DELTA software technology

AMELIO LOGIC DISCOVERY

"Information is not Knowledge"

- Frank Zappa

Zugegeben, Frank Zappa ist nicht gerade der typische Zitatgeber in der Informatik, aber diese Zeile aus einem seiner Liedtexte trifft das Thema sehr genau.

Bei dem Versuch die Funktion von Software zu verstehen beginnen wir sogar noch einen Schritt vor den Informationen, bei den Daten über die Software. Wobei klar ist, dass aus diesen Daten zunächst die Informationen erzeugt werden müssen, bevor daraus Wissen abgeleitet werden kann. Dieser Zusammenhang Daten, Informationen und Wissen wird oft als Wissenspyramide dargestellt.

Um es noch etwas schwieriger zu machen, sind die benötigten Daten im Quelltext von Programmen erst einmal schwer verdaulich verpackt . Sie müssen für die weitere Verarbeitung mittels Parser extrahiert werden. Die ersten Informationen entstehen, wenn diese Daten miteinander verknüpft und in einem Repository oder als ABSTRACT SYNTAX TREE (AST) abgelegt werden.

Egal wie umfangreich und detailliert sie auch sein mögen, diese Informationen reichen aber noch nicht, um qualifizierte Aussagen über die Logik von Softwareanwendungen zu treffen. Es genügt auch nicht, diese Informationen irgendwie zu sortieren, zu ordnen und zu verdichten. Wirkliche Erkenntnisse und damit das Wissen über die Anwendung entstehen dadurch nicht.

Anhand des folgenden Beispiels aus der Praxis möchten wir das darstellen. Ebenso soll gezeigt werden, wie der Übergang von den Informationen zum Wissen realisiert werden kann.

Eine ganz einfache Aufgabe

Ein Kunde fragt uns im Rahmen eines größeren Migrationsprojekts, ob wir nicht in der Lage sind herauszufinden, welche Programme schreibenden Zugriff und auf welche Datenbanken haben. Das Ziel ist, Programm-Cluster für die Migration zusammen mit den gemeinsam genutzten Datenbanken zu

finden. Für die lesenden Zugriffe hat sich bereits eine Lösung mittels Realtime-Replikation gefunden, sodass nur schreibende Operationen relevant sind.

Das erscheint auf den ersten Blick als eine relativ einfache Aufgabe, haben wir doch schon durch die automatische Migration die Daten über alle Programme und Datenbanken in ASTs und Repositories. Die Datendefinitionen und DB-Operationen sind bereits vollständig analysiert und alle Informationen dazu stehen detailliert zur Verfügung. Erste Auswertungen sind daher schnell bereitgestellt und ausgeführt.

Das Ergebnis entspricht im Prinzip dem folgenden kleinen Beispieldiagramm 1 (dabei stehen die roten Kanten für Schreiboperationen und die grauen für Leseoperationen, die aber nicht berücksichtigt werden müssen):

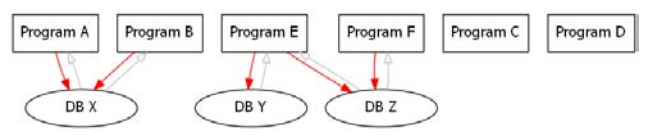

Diagramm 1

Das tatsächliche Resultat ist natürlich sehr viel umfangreicher und komplexer. Schließlich haben wir es in diesem Projekt mit mehreren tausend Programmen und hunderten von DB-Tabellen zu tun.

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

Die so ermittelten Informationen werden per Programm in eine Tabelle übertragen und so geordnet, dass aus den Schreiboperationen abgeleitet mögliche Cluster zu erkennen sind.

Schon während der Implementierung dieser Lösung wird aber klar, dass der gewählte Ansatz nicht ausreicht.

Eine nicht ganz so einfache Aufgabe

Viele der betroffenen Programme sind Unterprogramme, die natürlich ebenfalls DB-Operationen enthalten. Weil aber die entsprechenden Hauptprogramme nicht ohne die zugehörigen Unterprogramme ausgeführt und daher auch nicht ohne diese migriert werden können, müssen bei der Analyse auch die via Unterprogramm mittelbar aufgerufenen DB-Operationen berücksichtigt werden.

