

Data dependency elicitation in database reverse engineering

Jean Henrard, Jean-Luc Hainaut
Institut d'Informatique, University of Namur
rue Grandgagnage, 21 - B-5000 Namur - Belgium
tel: +32 81 724985 - fax: +32 81 724967
db-main@info.fundp.ac.be

Abstract

Database reverse engineering (DBRE) attempts to recover the technical and semantic specifications of the persistent data of information systems. Dependencies between records (data dependency) form a major class that need to be recovered. Since most of these dependencies are not supported by the DBMS, (foreign keys are the main exception, at least in modern relational DBMS), they have not be explicitly declared in the database schema. Careless reverse engineering will inevitably ignore them, leading to poor quality conceptual schema.

Several information sources can contribute to the elicitation of these hidden dependencies. The program source code has long been considered the richest, but also the most complex, of them. In this paper, we analyze and compare, through their respective quality and cost, different program understanding techniques that can be used to elicit data dependencies.

1. Introduction

Reverse engineering a piece of software consists, among others, in recovering or reconstructing its functional and technical specifications, starting mainly from the source text of the programs. Recovering these specifications is generally intended to redocument, convert, restructure, maintain or extend legacy applications.

In information systems, or data-oriented applications, i.e., in applications the central component of which is a database or a set of permanent files, the complexity can be broken down by considering that the files or database can be reverse engineered (almost) independently of the procedural parts as such, through a process called *Data Reverse Engineering* (DBRE in short).

This proposition to split the problem in this way can be supported by the following arguments.

- The semantic distance between the so-called conceptual specifications and the physical implementation is most often narrower for data than for procedural parts.
- The permanent data structures are generally the most stable part of applications.

- Even in very old applications, the semantic structures that underlie the file structures are mainly procedure-independent (though their physical structures may be highly procedure-dependent).
- Reverse engineering the procedural part of an application is much easier when the semantic structure of the data has been elicited.

Therefore, concentrating on reverse engineering the application data components first can be much more efficient than trying to cope with the whole application.

Even if reverse engineering the data structure is *easier* than recovering the specification of the application as a whole, it still is a complex and long task, especially when attempting to recover the implicit constructs, i.e., the data properties that hold in the database without being explicitly declared in the DBMS¹ schema.

One kind of constructs that are interesting to recover are the data dependencies, i.e., relations that exist between fields of the database. Foreign keys and redundant/derived fields are some examples of such relations. Most data dependencies are difficult to discover because they are not explicitly declared (except for foreign keys in modern relational database). Instead, evidence of their existence is buried in the code of the application, and can only be found by analyzing this code. On the other hand, these data dependencies generally translate business rules, so that their elicitation is essential to produce a high quality conceptual schema.

Discovering data dependency in the source code requires powerful program understanding techniques and tools, especially for real size projects that can score several millions LOC and several hundreds files and records. Those tools discover potential dependencies that need to be validated (check if the dependency really exists) and qualified (which kind of dependency, foreign key, redundancy) by the analyst.

None of these techniques are perfect. On the contrary, they generate noise (they detect dependencies that does not stand) and silence (they does not detect dependencies that hold). The noise and silence have a cost. For the noise, it

¹ DataBase Management System.

is the time needed to detect that the proposed dependency does not exist. The silence cost is more difficult to evaluate: it is the organizational cost of the incompleteness of the schema. If an incomplete schema is used to migrate data or to write a new application, it can lead to incorrect result that may have high cost. It can also be the cost of an application giving wrong answers and lacking the functions expected by the users.

This paper is organized as follows. Section 2 is a synthesis of a generic DBMS-independent DBRE methodol-

ogy. Section 3 explains what is data dependency and why it must be elicited. Section 4 presents different program understanding techniques. In section 5, those techniques are specialized to data dependency elicitation. Section 6 presents the DB-MAIN CASE tool that implements these techniques. Experiences gained by using these techniques on real projects are described in section 7. Section 8 shows the economic challenge of data dependency elicitation techniques.

