A Pilot Project for migrating COBOL Code to Web Services

by Harry M. Sneed
ANECON GmbH, Vienna
University of Regensburg
harry.sneed@T-Online.de

Abstract: *This paper describes a pilot project conducted to test the feasibility of constructing web services from existing mainframe COBOL programs. The project involved the use of four tools. The first tool* - ***COBAudit*** - *was intended to identify candidates for web services. The second tool* - ***COBStrip*** - *served to extract only that portion of the code required to fulfill the service. The third tool* - ***COBWrap*** *– wrapped the code extracted from original code and converted it to an executable component. The fourth tool –* ***COBLink*** *– connected the wrapped component to the web by generating a WSDL interface from either the COBOL linkage section or from the original map definition. The tools were applied to a legacy life insurance system with more than 20 million lines of COBOL code running under IMS on the IBM mainframe.*
Keywords: *Web services, Migration, COBOL, Code Stripping, Wrapping, Legacy Systems.*

1. Rationale for the project

Currently, many user organizations, especially in the traditional IBM mainframe market, are trapped in their old legacy systems. Over the years these systems have continued to grow and now have such dimensions that render them practically impossible to replace. To redevelop them would drive the owner company into bankruptcy. Alone in Germany there are some 240 billion lines of COBOL code in such legacy systems [CW08]. To redevelop them would cost more than the annual state budget. Even to reengineer them would cost at a rate of 4 Euros per Line close to one trillion Euros. To redevelop them would require that their requirements be completely recovered and that is currently beyond the state of the art. There is at present no technology available for automatically recovering requirements from legacy code. The architecture may be recoverable but not the content. That leaves the IT department with the burden of having to recollect the requirements from the end users, who in the meantime have long forgotten what they originally wanted.

To migrate these systems, for instance from COBOL to Java, is out of the question due to the high costs and risks involved. There are tools available for converting COBOL to Java but none of them have ever proved themselves in practice. The Moore tool set worked in the laboratory for a few selected COBOL-85 programs, but it never passed the practice test [FRS94.] The tool set from Fantechi and Nesi at the University of Florence – C2O2 - never even got out of the laboratory [FaNe97]. This author worked on the problem for several years and had a working prototype – COBTran – which produced executable OO-COBOL programs from COBOL-85. However this proved to be a dead end since Object-COBOL was never accepted by the market [Sned99]. Triangle Park Technologies worked together with the St. Petersburg State University to develop a tool set known as RescueWare for migrating COBOL and PL/I programs to Java [Tere02]. There was some initial success with this tool set, but for some reason the development was discontinued. Current efforts to migrate legacy languages to Java are focused on a statement to statement conversion so that the original structure of the code remains [CTM08]. What comes out is a procedural program in Java syntax. Such tools can generate a method to emulate every statement type of the original language, but the result is far from being object-oriented. Because of these inherent difficulties of language conversion, in the past years the focus of migration has moved from transformation to integration. [DeLu08]

That leaves the question open as to how to proceed with legacy systems. If they are small enough they can be redeveloped. If there is no need to ever change them, they can be emulated in a modern Java environment. Otherwise it is hard to say what to do with them. Once such systems have surpassed a given size of about 10 million statements there is little an organization can do to salvage them without a tremendous investment, an investment that most IT user departments are not willing to finance. Wrapping and integrating is a solution which promises to be cheap and quick.

To redevelop the system under discussion, it was estimated with several methods including COCOMO, Function-Point and Data-Point that the redevelopment would cost at least 2000 person years. Migrating the system would cost some 500 person years. Just maintaining the status quo of such a monster system with a 9% annual change rate requires more than 200 programmers. Especially in the financial service world laws and regulations are changing at an increasingly rapid rate. The Basel-II agreement is typical of the changes which have been imposed upon the European financial institutions. To comply with the new regulations large investments are required at a time when funds are scarce. This leaves little resources for redeveloping or migrating existing applications. If anything is done at all, it must be done cheaply. The preferred solution is here as elsewhere in industry a readymade standard solution. However, for the life insurance business there is as yet no such solution in sight.

On the other hand, competition is increasing. Big financial service providers with a large customer base are under tremendous pressure. There is a great need to provide Internet services both to the sales representatives in the field as well to the customers themselves. This means providing web access to the existing systems. Furthermore, to cut costs and reduce personnel it is imperative to restructure and streamline the existing business processes. Neither objective can be attained without adapting the underlying application systems.

Thus, there are two good reasons for wanting to open up existing legacy systems and to offer them as web services. One is to provide Internet access to certain online transactions. The other is to render it possible to integrate existing programs as steps in new business processes. The business processes must be revised in order to satisfy customer demands. There is, therefore, a definite need on the part of large financial service providers to move toward a service-oriented architecture, as it would mean cutting costs by eliminating unnecessary redundancy while at the same time increasing the quality of the service, but this move can only be done gradually without disrupting the continuity of ongoing IT operations and it must be done with the existing software. Those restrictions pose a great challenge to software reengineering technology. [SPL2003]

2. Migration or Integration

It should be pointed out that what is being described here is not really migration in the original sense of the word. The word “migration” has until now been used to describe the moving of a software system from one environment to another. It might mean porting to another platform, transforming the code from one language to another, transferring the data from one database to another or moving everything including programs, data and user interfaces over to a new operating system or middleware [Horo98].

Integration, is on the other hand, the use of existing software components in a new environment without actually physically moving them into that environment. They remain in their original environment and are only called upon to remotely provide a given functionality. The function results are integrated via the network. Mashups are a good example of integration [Zou07].