Selbst an dem einfachen Beispiel ist zu sehen, dass das resultierende Diagramm 2 deutlich komplexer wird.

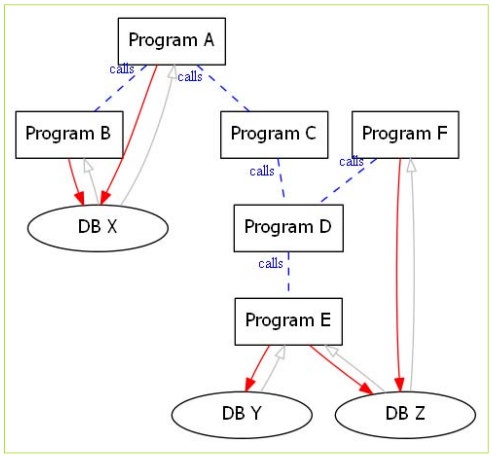

Diagramm 2

Besonders die Cluster-Tabelle ändert sich gründlich, denn die mittelbar via Unterprogramm ausgeführten DB-Operationen werden sichtbar (blau markiert).

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

Das Ergebnis dieser Analyse ist deutlich aussagefähiger und zeigt, dass die Beziehungen zwischen den Programmen und den Datenbanken komplexer sind, als zunächst angenommen. Vor allem gibt es

aber viel mehr Abhängigkeiten als auf den ersten Blick zu sehen war.

Eine sehr viel schwierigere Aufgabe

Bei der Diskussion der Resultate kommen aber Zweifel auf, ob es richtig ist anzunehmen, dass der Aufruf eines Unterprogramms auch automatisch dazu führt, dass alle darin dargestellten Funktionen immer ausgeführt werden. Ist es nicht wahrscheinlich, dass beispielsweise die Ausführung von DB-Operationen davon abhängt, wie das jeweilige Unterprogramm aufgerufen wird und dass nicht jedes Hauptprogramm alle möglichen Funktionen der Unterprogramme nutzt?

Ein typischer Fall sind die sogenannten CRUD-Module (CRUD steht für Create, Read, Update und Delete), mit allen möglichen DB-Operationen pro DB-Tabelle. Diese werden in der Regel von vielen Programmen aufgerufen, wobei häufig nur Leseoperationen angefordert werden. Das heißt, davon auszugehen, dass der Aufruf eines solchen CRUD-Moduls auch die Ausführung von Update- oder Delete-Anweisungen einschließt, führt zu falschen Aussagen.

Für das kleine Beispiel gilt daher:

Aus den zwei Informationen

1. Das Programm *A* ruft via *C* und *D* das Programm *E* auf
2. Programm *E* verändert die Datenbanken Y und *Z*

kann nicht automatisch geschlossen werden, dass die Ausführung des Programms *A* zu Änderungen an den Datenbanken Y und *Z* führt bzw. führen kann!

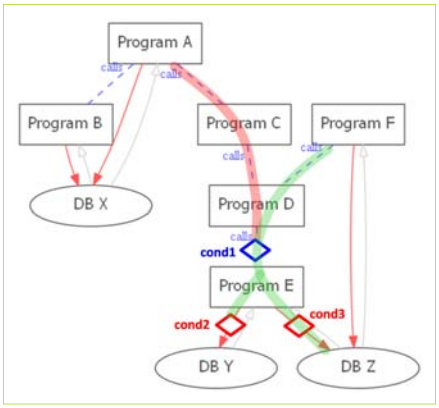

Diagramm 3

Um das zu verdeutlichen sind in das Beispieldiagramm 3 Bedingungen für den Unterprogrammaufruf (*cond 1*) und die Schreiboperationen (*cond 2* und *cond 3*) eingefügt. Offensichtlich wird die Schreibanweisung für Y nur ausgeführt, wenn *cond 1* und *cond 2* erfüllt sind. Entsprechendes gilt für *Z*, *cond 1* und *cond 3*.