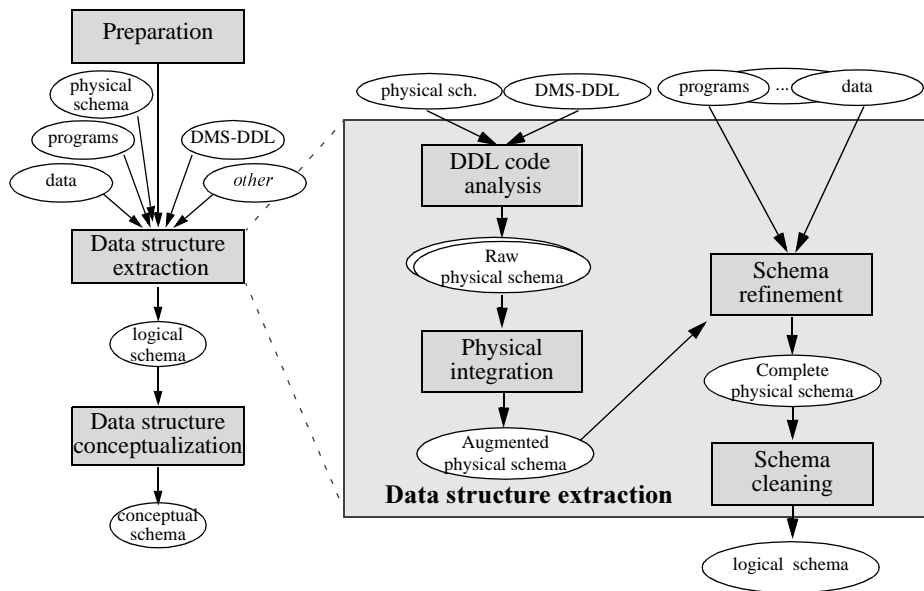


Figure 1. The major processes of the reference DBRE methodology (left) and the development of the data structure extraction process (right).

2. A Generic Methodology for Database Reverse Engineering

The reference DBRE methodology [5] is divided into three major processes, namely *preparation*, *data structure extraction* and *data structure conceptualization* (figure 1, left). The preparation process is not exactly a DBRE process, but its purpose mainly is to gather and to evaluate the relevance of the necessary information sources and for the analyst to get acquainted with the domain (interviews, demonstrations,...). The last two processes, that will be developed in the following, address the recovery of two different schemas and require different concepts, reasoning and tools. In addition, they grossly appear as the reverse of the physical and logical design usually considered in database design methodologies [1].

2.1. Data Structure Extraction

The data structure extraction process consists in recovering the complete DMS schema, called the *logical schema*, including all the explicit and implicit structures and constraints.

It is interesting to note that this schema is the document the programmer must consult to fully understand all the properties of the data structures (s)he intends to work on. In some cases, merely recovering this schema is the main objective of the programmer, who will find the conceptual schema itself useless.

In this reference methodology, the main processes of data structure extraction are the following (figure 1, right):

- *DDL¹ code analysis*: parsing the data structure declaration statements to extract the explicit constructs and constraints, thus providing a *raw physical schema*.
- *Physical integration*: when more than one DDL source has been processed, several schemas can be extracted. All these schemas are integrated into one global schema. The resulting schema (*augmented physical schema*) must include the specifications of all these partial views.

```

select customer....      ask-ord.
  record key is c-num.    accept c-num.
select ordre... .       read customer
fd customer.             invalid key
  01 cus.                 display "error"
  02 c-num.               go to ask-ord.
  ...                     move c-num to o-cus.
fd order.                ...
  01 ord.                 write ord.
  ...
  02 o-cus.

```

Figure 2. Example of code that implements an implicit foreign key from *ord-cus* to *cus-num*.

- *Schema refinement*: the most challenging problem of the data structure extraction phase is to discover and to make explicit the structures and constraints that were either implicitly implemented (figure 2) or merely discarded during the development process. The physical schema is enriched with implicit constructs made explicit, thus providing the *complete physical schema*.
- *Schema cleaning*: once all the implicit constructs have been elicited, technical constructs such as indexes or clusters are no longer needed and can be discarded in order to get the *complete logical schema* (or simply the logical schema).

The final product of this phase is the complete logical schema, that includes both explicit and implicit structures and constraints. Generally, this schema is no longer DBMS-compliant since the complete logical schema is the result of the refinement process, that enhances the schema with recovered implicit constraints, that are not necessarily DBMS compliant.

