10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing

Towards a Reverse Engineering Dataflow Analysis Framework for Java and C++

Petru Florin Mihancea
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
Email: petru.mihancea@cs.upt.ro

***Abstract*—Due to the software aging phenomenon, understanding a program is increasingly difficult. Many high-level analysis methods have been developed to support program comprehension, some of them based on dataflow analysis. Conventional dataflow analysis infrastructures could be used as a basis to implement these reverse engineering methods. However, none of them meet specific reverse engineering requirements. In this tool demo we introduce MEMBRAIN, a dataflow analysis infrastructure for reverse engineering. A brief insight over this infrastructure is presented along with some essential implementation details. The demo also includes a practical usage example and a performance comparison in order to prove the tool usability.**

***Keywords*-dataflow analysis; reverse engineering; static analysis**

I. MOTIVATION

Dataflow analyses collect runtime information about data in programs without actually executing them [1]. Usually, such static analyses are used in the context of optimizing compilers and program verifiers. However, they also have an important and increasing role for reverse engineering legacy systems.

The state of the art literature presents many high-level reverse engineering methods that rely on dataflow analyses. In [2], the authors present a method to understand class hierarchies implementation. Manipulating points-to information, their approach produces behaviorally equivalent hierarchies in which each object contains only the members that it needs. In [3], *Static Class Analysis* [4] is used to characterize the type hierarchy nature of a class hierarchy. However, in order to be applicable, such reverse engineering techniques need a dataflow analysis infrastructure as basis for their implementation. From the perspective of a reverse engineering practitioner / researcher such an infrastructure should have some particular traits:

1) To avoid duplication, when possible, the implementation of a reverse engineering analysis should be independent of the language in which the analyzed program is written. Thus, a dataflow analysis engine for reverse engineering should function on a common (language-neutral) representation of programs.
2) A reverse engineer wants an easy way to precisely implement concrete dataflow analyses. As a consequence, the common representation of programs must be a sufficiently low-level one: derived language constructs (*e.g.*, ++ operator) must be eliminated, the representation must be explicit (*e.g.*, implicit calls to copy-constructors in C++ must be made explicit), etc. This is because such language particularities complicate the implementation of dataflow analyses (*e.g.*, the ++ operator is a definition of the associated variable that must be considered by a precise implementation of *Reaching Definitions* [1]).
3) Usually, a maintainer wants to understand a program at the implementation level. Thus, a dataflow analysis infrastructure for reverse engineering should be able to present the data facts at the level of the analyzed source code.
4) Often, reverse engineering is performed on incomplete code. As a consequence, a reverse engineering dataflow analysis infrastructure should be employable even when the code of the reverse engineered program is not entirely available.

In the following, we are going to briefly present our state-of-the-art investigation in order to see if these requirements are or can be met by current dataflow analysis infrastructures.

II. RELATED WORK

A dataflow analysis framework for Java programs is presented in [5]. However, because it directly manipulates JVM code, it can be used to analyze only complete Java programs. *BML*[1] has similar capabilities, but it also analyzes Java bytecode.

In [6], common representations of programs written in various programming languages (at different levels of abstraction) are proposed. These representations can be used as the basis for building a generic dataflow analysis engine. However, they are too close to the original code (*e.g.*, derived operators such as ++ are not eliminated, etc.) making dataflow analyses less precise and harder to implement. Similar representations (but only for C++) are presented in [7].

JKit [2] is an implementation of a Java compiler and it uses an intermediate representation for Java programs (JKil). However, since it is a compiler, it has not been constructed as a dataflow analysis framework for reverse engineering (*e.g.*, it cannot be used to analyze incomplete code).

