Data Reverse Engineering using System Dependency Graphs

Anthony Cleve  
Computer Science Institute  
University of Namur  
21, rue Grandgagnage  
5000 Namur, Belgium  
ac1@info.fundp.ac.be

Jean Henrard  
ReveR S.A.  
130, Boulevard Tirou  
6000 Charleroi, Belgium  
jhe@rever-sa.com

Jean-Luc Hainaut  
Computer Science Institute  
University of Namur  
21, rue Grandgagnage  
5000 Namur, Belgium  
jlh@info.fundp.ac.be

Abstract

Data reverse engineering (DRE) is a complex and costly process that requires a deep understanding of large data-intensive software systems. This process can be made easier with the use of program understanding methods and tools. In this paper, we focus on the program slicing technique and show how it can be adapted to support DRE. We present a DML-independent SDG construction approach involving the analysis of database operations as a first stage. We describe a tool based upon this approach and we report on two industrial DRE projects.

1. Introduction

Data Reverse Engineering (DRE) is defined by Chikofsky [5] as "a collection of methods and tools to help an organization determine the structure, function, and meaning of its data". In particular, DRE aims at recovering the precise semantics of the data, by retrieving implicit data constructs and constraints (i.e. not explicitly declared in the database declaration but verified in the procedural code [7]), among which:

- fine-grained structure of entity types and attributes;
- referential constraints (foreign keys);
- exact cardinalities of attributes;
- identifiers of multi-valued attributes.

DRE is a complex and expensive task, that needs to be supported by program understanding techniques and tools. In this paper, we particularly focus on the use of program slicing as a mean to discover/validate hypotheses about implicit data structures and constraints. In the DRE context, program slicing typically helps to reach two intermediate objectives. The first one is the more traditional objective of reducing the visual search space when semi-automatically inspecting source code to discover complex business rules. The second goal concerns the automatic computation of data dependencies. Indeed, discovering dependencies between variables can be used as a basis to retrieve implicit constraints such as undeclared foreign keys. To this end, program slicing provides a very useful intermediate result: the System Dependency Graph (SDG).

Program slicing, initially introduced by Mark Weiser [17], has proved to be a valuable technique to support both software maintenance and reverse engineering processes [6, 3]. However, usual slicing algorithms fail to deal correctly with database application programs. Such programs do not only use standard files, but also store and manipulate data from external Data Management Systems (DMS). Therefore, a traditional slice computed from a database application program will potentially be incomplete or imprecise, due to the data dependencies hidden by the DMS.

In order to improve the accuracy of computed slices, additional data dependencies must be taken into account when constructing the SDG. These dependencies, induced by the execution of database operations, arise due to the interaction between program variables and database entity types and attributes.

In this paper, we propose an approach to compute accurate slices in the presence of database statements. We mainly concentrate on the core of our program slicing approach, which is the construction of an SDG as precise as possible.

The remainder of this paper is structured as follows. Section 2 defines some basic concepts used in the paper. In Section 3, the nature of the problem to be tackled is described. In Section 4, we present a method of computing accurate program slices in the presence of an embedded data manipulation language. Section 5 shows how this approach can be extended to handle programs that invoke data access...
modules to manipulate the database. In Section 6 we give an overview of the tools that we developed to support our methodology. Industrial reverse engineering projects are reported in Section 7. We discuss related work in Section 8. Section 9 concludes the paper and anticipates future work.

2. Basic Concepts

System Dependency Graph Horwitz and al. [11] introduced a new kind of graph to represent programs, called a system dependency graph, which extends previous dependency representations to incorporate collections of procedures with procedure calls.

The system dependency graph (SDG) for program P is a directed graph whose nodes are connected by several kinds of arcs. The nodes represent assignment statements, control predicates, procedure calls and parameters passed to and from procedures (on the calling side and in the called procedure).

The arcs represent dependencies among program components. An arc represents either a control dependency or a data dependency. A control dependency arc from node \( v_1 \) to node \( v_2 \) means that, during execution, \( v_2 \) can be executed/evaluated only if \( v_1 \) has been executed/evaluated\(^1\). Intuitively, a data dependency arc from node \( v_1 \) to node \( v_2 \) means that the state of objects used in \( v_2 \) can be defined/changed by the evaluation of \( v_1 \).

