Electronic Communications of the EASST
Volume 8 (2008)

Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007)

Lightweight Visualisations of COBOL Code for Supporting Migration to SOA

Joris Van Geet and Serge Demeyer

12 pages

Guest Editors: Tom Mens, Ellen Van Paesschen, Kim Mens, Maja D'Hondt
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Lightweight Visualisations of COBOL Code for Supporting Migration to SOA

Joris Van Geet[1] **and Serge Demeyer**[2]

[1]Joris.VanGeet@ua.ac.be, [2]Serge.Demeyer@ua.ac.be
http://www.lore.ua.ac.be/
University of Antwerp, Belgium

Abstract: In this age of complex business landscapes, many enterprises turn to Service Oriented Architecture (SOA) for aligning their IT portfolio with their business. Because of the enormous business risk involved with replacing an enterprise's IT landscape, a stepwise migration to SOA is required. As a first step, they need to understand and assess the current structure of their legacy systems. Based on existing reverse engineering techniques, we provide visualisations to support this process for COBOL systems and present preliminary results of an ongoing industrial case study.

Keywords: SOA migration legacy reverse-engineering visualisation COBOL

1 Introduction

Mismatches between business and IT pose a threat to the agility with which enterprises can adapt to changing requirements. In an attempt to ensure their competitive advantage, some enterprises are turning to Service Oriented Architectures as a means to align their IT portfolio with their business. Because of the enormous business risk involved with replacing an enterprise's IT landscape, a stepwise migration to SOA is required.

The *Service Oriented Migration and Reuse Technique* (SMART) [LMS06] is an existing methodology for defining a migration process consisting of five distinct phases. First you should (1) establish the stakeholder context, (2) describe the existing capabilities, (3) and describe the target SOA state in an iterative and incremental manner. Based on these results you can (4) analyse the gap between the existing capabilities and the target state, after which you can (5) develop a migration strategy. All these phases can — and should — be supported by reverse engineering techniques and tools.

As a first step, enterprises need to assess the current structure of their legacy system(s). Such a *First Contact* [DDN02] typically aims at building an overall mental model of the system at a high level of abstraction. During this process exceptional entities become apparent, which can be studied afterwards. As building a mental model of large legacy systems is not a trivial task, this process is usually supported by (visualisation) tools [SFM97].

Studies show that COBOL mainframes process more than 83% of all transactions worldwide and over 95% of finance and insurance data [AAB+00]. Needless to say that maintaining these systems is of vital importance, which is why we focus our efforts on COBOL. Unfortunately, there is not one standard COBOL language: many variants and dialects have manifested them-

selves over the years. To cope with this plethora of COBOL variations, lightweight parsing techniques are preferred.

In this position paper we present such a lightweight technique for visualising functional dependencies and data dependencies between COBOL programs. After a short introduction to services and what it could (should?) mean for reengineering legacy systems (Section 2) we present our data model on COBOL, define structural properties of this data and explain their usefulness in the context of migration to SOA (Section 3). Then we define views on this structure and explain how they should be interpreted (Section 4). We provide an initial experience report of an ongoing industrial case study in Section 5 and suggest improvements to our approach in Section 6. Related work is discussed in Section 7 after which we conclude.

2 The Service Concept

As a result of the increasing interest in SOA, many interpretations have been given to the concept of a service. Ranging from an enabler of a capability accessible through a prescribed interface and consistent with predefined constraints and policies [OAS06] to technical, network-centric implementations typically based on web-services[1]. In fact, the term *services science* has been coined as it reaches far beyond software engineering or even computer science [SR06].

In an enterprise context, however, a service can be best described as a way to specify encapsulated business functionality independent from concrete implementations. In this context, a service is more of a business concept than an IT concept. This means that we cannot simply identify services from the source code of a legacy system, because a thorough understanding of the organisation and the domain is required. We can, however, make the functional dependencies within the legacy system explicit and highlight the exceptional entities. This information can generate discussion with the domain experts and enterprise architects to find out which of these entities pose a threat for migrating to SOA. In this context a service would, ideally, be implemented as a loosely coupled component [KBS04]. Indeed, the effort required to extract a 'service' can reasonably be expected to increase with the number of dependencies originating from or targeting the associated source code.