[1]homepages.mcs.vuw.ac.nz/∼djp/bml/
[2]homepages.mcs.vuw.ac.nz/∼djp/jkit/

978-0-7695-3523-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SYNASC.2008.7

IEEE computer society

```
// i and j are local variables
if(2 == j) { i = j++; }
// MemBrain representation
1 :Copy          [j]
2 :Equal         2    (-1)
3 :CGoto         (-1) L1
4 :Goto          L2
5 :Label         L1
6 :Copy          [j]
7 :Add           (-1) 1
8 :AssignToName  [j]  (-1)
9 :AssignToName  [i]  (-3)
10:Label         L2
// Explanation
[i] - reference to local variable i
(-x)- temporary reference eg., (-3) in the
9th instruction identifies the result of
the 9 - 3 = 6th instruction
```

Fig. 1. Representation Example

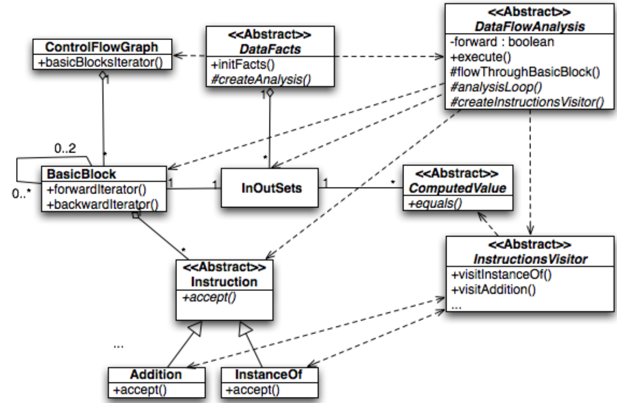

Fig. 2. The Dataflow Analysis Framework

III. MEMBRAIN INFRASTRUCTURE

In essence, MEMBRAIN (Memoria Extension for Method Body Representation, Analysis and INspection) means two things. First, it defines a common representation for programs written in Java and C++[3]. On the other hand, it is a framework that allows to easily implement intra-procedural flow-sensitive dataflow analyses on MEMBRAIN representation. In this section we present this representation and the main abstractions of the framework. Our prototypical infrastructure is implemented in Java.

A. The Representation

1) Instruction Set: The MEMBRAIN instructions can be viewed as an elementary low-level representation of Java and C++ operators and statements (*e.g.*, Addition, StaticCall, VirtualCall, etc.). This representation does not include derived operators (*e.g.*, ++, +=) or structured statements such as *for*, *do*, etc. All these statements are expressed in terms of *goto* and *conditional goto* instructions. Moreover, every instruction has only one meaning (*e.g.*, + means addition between numbers and cannot be overloaded as in C++). These simplifications allow an easier dataflow analysis implementation.

MEMBRAIN instructions are a form of three-address code [1]. To identify the operands of an instruction, different types of references are used. Usually, a reference models an entry from the symbol table (*e.g.*, a variable, a type). Additionally, because we have adopted an implementation of the three-address code using *triples* [1], we also use a special kind of reference (*i.e.*, temporary reference) in order to model the result of an instruction. As an example, we present in Fig. 1 a code fragment and its MEMBRAIN representation.

2) Translators: Our infrastructure also includes two translators that convert Java 1.5 and ISO C++ code into MEMBRAIN representation (the C++ translator is under development). We emphasize that it is the translator's responsability to perform all the transformations mentioned above (*e.g.*, in C++, when a + operator refers to an operator method, it must be translated into an explicit call).

The translators are built on top of the *Recoder*[4] respectively *Columbus* [8] parsers. One reason for using these tools is that they can parse incomplete code. Additionally, our translators are also written in such a way that they can recover from errors (*e.g.*, references to undefined types) produced by incomplete code. The final effect is that our infrastructure can be used even when the code of the analyzed program is incomplete.

The translator's primary output consists of a code table with MEMBRAIN instructions for the translated method. Next, the code table is used to build the control-flow graph of that method. The translators also provide a mapping between the generated instructions and the source code. In essence, each generated instruction is associated with a code stripe (*i.e.*, continuous or discontinuous sequences of characters identified by their start / stop line and column) representing the source code translated into that instruction. Thus, the mapping can be used to present data facts at the level of the original source code, an essential prerequisite for reverse engineering.

B. The Framework

In Fig. 2 we present the MEMBRAIN dataflow analysis framework. The result of a dataflow analysis consists in sets of *ComputedValues* associated with any particular point in a method (*e.g.*, at this execution point, local variable *x* may have the value assigned by the *instr* instruction). To implement a dataflow analysis, an engineer must define first what a *ComputedValue* means for that analysis (in a subclass) and to define the equality between two such instances (*i.e.*, to implement the *equals()* method).

