Program Slicing in the Presence of Database State

David Willmor and Suzanne M. Embury
Department of Computer Science,
University of Manchester,
Oxford Road, Manchester,
M13 9PL, United Kingdom
DWillmor|SEmbury@cs.man.ac.uk

Jianhua Shao
Department of Computer Science,
Cardiff University,
P.O. Box 916, Cardiff,
CF24 3XF, United Kingdom
J.Shao@cs.cf.ac.uk

Abstract

Program slicing has long been recognised as a valuable technique for supporting the software maintenance process. However, many programs operate over some kind of external state, as well as the internal program state. Arguably the most significant form of external state is that used to store data associated with the application, for example, in a database management system. In this paper, we propose an approach to supporting slicing over both program and database state, which requires the introduction of two new forms of data dependency into the standard program dependency graph. Our method expands the usefulness of program slicing techniques to the considerable number of database application programs that are being maintained within industry and science today.

1. Introduction

Program slicing [10] is a well known technique for assisting programmers in the comprehension of source code. Briefly, a program slice is a subset of a program which contains only those statements that determine the value a given variable (or set of variables) has at a particular point in the program. The slicer filters out those statements that are irrelevant to the task of understanding the variables of interest and thus allows the programmer to concentrate his or her effort on understanding a much smaller, simpler version of the program. Many different variants of the original slicing technique have been investigated, including slicing in the presence of procedures [5], slicing over complex data structures such as arrays [2] and dynamic slicing [1].

Traditional program slicers concentrate on the effect which a program has on some portion of its state. The user specifies the portion of state that is of interest and the slicer computes from the original program a second program that is simpler than the first, but equivalent in terms of its effect on the given portion of program state. The manipulation of *program state*, however, is not the only means by which programs achieve their intended effects; other forms of state may also have an important role to play. For example, many modern business programs operate over a database (DB), with program state being used as a temporary holding site for data that has been retrieved from the database or that may be written back to it. Such programs manipulate two kinds of state at the same time (the program state and the persistent state), often in closely inter-related ways.

Traditional program slicing algorithms do not take into account the presence of these additional forms of state. This leads to two problems. One of these is that statements which bridge the gap between the program state and some other form of state (e.g. by reading data from a DB into program variables) will not be included within the slice because the slicing algorithm is unable to detect their relevance to the slicing criteria in hand [9]. Therefore, the slices produced are not complete descriptions of the effect of the program on the variables/statement specified by the user. The second problem is that, when examining this kind of program, the user is often more interested in the effect of the program on the external state than on the program state. Rather than asking for the lines of code which define a particular variable at a particular statement, for example, we might wish instead to obtain the lines of code that are responsible for producing the DB state that exists at some particular statement (such as a transaction commit statement, for example). A traditional program slice can give us some of the relevant lines of code, but it will omit those statements which affect the DB state without affecting the program state; thus, the slice will again be incomplete.

In this paper, we describe two extensions to one of the standard methods of computing slices (the Program Dependence Graph approach of Ottenstein and Ottenstein [7]), that produce a slicing algorithm capable of navigating the additional data dependencies that are present in typical DB

programs. We begin, in Section 2, by briefly reviewing existing work on program slicing that treats program or other kinds of state in a non-traditional manner. In Section 3, we describe the extensions to the PDG that allow us to slice over both program and DB state. Finally, in Section 4, we conclude and present possible directions for future work.

2. Slicing in the Presence of External State

The basic concepts of program slicing were proposed by Weiser in 1979 [10], but since then many other forms of slicing have been proposed. In particular, researchers began to realise that slicing could extend its reach beyond program state to other stateful aspects of a software system. For example, Sivagurunathan *et al.* [8] recognised the complications introduced by programs which perform input/output operations, such as reading from the standard input stream. The problem here is that the standard I/O stream is acting as a form of state but, unlike program state, its semantics are not captured by the definition and use of variables. In fact, variables are probably used at some level to implement the behaviour, but they do not appear explicitly in the program and therefore are not visible to a traditional slicer.

Rather than modify the slicing algorithm itself, Sivagurunathan *et al.* chose to introduce a number of *pseudo-variables* into the program in order to make the hidden I/O state accessible to the slicer. The variables are added to the program by means of an "implicit state removal" transformation, which must be tailored to the specific semantics of each particular form of IO access. Essentially, to use this method, a transformation schema must be defined that maps the original language of the program into a new language that includes the pseudo-variables explicitly.

A similar problem occurs when programs access data that is stored externally, in some persistent state (e.g. flat files or a database management system). This was recognised by Tan and Ling, who also proposed a solution based around the introduction of pseudo-variables, although they did not adopt a transformational approach [9].