2.2. Data Structure Conceptualization

The third phase addresses the conceptual interpretation of the logical schema. Its goal is to propose a conceptual schema, of which the logical schema obtained so far could be a correct implementation. It consists in detecting and transforming (or discarding) non-conceptual structures, reducing redundancies, detecting and interpreting technical optimization and DBMS-dependent constructs and finally

in replacing the DBMS constructs with their abstract equivalent in the target conceptual model (ERA, UML, ORM, *etc.*).

The final product of this phase is the conceptual schema of the persistent data of the application. This process is outside the scope of this paper and will not be discussed further. Detail can be found in [3].

3. Data dependency elicitation

One kind of implicit constraints that need to be found during the *refinement phase* is made up of the *dependencies between fields* (of the same record or not). The most important classes of data dependencies certainly are foreign keys, redundant fields and existence dependencies.

A *foreign key* is a set of fields that is used to reference records in another file (or in the same file). As a consequence, at any time, for each source record, the value of the foreign key must be that of the identifier of a record in the target file. A *redundant field* has a value that is derived from source data that are part of the database itself. Such a field can be a copy of another field, or can be derived by a computation that takes other fields of the record (or of another record) as input. An *existence dependency* holds when the interpretation of the value of a field depends of the value of the other one.

Redundancy (computational) and existence dependencies are very frequent in legacy databases that are the result of different migration, integration, long evolution and maintenance. Due to lack of time and money, programmers are obliged to modify as quickly as possible the programs and the databases to follow the business evolution. This fast evolution forces them to modify the data structure and program without any concern about future evolution and the coherence of the database. So each modification increases the degree of denormalisation of the database.

Data dependencies are not supported (explicitly declared) by the DBMS, except foreign keys that can be controlled by modern relational DBMS. Since one of the goals of the conceptualization process is to translate, reduce or discard all those redundancies and dependencies, it is very important to discover them during the extraction phase. This explains the importance of their comprehensive elicitation to obtain a good conceptual schema.

Most often, undeclared data dependencies are checked and implemented in the procedural code of the applications that use the data. One of the *hottest* places where those constraints are, at least, verified is in the section of the code that precedes the record storage or modification. Indeed, before storing or updating of a record, the program has to verify that the new data comply with all the integrity constraints attached to the record type.

¹ Data Description Language.

For instance, to check a dependency constraint between fields of two records, the first record has to be read to check the constraint before the storage of the second record. Therefore, the control flow that ends with the *write* or *update* statement should include a *read* statement.

Dependency elicitation methods check if there exists a data or/and control flow between a read and a write instructions in a possible execution path of the program. If such a path exists, this mean that there exists a dependency between the two records. It is possible to refine the result by a closer analysis of the data flow between the two records to discover which fields are in relation.

To apply this method to real size projects, we need program understanding tools that discover automatically the fields in relation. In the next section, we will present three different techniques, namely variable dependency graph, system dependency graph and program slicing, that can be used to elicit data dependencies.

4. Program understanding techniques

4.1. Variable dependency graph

The *variable dependency graph*, VDG, is a weak - easy to compute - version of dataflow diagram. In this graph, each variable of a program is represented by a node, while an arc (directed or not) represents a direct relation (assignment, comparison, etc.) between two variables. To construct this graph, we only need to search the program for definite statement patterns without worrying to write a complete parser that analyzes the whole program. Figure 3.b illustrates the variable dependency graph of the program fragment of figure 3.a. If there is a path from variable A to variable C in the graph, then there is, in the program, a sequence of statements such that the value of A is in relation with the value of C.

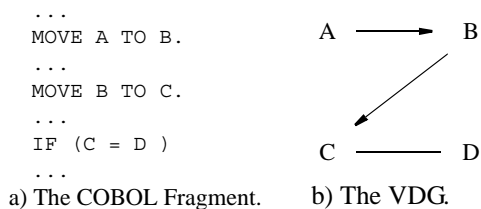


Figure 3. The variable dependency graph.

The very meaning of this relation between the variables is defined by the analyst depending on the pattern used to construct the graph. The interpretation of A being in relation with C can be one of the following: the structure of one variable is a variant of the other one, the variables share the same values, they denote the same real world object, etc.