3 Characterising Source Dependencies

3.1 COBOL Artefacts

As a means to characterise functional and data dependencies in COBOL source code, the following artefacts are of interest.

- A *program* is a functional block of COBOL code uniquely identified by a `Program-ID`. Programs are the basic building blocks of a COBOL system.
- A *copybook* is a reusable *snippet* of COBOL code, contained within one source file, that usually consists of a data declaration to be shared by different programs.
- A *`CALL` statement* is responsible for invoking a program from within another program using its unique `Program-ID`. The thread of execution is then passed on to the called

[1] http://www.w3.org/2002/ws/

program until the execution ends and the control returns to the calling program.
- A *COPY statement* is responsible for *copying* the contents of a copybook directly into the COBOL source. This construct enables code level reuse of data declarations.
- A *missing program or copybook* is a program or copybook that is referenced from within the system under study but not part of the system itself. They have been identified as IDs used in CALL and COPY statements which we could not map to available source files.

Currently, we extract this information from the COBOL sources with a PERL script using simple regular expression functionality implemented in the standard UNIX tool GREP. This provides us with the necessary robustness for parsing COBOL.

Note that there are typically two usage modes of COBOL programs on mainframe, namely *online* and *in batch*. In online mode programs interact via COBOL calls, in batch these programs can also be invoked from mainframe scripts, usually JCL[2]. We have not taken into account the JCL artefacts, thus we are missing some, perhaps important, information.

3.2 Structural Properties

Using the same PERL script, we extract the following dependencies from these source code artefacts.
- A *functional dependency* is a relationship between two programs implemented by a CALL statement in the first program referencing the second program. This dependency has a weight associated with it equalling the number of CALL statements from the first to the second program.
- A *data dependency* is a relationship between a program and a copybook implemented by a COPY statement in the program referencing the copybook. The same dependency can occur between two copybooks.

These functional dependencies are interesting because they provide a measure for the ease of separating programs: a group of programs sharing a lot of functional dependencies will be harder to separate than a group of programs sharing little or no dependencies. Furthermore, strongly interdependent programs might be working together to implement a certain functionality. In the same way, data dependencies might reveal groups of programs working on the same data.

Besides these dependencies, we also define metrics on a COBOL program.
- The *number of incoming calls* (NIC) measures the total number of CALL statements referencing that program.
- The *number of outgoing calls* (NOC) measures the total number of CALL statements within that program to any other program.

We study these metrics because they are an indicator for the types of usage of COBOL programs[3]. Programs with a high NIC can be seen as *suppliers* as they supply a functionality that is popular among other programs. As a special case of suppliers, programs with a high NIC and a NOC of zero can be seen as *libraries* as they are likely to supply core functionalities since they do not rely on other programs. Programs with a high NOC, on the other hand, can be seen as *clients* as they use a lot of functionality from other programs. Putting these concepts in a

[2] JCL stands for Job Control Language.

[3] The terminology of *suppliers* , *clients* and *libraries* has been adopted from [MM06].

service-oriented mindset, suppliers (and libraries) might provide good starting points for services whereas clients will most likely be service consumers.

4 Views

We use a visual representation of this information as it provides the ability to comprehend large amounts of data [War00]. More specifically, directed graphs have been shown to be natural representations of software systems. Enriching these entities with metrics information provides valuable visual clues for early structure comprehension [LD03].

To obtain such visualisations we export the structural information to GDF (Guess Data Format) after which it can be visualised in GUESS [Ada06], a flexible graph exploration tool. This environment assists a user in exploring graphs by providing capabilities such as applying graph layouts, highlighting, zooming, moving and filtering.