Tan and Ling focussed on persistent state in the form of flat files, accessed through specific COBOL file handling commands. They argued that statements such as a "fetch", which retrieves a data record from a file into program state, result in additional data dependencies (i.e. in this case, the definition of a variable's value) which the slicer must be made aware of. Their suggested approach for making these dependencies visible is similar to the work just described, in that they introduce a number of *implicit variables*, which are assumed to be updated when the file handling statements are executed. For each data file, a new variable is assumed to exist, and the various file access commands are characterised in terms of whether they use (i.e. retrieve from) or define (i.e. update) the data file.

While this approach solves the immediate problem of missing file access commands in slices, it has some limitations. In particular, if used on programs that operate with a more sophisticated data management system (such as a relational database), the slice produced by this approach will be larger than is necessary. For example, this method would not be able to detect that an update to a single column of a table does not have an effect on queries to that table that do not include that column. Using a single variable to characterise the use/defines relationships of a whole table is too coarse-grained for analysing more modern, flexible data management commands.

Of course, it is possible to introduce additional variables in order to model the interactions between parts of the state at a more fine-grained level. However, for even moderately complex database application, determining the exact set of variables required is non-trivial. Moreover, a more serious limitation of this pseudo-variable approach is that by representing the semantics of the data manipulation performed by the program as a set of faked data dependencies, we lose the ability to distinguish between them and to generate the different kinds of slice mentioned in the introduction. For example, we would not be able to compute a slice that describes *only* the effect of the program on the DB state, since we cannot distinguish which kind of state is affected by each data dependency.

An alternative approach is to create a new kind of data dependency from a direct analysis of the data manipulation semantics of the program. In the remainder of this paper, we explore this alternative approach to computing a variety of kinds of program slice in the presence of both program and DB state.

3. Incorporating Database Semantics into Program Slicing

Perhaps the most well-known program slicing algorithm is that proposed by Ottenstein and Ottenstein [7], which is based upon the notion of a *program dependence graph* (PDG) [6]. In a PDG, the nodes correspond to the statements of the program being sliced, and the edges describe two kinds of dependency between statements: *control dependencies* and *data dependencies*. Informally, a statement A is control dependent on a statement B if B is a conditional statement whose outcome determines (amongst other things) whether statement A will be executed or not. For example, a statement within the *then* part of an *if* statement is control dependent upon that *if* statement.

Equally informally, statement A is said to be data dependent on statement B if a variable whose value is used by statement A could have been given that value by the execution of statement B. For this condition to hold, the variable needs to be used by statement A and defined by state-

ment B, and there must be an execution path from B to A in which the variable is not redefined.

Given a PDG of a program p, a slice of p relative to a statement s of p and a set of variables Vs contains all nodes of the PDG from which s is reachable either by a control dependency or by a data dependency involving some variable in Vs.

This algorithm fails when dealing with programs which access DB state for fairly obvious reasons. The control dependencies are correct but the data dependencies in the PDG take account of only one form of state, i.e. the program state. Thus, the PDG can be seen as being incomplete because the program's full behaviour is defined in terms of both program state and DB state. In order to compute correct slices over DB state, we need to extend the PDG to take the missing dependencies into account.

In fact, two forms of data dependency are missing from a standard PDG. The first, and most obvious, of these are dependencies caused by the interaction between program and DB state. The second are dependencies which exist between pairs of statements which both manipulate DB state. If we extend the PDG with both these forms of dependency, we can compute more accurate slices for both kinds of state.

3.1. Program–Database Dependencies

Computation of data dependencies that arise due to the interaction between program state and DB state is relatively straightforward, provided the specific semantics of the commands used to access the persistent state are understood. Most database programming languages include some notion of special variables which are used as the point of connection between the program and the DB. For example, when using COBOL to access an IDMS database, special data structures are defined which mirror the structure of each record type in the DB. However, in more modern database programming languages, some of these implicit connections between program and DB state are made explicit. For example, in COBOL with Embedded SQL, program variables within the SQL code are are clearly designated (using the colon prefix). However, the slicer still needs to take into account the specific semantics of the DB command in order to determine which variables are used and which are defined by it.

Even in the embedded SQL examples, however, there is still a hidden variable, which must be considered. Almost all DB operations return a special error code variable, which is set to a value that describes its outcome. For example, in the version of COBOL we have been using for our own work, DB operations set a variable called `SQLCODE` on exit. These error variables must also be included in the set of data dependencies computed by the slicer.