A *DataFacts* object represents the result of a dataflow analysis applied on a particular method. Such an instance is an aggregation of *InOutSets* (*i.e.*, sets of *ComputedValues* associated with the input / output of each basic block). To define the result of a particular analysis, we must implement the *createAnalysis()* factory method [9]. This method will

[3]The representation will be extended in order to address full C++

[4]recoder.sourceforge.net

only have to create an object representing the implemented analysis. We also emphasize that the *DataFacts* class provides methods that can be used to find the input / output sets of *ComputedValues* at the basic block and instruction level. Thus, this class represents the way through which a user can (programatically) extract the results of a dataflow analysis for further exploitation.

A *DataFlowAnalysis* object models a particular dataflow analysis. The framework uses a classical worklist algorithm [1] to approximate the sets of *ComputedValues* at the start / end of each basic block. In order to implement a dataflow analysis, an engineer must provide an implementation for the transfer functions of the analysis (via the *createInstructionVisitor()* factory method [9]) and to define the analysis loop (*i.e.*, the manner in which the sets of *ComputedValues* are combined) in the *analysisLoop()* template method [9].

The purpose of an *InstructionsVisitor* object is to implement the transfer functions of a dataflow analysis for each relevant MEMBRAIN instruction. Usually, it is going to manipulate (*i.e.*, to compare, instantiate, etc.) analysis specific *ComputedValue* objects. At implementation level, such an instance is a *Visitor* [9] for the hierarchy of MEMBRAIN instructions.

IV. PRACTICAL USAGE

MEMBRAIN has been integrated into the IPLASMA reengineering environment [10]. In this section we present a reverse engineering practical application implemented using our tool and we compare MEMBRAIN execution times with those of *BML*[5].

1) Type Highlighting: In order to capture the extent to which the concrete types of objects defined in a class hierarchy are polymorphically / non-polymorphically treated by the clients of that hierarchy, we have developed the *Type Highlighting* visualizations [11].

The principle of one of these software views (*i.e.*, *Group Discrimination*) is presented in Fig. 3. The body of a client method is transformed into an image by mapping each source code character into a pixel. Next, based on the *Static Class Analysis (SCA)* [4] computed by MEMBRAIN and on the tool capabilities to map data facts at the level of source code, we use distinct colors to emphasize code areas where particular types of objects are non-uniformly treated (*e.g.*, the true branch of the *if* statement is green because only *B* instances can be referred by *x*; the false branch is blue because *x* may refer only to *C* instances in that code area; the last call is red because *x* may refer to both concrete types of objects). Using this visual codification simultaneously for all the clients of a class hierarchy and preserving the color code (*i.e.*, green is used only for code areas dedicated to *B* instances, etc.), we can easily detect, for example, client type checking design problems and their prevalence into the analyzed software.

The *Type Highlighting* views have been applied to several Java[6] case studies. During our experiments, we have also manually analyzed the code of many client methods of different class hierarchies in order to validate both the *Type Highlighting* technique and the results of *SCA* obtained by MEMBRAIN.

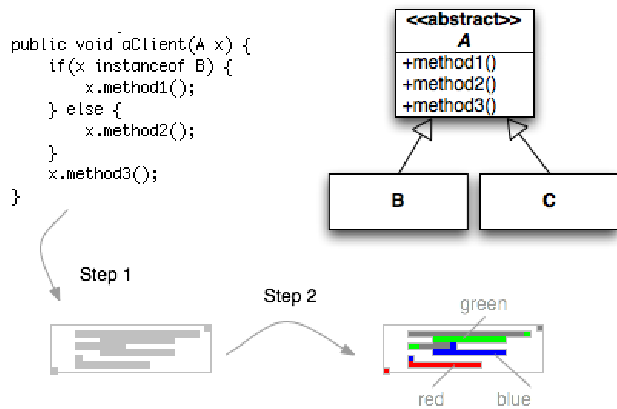

Fig. 3. Group Discrimination Sketch

TABLE I
MEMBRAIN EXECUTION TIMES

System	Lines of code	Concrete methods	CFG building (sec)	SCA (sec)	RD (sec)
Recoder	42 259	5 598	3.0	31.5	4.3
Jung	22 447	2 754	1.5	28.7	2.3