In what follows we define two views based on the structural relations defined in Subsection 3.2, namely the Functional Dependency View and the Data Dependency View. For each view we identified some visual indicators of structural properties and grouped them according to a specific task. We motivate the task, present visual indicators and interpret them in light of the specific task.

4.1 Functional Dependency View

The building blocks of this view are COBOL programs and the functional dependencies between them. To distinguish programs from missing programs we use *white* rectangles and *black* rectangles respectively. A functional dependency between two programs is implemented as a directed edge between two nodes. Furthermore, the height and width of the nodes are relative to the NIC and NOC metrics of the corresponding program respectively. We use the Graph EMbedder (GEM) algorithm [FLM94] as a layout algorithm because it tries to minimise the sum of the edge lengths in order to provide an easy to interpret layout. This layout positions highly connected nodes closer together thereby providing a visual indication of possible clusters and groups.

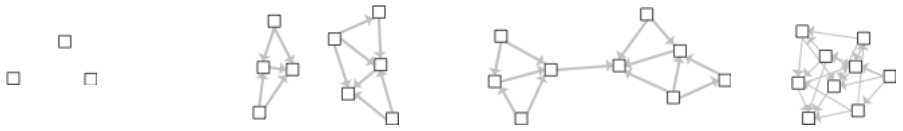
4.1.1 Overall Design

Intent — Get a feel for the overall design of the system.

Motivation — Functional dependencies reveal first hand information on the complexity of the design. Identifying strongly interdependent groups of programs can reveal opportunities or risks early in the project.

Visual Indicators (Figure 1) — Isolated nodes (a) have no edges, whereas a monolith (d) has an abundance of edges between all nodes. Disconnected clusters (b) are isolated groups of interconnected nodes, whereas with connected clusters (c) the groups are connected by few edges.

Interpretation — Assuming that functional dependencies are the only way to access a program,



(a) Isolated nodes (b) Disconnected Clusters (c) Connected Clusters (d) Monolith

Figure 1: Overall Design Indicators

isolated nodes and small disconnected clusters would reveal dead code. On the other hand, if there are other functional dependencies which are not part of our model, these nodes might reveal clearly separated functional blocks. Connected but loosely coupled clusters of nodes indicate functional blocks that are easy to separate.

4.1.2 Exceptional Entities

Intent — Locate clients, suppliers and libraries.

Motivation — Easily detecting exceptional entities helps in quickly gaining focus in the usual abundance of information.

Visual Indicators (Figure 2) — Clients are visually represented as very wide nodes (a), usually with many outgoing edges, whereas suppliers can easily be identified as very tall nodes (b), usually with many incoming edges. A combination of both results in a tall and wide node (c). Libraries are a special form of supplier as they have no outgoing edges (d).

(a) Client (b) Supplier (c) Client & Supplier (d) Library

Figure 2: Exceptional Entities

Interpretation — *Suppliers* and *libraries*, both characterised by a high NIC, indicate a popular functionality in the system. In the best case it could be an interface into an important functionality, although it might as well be a smell indicating bad separation of concerns. *Clients* (characterised by a high NOC), indicate more high level functionalities. Typically, (graphical) user interfaces exhibit such characteristics. A combination of both *client and supplier* possibly indicates an interface into lower level functionality (thus, making it a supplier) but delegating the real work to the core programs (thus, making it a client). A *library* is a *lowest level* functionality as it does not delegate anything, thus indicating a core functionality of the system.

4.2 Data Dependency View

The building blocks of this view are COBOL programs, (missing) COBOL copybooks and the data dependencies between them. To distinguish programs from copybooks we use the *rectangular* and *circular* shape respectively. The distinction between copybooks and missing copybooks

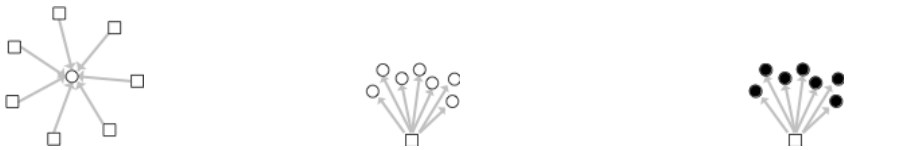
is implemented using *white* and *black* nodes respectively. A data dependency between a program and a copybook or between two copybooks is implemented as a directed edge between two nodes. Note that we do not show missing programs in this view, as they will never contain any data dependencies. As with the Functional Dependency View, we also use the GEM layout for positioning the nodes. Also note that the colour and the dimensions of a program node are intentionally kept in accordance with the Functional Dependency View to easily identify the same programs in the two views.