In the literature many authors use the word migration when what they are really talking about is integration [CFF06]. Such is not the case here. In this pilot project the databases are not affected. The functionality remains as it is. The code is being modified but not converted. Nothing is being ported. The programs continue to run in the IMS environment on the IBM mainframe. They are only exchanging data with the frontend via the company intranet. Therefore, this is not a migration in the true sense of that word.

What is being described here can best be termed as “integration”. The existing mainframe transactions are being integrated into a service-oriented architecture. Instead of interacting directly with human users via MFS masks on a 3270 terminal, they are interacting with web clients via a WSDL interface. These web clients may or may not be interacting with human users. They are nodes in a business process network. Consequently the word integration fits much better to the task at hand. [Sned06]

3. Previous Work in this Area

It goes without saying that this is not the first project of this type. The vision of a Service-Oriented Architecture is like a fata morgana for many over burdened IT organizations. For many it seems to be the only hope of breaking out of the legacy trap in which they find themselves. Commercial vendors and consultants keen on exploiting the opportunity are feeding this hope [KBS04]. The SOA vision has also provoked some serious work on moving from client/server systems to web-based systems. There have been a number of research projects aimed at migrating existing applications to web services. Leading in this field are the universities of Benevento and the RCOST institute in Southern Italy, the universities of Victoria and Waterloo in Canada and the Software Engineering Institute (SEI) in the U.S.A [KLS08].

As early as 2001, researchers at the RCOST Institute in Italy have begun with pilot projects

aimed at converting COBOL applications into Web applications [ACde01]. Their work lead to the migration of the first local government systems by replacing conventional user interfaces with web pages [BGT02] [BGT03]. Across the Atlantic researchers at the University of Victoria have been working for many years developing strategies for implementing web services, in particular from existing systems [TDH04]. In the meantime the Software Engineering Institute has picked up the subject and set up a research community focused on the evolution of service-oriented architectures [KLS07] to which this author has also contributed [Sned07]. Thus, it cannot be said that there is a lack of research in this field. What is lacking is large scale industrial applications. The theoretical approaches developed in the laboratories have to be applied in industry in order to prove their viability. The work described in this paper is a step in that direction.

4. Goals of the pilot project

Before launching a major migration project this user wanted to have a proof of concept, i.e. a study on the feasibility of migrating individual business functions embedded in legacy programs to a service-oriented architecture. Like many users, this user was keen on introducing a modern architecture, but without having to redevelop all of the contents. The exact goals of the study were:

1) to measure the sizes, complexities and qualities of the candidate IMS/COBOL programs in order to assess the extend of the work to be done.
2) to access the feasibility of reusing the candidate components as web services.
3) to determine if it is possible to extract slices of code from existing COBOL-IMS-DC programs for reuse as web services.
4) to demonstrate that the selected programs can be wrapped to work without the IMS-DC TP monitor.
5) to test whether the wrapped components can be accessed as web services within the IBM WebSphere environment.

For the sake of the feasibility study five programs were selected on the basis of their business content and their degree of reusability. It was decided not to deal with the non reusable programs as it would have cost too much to reengineer them. There were tools available to automatically reengineer them – COBRedo – but then the reengineered programs would have to have been retested before migrating them. This would have delayed the pilot project and caused additional costs.

The intention was to gain enough information for top management to decide whether this was a viable and affordable approach for migrating to a service-oriented architecture or not. The only way to judge such an approach was to try it out.

4.1 Measuring the candidate code

The first goal to be attained was to measure the existing software in terms of its size, complexity and quality. All together 7,297 COBOL programs were processed by the COBAudit tool. The size measurements taken were, among others, the lines of code, the number of statements, the number of data structures and data elements, the number of file and data base accesses, the number of decisions, the number of subroutines and the number of subroutine calls. In all 56 different size measurements were taken.

The program complexity was measured in terms of eight complexity metrics, Chapin's data complexity, Elshof's data flow complexity, Card's data access complexity, Henry's interface complexity, McCabe's control flow complexity, McClure's decisional complexity, Sneed's branching complexity and Halstead's language complexity. These complexity metrics have been described in detail in previous papers [Sned95]. Complexity in software is a question of the relation between software elements and their relationships to one another. The more relationships there are relative to the number of elements, the higher the complexity. The McCabe metric is measuring the relation between edges and nodes of a graph, the Henry metric is measuring the relation of module interactions to the number of modules and the Halstead metric is measuring the relation between operands and operators on the one side and references to them on the other. From this point of view, complexity of software can be well defined and easily measured [Sned08].

With the SoftAudit tool all complexity measurements are normalized to a rational scale, i.e. they are expressed as a coefficient on a scale of 0 to 1 with 0.5 being the median complexity. Being over 0.5 indicates that this aspect of the program is overly complex. Being under 0.5 indicates that the complexity is not a problem. By inverting the scale for quality originally proposed in the ISO-9126 standard for product assessment, 0 to 0.4 indicates low complexity, 0.4 to 0.6 indicates average complexity, 0.6 to 0.8 indicates high complexity and over 0.8 indicates that the code is overly complex [ISO93].

The following complexity measures represent the average complexities of the COBOL application system under consideration. They indicate that the data flow and data access complexities are very high as a result of the many dependencies on the underlying data bases. The control flow complexity is also high because of the many GOTO branches within the code. Interface complexity is low since

there are few direct interactions between programs. They are linked via the databases. The language complexity is low because the same operators and operators are used over and over again. Decisional complexity is also low because the business rules applied are actually very simple. This can be considered a typical complexity profile of an average legacy business application working with a relational database.

