() : Void]

 Time taken: 47 ms.
```

Figure 4. Execution of call graph analyzer with class hierarchy analysis

```
(Classes seen during traversal/in object map: No = 1983)
(Methods seen during traversal/in object map: No = 20042)

Call Graph:
Nodes: (No = 9)
    [AnimalInheritance.Main(String[]) : Void]
    [Terrier..ctor() : Void]
    [Dog..ctor() : Void]
    [Mamal..ctor() : Void]
    [Object..ctor() : Void]
    [AnimalInheritance.doATrick(Mamal) : Void]
    [Cat.rollOverAndPlayDead() : Void]
    [Hamster.rollOverAndPlayDead() : Void]
    [Dog.rollOverAndPlayDead() : Void]

Edges: (No = 8)
    [AnimalInheritance.Main(String[]) : Void] ------->[Terrier..ctor() : Void]
    [Terrier..ctor() : Void] ------->[Dog..ctor() : Void]
    [Dog..ctor() : Void] ------->[Mamal..ctor() : Void]
    [Mamal..ctor() : Void] ------->[Object..ctor() : Void]
    [AnimalInheritance.Main(String[]) : Void] ------->[AnimalInheritance.doATrick(Mamal) : Void]
    [AnimalInheritance.doATrick(Mamal) : Void] ------->[Cat.rollOverAndPlayDead() : Void]
    [AnimalInheritance.doATrick(Mamal) : Void] ------->[Hamster.rollOverAndPlayDead() : Void]
    [AnimalInheritance.doATrick(Mamal) : Void] ------->[Dog.rollOverAndPlayDead() : Void]

 Time taken: 7953 ms.
```

Copyright © 2007, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 5. Execution of call graph analyzer for reachable types

```
(Classes seen during traversal/in object map: No = 5)
(Methods seen during traversal/in object map: No = 8)

Call Graph:
Nodes: (No = 7)
      [AnimalInheritance.Main(String[]) : Void]
      [Terrier..ctor() : Void]
      [Dog..ctor() : Void]
      [Mamal..ctor() : Void]
      [Object..ctor() : Void]
      [AnimalInheritance.doATrick(Mamal) : Void]
      [Dog.rollOverAndPlayDead() : Void]

Edges: (No = 6)
      [AnimalInheritance.Main(String[]) : Void] ------->[Terrier..ctor() : Void]
      [Terrier..ctor() : Void] ------->[Dog..ctor() : Void]
      [Dog..ctor() : Void] ------->[Mamal..ctor() : Void]
      [Mamal..ctor() : Void] ------->[Object..ctor() : Void]
      [AnimalInheritance.Main(String[]) : Void] ------->[AnimalInheritance.doATrick(Mamal) : Void]
      [AnimalInheritance.doATrick(Mamal) : Void] ------->[Dog.rollOverAndPlayDead() : Void]

 Time taken: 47 ms.
```

Figure 6. Call graph analyzer multi-module example

```
(Classes seen during traversal/in object map: No = 4)
(Methods seen during traversal/in object map: No = 6)

Call Graph:
Nodes: (No = 6)
      [CSAppRefDelphiLib.Main(String[]) : Void]
      [TFact..ctor() : Void]
      [Object..ctor() : Void]
      [TFact.Fact(Int32) : Int32]
      [Environment.Exit(Int32) : Void]
      [Environment.ExitNative(Int32) : Void]

Edges: (No = 6)
      [CSAppRefDelphiLib.Main(String[]) : Void] ------->[TFact..ctor() : Void]
      [TFact..ctor() : Void] ------->[Object..ctor() : Void]
      [CSAppRefDelphiLib.Main(String[]) : Void] ------->[TFact.Fact(Int32) : Int32]
      [TFact.Fact(Int32) : Int32] ------->[TFact.Fact(Int32) : Int32]
      [CSAppRefDelphiLib.Main(String[]) : Void] ------->[Environment.Exit(Int32) : Void]
      [Environment.Exit(Int32) : Void] ------->[Environment.ExitNative(Int32) : Void]