4.2.1 Data Usage

Intent — Get a feel for the data usage between different programs.

Motivation — Especially in data driven environments, data dependencies can provide an entry point into the organisational structure of a company. Quickly identifying common data dependencies or facades clearly separating data usage can provide necessary focus in the remainder of the project.

Visual Indicators (Figure 3) — A common data concept is visually represented as a group of programs directly connected to the same copybook (a). A data facade is represented as a program connected to a set of copybooks which are only connected to that program. If the copybooks are mainly missing copybooks we call it an external data facade (c), otherwise it is an internal data facade (b).



(a) Common Data (b) Internal Data Facade (c) External Data Facade

Figure 3: Data Usage

Interpretation — As opposed to grouping programs by shared functional dependencies (Subsubsection 4.1.1), we can also group them according to shared data dependencies. Programs sharing data are likely to work on a similar concept, especially when the data represents a business concept. On the other hand, programs *hogging* data for themselves are likely to encapsulate certain data and the corresponding functionality, making them more easy to separate.

5 Experience Report

This section describes some preliminary results we obtained during an ongoing case study at a Belgian insurance company. The system under study is a document generation and management system consisting of 200k lines of COBOL code with embedded SQL (including whitespace and comments), divided over 401 COBOL programs and 336 COBOL copybooks, and 38k lines of JCL (from which we did not extract any information). Extraction of the COBOL artefacts and creation of the data model, as described in Section 3, was completed in less than five seconds