DATA COMPLEXITY	0.524
DATA FLOW COMPLEXITY	0.768
DATA ACCESS COMPLEXITY	0.900
INTERFACE COMPLEXITY	0.125
CONTROL FLOW COMPLEXITY	0.678
DECISIONAL COMPLEXITY	0.362
BRANCHING COMPLEXITY	0.578
LANGUAGE COMPLEXITY	0.215
AVERAGE PROGRAM COMPLEXITY	0.518

The program quality was measured in terms of the quality characteristics modularity, portability, reusability, convertibility, flexibility, testability, conformity and maintainability. These quality metrics too have been described in previous papers [Sned08]. Judging the quality of a software system depends very much on the goals one is striving for. Of particular importance for the sake of reuse as a web service are the qualities modularity, reusability and flexibility. Modularity is defined in terms of high cohesion and low coupling. The fewer dependencies there are between the individual program parts, the easier it is to extract them. The reusability metric has been the subject of a special paper on reuse measurement. It is concerned with the self containment of the program parts and their independence from the environment. Reusable code blocks should contain no IO operations and no direct branches into other blocks [Sned98]. Finally flexibility is an indicator of data independence. The code should be void of hard coded data to allow it to be used in a different context.

The following quality measures are representative of legacy mainframe applications. They indicate that the programs are inflexible because of their high use of hard coded data. The reusability is low because of the many interconnections between code blocks within the modules and because of the high usage of global data. Modularity and testability are also below average as a result of the large size of the modules.

MODULARITY	0.498
PORTABILITY	0.668
FLEXIBILITY	0.100
CONFORMITY	0.774
TESTABILITY	0.498
CONVERTIBILITY	0.821
REUSABILITY	0.150
MAINTAINABILITY	0.464
AVERAGE PROGRAM QUALITY	0.448

The measurement of the code was intended to indicate which programs would be problematic when it came to wrapping them and what would be the cost of the wrapping project as a whole. The problematic programs became candidates for reengineering. The costs of wrapping is dependent on the size of the programs, their data access and interface complexity as well as on their modularity, reusability and flexibility.

This step was fully automated. The input was the original legacy code, the output was the metric reports and the code deficiency lists.

4.2 Assessing the Reusability

Once the code had been measured it could then be accessed whether it was feasible to reuse the programs as web services or not. As pointed out above, the key issues here were modularity, data independence and self containment. Since much of the code was redundant, there was no reason to include it in the wrapped service. It could be commented out. But this also meant that any branches or performs to that portion of the code had to be capped. So, the fewer there were the better.

It was possible to see on hand of the metrics if a program was reusable or not. If there were mostly self contained sections and paragraphs invoked via a central control unit, then these code blocks could be reused in another context. Also, if the program had only one entry with a limited number of input parameters it could be reused as a whole. Of vital importance for the online transaction programs was that each program processed only one map. The one map could be emulated via a single wrapper. If multiple maps were processed, then several wrappers would be required.

The reusability assessment was intended to
a.) select programs for reuse and
b.) gain an insight into what portion of the programs could be readily wrapped without intensive rework.

As it turned out only 2,863 of the 7,297 programs analyzed were really reusable. That is less than 40%. The remainder of the programs were either too intertwined or too dependant on global data. That indicates that legacy code may not be such a good source for obtaining web services after all. It would require a large scale reengineering of the code to make it more reusable, something most users are not prepared to finance.

Normally, if the portion of reusable programs is too low, the project should be abandoned and another approach taken. An alternative strategy would be to

first conduct a reengineering project to put the programs in a state where they could be reused. However since this was only a pilot study, it was decided to go ahead and process those few programs which were structured and modular enough to be reused.

The decision as to whether a program is reusable or not was based first on a manual assessment of the metrics, in particular the modularity and reusability metrics. The next step was to look at the program code itself to detect whether the code blocks could be easily separated from another. A final selection step was to decide whether the program contained a function worthy of being turned into a web service. If these criteria were all fulfilled, the program was selected. This reusability assessment was supported by the tool SoftRedoc which documents both the data and the functional dependencies that COBOL code blocks have among each other. Of course the decision as to how to classify a program was made manually. (see Figure 1)

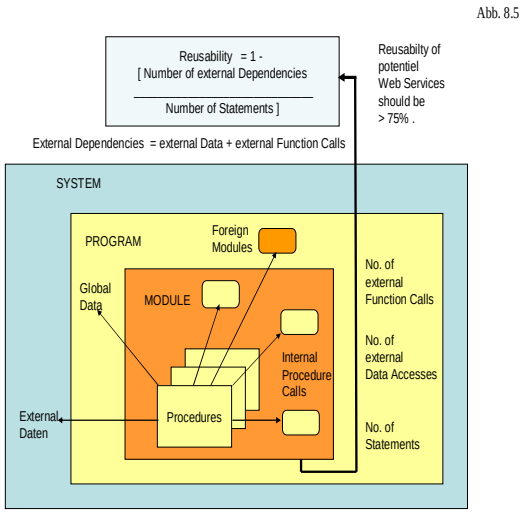

Fig. 1: Evaluating Reusability of Code

4.3 Stripping the Code

Code stripping is a technique used to select given paths through a program by blending out all of the data and statements not used. It was originally used for testing purposes. The idea was to test one path through a program at a time and not to be concerned with the rest. The code was submitted to an automatic slicing machine which left the slice selected as it was, while commenting out the rest. Then, the module was compiled and tested for that one path. This was then repeated for each path until all control paths had been tested. This technique was refined and used in a European research project - TRUST - for testing embedded, realtime software, where it was not possible to instrument the code [PuSn89]. However, the technique can also be applied to business systems as well to isolate strips of code to be extracted.