Program Slicing The slice (or backward slicing) of a program with respect to program point \( p \) and variable \( x \) consists of all statements and predicates of the program that may affect the value of \( x \) at point \( p \). This concept, introduced by M. Weiser in [17], can be used to debug programs, maintain programs and understand programs behavior [8]. In Weiser’s terminology, a slice criterion is a pair \( < p, V > \), where \( p \) is a program point and \( V \) is a subset of the program variables. Weiser claims that a slice corresponds to the mental abstractions that people make when they are debugging a program.

Horwitz and al. [11] also proposed an algorithm for interprocedural slicing that uses the system dependency graph. In their approach, a program is represented by a graph (the SDG) and the slicing problem is simply a node-reachability problem, so that slices can be computed in linear time.

Host Variables Data exchanges between the program and the database is performed by means of so-called host variables. We distinguish two kinds of host variables, depending on their role in the database operation:

- **input host variables** are used by programs to pass data to the DMS. For instance, COBOL variables occurring in the **where** clause of an embedded SQL query are of that kind.
- **output host variables** are used by the DMS to pass data and status information to programs. In embedded SQL, typical examples of output host variables are the status variable **SQLCODE**, as well as the COBOL variables occurring in a **into** clause.

3. Problem Statement

The complexity of the database-aware program slicing task lies on the nature of the Data Manipulation Language (DML). We identify four categories, according to the distance between the DML and the host programming language, which we assume to be COBOL in the remainder of this paper.

- **Native** In the case of standard files, programmers use native COBOL data access statements (e.g., **READ**, **WRITE**, **DELETE**, ...). Since all relevant information is contained in the program, traditional slicing on top of pure COBOL is sufficient.
- **Built-in** For DMS like IDMS/CODASYL, the COBOL language provides built-in instructions (i.e., **FIND**, **STORE**, ...) to access the database. Since they are seamlessly incorporated in the COBOL language, such instructions can be analyzed by traditional COBOL slicers as well. The only needed extension is an option to load the definition of the physical schema referenced by the **SUB-SCHMEA SECTION** of the **DATA DIVISION**. This physical schema is obtained through an additional DDL\(^2\) analysis phase.
- **Embedded** The third category collects DMS such as SQL or IMS, for which COBOL does not provide built-in data access statements. To access such DMS programmers write embedded instructions in the COBOL program. Such embedded code fragments belong to an external Data Manipulation Language (DML). They are embedded as is in the main programming language, called the host language. The DMS manufacturer provides a pre-processor (precompiler) that translates the embedded DML instructions into COBOL calls to external functions (programs) that implement the access to the DMS. Typical intricacies arise when analyzing programs with embedded DML code:
  - The embedded instructions are not standardized and thus vary from one DMS to another;

\(^1\)The definition is slightly different for calling arcs, but this does not change the principle.

\(^2\)DDL stands for Data Description Language
<table>
<thead>
<tr>
<th>Category</th>
<th>COBOL Code</th>
</tr>
</thead>
</table>
| Native     | MOVE 7 TO CUS-ID
READ CUSTOMER KEY IS CUS-ID
DISPLAY CUS-NAME |
| Built-in   | MOVE 7 TO CUS-ID
FIND CUSTOMER USING CUS-ID
DISPLAY CUS-NAME |
| Embedded   | MOVE 7 TO CUS-ID
EXEC SQL
SELECT NAME
INTO :CUS-NAME
WHERE ID = :CUS-ID
END-EXEC
DISPLAY CUS-NAME |
| Call-based | MOVE 7 TO CUS-ID
MOVE "CUSTOMER" TO REC-NAME
MOVE "FIND-BY-ID" TO ACTION
CALL "DAM" USING ACTION
REC-NAME
CUSTOMER
STATUS
DISPLAY CUS-NAME |

Figure 1. Illustration of native, built-in, embedded, and call-based DMLs

- The embedded code does not conform to the host language syntax;
- The physical database schema is not explicitly declared in the program itself.

