DELTA software technology

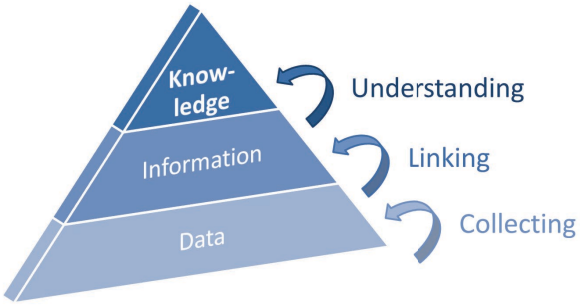
AMELIO LOGIC DISCOVERY

"Information is not Knowledge"

- Frank Zappa

Well, Frank Zappa is not particularly known for IT quotes but this line from one of his song texts is very true.

In an attempt to comprehend the functionality of a certain software, we begin one step before the information: with the data from the software. But first, of course, information must be generated from this data before you can derive knowledge from it. This connection between data, information and knowledge is often depicted as knowledge pyramid.



To make it even more difficult the required data is at first hidden in source code or the programs. To enable further processing the data must be extracted by a parser. The first information is generated as soon as this data is interlinked and stored in a repository or as ABSTRACT SYNTAX TREE (AST).

No matter how extensive and detailed it might be, this information is not enough to make qualified statements about the logic of a software. It is also not enough to somehow sort or arrange the information and to consolidate it. Real knowledge about the application doesn't show itself in this way.

We want to show this by using the following practice example. We also want to show how the transition from the information to the knowledge can be realized.

A Very Simple Task

As part of a large migration project a customer asks us whether we are able to find out which programs have write access and to which part of the database. The aim is to find program clusters for the migration that have a certain part of the database in common. For the read accesses the customer has already found a solution by using a real time replication so that only the write accesses are relevant.

This seems to be a relatively simple task at first glance because we have already the data about all programs and databases in ASTs and repositories in preparation of the automated migration. The data

definition and DB operations are already completely analysed and any information about it is available in detail. First evaluations are therefore provided and performed quickly.

In principle, the result complies with the following small exemplary diagram 1 (red arrows for write operations and grey arrows for read operations which must not be considered):

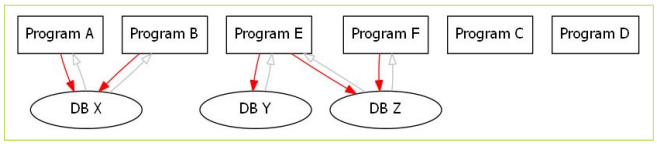


Diagram 1

The actual result is a lot more extensive and more complex, of course, because we have several thousand programs and hundreds of DB tables in this project.

Program	DB X	DB Y	DB Z
A	upd	-	-
B	upd	-	-
E	-	upd	upd
F	-	-	upd

The information collected in this way is transferred to a table per program and is arranged in the way that possible clusters to be derived from the write operations can be recognized.

But already during the implementation of this solution is gets clear that this selected approach is not sufficient.

A Not So Simple Task

Many of the programs involved are sub-programs which also contain DB operations. As the appropriate main programs cannot be executed without the associated sub-programs and can therefore not be migrated, the DB operations called via sub-program must be considered in the analysis.

Even in the simple example you can see that the resulting diagram 2 gets more complex.

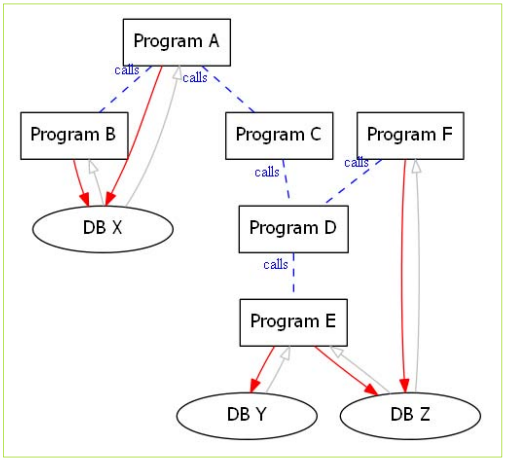


Diagram 2

Especially the cluster table changes completely as the DB operations which are indirectly executed via sub-program get visible (marked in blue).

Program	DB X	DB Y	DB Z
A	upd	(upd)	(upd)
B	upd	-	-
C		(upd)	(upd)
D	-	(upd)	(upd)
E	-	upd	upd
F	-	(upd)	upd

The result of this analysis is very significant and shows that the relationship between programs and databases are more complex than initially expected. In particular, there are more dependencies than expected.

A Much More Difficult Task

During the discussion of the results doubts arise whether it is right to assume that the call of a sub-program automatically leads to the fact that all functions presented therein are always executed. Isn't it likely that, for example, the execution of DB operations depend on how the respective sub-program is called and that not every main program uses all possible functions of the sub-programs?

Typical cases are the so-called CRUD modules (CRUD means Create, Read, Update and Delete) with all kinds of DB operations per DB table. These modules are normally called by many programs but often only read operations are requested. That means assuming that the call of such a CRUD module also includes the execution of update or delete statements leads to wrong conclusions.

The following applies to the small example:

The information

1. The program *A* calls the program *E* via *C* and *D*
2. The program *E* changes the databases Y and *Z*

does not automatically mean that the execution of the program *A* results in or might result in changes of the databases Y and *Z*!