In the context of wrapping, the technique of code stripping is used to generate multiple instances of the same program. Each instance can be a separate web service. An instance corresponds to a particular business rule. In the past, for efficiency reasons, programs were written to fulfill several business rules at one time. This led to the intertwining of business functions with one another. This, among other things, is one of the main deterrents to reusing existing programs as web services, since a web service should correspond to one and only one business rule, so that the sequence and combination of business rules can be determined in the business process which uses the services. This is after all, one of the major goals of a service-oriented architecture [BiKw98]. Business processes should be able to arbitrarily combine individual web services to meet different requirements.

To determine if this was possible sample complex programs had to be selected and run through the stripping machine – COBStrip. COBStrip requires some human interaction. The user must mark which results he wishes to obtain from the program. For this he is given a view of the output data in the Data Division. For online programs this output data is normally to be found in the data structure corresponding to the map of the user interface. In AS400 programs this is the screen section. For CICS programs it is the Basic Map Service data definition. For these programs it was the Message Format Service map together with the map attribute bytes. In the case of batch programs, the structure of the output files is displayed.

By means of data slicing, the tool locates all COBOL paragraphs required to produces those results selected by the user. These include not only the paragraphs or blocks of code where the map fields are set, but also those paragraphs which produce intermediate results used by the paragraphs that set the map fields. The intermediate results are stored in data tables where it is possible to trace the final result selected back through the intermediate results to the original input data by means of cascading. At the other end of the data flow are those paragraphs which receive and check the input maps. Very often output map fields are set from database contents. So those paragraphs which access the database also have to be included. What is left of the COBOL code is placed in comments so as not to be compiled..

What remains is a partial procedure division consisting of selected code blocks and a data division containing only those data structures processed by the selected paragraphs. It most cases this amounted to less than 1/3 of the original code.

This was then compiled as a separate stand alone module. (see Figure 2)

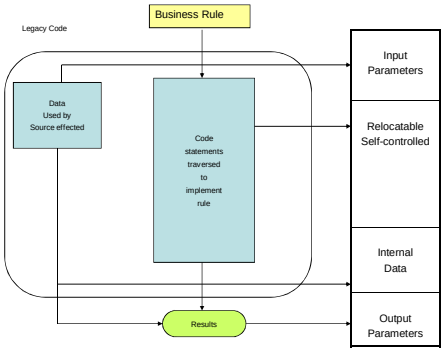

Fig. 2: Selecting Code to be stripped

If the user wanted to extract several different business rules from one and the same COBOL program, then several passes were required, one for each rule resulting in a different version of that program for each potential web service. In the case of particularly large programs as many as 4 different web services could be extracted from one source code member, one for each desired result set or service response.

The manual effort involved in this step was the selection of the desired results. This was done by displaying the output data structures on the screen and allowing the user to mark them as depicted in the following screen shot.

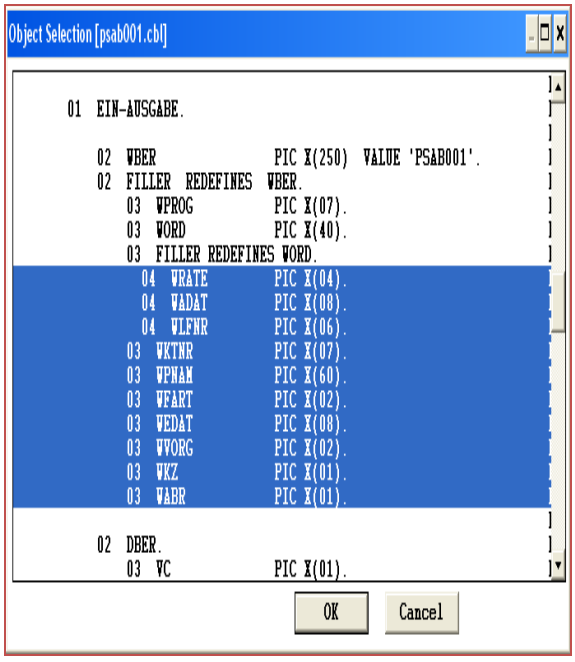

Fig. 3: Marking the desired Results

4.4 Wrapping the Code

Once the stripped versions of the existing COBOL programs were available and compiled, the next step was to wrap them behind a WSDL interface. For this two tools are required: COBWrap and COBLink.

COBWrap processes a stripped module to replace the terminal input/output operations with calls to a wrapper module and to move the input/output data from the Linkage section to the Working-Storage section. In the case of IMS the input maps are received via a call to the IMS transaction monitor:

CALL ‘CBLTDLI’ USING PARAM-NR, IO-PCB, INPUT-MAP.

The same type of call is used to send the output map to the terminal.

CALL ‘CBLTDLI’ USING PARAM-NR, IO-PCB, OUTPUT-MAP.

COBWrap simply places the original call in comments and inserts another call to the generated wrapper module behind it.

CALL ‘CBL2WSDL’ USING PARAM-NR, IO-PCB, INPUT-MAP.

The data in the Linkage-Section of CICS programs is moved to the Working-Storage Section as CICS programs are actually subprograms of the CICS TP-Monitor. In the case of IMS-DC the opposite is true. The COBOL program is the main program and the IMS-DC monitor is implemented as a subprogram to handle the data flow to and from the user terminal. Therefore, IMS-DC programs, such as those here, are much easier to wrap. It is only a question of replacing the IMS-DC calls with calls to the wrapper. The author has already reported on a similar project to wrap Assembler-IMS programs in an earlier project in 1997 [SnMa98]

This step was fully automated. There was no need for any manual intervention since the wrapping is based on an analysis of the program interfaces and these are as IMS macros readily recognizable.