- **Call-based** Many data-intensive programs invoke another program, called a Data Access Module (DAM), to access the database. The Call-based DML category accounts for this frequent situation. In this case, the slicing problem becomes very complex, especially since the DAM may in turn use a Call-based DML. Furthermore, two additional issues must be considered:
  - Unlike embedded DML instructions, the precise behaviour of DAM calls is not explicitly specified in a user manual;
  - When analyzing a DAM invocation it is necessary to determine the actual value of each input parameter, which is not always statically decidable.

Figure 1 shows similar COBOL fragments illustrating the differences between the four DML categories. Each fragment looks for a customer according to a given identifier and displays its name.

4. Slicing with Embedded Code

In this section, we elaborate a general methodology to slice programs in the presence of embedded DML code.

The proposed methodology, depicted in Figure 2, includes the analysis of database operations occurring in the program as a first stage. This program analysis phase, discussed in Section 4.1, extracts implicit data dependencies from DML code fragments. These dependencies are then used as input to the SDG construction process, as explained in Section 4.2. Once the SDG is available, a traditional but database-aware program slicing can be performed.

4.1. DML Code Analysis

The DML code analysis phase consists in analyzing the database operations performed by the program. This process aims at extracting data dependencies between host program variables (input or output) and database variables. Such dependencies are of two possible kinds:

- **direct mappings**
  
  \[ \text{form: } \text{DIRECT-MAP } \text{var}_{in} \text{ TO } \text{var}_{out} \]
  
  \[ \text{meaning: the value of variable } \text{var}_{in} \text{ directly affects the value of variable } \text{var}_{out}. \]  
  \[ \text{(var}_{in} \text{ and } \text{var}_{out} \text{ must be of compatible types). } \]
  
  Directly means that there exists a clear function to compute the value of \( \text{var}_{out} \) from the value of \( \text{var}_{in} \).

- **indirect mappings**
  
  \[ \text{form: } \text{INDIRECT-MAP } \text{vars}_{in} \text{ TO } \text{vars}_{out} \]
  
  \[ \text{(with } \text{vars}_{in} \text{ being optional)} \]
  
  The form \( \text{INDIRECT-MAP } \text{vars}_{in} \text{ TO } \text{vars}_{out} \) is used to indicate that the value of \( \text{vars}_{out} \) is indirectly affected by the values of \( \text{vars}_{in} \). This can occur through intermediate variables or through indirect database operations.
As an illustration, let us consider the COBOL/SQL program fragment shown in Figure 3. COBOL instructions and host variables are in upper-case, while embedded SQL code is in lowercase. The program fragment displays the name and the phone number of each customer living in a given city. It first accepts a zip number from the command-line, opens a SQL cursor and fetches each of its rows.

The open statement (lines 52-54) involves two different kinds of data dependencies. First, the value of the input host variable CUS-ZIP directly affects the value of column zip. Second, the value of column zip has an indirect influence on the values of the columns occurring in the select clause (name and phone). In addition, the value of status variable SQLCODE is also affected. Thus we can extract the following mapping statements from the embedded open statement:

DIRECT-MAP CUS-ZIP TO zip
INDIRECT-MAP zip TO name phone SQLCODE

When analyzing the fetch query (lines 79-83), implicit data dependencies can be detected. First, fetching a cursor indirectly influences the value of the status variable SQLCODE. Second, the values of the output host variables CUS-NAME and CUS-PHONE are directly affected by the values of the corresponding selected columns (name and phone). These implicit dependencies can be formalized by the following dependency pseudo-instructions:

INDIRECT-MAP columns(input-couple(q))
TO columns(output-couple(q)) ∪ output-host-var(q)

We will now express the general rules that can be used to extract dependencies from embedded SQL code. We first need to give some definitions. Let \( q \) be an embedded SQL query:

- \( \text{input-couple}(q) \) is the list of the couples \((h v, c)\), such that \( h v \) is an input host variable used in \( q \) and \( c \) is the column associated with \( h v \) in \( q \).
- \( \text{output-couple}(q) \) is the list of the couples \((h v, c)\), such that \( h v \) is an output host variable used in \( q \) and \( c \) is the column associated with \( h v \) in \( q \).
- \( \text{columns}(\text{list-couples}) \) is the list of the columns occurring in \( \text{list-couples} \), a list of couples \((h v, c)\), such as \( h v \) is a host variable and \( c \) is a column.
- \( \text{output-host-var}(q) \) is the list of all the output host variables used in \( q \) and that do not appear in \( \text{output-couple}(q) \). In other words, that list contains the output host variables that are not directly associated with a column, such as the status variable SQLCODE or variables resulting from aggregation queries.

