Representing Procedural Source in UML

Patrick DJ Kulandaisamy (patrickj@infosys.com)
N. S. Nagaraj (nagarns@infosys.com)
Srinivas Thonse (srinit@infosys.com)

Comfactory, SETLabs
Infosys Technologies Limited,
Electronics City, Hosur Road, Bangalore, India 561 229

Abstract

Traditional procedural systems have been designed using structured system analysis methods and function decomposition techniques. Reengineering them to new architectures has been a challenge because recovery of abstractions from the traditional systems has been complex. Language translation does not scale for this reengineering problem. MDA and UML based techniques can help in reengineering these applications. This paper explores aspect based reengineering approach for using these MDA based techniques in reengineering procedural applications.

Introduction

Reengineering can be defined as a process that takes source program and subjects it to a set of conversion methods to deliver target programs in a new language and architecture.

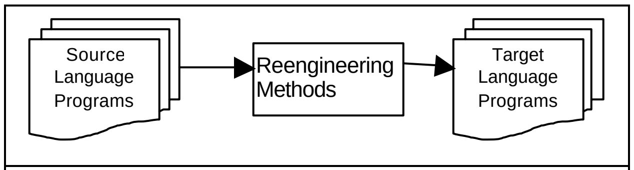

Figure 1: Concept of Reengineering

Current approaches for reengineering can be classified as:

- Source analysis and Recoding

Programs in the application are analyzed to understand business functionality and design. The knowledge gained drives redesign and recoding in new architecture and language

- Language Translation

Application programs are input into a language conversion tool that recognizes source language constructs and generates programs in target language

These approaches have many limitations and are not scalable for large application reengineering. Section 2 elaborates on these methods.

MDA proposed by OMG, promises software development lifecycle as set of models and translations. MDA is driven by separation of concerns theme that is achieved through PIMs (platform independent models) and PSMs (platform specific models).

This paper describes aspect-based reengineering that adopts these techniques.

Current Methods of Re-engineering

Structurally procedural languages have many similarities. They have single entry point into the main program and organized as set of callable functions and subroutines. Majority of data managed in these programs are global data. Persistent data is managed through SQL and file IO operations embedded in the source code. Transaction boundaries and error management facilities are hard-coded. User interface is often tightly coupled with processing logic. Hence the design artifacts of procedural systems have often been simple word documents and program source code is the only dependable source of input for reengineering automation.

Source Analysis and Recoding

Current reengineering is about gathering detailed knowledge about the application and rewriting the business functions in target environment. Figure 2, depicts various stages of conventional reengineering.

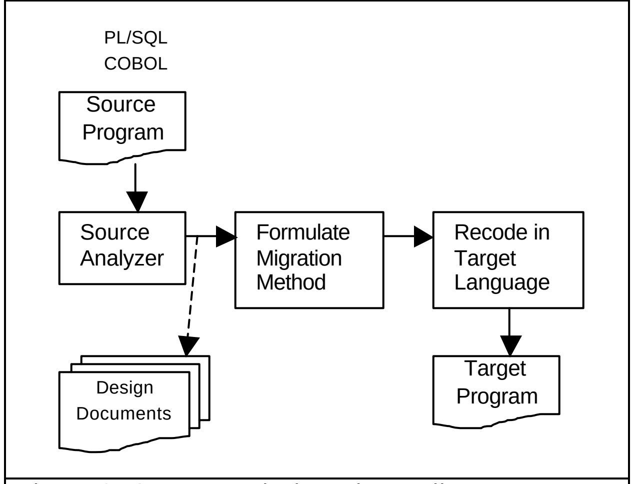

Figure 2: Source Analysis and Recoding

As depicted, source language programs of an application are parsed and analyzed using a source analyzer. The diagramming facilities in analyzer tools present various views of the application such as program call graph, data usage matrix, algorithm flow chart and so on. These views are primarily read-only and help in gaining business functions implemented in the system and documenting application design in text format.

Thorough knowledge of the source application is the basis for formulating migration approach. Such a migration approach normally consists of techniques for mapping data structures, methods for optimizing user interface and guidelines for translating program logic. It would also include activities for business process workflow identification, transaction boundary identification, validation & error conditions, and data table to business entity modeling, data access separation and data schema migration.

The application is recoded manually in target language and architecture.

Language Translation

Compiler construction tools and language translation techniques have resulted in variety of tools that translate programs from one language to another. These tools use source language's grammar to parse and recognize input programs. The tool traverses the parse tree and applies language translation rules to generate target language program. A pictorial representation of such a method is given below.