4.5 Linking the wrapped Services

The final step of the extraction process was to connect the wrapped components to the Internet via the Websphere middleware. This is where the tool COBLink comes into play. COBLink generates two wrapper modules to be linked to the wrapped program. The input to the generation is the source code of the altered COBOL program. Based on the declarations of the input parameters, COBLink creates a WSDL schema for the web service request and at the same time generates a COBOL module for translating that request into the input parameters of the server program. In a second run, COBLink creates another WDSL schema for the web service

response while, at the same time, generating a COBOL module for transferring the output parameters of the server program into the WSDL response. (see Figure 4)

```
<XSDCOB:complexType type = "#params" name = "PSABO01V-PARAMS"
          content = "eltOnly" model = "closed" level = "02"
          occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "unbounded">
    <XSDCOB:complexType type = "#group" name = "WBER"
           content = "eltOnly" model = "closed" level = "02"
           occurs = "ONEORMORE" minOccurs = "0001" maxOccurs = "0001">
      <XSDCOB:element type = "#char" name = "WRATE"
            content = "TextOnly" model = "closed" level = "03"
            occurs = "ONEOFMORE" minOccurs = "0001" maxOccurs = "0001"
            pos = "0007" lng = "0004"
            pic = "X(4)" usage = "DISPLAY"/>
      <XSDCOB:element type = "#char" name = "WADATE"
            content = "TextOnly" model = "closed" level = "03"
            occurs = "ONEOFMORE" minOccurs = "0001" maxOccurs = "0001"
            pos = "0011" lng = "0008"
            pic = "X(8)" usage = "DISPLAY"/>
      <XSDCOB:element type = "#dec" name = "WLFNR"
            content = "TextOnly" model = "closed" level = "03"
            occurs = "ONEOFMORE" minOccurs = "0001" maxOccurs = "0001"
            pos = "0019" lng = "0006
            pic = "X(6)" usage = "DISPLAY"/>
      <XSDCOB:element type = "#dec" name = "WBTNR"
            content = "TextOnly" model = "closed" level = "03"
            occurs = "ONEOFMORE" minOccurs = "0001" maxOccurs = "0001"
            pos = "0025" lng = "0007"
            pic = "X(7)" usage = "DISPLAY"/>
    </XSDCOB:complexType>

<message name="getPSAB001VResponse">
   <part name="getPSAB001VResponse" name = "PSABO01V-PARAMS" type="types:PSAB001VResponseTypes" />
</message>

<portType name="RSFServiceTemplatePort">
   <operation name=„PSAB001V">
         <input message = "PSAB001VRequest"/>
         <output message= „PSAB001VResponse"/>
   </operation>
```

Fig. 4: Generated WSDL Interface

The four results of COBLink are the two WSDL schemas – one for the web service request and the other for the web service response – plus the two COBOL wrapper modules – one for handling the inputs and the other for handling the outputs. The two COBOL modules are generated from a template modified and enhanced by the parameter data taken from server program source. The two WSDL schemas are generated from the COBOL interface definitions in the Linkage Section. This technique has also been published by the author in an earlier paper [Sned01].

4.6 Testing the wrapped Services

In a subsequent test the wrapped services were subjected to a number of test requests from a remote web client. For that the service requests had to be simulated. Requests were manually edited and dispatched via a Java AJAX driver. As it turned out, it was not possible to reuse the existing test data since the interfaces of the services had changed. Instead of submitting a MFS map, one now had to submit a WSDL message. This message differed not only in structure but also in content. The input messages contained only a subset of the original map contents. Thus, creating test cases to test the wrapped services resulted in much more effort than was planned. The existing test inputs had to be manually transformed over into web service requests.

For the five programs making up the pilot project, this manual testing was possible but it became obvious that the test required more effort than what went into all of the proceeding steps. The reason for the high costs of testing lies in the changing of the program interface. Having a new interface, in this case the WSDL interface, requires setting up test data which covers the parameters in that interface. There are tools for randomly generating data based on the specified data types, but this is not sufficient to test the business logic. To test the business logic requires that the data which was originally submitted via the IMS maps, be refitted to the web service interface. This has to be done manually by someone familiar with the application.

The measuring, stripping, wrapping and linking of the code components could be done automatically with the use of tools, whereas the testing had to be done manually. This came as a surprise to the project stakeholders, who had not contemplated what it would cost to test all of the web services taken from the existing code base. For this reason the migration project was put on hold until a solution could be found for automatically testing the services.

In the meantime, the financial crisis brought a stop to all of the activities aimed at introducing a service-oriented architecture. The user organization has decided to remain in their legacy world until the storm has passed over. The author had to move on and deal with something else, namely the conversion of an ancient COBOL-74/VSAM system into COBOL-85 with DB2.

5. Results of the pilot project

New methods can be easily conceived and propagated, especially in software technology where it is very difficult to demonstrate their feasibility without having access to a real world industrial environment. In the end it is not the method which counts, but only the results. No matter how appealing a method may be, it is worthless without being able to produce the right results for the right environment within the time and budget constraints imposed by the user. The results of the project presented here were defined from the start as being:

1) A set of metric reports and graphs depicting the sizes, complexities and qualities of the COBOL programs slated to be reused as web services in a service-oriented environment.