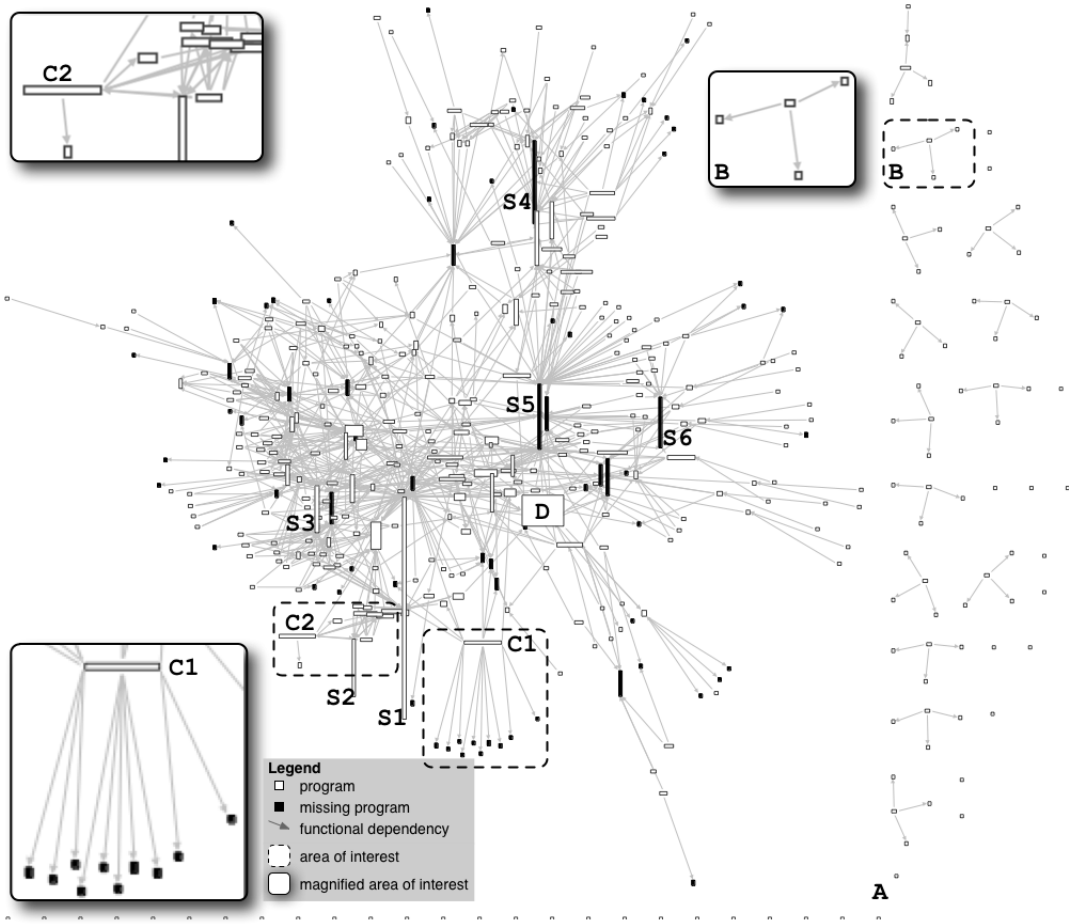


Figure 4: Functional Dependency View, height and width of the nodes represent NIC and NOC respectively.

indicating that the approach is indeed very lightweight and likely to scale well on even bigger systems.

Figure 4 depicts the Functional Dependency View of the system. The first thing you notice is the typical monolithic nature of the legacy system. Almost everything seems to be interconnected with no apparent order or hierarchy. There are some isolated nodes on the bottom and the right (A) that would seem to constitute dead code, as they have no functional dependencies. Although closer investigation did reveal some empty files, not all nodes are by definition *dead code* as they can be called directly from JCL scripts running on mainframe. The small disconnected clusters on the right show a distinct pattern (B): one parent node functionally depending on three child nodes. Closer investigation revealed that each cluster is responsible for accessing and modifying a specific part of the *documents database*. Each group of three child nodes is responsible for respectively inserting, deleting and updating corresponding records.

When looking for exceptional entities, the most pertinent supplier is the tallest *white* node (S1). Closer investigation revealed that this program is indeed a supplier as it provides the interfacing program into all the disconnected clusters on the right. It collects all user and application requests for adjusting the documents database and forwards those request to the correct cluster. Although one would expect a functional dependency between this supplier and the clusters, this is not