Figure 4 specifies the general rule used for extracting data dependencies from an arbitrary embedded SQL query. The first part of the rule derives direct mappings between input host variables and their corresponding SQL columns. Those mappings typically occur in the \text{where} clause of the query. The second part is the indirect mapping from the columns associated with input host variables to the columns associated with output host variables and the output host variables that are not associated with any column. The last part of the rule represents the direct mappings between the
selected columns and their corresponding output host variables. Note that, since both input-couple and output-couple lists may be empty, the minimal dependency that can be extracted from an embedded SQL query is the following:

$$\text{INDIRECT-MAP TO SQLCODE}$$

meaning that the execution of the query indirectly influences the value of the status variable SQLCODE.

4.2. SDG Construction

To compute program slices for a program that contains embedded code, the SDG must be produced. This SDG must represent the control and the dataflow of both the host language and the embedded DML. In order to produce such an SDG, we use the results obtained by the DML code analysis process. The dependency pseudo-instructions (DIRECT-MAP and INDIRECT-MAP) are used (instead of the original code) to construct the SDG nodes and the data dependency arcs corresponding to embedded DML fragments. All SDG nodes are linked to the initial source code locations.

The use of this technique allows the SDG construction process to become DML-independent. The only language to be considered is the host language (here COBOL), augmented with the two additional pseudo-instructions. These pseudo-instructions are added to the COBOL grammar to stress the fact that they do not really implement the embedded code but are used to construct an equivalent SDG. This ad-hoc grammar augmentation is inspired by the scaffolding technique proposed by Sellink and Verhoef [14].

Once the full SDG has been built, the slices are computed using the usual algorithm.

5. Slicing with Data Access Module

In this section, we show how our methodology, depicted in Figure 2, can be extended to program slicing in the presence of a data access module (DAM). In other words, we will show how we can adapt our approach so that we can correctly slice programs that use a Call-based DML.

We can distinguish three possible kinds of DAM-based data access:

- **Global DAM**: there is only one DAM used to access all the tables/records and to perform all the possible actions (read, write,...);
- **One DAM per table/record**: each table/record has a dedicated DAM, which implements all the possible actions;
- **One DAM per action**: each action has a dedicated DAM, which can be called to access all the tables/records.

Our approach to program slicing with DAM invocations (depicted in Figure 5) extends the methodology described in Section 4. The extension is based on the following facts:

- Any data access module can be seen as a traditional DMS;
- Any call to a data access module can be seen, and analyzed, as a traditional DML instruction.

However, analyzing DAM invocations requires, as input, an additional knowledge of the DAM behavior itself. As already stated, we cannot assume that such a knowledge is available. Consequently, our methodology needs an extra step that consists in recovering a deep understanding of the DML semantics. Once that knowledge has been recovered, we can then retrieve, for each DAM invocation, the DML instructions actually executed by the DAM. This requires to determine the actual value of each input parameter of the DAM call.

Slicing in the presence of DAM invocations might lead to very imprecise results. We do not always know the actual value of the parameters used to call a DAM. Typically, the table/record to access and the action to be performed are unknown. Therefore, the computed slice will contain all the possible execution paths of the DAM and thus a lot of noise.

Let us consider the code fragment shown in Figure 6. In this example, a program invokes a data access module
to find a customer, using its identifier. The DAM code is provided in Figure 7. If the customer is found, the calling program displays its name and phone number.

In this example, if the slice w.r.t. the DISPLAY instruction (line 213 of Figure 6) is computed, it will include both select instructions (lines 956 and 1125 of Figure 7), since the slicer cannot make any assumption about the value of REC-NAME and ACTION.

Replacing DAM invocations with corresponding DML instructions can significantly improve the accuracy of computed slices, since this allows to focus on relevant control and data flows. For instance, if we replace the DAM call (lines 207-211 of Figure 6) with the corresponding select clause of Figure 7 (lines 955-960), then the slice w.r.t. the DISPLAY instruction will be minimal.