2) Stripped COBOL programs in which only the selected functions and their data are contained. All of the other undesired code and data definitions should have been removed.
3) Wrapped COBOL programs to be reused as web services in which the calls to the TP-Monitor, in this case the IMS-DC calls, are replaced by wrapper calls with a new type of interface.
4) A wrapper module for each COBOL web service to receive the web service request and to convert the XML data contained therein to the COBOL input data expected by the web service.
5) A wrapper module for each COBOL web service to create a web service response from the COBOL output data and to dispatch it back to the client.
6) A WSDL schema for the web service request.
7) A WSDL schema for the web service response.

5.1 Metric Reports

The metric reports for each program and each subsystem included rankings and comparisons as well as various graphics such as kiviat diagrams depicting the degree of fulfillment of the eight quality metrics as well as the mutual relations of the eight complexity metrics. There are also management dashboards with gauges for the different program characteristics. Of particular importance here was the reusability of programs. The median reuse rating of 0,5 is divided by the measured reuse rating to give the effort multiplication factor to adjust the unadjusted effort required to adapting the target program to a web service. All of the 7,297 COBOL programs were ranked according to this criteria to select the most likely candidates for web services. (See Figure 5)

RANK	PROGRAM	QUALITY
0001	P20FIVS2	0.591
0002	P22FIVS3	0.526
0003	P17FIVS1	0.487
0004	P18FIVS5	0.432
0004	P10FIVS0	0.383
Average	Quality =	0.484
Median	Quality =	0.487

Fig. 5: Ranking Programs for Reusability

5.2 Stripped Programs

For the pilot project to test the feasibility of the wrapping approach only five of the 7,297 programs were selected to be stripped. The reasons for selecting only five programs have already been given. There were both time and budget restrictions to this pilot project.

The tool COBStrip displayed the data division of these programs and the specialist for defining web services selected those variables which he would like to have in one web service response, i.e. the results of a service invocation. The technique used was similar to that proposed by the author for extracting business rules in a previous project [SnEr96]. Then the code was stripped to contain only those paragraphs required to produce the desired results. Data not used by these paragraphs was deleted. What remained was a subset of the original program. The following example illustrates a section of code stripped out of a COBOL program for obtaining the day of the week based on the date. (see Figure 6)

```
XM059-C.
* FOR SETTING LANGUAGE
   EVALUATE TRUE
   WHEN SPC = 1
     MOVE LANG-1 (TAB-I) TO DAY-NAME
   WHEN SPC = 2
     MOVE LANG-2 (TAB-I) TO DAY-NAME
   WHEN SPC = 3
     MOVE LANG-3 (TAB-I) TO DAY-NAME
   END-EVALUATE
* FOR LEFT SCHIFT
   IF LRS = 'L'
      CONTINUE
   ELSE
      MOVE DAY-NAME TO WW
      MOVE 10 TO TAB-I
      PERFORM WITH TEST BEFORE UNTIL W1
(TAB-I) NOT = SPACE
         SUBTRACT 1 FROM TAB-I
      END-PERFORM
      PERFORM WITH TEST BEFORE VARYING W-I
FROM 10 BY -1
      UNTIL TAB-I = 0 OR TAB-I = 10
               MOVE W1 (TAB-I) TO W1 (W-I)
        MOVE SPACE TO W1 (TAB-I)
        SUBTRACT 1 FROM TAB-I
      END-PERFORM
      MOVE WW TO DAY-NAME
   END-IF.
   GOBACK.
```

Fig. 6: Sample of stripped Code

5.3 Wrapped Programs

The third result was the wrapped programs themselves. The tool COBWrap scans through the code to identify all of the IMS-DC calls and to replace them with calls to the wrapper module. The original IMS-DC calls are placed in comments. Of all the results produced this was the easiest to produce, since it only meant recognizing and replacing IO macros. The result of this step is depicted schematically in Figure 7.

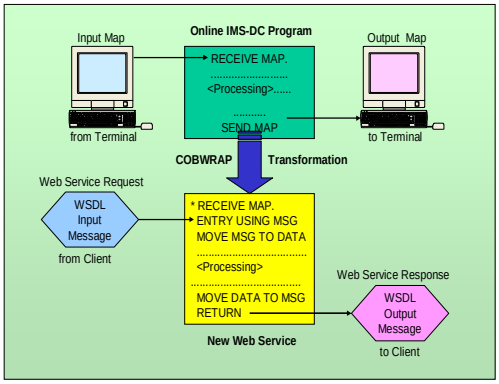

Fig. 7: Sample of wrapped IMS-DC Code

5.4 The Wrapper Modules

The wrapper modules, one for converting the web service request and the other for creating the web service response were generated by COBLink from the source of the wrapped COBOL program. These modules are called by the wrapped COBOL program, the input module to provide the input data from the web service request and the output module to create the web service response from the output data of the COBOL program. The main task is that of the data conversion. The XML data types in the WSDL message have to be mapped to the COBOL data types in the linkage section of the wrapped program. For this purpose the COBOL data types, lengths and positions are defined as attributes in the WSDL schema. The other tasks are to queue the incoming SOAP messages, to unpack the service requests from the SOAP messages, to pack the service responses into SOAP messages and to dispatch those outgoing messages. The technique of wrapping is well documented in the pertinent literature [Keys89]. The structure of a wrapper module is displayed in Figure 8.