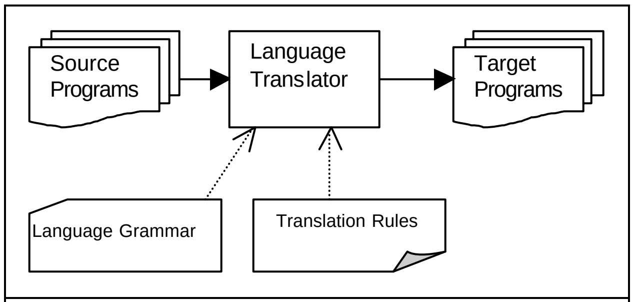

Figure 3: Conventional Reengineering - Translator

Application reengineering does not end with source language to target language conversion. It extends into runtime environment migration as well. There is a vast disconnect between source and target environments in architecture, transaction monitor, scripting environment, development tools, performance monitoring tools and to the level of network and operating system APIs. Mapping platform services from source to target environment has always been a manual task as this is difficult to automate.

Limitations of current reengineering approaches

- Language translation works on line-by-line conversion concept. It is suited for translation into similar architecture only.
- Programs produced by language translator are poorly structured, contain cryptic variable names, use non-optimal data structures. Maintainability of such programs is difficult.
- Language translation does not provide design models of the application.
- Execution effectiveness and artifacts consistency is not repeatable as it depends on development team's skill set and application knowledge the team possesses.
- Conventional reengineering is equivalent to new application development with respect to manual recoding phases.
- Compliance to component architectures like J2EE is not feasible in language translation approach.

MDA Concepts for Re-engineering

OMG defines MDA as an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of the functionality on a specific technology platform. Models in MDA are of two types – Platform Independent Model (PIM) and Platform Specific Model (PSM). PIM describes a platform independent specification of the functionality of the application. PSM defines a platform specific model of the system incorporating technology specific details. Other key elements of MDA are, UML profiles that specify formal semantics to UML extension and model transformation rules.

Elaborating MDA concepts, reengineering approach can be refined as set of model translations as depicted in the diagram below.

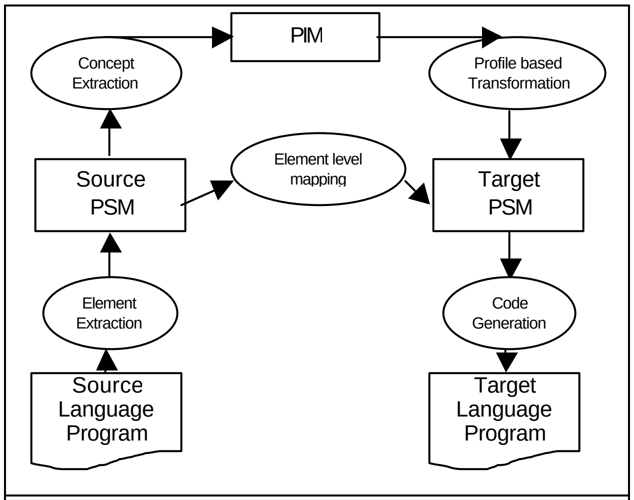

Figure 4: MDA concepts for reengineering

MDA concepts suggest that application reengineering be set of model translations. Elements from source program will be extracted and represented as source PSM. UML profile for the source environment will be used in identifying and extracting relevant code segments that match profile elements. Source PSM thus obtained will be subjected to PSM-PIM translation rules to segregate core abstraction thereby platform independent model for the application. Target environment's UML profile will be applied on this abstract model to arrive at a model that is compliant with target UML profile. This PSM subjected to code generation algorithms yields target language programs. For some PSM elements where abstraction to PIM is not optimal, it is advisable to provide a direct mapping to target PSM through element level mapping rules.

Aspect based Re-engineering

A program can be viewed as comprising of many types of elements. Aspect identifies types of code fragments that can be considered as logical units for translation. The aspects typically are business rules, transactions, errors, user interface, SQL operations, computations, workflow, functions, environment API calls and so on. For example, in a COBOL program a paragraph that implements database insert operation would be considered as SQL aspect and translated to an equivalent API call in target architecture such as JDBC call. Similarly, record structure definitions in data division would be considered as TABLE aspect and translated to entity classes in target architecture.

Hence re-engineering a procedural system can be defined as identifying aspects in a program and transforming them into relevant objects guided by aspect translation rules and target architecture. Diagrammatic representation of aspect based re-engineering approach is given in Figure 5.