Wenn man beispielsweise annimmt, dass Aufrufe aus dem Programm *A* die Bedingung *cond 1* und Aufrufe aus dem Programm *F* die Bedingung *cond 2* nicht erfüllen, dann ergibt sich daraus die untenstehende erneut veränderte Cluster-Tabelle.

Programm	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

Damit aber entschieden werden kann, ob die Programme *A* oder *F* die DB-Tabellen Y oder *Z* verändern, muss der komplette Pfad von den Programmen *A* und *F* bis zu den entsprechenden Operationen ausgewertet und „verstanden“ werden.

Dieses „Verstehen“ ist jedoch mit einer ganzen Reihe von Komplikationen verbunden:

- Die Pfade, die analysiert werden müssen, können prinzipiell über beliebig viele Programme verlaufen, mit Verzweigungen, Schleifen und alternativen Wegen.
- Die Bedingungen, die im Diagramm vereinfacht an den Verbindungen zum Unterprogramm und den Datenbanken stehen, setzten sich in der Praxis aus mehreren, teilweise alternativen oder gar widersprüchlichen Bedingungen zusammen.
- Die Bedingungen können von Datenelementen abhängen, die in der Programmschnittstelle definiert sind. Was wiederum bedeutet, dass

auch der Datenfluss in den übergeordneten Programmen analysiert werden muss.

Information is not Knowledge

An diesem Punkt angekommen wird deutlich, dass die gespeicherte große Menge umfassender und detaillierter Informationen über das Anwendungspaket zwar ein guter Ausgangspunkt für die Analysen darstellt, allein aber nicht ausreicht, um eine qualifizierte Antwort auf die gestellten Fragen zu geben. Um die richtigen Antworten geben zu können, muss die Logik der Anwendungen verstanden und auswertbar gemacht werden.

Wenn es bei einer begrenzten Wartungsaufgabe nur um einzelne Programme geht, kann man das natürlich manuell erreichen. Es gibt entsprechende Werkzeuge, die es erlauben, den Anwendungsfunktionen Schritt für Schritt kontrolliert zu folgen. Bei der Auswertung von hunderten und tausenden von Programmen ist das aber ein völlig undenkbarer Ansatz. Wir brauchen eine Lösung, die automatisch aus der großen Menge von Informationen Erkenntnisse ableitet und buchstäblich diese Informationen in Wissen transformiert.

Um das zu erreichen, ist ein nächster, großer Abstraktionsschritt notwendig. Dazu werden aus den Informationen über die Implementierung der Software, wie wir sie in Repositories und ASTs finden, neue Modelle abgeleitet, welche die Anwendungsfunktionen abbilden und die Logik auswertbar machen.

Dabei ist das geschilderte Beispiel kein exotischer Sonderfall, sondern sehr typisch für die Art von Fragen, die sich bei der Modernisierung von Softwaresystemen immer wieder stellen.

- Welcher Programmcode ist unbenutzt oder überflüssig?
- Wie können Programme zur besseren Wartbarkeit restrukturiert werden?
- Welche Anwendungsfunktionen enthält ein Programm? Was tut das Programm und unter welchen Voraussetzungen?
- Welche Programmbausteine und -Funktionen können als Services gekapselt und weiterverwendet werden?
- Welche Programmbausteine und -Routinen bieten sich an, in einer objektorientierten Re-Implementierung als Klassen oder Methoden wiederverwendet zu werden?
- ...

In all diesen Fällen werden ein oder mehrere der logischen Modelle benötigt, die es uns auch erlaubten, die Fragen zur Cluster-Bildung qualifiziert zu beantworten. Das sind vor allem das *Kontrollflussmodell*, das *Bedingungsmodell* und das *Datenflussmodell*.

Die logischen Modelle

Das Kontrollfluss- und Prozedurmodell

Ein Kontrollflussmodell erlaubt es etwa herauszufinden, ob es einen oder mehrere Pfade vom Eingang eines Programms zu bestimmten anderen Punkten in diesem Programm gibt. Es stellt dar, welche Verzweigungen, Schleifen und Alternativen es auf diesen Pfaden gibt und von welchen Bedingungen diese abhängen.