 Time taken: 187 ms.
```

of magnitude slower, both in space and time required, comparing the number of classes and methods processed and the time taken to the previous run. Time taken was approximately 7593ms (with an estimated variance of about 200ms based on observing repeated runs). This is at least 2 orders of magnitude larger than before. The discrepancy between the storage requirements here and previously is even more staggering: approximately 2,000 classes processed versus just 5 before and about 20,000 methods seen versus just 6 before.

Here we can see the analyzer now being very conservative, by now including all three of Cat, Hamster and Dog classes as being receiver classes of the rollOverAndPlayDead method call (together with also including for safety, all visible types in all referenced assemblies. Finally, we show an example where the analyzer was run to strictly include only the reachable types when building the *Class Hierarchy Graph* (CHG). (This is in terms of processing result much like that of the *RTA* algorithm by Bacon and Sweeney (1996), however the actual algorithm is rather different.): The output was as follows:

The performance is markedly improved, evidently back to where it was in the beginning in terms of time, with the class count also being the same as the first experiment and the method count being marginally higher. More importantly, the result is also a lot more accurate, thanks to the extra intelligence employed. Clearly the cost of blindly traversing all of the "visible" program code, without analyzing whether it is in fact reachable from the main entry point is enormous.

One of the key goals of this research was to prove that it was possible to do multi-language/multi-module full-program analysis. We therefore also tested the analyzer with several multi-language scenarios, including this one where a C# application calls on a Borland Delphi library. The source for the both functions are given in Table 2.

Table 2. Call graph analyzer multi-module example: Program code

```
using System;
using System.Collections.Generic;
using System.Text;
using DelphiLib;

namespace CSAppRefDelphiLib {
  class CSAppRefDelphiLib {
    static void Main ( string[] args ) {
      TFact fact = new TFact();
      System.Environment.Exit( fact.Fact( 5 ) );
    }
  }
}
```

C# Main Program Code

```
library DelphiLib;
 type
   TFact = class
   public
     function Fact (n : integer) : Integer;
   end;

 function TFact.Fact( n : Integer) : Integer;
 begin
   if n = 0 then
     result := 1
   else
     result := n * Fact(n -1 );
 end;