4.2. 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 might affect the value of x at point p . This concept, originally discussed by M. Weiser in [9], can be used to debug programs, maintain programs, understand programs behaviour [4]. In Weiser's terminology, a *slicing criterion* is a pair $\langle p, V \rangle$, where p is a program point and V is a subset of the program's variables.

Horwitz and al. [7] introduce a new kind of graph to represent programs, called a system dependency graph, which extends previous dependency representations to incorporate collections of procedure, with procedure calls. They also give an algorithm for interprocedural slicing that uses the system dependency graph. We extend this graph construction algorithm to add arbitrary control flow (Go To's) as presented by Ball and Horwitz in [2].

In their approach, they represent a program by a graph (the system dependency graph) and the slicing problem is simply a node-reachability problem, so that slices can be computed in a linear time.

```

0 main.           3  if(B = 2)       7 P(X,Y) .
1  A = 1.         4  print B.       8  Y = X.
2  call P(A,B) .  5  else
                  6  call P(B,C) .
                  6  print C.

```

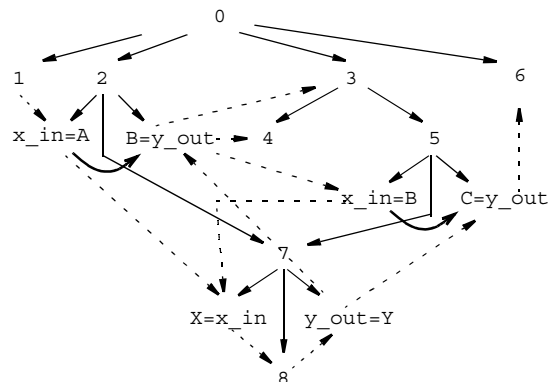


Figure 4. An example program and its corresponding SDG.

The *system dependency graph* (SDG) for program P is a direct graph whose nodes are connected by several kinds of arcs. The nodes represent the 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,

the component v_2 can be executed/evaluated only if v_1 has been executed/evaluated¹.

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 . As a consequence, program computation might be changed if the relative order of the component represented by v_1 and v_2 were reversed.

Figure 4 shows a program and its corresponding system dependency graph. Control dependencies are represented using plain arrows and data dependencies are represented using dashed arrows. On the calling side, information transfer is represented by a set of nodes ($x_{in}=A$, $B=y_{out}$, $x_{in}=B$ and $C=y_{out}$) that are control dependent on the call-side. Similarly, information transfer in the called procedure is represented by a set of node ($X=x_{in}$ and $y_{out}=Y$) that are control dependent on the procedure entry node. The bold arrows represent the transitive data dependencies due to the procedure call on the calling side. The presence of such edges permits the slicing operation to move “across” a call without having to descent into it.

The program slicing is computed by the traversal of the SDG. The computation is performed in two phases. Both Phases 1 and 2 traverse the system dependency graph to find the set of nodes that can reach a given set of nodes along certain kinds of arcs. Phase 1 identifies nodes that can reach s , and are either in P itself or in a procedure that calls P (either directly or transitively). Phase 2 identifies nodes that can reach s from procedures (transitively) called by P or from procedures called by procedures that (transitively) call P .

5. Program understanding application to dependency elicitation

In this section the different program understanding techniques described in the previous section will be specialized to data dependency elicitation. At the end of the section, the quality of the results obtained by each technique will be compared to show the strength and weakness of each of them. One of the criteria to compare two techniques is to compare the silence and noise generated by both of them. *Silence* is a constraint that exists but is not discovered. On the other hand, *noise* is a constraint that is suggested by a technique but does not exist.

Every technique presented finds couples of fields that are possibly in relation, through source code analysis. The analyst has to check manually each couple to validate the result. If (s)he decides that the couple is a valid dependency, (s)he has to qualify it (as foreign key, redun-

dancy,...) and adds this dependency to the database schema. In a comprehensive schema refinement process, other sources must be used as well, such as schema analysis and data analysis. These additional techniques can help finding and qualifying the dependencies [5].

5.1. Variable dependency graph

If a data flow exists between two fields of the database, we need to construct the variable dependency graph where the arcs represent assignments (move) between two variables. A dependency between two fields is identified whenever a path exists between the two fields in the graph.