5.1. DAM Semantics Recovery

Analyzing a DAM call requires a correct understanding of how the DAM is implemented and to know the meaning of the different parameters. The name of the DAM and the value of the input parameters are usually sufficient to translate the call into a corresponding DML instruction.

There is no universal method or tool to recover the semantics of the parameters. It is, as usual in program understanding, a mixture of program analysis, domain knowledge, DAM construction knowledge and interviews of users (or programmers).

In our example of Figure 6, the program invokes the data access module using four parameters. Through the DAM semantics recovery phase, we can learn that:

- the first parameter (ACTION) specifies the database operation to be performed (read, write, etc.);
- the second parameter (REC-NAME) contains the name of the record type to be accessed;
- the third parameter (CUSTOMER) is the resulting record itself;
- the last parameter (STATUS) is an output status variable.

5.2. Input Parameters Resolution

In order to analyze DAM invocations correctly, we need to determine the actual value of each input argument (here, ACTION and REC-NAME). This can be supported by a customized backward slicing phase, searching for predecessor instructions that initialize DAM input parameters.

In our example, the MOVE statement of line 205 (resp. 206) will be found and analyzed, to determine the actual value of ACTION (resp. REC-NAME) at the time of DAM invocation.
5.3. DAM Call Analysis

Figure 8 summarizes the successive steps that are needed to analyzing DAM invocations. First, input parameters are resolved. Second, the DAM call is replaced by its corresponding DAM fragment, including the actual DML instructions. Finally, the DML code is analyzed as described in Section 4.1. The resulting code/pseudo-code will be used, instead of the original DAM invocation, to build the SDG.

6. Tool Support

6.1. Program Analysis

We implemented DML code analyzers for embedded SQL and IMS. Both analyzers rely on the ASF+SDF Meta-Environment [4]. The ASF+SDF Meta-Environment is an interactive development environment dedicated to the automatic generation of interactive systems, the manipulation of programs, specifications or other texts written in a formal language. We reused an SDF version of the IBM-VSII COBOL grammar, which was obtained by Lammel and Verhoeof [13]. We wrote SDF modules specifying (a sufficient subset of) the syntax of embedded SQL (resp. embedded IMS). On top of this augmented COBOL grammar, we implemented traversal functions [16] analyzing DML fragments and accumulating implicit data dependencies.

Figure 9 shows an example of ASF equation. This rewrite rule analyzes a select query and extracts its direct and indirect data dependencies. It implements the general rule given in Figure 4 in the particular case of a select statement. The rule can be applied to the example shown in Figure 8, for which the value of in-couple is [\{CUS-ID, id\}], while the value of out-couple is [\{CUS-NAME, name\}, \{CUS-PHONE, phone\}].

The result of the analysis consists of a set of lines. Each line represents a single data dependency extracted from a given DML fragment, and provides the following information:

- full path name of the program;
- fragment location (begin and end line numbers);
- dependency order number in the fragment;
- kind of mapping (direct or indirect);
- input variables (host or DML variables);
- output variables (host or DML variables).

The extracted dependencies are finally loaded in a relational database, which is used as an input to the SDG construction process.

6.2 SDG Construction and Slicing

The program slicing tool analyzes COBOL program with procedures (PERFORM), sub-routines calls (CALL) and arbitrary control flows (GO TO). A program is represented by a graph (the system dependency graph) and the slicing problem is simply a vertex-reachability problem. Therefore, slices may be computed in linear time in the number of edges when the graph has already been computed.

The computation of the graph is more costly. For the interprocedural slice, we use the SDG to represent the program and the algorithm that was described by S. Horwitz et al. in [11] to compute the slice. This algorithm can handle procedures and sub-routines.
The SDG construction algorithm can only manipulate variables that are local to procedures but cannot handle global variables present in COBOL programs. Therefore, for each procedure, we recover all the variables used (referenced and modified) by the procedure and create corresponding formal-in and formal-out parameters.

We use the augmented system dependency graph as proposed by Ball et al. in [2] to solve the orthogonal problem of slicing procedures with arbitrary control flows. The variables are represented by their physical position and their length. Indeed, we cannot use the names of the variables because they can be made of sub-level variables and can be redefined.

