Volume 5, Issue 7, July 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Improving Regression Test Coverage Using Parse Tree

G. M. Malik Basha, M. Tech, **Dr. V. PurnaChandra Rao,** PhD
Dept. of Computer Science and Engineering, Lords Institute of Engineering & Technology,
Himayath Sagar, Hyderabad, Telangana, India

Abstract:-In this project investigation is done on efficient dataflow and slicing techniques to further reduce the instrumentation for getting even greater savings in the time for the regression testing. Dynamic Slicing algorithms can greatly reduce the debugging effort by focusing the attention of the use, on a relevant subset of program statements. In this project the design and evaluation of three precise dynamic slicing algorithms called the full preprocessing (FP), no preprocessing (NP) and limited preprocessing (LP) algorithms. The algorithms differ in the relative timing of constructing the dynamic data dependence graph and its traversal for computing requested dynamic slices. These experiments show that the LP algorithm is a fast and practical precise slicing algorithm. In fact it is shown that while precise slices can be orders of magnitude smaller than imprecise dynamic slices, for small number of slicing requests, the LP algorithm is faster than an imprecise dynamic slicing algorithm proposed by Agarwal and Horgan [1,2].

Index Terms: - Static and Dynamic Slicing, Regression Testing, Imprecise Algorithm, Full Preprocessing Algorithm, No Preprocessing Algorithm, Limited Preprocessing Algorithm.

I. INTRODUCTION

The technique used for debugging and understanding [3] a program is named as program slicing. It works as follows. For any given program P, a slicing criterion is provided by the programmer of the form (l, V), where ‘l’ is a control location in the program and ‘V’ is a set of program variables referred at l. The purpose of slicing is to find out the statements in P which can affect the values of V at l via control and/or data flow. So, if during program execution, the values of V at l were unexpected, the corresponding slice can be inspected to explain the reason for the unexpected values.

The two types of slicing techniques are:

Static slicing and

Dynamic Slicing.

Static Slicing & Dynamic Slicing

For the slicing criterion, static slicing techniques typically operate on a program dependence graph (PDG). The nodes of the PDG are simple statements/ conditions and the edges correspond to data/control dependencies [Horwitz et al. 1990]. For a given input of a program P dynamic slices are often smaller than the staticslice of the ProgramP. Dynamic slicing with respect to an input I on the otherhand, often proceeds bycollecting the execution trace corresponding to I. The dynamic data andcontroldependencies between thestatement occurrences inexecution trace can be precomputed or computed on objects and pointers in programs, static slices may bevery large. On the other side, dynamic slices capture the closure of dynamic data and control dependencies. Hence they are much more precise, and more helpful for narrowing the attention of the programmer.The slicing criterion for particular input may affect the dynamic slices of the program fragments,they naturally support the taskof debugging via running of selected test inputs.

Dynamic slicing techniquewas proposed for debugging [Korel and Laski 1988Agrawal and Horgan 1990], subsequently it has been used for program comprehensions in many other innovative ways.[Zhang et al. 2005].Static data dependence computations are often conservative due to the presence of gram comprehension in many other innovative ways. In particular, dynamic slices or their variants which also involve computing the closure of dependencies by trace traversal have been used for studying causes of program performance degradation [Zilles and Sohi 2000], identifying isomorphic instructions in terms of their runtime behaviors [Sazeides 2003] and analyzing spurious counter-example traces producedby software model checking [Majumdar and Jhala [2005]. Even in the context of debugging, dynamic slices have been used in unconventional ways e.g. [Akgul et al. 2004] studies reverse execution along a dynamic slice. Thus, dynamic slicing formsthe core of many tasks in program development and it is useful to develop efficientmethods for computing dynamic slices.