The usage of this graph can lead to three kinds of silence and to two kinds of noise.

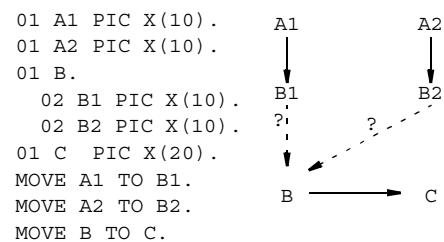


Figure 5. Example of silence in variable dependency graph.

The *first* source of silence lies in the relations that are represented by the arcs. If we use assignment statements only, then all the other instructions that contribute to the dataflow (compute, multiply, string,...) are ignored. This kind of silence can be reduced by increasing the number of statements we are looking for.

The *second* source of silence is that the graph is not aware of the structure of the variables. Figure 5 gives an example where such silence appears. The decomposition of B in $B1$ and $B2$ is not represented, so that the path between $(A1, A2)$ and C remains undetected.

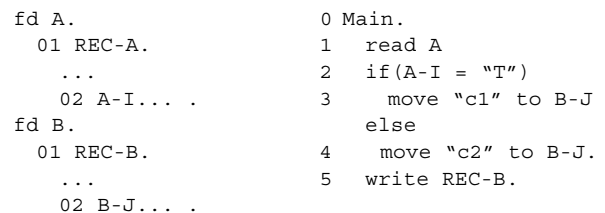


Figure 6. The dependency between $A-I$ and $B-J$ is implemented using a test (if).

Finally, ignoring control flow can also generate silence. For example in figure 6, the result of the test on $A-I$ ($if(A-I = "T")$) is necessary to discover the dependency (a computed dependency) between $A-I$ and $B-J$. The correspond-

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

ing VDG is empty because there is no assignment between variables in this examples, but there is a dependency. The last two kinds of silence are very difficult to address with this technique.

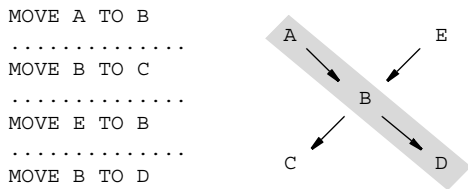


Figure 7. Example of variable dependency graph noise.

Noise can be generated because the graph only represents dataflow and not the control flow. There exist variables that are connected by a path in the graph though they are not in relation at execution time. As show in figure 7, if a path exists between two variables in the graph, that does not mean that there exists a corresponding valid execution path in the program.

The second source of noise is that if a variable represents a record field, it does not necessarily contain a value that appears in the database. Let us consider the tricky program :

```

read A.
move "cst" to A1.
move A1 to B1.
write B.

```

... where *A1* is a field of record *A* and *B1* of record *B*. The graph shows a relation between *A1* and *B1*, so we can conclude erroneously that there is a dependency between *A1* and *B1*.

5.2. Program slicing

There are several usage of the program slicing techniques and its underlying SDG to detect dependencies in a program. This section presents three different SDG querying techniques: program slicing, dataflow program slicing and dataflow program slicing with variable follow-up.

Program slicing.

The first program slicing application, that is to detect dependencies between fields, requires the computation of the slice with respect to a *write* instruction and the written record (*write B*). If the slice contains a *read* instruction (*read A*), then there exists at least one execution path such that the read instruction influences (is in the slice computed with respect to) the write instruction. The result is not very precise because we do not know which fields of *A* influence which fields of *B*. This technique generates noise: *A* influencing *B* does not imply that there is a dependency between *A* and *B*.

```

L1.
....
read next A
  at end go to L2.
...
go to L1.
L2.
* the computation of B
* does not use A
...
write B.

```

Figure 8. The statement *read next A* is in slice w.r.t. *write B*. However, there is no field dependency between both records.

Figure 8 shows such an example where a first loop reads each record of *A*. This first loop is followed by a section that writes *B*, the computation of the value of *B* does not rely on reading *A*. The *read A* appears in the slice w.r.t. *write B* because all the records of *A* need to be read before the execution of the second loop but there is no dependency between the fields of *A* and *B*.

Program slicing is one example of SDG querying. It is possible to query it differently to extract other information.