 begin
 end.
```

Delphi Library Code

These two programs were compiled by completely different compilers written by completely different vendors. The main program was fed into the analyzer as input. The output is shown below in Figure 6.

As can be seen, the analyzer had no problem with the fact that the library was originally written in another source language, as should be expected. It can be inferred that a different language was used through visual inspection from the type names, but that is all. There are no other obvious differences. It would be interesting to re-run some of these tests against/on the Mono platform.

5. BRIEF DISCUSSION

Most conventional program analysis approaches employ source based approaches, either simple text I/O or actual scanning and parsing techniques. In some cases byte-code analysis is also used, although this (with a few exceptions) focuses on Java bytecode rather than .Net CIL. While a detailed quantitative comparison of techniques is outside the scope of this research, it is nevertheless useful to try and establish qualitatively the relative strengths of the various approaches, so as to establish whether the .Net CIL based approach is comparable, better or poorer than conventional techniques. There does not appear to exist any comparative study in the literature that compares both source and bytecode based approaches as part of the one study. Murphy (1998) presents an empirical study of static call-graph constructors and states that "four choices of input format are available for the developer of an extractor for a system implemented in C: unprocessed source, preprocessed source, object code with symbol table information, and executable code with symbol table information." Unfortunately they also state that their focus was on source based processing in their study and that object-code based analysis was thus out of scope.

The first observation to make is that .Net CIL files tend to be very compact. The IL assembler language is actually quite simple, and while op-code's can be multiple bytes, most of them (90%) are in fact single bytes. To be precise, there is at present only 250 op-codes, 225 of which occupy a single byte, the remainder occupying 2 bytes (Lidin 2002, pp 422 - 428). Roughly speaking, based on estimates observing the ratios of source code to binary size of the Analyzer code, test case libraries and example libraries produced as part of this research, it would appear that the ratio of IL binary size vs. Original Source code size would be about 2:3. More investigation would be needed to establish whether this observation is borne out in larger systems. But in any case, in terms of I/O overheads, it can therefore be argued that IL will probably be as easy or easier to deal with than text based source code.

Of course, on today's machines, I/O throughput is unlikely to be a significant bottleneck during program analysis. However, as can be observed by trying to analyze even trivial programs with the analyzer while employing a naive approach to what is read as "potentially callable," it is quite possible to have the analyzer consume several hundred megabytes of memory and take several minutes of processing time. So the real problem lies with managing the intermediate in memory representation, and with the algorithms employed during analysis. Here the IL

Copyright © 2007, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

based approach will suffer the same challenges faced by text based approaches – there is essentially no advantage to either approach from this point of view.

Comparing Java bytecode based analysis to .Net bytecode based analysis, it appears that some of the Java bytecode approaches suffer somewhat partly because of weaker reflection support (whether third party libraries or in the platform.) Lance (1999, pg. 5) presents their "JAristotle" bytecode based Java program analyzer. He remarks that the development of the bytecode based prototype required the writing of (only) 13,700 lines of Java code, and contrast this with the prior "Aristotle" based front-end (which was source based) and required modification to some 30,000 lines of an existing C parser to implement. While probably not directly comparable (since that analyzer computes intra-procedural flow-graphs, not call-graphs) it is nevertheless instructive to note that the .Net CGAnalyzer source code consist of approximately only 1500 lines of C# code, and this includes more than one approach to the analysis as well. This appears to be due in part to to the stronger reflection and introspection support in the .Net platform which enabled us to avoid writing code to directly deal with IL bytecode. This advantage is in addition to and apart from the implied advantages of being able to analyze whatever language target the .Net platform.

6. CONCLUSION AND FUTURE WORK

This research project investigated the possibility of leveraging .Net's CIL bytecode together with reflection support as a vehicle for static program analysis, in particular call-graph analysis, and successfully implemented a prototype to prove the concept. Like Lance (1999) this approach has the benefit of sidestepping the usual lexical and syntax analysis that is associated with conventional source based analyzers, with a consequent lowering of the amount of effort required to get a working analyzer going. Unlike Lance (1999) our approach focuses on .Net, a platform that is deliberately multi-language, and one that is likely to be increasingly used as a platform for legacy migration. In this way, this work therefore will ultimately contribute towards easing the maintenance burden for legacy systems.

The focus on .Net has also made the entry to program analysis easier in other ways. There are a number of API's and libraries available to choose from that can shield one from having to even deal with the bytecode oneself. All of this is reflected in the number of lines of code that was required to effectively implement a protoype analyzer with several analysis features, including a form of Class Hierarchy Analysis and an RTA-like call-graph generation algorithm (that uses a "reachable types only" approach to limit the amount of analysis work done.)