Both the SDG construction and the slicing algorithm are implemented in C++.

6.3. Parameter Resolution

During the construction of the SDG, both the variables used (referenced and modified) by each instruction and the constants used are stored. This allows to automate the resolution of DAM input parameters. To do so, the SDG is queried for the possible values of each input parameter (of the call to the DAM).

To determine which values (constants) can be stored in an input parameter, the dataflow edges of the SDG are followed backwards starting form the DAM call instructions until a constant is reached.

7. Case Studies

In this section, we describe two industrial reverse engineering projects for which our approach and tools were used.

7.1. COBOL/Embedded SQL

SDG construction with embedded SQL was used successfully, among others, within the context of a small data reverse engineering project. The purpose of this project was to convince a customer that it is possible to recover data dependencies using automated program analysis techniques.

The application we analyzed is written in COBOL with embedded SQL fragments. The relational database does not contain any (explicit) foreign key. The knowledge of the implicit constraints on the database was lost.

The customer plans to migrate their whole application. Before starting the actual migration process, they are looking for a methodology and tools, and need to evaluate the size and the complexity of the project.

We applied our methodology and tools to a small subset of the application: 27 tables, 784 columns, 95 programs, 150 000 lines of code, 124 declared cursors, 17 delete queries, 38 insert queries and 41 update queries.

The analysis of the embedded SQL code allowed us to extract more than 5000 dependencies (direct-maps and indirect-maps). Through the analysis of the SDG and the program slicing we were able to collect the following information:

- the set of tables (only 15) used by the 95 programs;
- the set of columns of these tables that are actually used by each program;
- 32 implicit data dependencies.

This case study proved that the analysis of embedded SQL code help to recover implicit referential constraints between tables (i.e., foreign keys). Indeed, analyzing SQL fragments allows, as a second step, to detect dataflows between columns of distinct tables. From such dataflows we can derive foreign key candidates, to be confirmed through a validation phase.

7.2. COBOL/DAM

The second project concerned a quite large application written in COBOL, using a IDS/II (CODASYL) database. The database contains 232 records types, 150 sets and 648 fields. The application is made up of 2287 programs totaling more than 2 000 000 lines of code and including 5952 DAM invocations.

It was absolutely necessary to analyze the DAM calls for the following reasons:

- There is only one data access module for all the records and all the actions. This DAM has the record name, a variable to store the record and the type of access (get first, get next, store, ...) among its parameters. If we computed the program slice for one of its calls, it would contain read and write access to each record type!
- The DAM also computes the physical position (DB-KEY) before writing a record. This is done to store records in different areas (files) according to a complicated rule made to optimize data access speed. All the related records are stored on the same disk.

Thanks to the analysis of DAM calls it was possible:

- to draw the usage graph, specifying which program uses which record type;
- The DAM also computes the physical position (DB-KEY) before writing a record. This is done to store records in different areas (files) according to a complicated rule made to optimize data access speed. All the related records are stored on the same disk.

Thanks to the analysis of DAM calls it was possible:

- to find a finer-grained structural decomposition for each record type;
- to discover more than 2000 implicit data dependencies.
In this particular project, the DAM input parameter resolution was very important. Without this step, it would have been impossible to determine which record is accessed and which operation is performed. Resolving the input parameters allowed us to build a precise usage graph and to derive a finer-grained decomposition of each record type. In the DDL code each record type contains only two fields (the access key and a big field for the remainder of the record). On the other hand, the COBOL variable used to store the record (when invoking the DAM) has a more precise decomposition.

7.3. Lessons Learned

Thanks to the two industrial case studies, we have proved that it is possible to construct accurate SDGs (and thus to compute precise program slices) for programs involving database operations, especially when data access is performed using an extension of the programming language (embedded SQL) or through a data access module.

To do this, we have used the divide and conquer method, by analyzing separately the database manipulation fragments. The result of this analysis is materialized by pseudo-instructions that produce the same SDG as that of the original code. From that point it is possible to use traditional SDG construction as well as program slicing techniques.

One important lesson is that it may be necessary to analyze/transform a program (and to build an incomplete SDG) in order to construct, in a second stage, the correct SDG. This is especially true when analyzing DAM usage, since it is essential to instantiate input calling parameters.