Dataflow program slicing.

A *dataflow program slicing* can be defined as the program slicing where only the data dependency arcs are used and not the control arcs. If a dataflow program slice, computed with respect to a write instruction, contains a read, it means that there is a dataflow between some fields of the read to the fields of the write. As in the first program slicing usage, the result is not very precise because we do not know which part of the records are in relation. This technique does not generate noise, but it misses all the dependencies that rely on control such as instruction *if*. Figure 9 presents the SDG corresponding to figure 6 program. The dependency implemented in this program is not detected by the dataflow program slicing. The path that needs to be followed to detect the dependency between *read A* and *write B* are shaded and we notice that it contains control arcs.

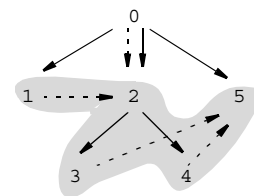


Figure 9. The system dependency graph of figure 6 program.

Dataflow program slicing with variable follow-up.

To increase the precision of this technique, when there exists a path (based on data dependency only) between a read and a write, it is possible to analyze each instruction to determine which parts of the records are used. This is called *dataflow program slicing with variable follow-up*.

Figure 10 shows how this can be done on an example. After each instruction, we adjust the part of the read record (A) that is used. When the write instruction is reached we know which part of A influences which part of B.

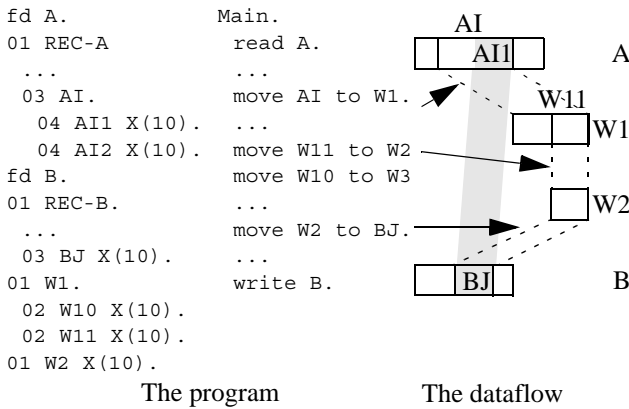


Figure 10. Field dependency detection using dataflow program slicing.

However, as in the previous technique, this one only finds dependencies that are implemented using dataflow between fields.

5.3. Silence and noise of each technique

Figure 11 is a comparison of the source of noise and silence of each technique presented in the previous section. In this comparison, without surprise, the techniques using the SDG are better than those that use VDG. They generate less silence, less noise and the noise and silence are included in those generated by the VDG.

A noise is a couple of fields detected but for which there is no dependency. Silence is a dependency between two fields that is checked or implemented in the analyzed code but that is not detected. Other kinds of silence may be due to an incomplete analysis of the code of the application or to the fact that they have not been controlled by program code though they could be elicited through data analysis.

The techniques using the SDG are immune from the silence generated by VDG due to the incomplete knowledge of the data structure and to the consideration of only some kinds of statements. Because the SDG computation does the complete parsing of the program and has the knowledge of the data structure.

VDG and dataflow program slicing do not analyze control flow, so that none of these techniques discover constraints implemented by control flow. Discovering such constraints requires control flow program slicing.

The VDG can generate noise because only the assignments between variables are analyzed, so when an assignment refers to a field, this field may not contain a valid value. The SDG techniques do not generate this kind of noise because, they only analyze paths that start at a *read* and end by a *write*. In this way, we are sure that all the referenced variables receive their value from the *read* (directly or indirectly).

Ignoring the control flow generates noise in the VDG, i.e. paths between two variables that are never followed in the execution of the program are impossible in the program slicing and dataflow, because, by construction, the SDG only represents valid execution paths.

This proves that techniques based on the SDG give better results than the VDG one.

One big advantage of dataflow program slicing with variable follow-up is that it gives more precise results than pure program slicing and dataflow program slicing. It gives exactly which fields are in relation while the others only give the records.