There are clearly many potentially interesting areas for future work: This research project was originally started with a view to call-chain analysis. Having now developed a basic call-graph one could now go directly forward and add some form of call-chain analysis. Call-chain analysis is sometimes used in the context of integration test coverage analysis (see for example (Rountev 2004b)), which requires dynamic analysis support as well to measure the actual chains occurring at runtime. As such, another avenue of work may be to investigate dynamic analysis support on the .Net platform with a view to fully supporting call-chain analysis as part of integration test coverage.

ACKNOWLEDGMENT

This research project was undertaken as part of a Master of Science degree with University of Liverpool and Laureate Online Education.

REFERENCES

Aiken (2005) Call Graphs in Higher-Order Languages (Lecture 14) <http://www.cs.berkeley.edu/~aiken/cs264_02/lectures/lecture14.ppt>. (7/9/2005)

Bacon, D.F. Sweeney, P.F. (1996) 'Fast static analysis of C++ virtual function calls.' ACM SIGPLAN Not. 31, 10 (October)., pp. 324–341

Bairagi, D. & Agrawal, Dharma P. & Kumar, Sandeep (1997) 'Precise Call Graph Construction for OO Programs in the Presence of Virtual Functions,' 1997 International Conference on Parallel Processing (ICPP '97) p. 412, IEEE , <http://doi.ieeecomputersociety.org/10.1109/ICPP.1997.622674> (15/09/2005)

Brown, P. (2000) An Object-Oriented Persistence Layer Design ,: <http://cc.borland.com/Item.aspx?id=15511> (24/1/2006)

Dean, J & Grove, D & Chambers C (1995) 'Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis', ECOOP, <http://www.cs.ucla.edu/~palsberg/tba/papers/dean-grove-chambers-ecoop95.pdf>. (05/09/2005)

Dean, J (1997) Call Graph Analysis in the Presence of Higher-Order Functions , <http://citeseer.ifi.unizh.ch/20731.html>. (6/9/2005)

Fowler, M., (2000) Refactoring: Improving the design of existing code, Upper Saddle River, New Jersey: Addison Wesley

Grove, D., Chambers, C., (2001) 'A framework for call graph construction algorithms' ACM Transactions on Programming Languages and Systems (TOPLAS) v.23 n.6, pp. 685-746, November

Lakhotia A., (1993) 'Constructing call multigraphs using dependence graphs.' Proceedings of the Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, New York, NY, pp. 273–284.

Lance D., Untch R. H., Wahl (1999) 'Bytecode-based Java program analysis.' Proceedings of the 37th Annual Southeast Regional Conference (Cd-Rom) ACM-SE 37. ACM Press, New York, NY, 14.

Lidin Serge (2002) Inside Microsoft .Net CIL Assembler Redmond, Washington, Microsoft Press, pp 421- 428

Losch F, (2005) Instrumentation of Java Program Code for ControlFlow Analysis , <http://elib.uni-stuttgart.de/opus/volltexte/2005/2256/pdf/DIP_2258.pdf>.(5/1/2006)

Maggi P., Sisto R., (2001) 'Using Data Flow Analysis to Infer Type Information in Java Bytecode' FirstIEEE International Workshop on Source Code Analysis and Manipulation, p. 213

Murphy G., (1998) 'An Empirical Study of Static Call Graph Extractors' ACM Transactions on Software Engineering and Methodology, Vol 7, No. 2, April, pp 158-191

Rayside D. (2001) A Generic Worklist Algorithm For Graph Reachability Problems in Program Analysis Waterloo MSc Thesis, Ontario, Canada: University of Waterloo

Rountev A, Kagan S, Gibas M (2004b) 'Static And Dynamic Analysis of Call Chains in Java' Software Engineering Notes Vol. 29(4), p. 1

Searle A., Gough J., Abramson D., (2003a) 'DUCT: An interactive Define-Use Chain Navigation tool for relative debugging' Fifth Int. Workshop on Automated and Algorithmic Debugging [Internet]

Shivers O., (1988) 'Control flow analysis in Scheme' Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, vol. 23(7) New York: ACM Press, July, pp 164-174,

Sommerville I., (2001) Software Engineering (6th edition), Upper Saddle River, New Jersey: Addison Wesley, p. 623

Zhao J., (2000) 'Dependence Analysis of Java Bytecode.' 24th international Computer Software and Applications Conference pp. 486-491, October

Copyright © 2007, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

0 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage: www.igi-global.com/proceeding-paper/call-graph-based-program-analysis/33187

Related Content

Collaborative Environments Based on Digital Learning Ecosystem Approach to Reduce the Digital Divide
José Eder Guzmán Mendoza, Jaime Muñoz Arteagaand Julien Broisin (2019). *Educational and Social Dimensions of Digital Transformation in Organizations (pp. 27-42).*
www.irma-international.org/chapter/collaborative-environments-based-on-digital-learning-ecosystem-approach-to-reduce-the-digital-divide/215134

Prediction of Ultimate Bearing Capacity of Oil and Gas Wellbore Based on Multi-Modal Data Analysis in the Context of Machine Learning
Qiang Li (2023). *International Journal of Information Technologies and Systems Approach (pp. 1-13).*
www.irma-international.org/article/prediction-of-ultimate-bearing-capacity-of-oil-and-gas-wellbore-based-on-multi-modal-data-analysis-in-the-context-of-machine-learning/323195

Philosophical Framing and Its Impact on Research
Eileen M. Trauthand Lee B. Erickson (2012). *Research Methodologies, Innovations and Philosophies in Software Systems Engineering and Information Systems (pp. 1-17).*
www.irma-international.org/chapter/philosophical-framing-its-impact-research/63255

Application of Methodology Evaluation System on Current IS Development Methodologies
Alena Buchalcevova (2018). *International Journal of Information Technologies and Systems Approach (pp. 71-87).*
www.irma-international.org/article/application-of-methodology-evaluation-system-on-current-is-development-methodologies/204604

The Effects of Sampling Methods on Machine Learning Models for Predicting Long-term Length of Stay: A Case Study of Rhode Island Hospitals
Son Nguyen, Alicia T. Lamere, Alan Olinskyand John Quinn (2019). *International Journal of Rough Sets and Data Analysis (pp. 32-48).*
www.irma-international.org/article/the-effects-of-sampling-methods-on-machine-learning-models-for-predicting-long-term-length-of-stay/251900