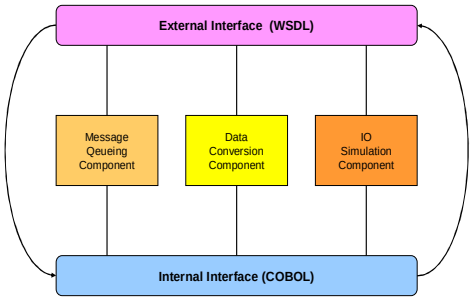

Fig. 8: Structure of a Wrapper Module

5.5 WSDL Schemas

Besides generating the wrappers, COBLink also generates the appropriate WSDL schemas, one for the request and one for the response. A sample request and the corresponding response are depicted in Figure 9. These schemas are intended for the web client to use in producing requests and consuming responses. Without them, it would be very difficult to create a proper WSDL which is consistent with the expectations of the COBOL web service. These schemas are also used by the wrapper modules to interpret the structure of the incoming requests, to convert the data from XML to COBOL and back from COBOL to XML and to generate the structure of the outgoing responses (see Figure 9)

```
<!DOCTYPE "xm059i" SYSTEM "xm059i.xsd">
<xm059i>
  <DayofWeekRequest>
     <DAY>12</DAY>
     <MONTH>10</MONTH>
     <YEAR>1977</YEAR>
     <LANGUAGE>3</LANGUAGE>
     <ALIGNMENT>1</ALIGNMENT>
  </DayofWeekRequest>
</xm059i>
```

Sample Request

```
<!—DOCTYPE XM059O  SYSTEM "XM059O.xsd"-->
<XM059O>
  <DayofWeekResponse>
    <RETURN-CODE>00</RETURN-CODE>
    <P4>
<DAYOFWEEK>Mercoledi</DAYOFWEEK>
   </P4>
 </DayofWeekResponse>
</XM059O>
```

Sample Response

Fig. 9: Generated WSDL Schemas

6. Status of the project and future work

As of this date the pilot project has been completed. The five selected programs were wrapped and tested. Altogether 12 independently executable web services were extracted from the 5 sample source programs. Among those 12 web services were:

- a service to confirm the insurance agent id
- a service to collect all of the policies sold by a given insurance agent in a given time frame
- a service to compute the bonus of an insurance agent
- a service to authorize access to a policy
- a service to create a match code
- a service to scan thru the policies by means of a match code
- a service to extract data from a customer record
- a service to update a customer record
- a service to archive a customer record
- a service to collect all beneficiaries of a policy
- a service to extract selected data from a policy

- a service to compute the day of the week

The stripping and wrapping of the services proved to be feasible, provided the programs selected satisfy the preconditions for wrapping. It was also possible to generate the appropriate WSDL interfaces. It was not possible to convert the test data. The testing of the web services required a lot of effort just to set up, let alone to execute, so testing turned out to be a major barrier to implementing this technology. If the effort required to test the 12 web services – 32 person days – is projected to the several hundreds of web services required to replicate the whole application, then the migration would be unfeasible.

It is one thing to create web services from existing code and another thing to prove that they perform correctly. The first can be automated. The second requires significant human effort even if it is automated. The tester must assign the input arguments in accordance with the pre conditions and define the expected results in accordance with the post conditions of a particular step in a business process which does not even exist yet. Whereas the actual wrapping of the web services costs less than a day per service, the testing of that services costs 2-3 person days. Stakeholders find it difficult to accept this. It is, therefore, absolutely essential to find a way to automate the testing of reused web services based on the test data of the original legacy components. The author is currently searching for a practical solution to this problem. In the meantime, it is only possible to wrap transactions as a whole at the level of the user interface.

This brings up the question of what should be done first – the design of the business processes or the development of the web services. If web services are made first as is the case here, it is not sure they will fit to the business processes being designed, especially since the business analysts doing the business process modeling are not concerned with the availability of the services. If, on the other hand, the business processes are designed first, then it is sure that none of the existing mainframe programs will fit to them. So here again, we are faced with another chicken and egg problem. Fortunately, it is not up to the author to solve that problem. His task was to demonstrate that web services can be created from existing mainframe programs and this task has been accomplished.

For the future, there is still much work to be done to make the COB2WEB tool set more reliable and more usable. There is also still some optimizing work to be done, for instance when a paragraph contains a GO TO into another paragraph, it might be better to include that code in the paragraph from whence the GO TO comes. At present, if only one field in a data structure is referred to, then the whole data structure is included in the stripped module. That too might be improved upon, but it is always dangerous to change the structure of data, since that can result in undesired side effects. In any case there is still much to be done, even if the approach appears to work for the sample taken. The biggest remaining task is, of course, to automate the testing of the migrated web services without having to create a whole new test data base. This will be no easy task.

The underlying tools of the COBWEB tool set are implemented themselves in COBOL. The graphical user interface to the underlying tools, which run in the background, is implemented in Delphi. The work data required by the tools is stored in local tables. The user interface could as always be improved to allow the user to better select the desired results. However this is a minor problem compared with that of testing the results.

7. Conclusion

This contribution has presented a tool supported approach to reusing existing COBOL programs as web services. For this purpose the author has developed a tool kit COB2WEB which performs the necessary adaptations to the target program and which generates the code required to wrap that program behind a WSDL interface. The tools have been applied in a pilot project to create web services for a large insurance company. The further use of the tools is for the time being suspended until the project is continued. Whether it continues or nor depends on a lot of factors, the least of which is the technical quality of the tools. The main factors are of a political nature, i.e. whether the user really wants to migrate his IT production to a service-oriented architecture. From a technical point of view, the current focus is on the test process in connection with the certification of web services [Sned08].