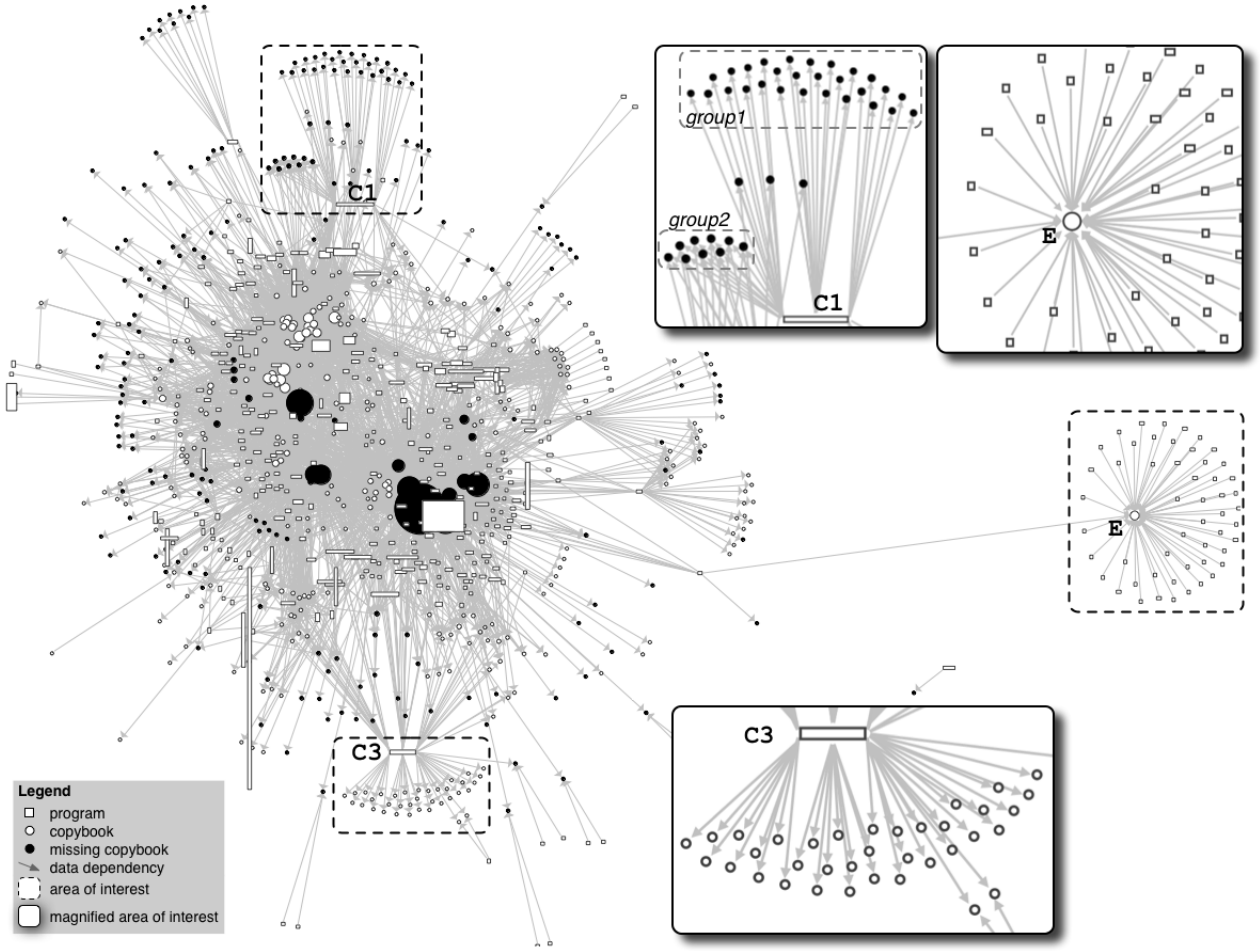


Figure 5: Data Dependency View.

the case because processing the request is performed asynchronously (in batch). Therefore, the functional dependency is not visible in the COBOL files, but rather in JCL scripts running on mainframe. Other suppliers include a program for visualising errors (S2) and an interface into other databases (S3). Besides the *white* suppliers there are also missing programs (*black* nodes) that are classified as suppliers. The tallest of them (S4), for example, provides an interface into core mainframe functionality. Many other tall black nodes (e.g., S5, S6) are core programs of the organisation.

Besides the very pertinent suppliers, Figure 4 also contains two clients that stand out. The first one (C1), connected to several *missing* programs, is responsible for automatically filling in documents with information from different databases outside the system under study. This program is classified as a client as it seems to be extensively using data services of other systems. Another client (C2) is responsible for checking the correctness of documents. Although, it delegates its functionalities to three child nodes, the main contribution for classifying it as a client are numerous calls to a module that visualises error messages (S2).

One last node that really stands out is the big *white* rectangle (D) classified as both a client and a supplier. It is the interface into the core of the system under study. It is a supplier for all the *end programs* that feed (graphical) user interface, and wide because it delegates a lot of its

responsibilities to lower level suppliers.