Technique name	Source of noise	source of silence
variable dependency graph	ignores control flow; does not verify that DB fields contain valid value	ignores control flow; ignores data structure; depends on the instruction we are looking for
program slicing	control dependency between records does not mean data dependency	none
dataflow program slicing	none	only dataflow dependency
dataflow + variables follow-up	none	only dataflow dependency

Figure 11. Noise and silence generated by each technique.

6. The DB-MAIN CASE tool

DB-MAIN is a general-purpose database engineering CASE environment that offers sophisticated reverse engi-

neering tool-sets. DB-MAIN is one of the results of a R&D project started in 1993 by the Database Engineering Laboratory of the University of Namur (Belgium). Its purpose is to help the analyst in the design, reverse engineering, migration, maintenance and evolution of database applications.

DB-MAIN offers the usual CASE functions, such as database schema creation, management, visualization, validation, transformation, as well as code and report generation. It also includes a programming language that can manipulate the objects of the repository and allows the user to develop his (her) own functions. Further detail can be found in [3] and [5].

DB-MAIN also offers several functions that are specific to the data structure extraction process [6]. The *extractors* extract automatically the data structures declared in a source text. Extractors read the declaration part of the source text and create corresponding abstractions in the repository. The *foreign key assistant* is used during the Refinement phase to find the possible foreign keys of a schema.

Program analysis tools include three specific program understanding processors.

- A *pattern matching* engine searches a source text for a definite pattern. Patterns are defined into a powerful pattern description language (PDL), through which hierarchical patterns can be defined.
- DB-MAIN offers a *variable dependency graph* tool. The relation between two variables is defined by a pattern.
- The *program slicing* tool computes the program slice with respect to the selected line of the source text and one of the variables, or component thereof, referenced at that line. There exists also a command line version of this tool that allows users to execute complex analysis in a batch mode (without user interaction), to query the SDG and to save the result for further analysis.

One of the lessons we (sometimes painfully) learned is that they are no two similar DBRE projects. Hence the need for easily programmable, extensible and customizable tools. The DB-MAIN CASE tool (and more specifically its meta functions) includes sophisticated features to extend its repository and its functions. In particular, it offers a 4GL language (*Voyager2*) through which analysts can develop their own customized functions.

7. Case study / experiences

This section will present some industrial projects together with facts we learned during the application of the program analysis techniques described in this paper. The applications analyzed are COBOL programs using files and/or

databases. We have restricted our study to COBOL programs due to the cost of the development of program slicing for other language.

To compare the number of possible dependencies discovered by the variable dependency graph and the dataflow program slicing with variable follow-up (dataflow for short), we have tried both techniques on two quite different applications. The first one is a small COBOL program (1600 LOC) that acts as a gateway between two sub-schemas of an IDMS database. It copies records of one sub-schema into the other one with some filtering and reorganization of the records. The second one is an application that manages a warehouse. It is composed of 13 COBOL programs, totaling 41151 LOC and uses only files to store data.

	VDG		dataflow		real dep.
	valid dep.	false dep.	valid dep.	false dep.	
gateway	10	3	13	0	16
warehouse	631	381	1627	0	n/a

Figure 12. The number of valid and false dependencies found by VDG and dataflow with variable follow-up for two applications.

Figure 12 shows the number of valid and false dependencies generated by each technique and the total number of dependencies implemented in the analyzed applications. The valid dependencies columns contain the number of proposed dependencies that are effectively implemented. The false dependencies columns is the number of proposed dependencies that do not exist (noise). The relative performance of the different techniques heavily depends on the size of the program, the complexity of the algorithm and the programming style used. In the first program, the VDG find almost as many dependencies as the dataflow (77%). This can be explain by the relative simple algorithms used: it reads one record and stores its values into 2 or 3 other records after some simple tests and mainly uses *move* instructions to transfer value from one record to the other. None of the techniques discover three dependencies that are implemented using control flow.

The results obtained for the second application are quite different. This application is a lot more complex: there is a huge amount of ratio computation, tests and *goto's*. This explains why the VDG gives very poor results, i.e., much noise and even more silence (it only discover 631 valid dependencies for 1627 discovered by the dataflow program slicing with variable follow-up). Paths in the VDG that are impossible during a valid execution of the program generate the majority of the noise. The silence came mainly from the VDG construction, where we only use the *move*

instruction. We do not know the exact total number of dependencies in this application because its analysis is still in process and a complete manual analysis of the code would be too expensive.