In this paper, an infrastructure for dynamic slicing of Java programs has been provided. Our method operates on byte code traces, work is done at the bytecode level since slice computation may involve looking inside library methods and the source code of libraries may not always be available. First, the bytecode stream corresponding to an execution trace of a Java program for a given input is collected. The tracecollection is done by modifying a virtual machine. The Kaffe Virtual Machine has been used in our experiments. The backward traversal of the bytecodetrace is then performed to compute dynamic data and control dependencies on thefly. The slice is updated as these dependencies are encountered during trace traversal. Computing the dynamic data dependencies on bytecode traces is complicated due to Java's stack based architecture. The main problem is that partial results of a computation are often stored in the Java Virtual Machine's operand stack. This results in implicit data dependencies between bytecodes involving data transfer via the operand stack. For this reason, our backwards dynamic slicing performs a "reverse" stack simulation while traversing the bytecode trace from the end.

These methods of dynamic slicing typically involve traversal of the execution trace. This traversal may be used to pre compute a dynamic dependence graph or the dynamic dependencies can be computed on demand during trace traversal. Thus, the representation of execution traces is important for dynamic slicing. In the case for backwards dynamic slicing where the trace is traversed from the end . The traces tend to be huge. The work of [Zhang et al. 2005] reports experiences in dynamic slicing programs like gcc and perl where the execution trace runs into several hundred million instructions. It might be inefficient to perform postmortem analysis over such huge traces. Consequently, it is useful to develop a compact representation for execution traces whichcapture both control flow and memory reference information. During program executioncompact trace should be generated on-the-fly. Our method proceeds by on the fly construction of a compact byte code trace during program execution. The compactness of our trace representation is owing to several factors. First, bytecodes which do not correspond to memory access (i.e.data transfer to and from the heap) or control transfer are not stored in the trace. Operands used by these bytecodes are fixed and can be discovered from Java class files. Secondly, the sequence of addresses used by each memory reference bytecodeor control transfer bytecode is stored separately. Since these sequences typically have high repetition of patterns, such repetition is exploited to save space. A well-known lossless data compression algorithm called SEQUITUR [8] is modified for this purpose. This algorithm identifies repeated patterns in the sequence on-the-fly and stores them hierarchically.

II. REGRESSION TESTING

IEEE (Institute of Electrical and Electronics Engineers) defines regression testing as follows: "Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements." [IEEE]During software development projects, there are several design changes that typically result from added new features and error correction work. Customers want new features in the latest releases, but still expect the older features to remain in place. This is where regression testing plays a role. In order to prevent quality from degrading, the new versions of software are retested using a combination of existing test cases. To accomplish this time consuming task effectively, test selection techniques and test automation are recommended.

III. PRECISE DYNAMIC SLICING ALGORITHM

In precise dynamic slicing algorithm the program is executed once and produce the execution trace that is processed to construct dynamic data dependence graph which in turn is traversed to compute the dynamic slices. The execution trace captures the complete runtime information of the programs execution used by the dynamic slicing algorithm. There is sufficient information to compute the precise dynamic slices. The information that the trace holds the full control flow trace and memory reference trace. Thus the complete path followed during the execution can be known and each point where data is referenced through pointers the address from data is accessed can be found.

3.1 Full Preprocessing Algorithm

The main aim of developing full preprocessing slicing algorithm is to achieve dynamic data dependence graph representation that would not only allow for computation of precise dynamic slices. It supports computation of dynamic slices for variables and memory address at any point of execution. This property is not supported by the precise dynamic slicing algorithm of Agrawal and Horgan [1]. Here the statement level control flowgraph representation of the program are considered and add it to the edges corresponding to the data dependences extracted from the execution trace. The execution instances of the statements involved in the dynamic data dependence are explicitly indicated on the dynamic data dependence edges thus allowing the above goal to be met. it is equally easy to compute a dynamic slice atany earlier execution point.