Figure 5 depicts the Data Dependency View of the system. Besides the monolithic structure on the left, there is one group of nodes that is clearly separated from the rest. This cluster reveals one data concept (E) used by several programs. Closer investigation revealed that the programs in this cluster are the same programs that are responsible for managing the documents database (the disconnected clusters in Figure 4). The central copybook has the capability of uniquely defining one row within this entire documents database. So this database management responsibility is not only functionally clearly separated from the rest of the system, but also with regard to the data dependencies. When consulting the domain experts, they supported this observation and explained that this is actually a subsystem used by other systems as well. For historical reasons it is part of this bigger legacy system, but it was designed to be used as a separate system.

Client C3 has many data dependencies with copybooks that have no dependencies with other programs, therefore it acts as an internal data facade. Closer investigation revealed that C3 is the only program responsible for (re)creating and correctly formatting all types of documents. Client program C1, on the other hand, has a lot of data dependencies with *missing* copybooks (the *black* nodes above). This is the program that retrieves information from other systems to automatically fill in documents and was characterised by a lot of functional dependencies with missing programs in Figure 4. This link between functional and data dependencies over the two views is not surprising since data definitions are necessary to communicate with these programs. When looking at *group1* of copybooks, C1 acts like a data facade for the external data. The copybooks from *group2*, on the other hand, are also used by other programs, thereby apparently violating the facade property. Closer investigation revealed that the shared copybooks (*group2*) are necessary for creating the terminal screens, indicating that C1 not only implements business logic on top of external data, but also its GUI functionality.

6 Room for Improvement

Both views suffer from unnecessary cluttering. We will try to eliminate this by identifying and removing *omnipresent* programs [MM06]. Furthermore, some of the programs classified as *missing programs* result from unresolved dynamic calls rather than really being a program outside the system scope. We will try to resolve them if possible (using static techniques), otherwise we will remove them from the model.

After uncluttering the views, we believe more meaningful visual patterns will become apparent. Also, the synergy between the two views (e.g., the link between depending on missing programs and using missing copybooks) is something we would like to make more explicit, maybe by using a clearly focussed combined view.

Furthermore, our model is far from complete, therefore we would like to take other source code artefacts into account as well. For example, the functional dependencies resulting from the JCL scripts or the data dependencies implemented by global data accesses. Also, the data dependencies are only extracted from the copybooks. However, there is no guarantee that each copybook is uniquely mapped to a database table, thus, several copybooks could point to the same data. This can be solved by explicitly mapping each copybook to its originating source.

Finally, we would like to merge our scripts with FETCH [DV07] — an open-source fact ex-

traction tool chain currently geared towards C(++) but easily allowing extension via its *pipe and filter* architecture — as we can then take advantage of more advanced querying mechanisms and and a collection of views already available in FETCH.

7 Related Work

We based our views on the ideas of Lanza's polymetric views [LD03], aiming at both a lightweight and a visual approach. Although the views as proposed by Lanza are independent from the implementation language, they are mainly targeted at the object oriented paradigm. We apply this polymetric view concept to COBOL and target our views to anlysing the reuse potential in a service oriented context.

Nyáry et. al. [NPHK05] support the maintenance of legacy COBOL applications by setting up a COBOL repository. While using much of the same artefacts and dependencies as we use, they go into more COBOL detail (including file references and data fields). Their tool focuses on the repository with simultaneous code browsing resulting in low level abstractions, whereas we aim to create more high level abstractions and uncover visual patterns.

Another tool capable of reverse engineering COBOL code is RIGI [MTW93, WTMS95]. They also present graph based visualisations of the system. But while they aim at more general program comprehension activities, our approach is specifically targeted to finding opportunities and risks for migrating to SOA. Furthermore, RIGI has a finer-grained approach to extracting COBOL artefacts, making our approach more lightweight.

O'Brien et. al. [OSL05] support migration to services using software architecture reconstruction techniques. While they also visualise functional and data dependencies (using their tool ARMIN), they perform these analyses mainly on object-oriented systems and not on legacy COBOL systems.

8 Conclusion

While enterprises are turning to Service Oriented Architectures for aligning their IT portfolio with their business, migrating the current systems to such an architecture is not trivial. Therefore a stepwise migration is necessary. This research constitutes the initial steps for investigating COBOL systems in the early stage of such a migration project.