In Legacy-Sprachen wie COBOL gibt es keine expliziten Definitionen von Prozeduren oder Funktionen. Im erweiterten Kontrollflussmodell werden die implizit verwendeten Prozeduren sichtbar gemacht.

Bei der gegebenen Aufgabe werden je Programm alle Pfade vom Programmeingang zu den DB-Operationen bzw. zu den relevanten Unterprogrammaufrufen ermittelt.

Das Bedingungsmodell

Dieses Modell hat zwei Hauptaufgaben:

1. Alle Bedingungen so in eine sprachunabhängige und normalisierte Form zu bringen, dass sie ohne weiteres miteinander verglichen werden können. Selbst einfache Bedingungen können oft auf unterschiedliche Arten formuliert werden.

Bei größeren Ausdrücken gilt das umso mehr.

2. Es muss Mechanismen geben, um mehrere Teilbedingungen zu einer neuen normalisierten Bedingung zusammenzufügen. Dabei müssen Redundanzen und Widersprüche erkannt werden.

Bei der beschriebenen Aufgabe müssen alle Bedingungen auf dem Pfad vom Hauptprogramm zur DB-Operation so zusammengefasst werden, dass entschieden werden kann, welche Bedingungen erfüllt sein müssen, um diesen Punkt zu erreichen bzw. herauszufinden, dass diese Bedingungen überhaupt nicht erfüllt werden können.

Das Datenfluss- und Datenabhängigkeitsmodell

Dieses erweiterte Datenflussmodell beschreibt drei wesentliche, sich ergänzende Aspekte:

1. Der Datenfluss auf allen Ebenen, top-down vom Programm, über die verschiedenen Prozeduren, bis zu den einzelnen Statements. Es werden jeweils die eingehenden und ausgehenden Daten sichtbar.
2. Die Abhängigkeiten zwischen den einzelnen Programmschritten, die sich aus dem Datenfluss ergeben, werden festgehalten. Durch die Aufdeckung der impliziten Abhängigkeiten wird es beispielsweise möglich, Programme zuverlässig zu restrukturieren.
3. Einzelnen Datenelementen werden im Programm oft an mehreren unterschiedlichen Stellen Werte zugewiesen. Die Unterscheidung der dadurch entstehenden verschiedenen Rollen der Datenelemente und die Darstellung des Datenflusses pro Rolle ist aber eine unerlässliche Voraussetzung, um die Logik einer Anwendung zu verstehen.

Zur Analyse der Pfade von den Hauptprogrammen zu den DB-Operationen ist das Verständnis der Datenflüsse unabdingbar. Typischerweise finden sich in Programmschnittstellen Datenelemente, die abhängig vom Aufruf unterschiedliche Rollen haben. Diese Rollen müssen auseinandergehalten werden, um den weiteren Daten- und Kontrollfluss beurteilen zu können.

Das Resultat – Vom WIE zum WAS

Das genannte Migrationsprojekt wurde Ende 2011 erfolgreich abgeschlossen und ab 1.1.2012 ohne jegliche Probleme in Produktion genommen. Dabei war die Analyse zur Clusterbildung nur ein weiterer Baustein in einem durch vollständig modellgetriebene Automation geprägten Transformationsprojekt.

Die Erkenntnisse und Erfahrungen aus diesem und einigen anderen Projekten bildeten die entscheidenden Grundlagen für die Entwicklung unseres aktuellen Produkts **AMELIO Logic Discovery**.
Die üblichen Meta-Daten-Repositories beantworten vor allem die Frage: WIE ist ein Anwendung implementiert? In **AMELIO Logic Discovery** werden diese Informationen automatisch in logische Modelle transformiert und diese liefern Antworten auf die Frage: Was tut die Anwendung?

Dieser Übergang vom WIE zum WAS ist der entscheidende Schritt von der Information zum Wissen.

INFORMATION IS NOT KNOWLEDGE – aber es gibt Wege aus Daten Informationen zu erzeugen und aus den Informationen Wissen abzuleiten.

Bestellnummer: MA 14‘018.01 – Juli 2013