References:

[Acde01] Aversano, L./Canfora, G./deLucia, A.: “Migrating Legacy System to the Web”, in Proc. of CSMR-2001, IEEE Computer Society Press, Lisabon, March 2001, p. 148

[BGT02] Bodhuin, T./Guardabascio, E./Tortorella, M.: “Migrating COBOL Systems to the Web”, WCRE-2002, IEEE Computer Society Press, Richmond, Nov. 2002, p. 329

[BGT03] Bodhuin, T./Guardabascio, E./Tortorella, M.: “Migration of non-decomposable software systems to the Web using screen proxies” Proc. of, WCRE-2003, IEEE Computer Society Press, Victoria, B.C., 2003, p. 165

[BiKw06] Bichler,M./Kwei-Jay, L.: „Service oriented Computing“ IEEE Computer, March, 2006, p. 99

[CFF06] Canfora, G./Fasolino, H./ Frattolillo, G.: "Migrating Interactive Legacy System to Web Services", Proc. of CSMR-2006, IEEE Computer Society Press, Bari, March 2006, p. 23
[CTM08] Ceccato, M./Tonella,P./Matteotti,C.: "GoTo Elimination Strategies in the Migration of Legacy Code to Java", IEEE Proc. Of CSMR2008, Athens, April, 2008, p. 53
[Horo98] Horowitz, E.: "Migrating Software to the World Wide Web", IEEE Software, May 1998, p. 18
[CW08] Computerwoche, Nr. 32, Report of German Software Initiative, July, 2008, p. 5
[DeLu08] DeLucia et al.: "Developing Legacy System Migration Methods and tools for Technology Transfer" in Software: Practice and Experience, Vol. 38, No. 13, Nov. 2008
[FaNe97] Fantechi,A./Nesi,P./Somma,E.: „Object-Oriented Conversion of COBOL", Proc. of CSMR1997, Berlin, March, 1997, p. 157
[FRS94] Fergen,H./Reichelt,P./Schmidt,K.: „Bringing Objects into COBOL - Moore, a tool for migrating from COBOL to OO-COBOL", Proc. Of Int. Conference on Technology of OO Languages and Systems, TOOLS94, New Orleans,1994, p. 435
[KBS04] Krafzig,D./Banke,K./Schama, D.: Enterprise SOA, Coad Series, Prentice-Hall, Upper Saddle River, N.J., 2004, p. 6
[Keys89] Keyes, J.: Datacasting – How to stream Data over the Internet, Mcgraw-Hill, New York, 1989, p. 241
[KLS08] Kontogiannis, K. /Lewis, G./ Smith, D.: "The Landscape of Service-oriented Systems: A research perspective for Maintenance and Reengineering" in 2nd Workshop on SOA based systems in Proc. of CSMR2008, IEEE Computer Society Press, April, 2008, p. 336
[KLS07] Kontogiannis, K./ Lewis, G. Smith, D.: " A Research Agenda for Service-Oriented Maintenance" Workshop Proceedings of CSMR-2007, Amsterdam, 2007, p. 100
[PuSn89] Puhr, P./Sneed, H.: "Code Stripping as a means of instrumenting embedded systems" in EU ESPRIT Project 1258 – Report-1258-3, Liverpool, 1989
[SPD03] Seacord, R./Plakosh, D./Lewis, G.: Modernizing Legacy Systems, Addison-Wesley, Reading, 2003, p. 120
[Sned07] Sneed, H.: "Migrating to Web Services – A research framework", Workshop Proceedings of CSMR-2007, Amsterdam, 2007, p. 116
[Sned95] Sneed, H. "Understanding Software through Numbers", Journal of Software Maintenance, Vol. 7, No. 6, Nov. 1995, p. 405
[Sned98] Sneed, H.: "Measuring Reusability of Legacy Software" in Software Process, Volume 4, Issue 1, March, 1998, p. 43
[SnMa98] Sneed,H., Majnar, R.: „"A Case Study in Software Wrapping", Proc. of Int. Conference on Software Maintenance, IEEE Computer Society Press, Washington, D.C. Nov. 1998, p. 86-.93
[Sned99] Sneed, H. "Object-oriented Software Migration, Addison-Wesley Pub., Bonn, 1999
[Sned01] Sneed, H.: "Wrapping Legacy COBOL Programs behind an XML Interface", Proc. Of WCRE-2001, IEEE Computer Society Press, Stuttgart, Oct. 2001, p. 189
[SnEr96] Sneed, H./ Erdoes, K.: "Extracting Business Rules from Source Code", Proc. of IWPC-96, IEEE Computer Society Press, Berlin, March, 1996, p. 240
[Sned06] Sneed, H.: "Integrating legacy Software into a Service oriented Architecture", in Proc. of CSMR-2006, IEEE Computer Society Press, Bari, March 2006, p. 3
[Sned08] Sneed, H.: "Certification of Web Services" in 2nd Workshop on SOA based systems in Proc. of CSMR2008, IEEE Computer Society Press, April, 2008, p. 336
[Sned08] Sneed, H. "Measuring 75 million lines of code" Proc. of IWSM-2008, Munich, Springer Pub., Nov. 2008, p. 271
[TDH04] Tilley, S./ Distante, D./ Huang, S.: "Web Site Evolution via Transaction Reengineering", Proc. of WSE 2004, Chicago, Sept. 2004, p. 31
[Tere02] Terekhov,A./Koznov,D./Boulychev,D.: "Project specific languages and their application in Reengineering", IEEE Proc. Of CSMR2002, Budapest, March, 2002, p. 177
[Zou07] Ying Zou, Qi Zhang, Xulin Zhao: Improving the Usability of e-Commerce Applications using Business Processes, IEEE Trans. on S.E., Vol. 33, No. 12, Dec. 2007, p. 837