We call such dependencies *PD dependencies*, for short, and define them as follows:

Definition 1 *A database statement ds is PD dependent on a statement s if there exists some variable v such that:*

- *v is used as an input to ds,*
- *v is defined by s, and*
- *there is a v-definition free path from s to ds.*

Definition 2 *A statement s is PD dependent on a database statement ds if there exists some variable v such that:*

- *the execution of ds sets v to be equal to one of the outputs of ds,*
- *v is used by s, and*
- *there is a v-definition free path from ds to v.*

The set of all such dependencies computed for a program is referred to as a program–database dependency graph (PDDG).

3.2. Database–Database Dependencies

The second form of data dependency that we introduce captures the situation when execution of one DB statement affects the behaviour of some other DB state that is executed after it.

Such dependencies are important in computing the correct slice for a program variable that is involved in a DB command, but they are also useful for computing a new kind of slice as well. This is a slice which explains not how some variable got its value but how some state of the DB was computed. For example, when trying to comprehend a large database application program, it is common to examine the rollback points, as these indicate the cases in which an error has occurred. By requesting a DB slice on a rollback command, the user can see which of the program statements have directly contributed to forming the DB state at the time of the rollback, and this he or she can work out the error condition which triggered it much more easily. Similarly, the user might examine slices computed on transaction commit, to gain an understanding of which DB states are considered legal products of the program. Such an analysis is very useful when searching for data-oriented business rules, or other forms of data semantics that are often locked into source code by the processes of maintenance and evolution.

How, then, are such useful dependencies as these to be computed. DB state takes a rather different form from program state. There are no individual variables, and all data within it is stored in complex data structures with a well-developed (and often explicit) semantics. Can we create a form of data dependency over DB state that is analogous

to the define-use model that has proved to be so successful for program state?

One approach is to categorise each DB statement ds in terms of three properties:

- the subset of the DB state that is read by the statement (*ds.read*),
- the subset of the DB state that is inserted by the statement (*ds.add*), and
- the subset of the DB state that is deleted by the statement (*ds.del*).

Given these sets, we can formulate the following definition:

Definition 3 *A database statement ds_1 is DD dependent on another database statement ds_2 iff:*

- $ds_1.read \cap (ds_2.add \cup ds_2.del) \neq \emptyset$,
- *there is a rollback-free execution path p between ds_2 and ds_1 (exclusive) such that:*
  - $ds_2.add \cap p.del \neq \emptyset$ *or*
  - $ds_2.del \cap p.add \neq \emptyset$.

That is, ds_1 is DD dependent on ds_2 if it accesses a part of the DB state that may have been modified in some way by ds_2 and that is not cancelled out by any intervening DB operations. An update may be cancelled out by a rollback statement (i.e. a transaction abort) or by another update that reverses its effect.

The full set of DD dependencies for a program is known as its Database–Database Dependence Graph (DDDG), and by computing it and combining it with the PDDG and the original program dependence graph, we can compute full slices of programs based on either program or DB state. But how can we compute the *read*, *add* and *modify* sets for an arbitrary DB statement (or sequence of DB statements)? The computation of the DD dependencies would be made most simple if the sets could be represented extensionally (since the standard set operators are easy to implement efficiently). Unfortunately, there is no practical way to achieve such a representation during a static analysis, when information about the particular DB state being operated on is not available.

The alternative is to extract an intensional representation of the use/define sets from the code. In the case of programs with embedded SQL commands, for example, we can extract relational algebra expressions which describe the tuples accessed or updated by a statement. In this case, generation of the use/define sets is straightforward, but reasoning about the sets in order to determine whether there is any overlap or not is rather more complicated (and computationally expensive). As is often the case with static forms of program analysis, we can make a controlled sacrifice of accuracy in order to gain efficiency and ease of implementation. The nett effect of this is that some of our slices will contain DB commands which, strictly speaking, should not be included in the slice. However, we can be certain that all necessary statements *have* been included. Details of our approach to reasoning over the use/define sets can be found elsewhere [11].

3.3. Computing the Slices

The computation of the dependencies produces four distinct types: control, data, program-database, and database-database. Each of these dependencies are represented by individual graphs, that when combined form a specifal case of the PDG, which we call the Database-Oriented Program Dependency Graph (DOPDG). The user specifies the slicing criteria, which is either a statement or a statement plus a set of variables of interest, and the slicer computes the requested slice by filtering out all statements that correspond to nodes in the DOPDG from which the specified statement is not reachable. This produces a slice which is correct with regard to program and DB state.

As well as considering the DOPDG as a whole, it is also possible to consider subgraphs constructed from one or more of its constituent dependencies. Thus, a subgraph when constructed from control and database-database dependencies would allow us to construct a pure DB state slice (DB statements and those they are control dependent upon). This is of particular use for understanding which DB statements of a program may be affected by a particular abort or commit operation.