In preparation of talking to domain architects, it is important for a reengineer to quickly gain an understanding of the COBOL source code structure. The explorative views presented in this paper are a first contribution towards this goal as they aim at identifying areas obstructing or facilitating migration to SOA. We already described some noticeable phenomena and use them to make concrete observations. In the future we will conduct more empirical studies to evaluate this approach.

The logical next step in the migration process would be to inject domain and organisational knowledge into the views. This way we can see to what degree the structure of the existing systems corresponds to the target SOA state as proposed by the domain experts.

Acknowledgements: This work has been carried out in the context of the '*Migration to Service Ori-*

ented Architectures' project sponsored by *AXA Belgium NV* and *KBC Group NV*.

References

[AAB+00] E. Arranga, I. Archbell, J. Bradley, P. Coker, R. Langer, C. Townsend, M. Weathley. In Cobol's Defense. *IEEE Software* 17(2):70–72,75, 2000. doi:10.1109/MS.2000.10014

[Ada06] E. Adar. GUESS: a language and interface for graph exploration. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*. Pp. 791–800. ACM Press, New York, NY, USA, 2006. doi:10.1145/1124772.1124889

[DDN02] S. Demeyer, S. Ducasse, O. Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[DV07] B. Du Bois, B. Van Rompaey. Supporting Reengineering Scenarios with FETCH: an Experience Report. In *Third International ERCIM Symposium on Software Evolution*. October 2007. to appear.

[FLM94] A. Frick, A. Ludwig, H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. In *GD '94: Proceedings of the DIMACS International Workshop on Graph Drawing*. Pp. 388–403. Springer-Verlag, London, UK, 1994.

[KBS04] D. Krafzig, K. Banke, D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[LD03] M. Lanza, S. Ducasse. Polymetric Views-A Lightweight Visual Approach to Reverse Engineering. *IEEE Trans. Softw. Eng.* 29(9):782–795, 2003. doi:10.1109/TSE.2003.1232284

[LMS06] G. Lewis, E. Morris, D. Smith. Analyzing the Reuse Potential of Migrating Legacy Components to a Service-Oriented Architecture. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*. Pp. 15–23. IEEE Computer Society, Washington, DC, USA, 2006. doi:10.1109/CSMR.2006.9

[MM06] B. S. Mitchell, S. Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Trans. Softw. Eng.* 32(3):193–208, 2006. Brian S. Mitchell and Spiros Mancoridis. doi:10.1109/TSE.2006.31

[MTW93] H. A. Müller, S. R. Tilley, K. Wong. Understanding software systems using reverse engineering technology perspectives from the Rigi project. In *CASCON '93:*

Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research. Pp. 217–226. IBM Press, 1993.

[NPHK05] E. Nyáry, G. Pap, M. Herczegh, Z. Kolonits. Supporting the Maintenance of legacy COBOL Applications with Tools for Repository Management and Viewing. In *ICSM (Industrial and Tool Volume)*. Pp. 5–10. 2005.

[OAS06] OASIS Consortium. Reference Model for Service Oriented Architecture 1.0. July 2006.
http://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf

[OSL05] L. O'Brien, D. Smith, G. Lewis. Supporting Migration to Services using Software Architecture Reconstruction. *step* 0:81–91, 2005.
doi:10.1109/STEP.2005.29

[SFM97] M.-A. D. Storey, F. D. Fracchia, H. A. Mueller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization. In *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension*. P. 17. IEEE Computer Society, Washington, DC, USA, 1997.
doi:10.1109/WPC.1997.601257

[SR06] J. Spohrer, D. Riecken. Services science. *Commun. ACM* 49(7), July 2006.

[War00] C. Ware. *Information visualization: perception for design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[WTMS95] K. Wong, S. R. Tilley, H. A. Muller, M.-A. D. Storey. Structural Redocumentation: A Case Study. *IEEE Software* 12(1):46–54, 1995.
doi:10.1109/52.363166