When the tools give a list of possible dependencies, the manual part of the job only start. For each couple proposed by the tools, the analyst has to validate and to qualify the dependency. This job is very tedious, specially for large projects and time consuming due to the large number of couples to checks. This effort can be greatly reduced, if it is done with the help of a local programmer/analyst that has a very good domain and application knowledge. Indeed, (s)he can quickly validate most of the dependencies without the need to look at the code or data.

8. The economic challenge

As shows in the previous section, the dataflow technique gives better results than the VDG. But its initial cost is very high, because the construction of the SDG requires a code parser and the transformation of the syntactical graph of the program into the SDG. This is a very expensive job, since the parser and the graph transformation are language dependent, so they need to be done for each new language and to be adjusted to each new version of the language.

On the other hand, variable dependency graph is easy to implement. For each instruction we are looking for there is only one pattern to write.

Another aspect of the cost of data dependency detection is the evaluation of the noise and of the silence. Noise cost is easy to evaluate: it is the time needed to validate a proposed dependency by code or data analysis or by user interview. The difficulty to evaluate the silence cost, is that by definition the silence is not materialized and it is very difficult to know how much dependencies have been missed. The cost of the silence will not appear in the extraction but they will appear during the conceptualization and even later when the conceptual schema is used as an input for a data migration, datawarehouse or reengineering project. It can lead to incorrect results that may have high cost. The later the silence will be detected, the more expensive it will be to correct it.

9. Conclusion

In this paper, we have presented data dependency elicitation, why such constraints are important to be discovered and how some program understanding techniques and tools can help to discover them.

Those techniques generate noise (they detect false constraints) and silence (they miss existing constraints). Each

technique quality has been evaluated to find the one that gives the best results.

Cost evaluation shows fair but disappointing results: the more precise is the result, the higher is the price. However, getting a satisfying cost model still needs considerable research and experiments.

We have developed specific analyzers that are included in the DB-MAIN CASE tool. Information about the DB-MAIN project can be found at <http://www.info.fundp.ac.be/~dbm>.

10. Acknowledgment

This paper is the “formalized” result of several DBRE projects. We would like to thanks all the persons who patiently answer all our tedious questions about their applications, business domain, technical environment, programming practices and who validated our results.

11. References

- [1] Batini, C., Ceri, S. and Navathe, S.B.: “*Conceptual Database Design - An Entity-Relationship Approach*”, Benjamin/Cummings, 1992.
- [2] Ball T. and Horwitz S.: “Slicing Programs with Arbitrary Control Flow”, technical report tr1128, University of Wisconsin (<ftp://ftp.cs.wisc.edu/tech-reports/reports/92/tr1128.ps.z>) (1992).
- [3] Englebert, V., Henrard J., Hick, J.-M., Roland, D. and Hainaut, J.-L.: “DB-MAIN: un Atelier d'Ingénierie de Bases de Données”, Ingénierie des Système d'Information, 4(1), HERMES-AFCET, 1996.
- [4] Gallagher, K. B., Lyle, J. R.: “Using program slicing in software maintenance”, IEEE Transactions on Software Engineering, 17(8) p. 751--761, Aug. 1991.
- [5] Hainaut, J.-L., Roland, D., Hick J.-M., Henrard, J. and Englebert, V.: “Database Reverse Engineering: from Requirements to CARE Tools”, Journal of Automated Software Engineering, 3(1), 1996.
- [6] Henrard, J., Englebert, V., Hick, J.-M., Roland, D., Hainaut, J.-L.: “Program understanding in databases reverse engineering”, in Proc. of DEXA'98, Vienna, 1998.
- [7] Horwitz S., Reps T., and Binkley D.: “Interprocedural slicing using dependence graphs”. ACM Transactions on Programming Languages and Systems, 12(1):26--60, January (1990).
- [8] Tilley S: “A reverse-engineering environment framework”. Technical report CMU/SEI-98-TR-005, Carnegie Mellon University, <http://www.sei.cmu.edu/publications/documents/98.reports/98tr005/98tr005abstract.html>, 1998.
- [9] Weiser, M.: “Program Slicing”, IEEE TSE, 10, 352-357 (1984).