TABLE II
PERFORMANCE COMPARISON

System	Time/method MemBrain (ms)	Time/method BML (ms)
Recoder	0.768	0.181
Jung	0.835	0.569
Average	**0.802**	**0.375**

2) Performance: To evaluate the efficiency of the tasks accomplished by MEMBRAIN during the execution of *Type Highlighting* analyses, we present in Table I the execution time for (i) control-flow graph construction (includes translation time but not parsing time) (ii) computation of *Static Class Analysis (SCA)* [4] and (iii) computation of *Reaching Definitions (RD)* [1][7]. The measurements have been performed for all methods of two medium-sized Java programs, using a MacBookPro computer (Core 2 Duo 2.33 GHz, 2 GB of RAM) running MacOS 10.5.

In Table II, we compare our *RD* analysis with that of *BML* which however considers only local variables and not data fields. Our execution time per method is on average 2.14 times higher. We consider this difference of performance acceptable since it will be offset by the advantage of also analyzing C++ programs.

V. CONCLUSIONS AND FUTURE WORK

We have presented in this tool demo the MEMBRAIN dataflow analysis prototypical infrastructure dedicated to re-

[5]homepages.mcs.vuw.ac.nz/~djp/bml/

[6]Because the C++ translator is still under development, at the moment we cannot analyze realistic C++ systems

[7]The current implementations do not include an alias analysis

verse engineering. We have briefly described its internal structure and we have presented one of its successful practical applications. As a conclusion, we state that MEMBRAIN is a promising support for reverse engineering practice and research.

Our future efforts will be focused on the completion of the C++ to MEMBRAIN translator. To accomplish this task, our common representation needs to be extended (*e.g.*, in order to support C++ templates, pointers to functions, etc.). We also plan to extend our infrastructure to also support interprocedural dataflow analysis (context-sensitive and insensitive). Last but not least, we also plan to improve the execution times of our tool.

ACKNOWLEDGMENT

This work has been partially supported by the Romanian Ministry of Education and Research under the research grants CNCSIS TD (2007 & 2008 Code 126) and PN2 (357/1.10.2007).

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools (2nd Edition)*. Addison Wesley, 2007.

[2] G. Snelting and F. Tip, "Understanding Class Hierarchies Using Concept Analysis," *ACM Trans. on Programming Languages and Systems*, vol. 22, pp. 540–582, May 2000.

[3] P. F. Mihancea, "Towards a Client Driven Characterization of Class Hierarchies," in *Proceedings of IEEE International Conference on Program Comprehension (ICPC 06)*. IEEE Computer Society, 2006, pp. 285 – 294.

[4] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *Proceedings of European Conference on Object-Oriented Programming (ECOOP 95)*, ser. LNCS, vol. 952. Springer-Verlag, 1995, pp. 77 – 101.

[5] M. Mohnen, "An Open Framework for Data-Flow Analysis in Java : Extended Abstract," in *Proceeding of Inaugural Conference on the Principles and Practice of Programming (PPPJ 02), and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines (IRE 02)*. National University of Ireland, 2002, pp. 157 – 161.

[6] R. Al-Ekram and K. Kontogiannis, "An XML-Based Framework for Language Neutral Program Representation and Generic Analysis," in *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR 05)*. IEEE Computer Society, 2005, pp. 42 – 51.

[7] N. Kraft, B. Malloy, and J. Power, "Towards an Infrastructure to Support Interoperability in Reverse Engineering," in *Proceedings of Working Conference on Reverse Engineering (WCRE 05)*. IEEE Computer Society, 2005, pp. 196 –205.

[8] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus - Reverse Engineering Tool and Schema for C++," in *Proceedings of IEEE International Conference on Software Maintenance (ICSM 02)*. IEEE Computer Society, 2002, pp. 172– 181.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[10] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design," in *Proceedings of IEEE International Conference on Software Maintenance (ICSM 05), Industrial and Tool Volume*, 2005.

[11] P. F. Mihancea, "Type Highlighting: A Client-Driven Visual Approach for Class Hierarchies Reengineering," in *Proceedings of IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 08)*. IEEE Computer Society, 2008, pp. 207–216.