We have implemented a version of the modified slicing algorithm that operates over COBOL programs with embedded SQL commands. The tool supports backwards slicing from a statement specified by the user, or from a given statement and set of variables of interest. The tool also allows the slice to be constrained to specific edge types allowing the construction of pure DB state slices and other combinations of edge types.

Details regarding the tools implementation, examples of its use, and details regarding its testing over code supplied by BT can be found in the technical report[11].

4. Conclusions

Information systems make up a very large proportion of the source code that is currently being maintained throughout the world, and programs which manipulate or query a DB in some form make up a very large proportion of these information systems. We have shown how traditional slicing algorithms do not take into account the additional semantics of a DB state, and have proposed two additional

forms of dependency to address this. The two new forms of dependency help the slicer to take into account the program behaviour that is taking place wholly within the DB state (using the DD-dependencies) as well as the interaction between the program state and the DB state (the PD-dependencies).

The approach we have taken (introducing new forms of data dependency) is rather different from the previous work in this area. Rather than using pseudo-variables to fool the slicer into computing the correct data dependencies, we have chosen to compute the additional dependencies by direct analysis of the code, and to modify the slicing algorithm to take the new dependencies into account. In fact, the change to the slicing algorithm is minimal, and we foresee a number of advantages to our approach:

- A DB state is often large and complex. Schemas containing hundreds of tables and many thousands of attributes are not uncommon in real world information systems. A great many pseudo-variables must be introduced in order to model the complex interactions between the different elements of DB state at a fine grained level.
- As we have seen, in order to increase slicing accuracy, it is necessary to consider which subset of a DB table is currently being accessed, rather than simply recording that the table as a whole has been accessed. Pseudo-variables are too coarse-grained a mechanism for recording the details of the complex sets of interactions typically found in database application programs.
- It is entirely possible that the new kinds of dependency we have proposed for use in database-oriented slicing may also be useful for other forms of analysis over DB programs. For example, standard data dependencies are of great importance in computing the impact of a proposed change.

We would like to explore further improvements to our slicing algorithm, perhaps to improve the accuracy of slicing, or to include elements of transformation to improve comprehensibility of the slices. We would also like to investigate different forms of slicing criteria that are specific to databases. For example, rather than giving a set of variables of interest as part of the slicing criterion, one could envisage supplying a list of tables or attributes from the database schema. The slice produced would be limited to only those statements that were directly or indirectly related to the manipulation of those schema elements. This would allow users to answer such questions as: “how is this table populated?”, “what are the conditions under which I am allowed to delete items from this table?” and so on.

The program slicer we have described in this paper is just one component of a larger collection of tools (e.g. [3, 4]) that we are constructing with the aim of supporting the maintenance programmer in the understanding, maintenance and evolution of complex data semantics.

Acknowledgements

We would like to thank Mr Nigel Turner and Mr Owain Hughes of BT for their assistance. David Willmor is supported by a studentship from the EPSRC.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, June 1990.

[2] D. Binkley. Computing amorphous program slices using dependence graphs. In *Proceedings of the 1999 ACM symposium on Applied computing*, pages 519–525. ACM Press, 1999.

[3] S. Embury and J. Shao. Assisting the Comprehension of Legacy Transactions. In *Proceedings of 8th Working Conference on Reverse Engineering: Workshop on Data Reverse Engineering*, pages 345–355, Stuttgart, Germany, Oct. 2001. IEEE Computer Society Press.

[4] S. Embury and J. Shao. Analysing the Impact of Adding Integrity Constraints to Information Systems. In J. Eder and M. Missikoff, editors, *Proceedings of 15th International Conference on Advanced Information Systems Engineering (CAiSE 03)*, LNCS vol. 2681, pages 175–192, Klagenfurt, Austria, June 2003. Springer.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, 1990.

[6] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth annual ACM Symposium on Principles of Programming Languages*, pages 207–218. ACM, ACM, Jan. 1981.

[7] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, May 1984.

[8] Y. Sivagurunathan, M. Harman, and S. Danicic. Slicing, I/O and the Implicit State. In *Proceedings of 3rd International Workshop on Automatic Debugging (AADEBUG’97)*, volume 2 (009-06) of *Linköping Electronic Articles in Computer and Information Science*, Linköping, Sweden, May 1997.

[9] H. Tan and T. W. Ling. Correct Program Slicing of Database Operations. *IEEE Software*, 15(2):105–112, Mar./Apr. 1998.

[10] M. D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

[11] D. Willmor, S. Embury, and J. Shao. Program Slicing in the Presence of Database State. Technical report, Informatics Process Group, University of Manchester - http://www.cs.man.ac.uk/∼willmord/, 2004.