Experience has shown that every reverse engineering project is different from other ones. Hence the need for programmable, extensible and customizable tools. In this context, the reusability of our slicing approach and tools appears as an important advantage. Indeed, when one needs to slice programs using a new embedded language or DAM, the only adaptation required regards the analysis rules, while the SDG construction and the slicing algorithm remain unchanged. Our approach would also be well suited to the analysis of programs that use several embedded languages or that use embedded languages together with a data access module.

8. Related Work

8.1. Previous Work

Program slicing has long been considered as a valuable technique to support various software maintenance tasks such as debugging, regression testing, program understanding, and reverse engineering [6, 3]. A lot of researchers have extended the SDG to represent various language features and proposed variations of dependence graphs that allow finer-grained slicing. Among them, we mention [1] and [12].

Tan and Ling [15] recognised that traditional slicing methods cannot correctly deal with programs involving database operations. To face this limitation, they suggested the introduction of implicit pseudo-variables to capture the influence among I/O statements that operate on COBOL files. For each COBOL record type, a pseudo-variable is assumed to exist and to be updated when the file access statements are executed. Such pseudo-variables allow to introduce additional data dependencies at the record level.

More recently, Willmor et al. [18] proposed an approach to program slicing in the presence of database states. In particular, they introduced two new forms of data dependencies. The first category, called program-database dependencies, accounts for interactions between program statements and database statements. The database-database dependencies capture the situation in which the execution of a database statement affects the behaviour of another database statement.

As already stated, the way we simulate the dataflow behavior of DML code with pseudo-instructions is close to the scaffolding technique proposed by Sellink and Verhoef in [14]. Within the context of software renovation, source-code scaffolding consists in inserting some markup in the source code in order to store intermediate results of analysis/transformation processes and to share information between tools. In our case, we do not transform or scaffold the source code itself, but we store extracted dependencies into an external database.

8.2. Discussion

As Tan and Ling, we consider additional data dependencies involved in the execution of database operations. But our approach aims at extracting variable dependencies at a more fine-grained level. For instance, when analyzing an embedded SQL update statement, it is not sufficient to conclude that a table is updated. We also need to determine the set of columns that are actually affected and, if relevant, to make the link between these columns and the corresponding COBOL input host variables.

Our approach actually complements the work by Willmor et al. since we focus on recovering dependencies between program and database variables, while they introduced additional dependencies between program and database statements.

It is obvious that the data dependencies we extract from database statements allow, in a second stage, to compute statements dependencies in the style of Willmor. Indeed, we can determine for each database statement, the set of
variables that are used and defined. In addition, we also recover the links that hold between program variables and their corresponding database variables.

Another important difference lies on the fact that we clearly separate the analysis of the database operations from the SDG construction phase. This allows the latter to become DML-independent, and thus increases its reusability.

The main contribution of this paper with respect to previous work concerns our approach to slicing in the presence of DAM invocations. This immediately solves a real problem, since many programs do invoke other programs to access the database. The technique we propose allows to improve the accuracy of the computed slices by eliminating as much noise as possible when constructing the SDG.

9. Conclusions

In this paper, we presented a general methodology that allows to compute accurate slices in the presence of embedded database operations. The methodology is based on the combination of embedded code analysis and DML-independent SDG construction. We showed how this methodology can be generalized to deal with programs invoking data access modules (DAM).

While industrial reverse engineering projects have shown the suitability of our approach and tools, precisely measuring the positive effect of our methodology on the accuracy of constructed SDGs still remains to be done. Unfortunately, such an evaluation is particularly difficult in an industrial context. Indeed, case studies are very costly and require the cooperation of a customer ready to invest in comparative experiments.

We anticipate several directions for future work in database-oriented program slicing. In particular, we would like to explore the use of SDG querying techniques for database applications reengineering and evolution. For instance, SDG analysis seems to be a promising basis to understand the data access logic of a program, before adapting it to an evolving database schema or platform.

Acknowledgments Anthony Cleve was sponsored by the Belgian Région Wallonne and the European Social Fund via the RISTART project. We thank the WCRE’06 anonymous reviewers for their constructive criticism, as well as Jean-Roch Meurisse for proofreading drafts of our paper.

References