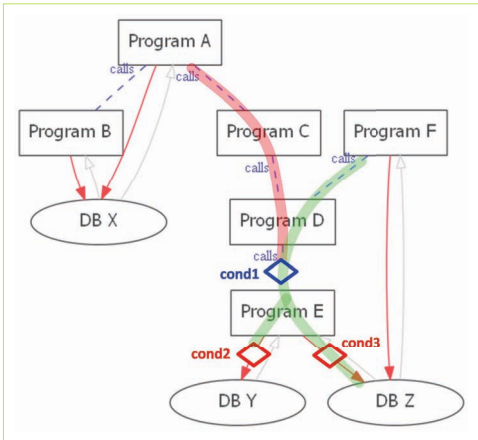


Diagram 3

To illustrate this, conditions for the sub-program call (*cond 1*) and the write operations (*cond 2* and *cond 3*) are inserted into the exemplary diagram 3. Obviously the write statement for Y is only executed if *cond 1* and *cond 2* are fulfilled. The same applies to *Z*, *cond 1* and *cond 3*.

When you suppose, for example, that calls from the program *A* don't fulfil the condition *cond 1* and that calls from the program *F* don't fulfil the condition *cond 2*, cluster table would look like diagram 4.

Program	DB X	DB Y	DB Z
A	upd	(upd)	(upd)
B	upd	(upd)	(upd)
C	upd	-	-
D	-	(upd)	(upd)
E	-	upd	upd
F	-	(upd)	upd

Diagram 4

To decide whether the programs *A* or *F* change the DB tables Y or *Z*, the complete path from the programs *A* and *F* to the appropriate options must be analysed and "understood".

But this "understanding" is associated with a large range of complications:

- The paths that must be analysed might run across any number of programs, with branches, loops and alternative paths.
- In practice, the conditions which stand in the diagram beside the connections to the sub-program and the databases consist of several, partially alternative or even inconsistent conditions.
- The conditions might depend on data elements which are defined in the program interface. This means that also the data flow in the higher-level programs must be analysed.

Information is not Knowledge

Arriving at this point it is clear that the large amount of comprehensive and detailed information stored about the application package is a good starting but it is not sufficient to give a qualified answer to the asked questions. The right answers can only be given if the application logic is fully understood and accessible.

If a limited maintenance task only covers some individual programs, you can achieve this manually. There are appropriate tools that allow you to follow the application functions stepwise in a controlled

manner. But this is a completely unthinkable approach for the analysis of hundreds and thousands of programs. We need a solution that automatically gathers knowledge from a large amount of information and literally transforms this information into knowledge.

To achieve this, a next, big abstraction step is necessary. For that purpose, new models are derived from the information about the implemented functionality in the software, as we can find them in repositories and ASTs. These models show the application functions and make the logic available.

The example above is not an exotic special case but is very typical of the type of questions that arise in conjunction with the modernisation of software systems.

- Which program code is unused or redundant?
- How can programs be restructured for a better maintainability?
- Which application functions are contained in a program?
  What does the program do and under what conditions?
- Which program modules and functions can be encapsulated as service and be re-used?
- Which program modules and routines can be re-used as classes or methods in an object-oriented re-implementation?
- ...

In all these cases one or more logical models are required which allow a qualified answer to the questions. These models are the control flow model, the condition model and the data flow model.

The Logical Models

The Control Flow and Procedure Model

A control flow model allows to find out whether there are one or more paths from the program input to a certain point in this program. It displays the branches, loops and alternatives on these paths and the conditions they depend on.

In legacy languages like COBOL there are no explicit definitions of procedures or functions. In the extended control flow the implicitly embedded procedures are made visible.

In the given task all paths from the program input to the DB operations or to the relevant sub-program calls are determined for each program.

The Condition Model

This model has two main tasks:

1. To bring all conditions into a language-independent and normalized form so that they can be compared with each other without further ado. Even simple conditions can often be formulated in different ways. That applies especially to large expressions.
2. There must be mechanisms to combine several partial conditions to one new normalized condition. In the process redundancies and inconsistencies must be detected.

In the described task all conditions must be combined on the path from the main program to the DB operation in the way that a decision can be made which condition must be fulfilled to achieve this point or to find out that these conditions can't be fulfilled at all.

The Data Flow and Data Dependency Model

This extended data flow model describes three essential, complementary aspects:

1. The data flow at all levels, top-down from the program across the different procedures to the individual statements. Each incoming and outgoing data is visible.
2. The dependencies between the individual program steps that result from the data flow are held. Uncovering the implicit dependencies makes it possible to re-structure programs reliably.
3. Values are often assigned to individual data elements at several different points in the

program. But the distinction of the resulting different roles of the data elements and the presentation of the data flow per role is an indispensable prerequisite for understanding the logic of an application.

To analyse the paths from the main programs to the DB operations the understanding of the data flows is indispensable. In program interfaces data elements can typically be found which have different roles that depend on the call. These roles must be kept separated to be able to evaluate the further data and control flow.

The Result – From the HOW to the WHAT

The mentioned migration project has been concluded successfully at the end of 2011 and has been made productive at 1.1.2012 without any problems. The cluster formation analysis just one step in a transformation project that was characterized by totally model-driven automation.

The knowledge and experience from this and some other projects are the decisive basis for the development of our current product **AMELIO Logic Discovery**. The usual meta data repositories answer especially the question: HOW is the application implemented? In **AMELIO Logic Discovery** this information is automatically transformed into logical models and provides answers to the question: Was does the application do?

This transition from the HOW to the WHAT is the decisive step to gain knowledge from information.

INFORMATION IS NOT KNOWLEDGE – but there are ways to generate information from data and to derive knowledge from the information.

AMELIO® Logic Discovery

helps to understand the existing COBOL- and PL/I-applications and thus reduces the costs for re-implementation of the existing functions and for the modernization of the applications.

Further information can be found under:

www.d-s-t-g.com/amld

Order number: MA 24'018.01 – July 2013