3.2. No Preprocessing Algorithm

The Full preprocessing algorithm first carries out all preprocessing and then starts slicing. For large program with long execution run it is possible that the dynamic dependence graph requires too much space to store and too much space to build. The experiment shows that too often run out of memory since the graphs are too large. For this reasons precise dynamic slicing algorithm is proposed that do not perform any preprocessing. To avoid prior preprocessing demand driven analysis of the trace can be used to recover dynamic dependences. when a slice computation begins the trace is traversed backwards to recover the dynamic dependences required for the slice computation. If the dynamic slices for the variable 'V' at the end of the program is needed, backward traversing is done until the definition of the variable v is found and include the defining statement in the dynamic slice. If 'V' is defined in terms of other variable W, The traversal of the trace is resumed starting from the point where the traversal had stopped upon finding the definition of 'V' and so on.

3.3 Limited Preprocessing

While the NP algorithm described above addresses the space problem of the FP algorithm, this comes at the cost of increased time for slice computations. The time required to traverse a long execution trace is a significant part of the cost of slicing. While the FP algorithm traverses the trace only once for all slicing requests, the NP algorithm often traverses the same part of the trace multiple times, each time recovering different relevant dependences for different slicing request. With the above discussion No Preprocessing algorithm does too little preprocessing lead to high slicing cost while Full Preprocessing perform too much preprocessing leading to the space problems. next an algorithm is proposed that strikes the balance between the preprocessing and slicing cost. In this precise algorithm limited preprocessing of the execution trace is carried out first augmenting the trace with summary information that allows faster traversal of the augmented trace. Demand driven analysis to compute the slice using the augmented trace is used. This algorithm is called as limited preprocessing algorithm (LP).

In LP algorithm the trace is divided into fixed size trace blocks .At the end of each trace block summary of all downwards exposed definitions of variables names and memory addresses are stored. During the backward traversal for slicing , when looking for variable names and the memory addresses, the presence in the summary downward exposed definitions are verified. If the definition is found then the trace block are traversed to locate the definition. Other wise using the size information method execution skip away to start of the trace block.

IV. CONCLUSION

In this paper the design of a dynamic slicing can greatly improve its practicality. Three different precise dynamic slicing algorithms designed and studied: FP, NP, and LP. demand driven analysis (with and without caching) is used and trace augmentation (with trace block summaries) to achieve practical implementations of precise dynamic slicing. It is demonstrated that the precise LP algorithm which first performs limited preprocessing to augment the trace and then uses demand driven analysis performs the best. In comparison to with other algorithms LP runs faster for small number of slices. In conclusion this paper it is shown that imprecise dynamic slicing algorithms are too imprecise and therefore not a better option, a carefully designed precise dynamic slicing algorithm such as the *LP* algorithm is practical as it provides precise dynamic slices at reasonable space and time costs.

REFERENCES

[1] H. Agarwal and J. Horgan, “Dynamic Program Slicing ” ACMSIGPLAN Conference on Programming language design and Implementation (PLDI), pages 246-256, 1990.

[2] H. Agrawal, R. DeMillo, and E. Sapfford,” Debugging with Dynamic Slicing and backtracking”,Software Practices and Experience SP&E, Vol. 23, No. 6, pages 589-616, 1993.

[3] Weiser. M. 1984. Program slicing. IEEE Transactions on Software Engineering 10, 4,352-357.

[4] Akgul, T., Mooney . V., and Pande , S. 2004. A Fast Assembly level reverse execution method via dynamic Slicing Conference on Software Engineering (ICSE), IEEE, computer Society, Edinberg, Scotland, UK, 522-531.

[5] Zaho, J. 2000. Dependence Analysis of Java bytecode, in IEEE Annual International Compute Software and Applications Conferences, Computer Society, Taipie, Taiwan, 486-491.

[6] Korel, B. and Laski, J. W. 1998. Dynamic program Slicing, Information processing Letters 29, 3, 155-163.

[7] Path Slicing Rupak Mujumdar and Ranjit Jhala proceedings of International Conference on Programming design and Implementation (PLDI), ACM press, 2005.

[8] Nevill Meaning, C. G. and Written, I, H, 1997, Linear time, hierarchy inference for e-conference. In data compression conference (DCC), IEEE Computer Society, Snowbird, Utab,3-11.