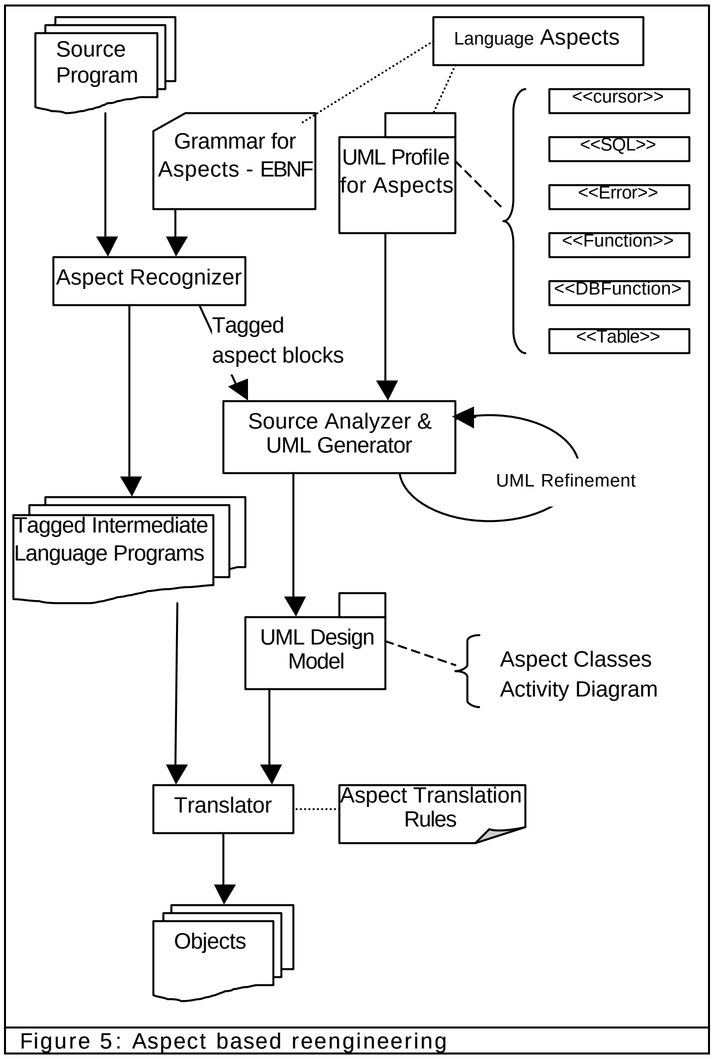

Figure 5: Aspect based reengineering

Aspect recognizer accepts source programs and identifies the language aspects. Language aspects are modeled in a grammar rule notation such as **EBNF**.

The aspect recognizer marks the language aspects as distinct areas. These areas are represented as **tagged aspect blocks**, which are group of source language statements that implement a specific aspect. These aspect blocks can be distinctly viewed in source browser.

The **source analyzer** matches the aspect blocks and builds a preliminary **UML model** by referring to UML profiles for the aspects.

The **UML profiles** define UML extensions as stereotypes for significant aspects in the programming language. For example, most procedural languages embed database operations as SQLs. It is beneficial to recognize this aspect of the language and define <<SQL>> stereotype in the UML profile.

The UML model provides class diagrams representing the aspects and activity diagram for modeling relationship among these aspects. For example, a block of statements that deal with errors in the source program would be modeled in an exception class. An aspect that interacts with database will be modeled as set of classes that represent entities and containing the database operations and queries. The developer can view and refine these diagrams.

The UML model generated by analyzer is refined manually to address additional elements of target architecture – like entity classes, class hierarchy, packaging and so on.

The **translator** working on this UML model, intermediate language representation for the aspects and guided by aspect specific translation rules, produces aspects for the target component environment, which are classes and interfaces.

Advantages
Aspect based reengineering approach delivers UML design models and translated code. Intermediate code representation and UML model representation leads to generation target objects for many languages with appropriate translation engine. This simplifies viewing of procedural programs through design tools. This method offers a way to extend aspect coverage incrementally. More aspects and intermediate code generation for those aspects can be easily added.

Illustrative Example

This section elaborates on a sample reengineering and migration work carried out to evaluate and compare various approaches.

The source environment was Oracle PLSQL stored procedures and the target was Java with embedded SQL. Many modules in higher layers invoked these PLSQL procedures.

Our approach had following set of blocks.
1. Language Recognizer
2. Source Analyzer

3. Translation Engine

Language Recognizer defines grammar for PLSQL using EBNF for a few core language constructs. This grammar was input into ANTLR toolkit to generate PLSQL parser. This language recognizer accepts PLSQL source input and builds an abstract syntax tree representation of the source.

The program structure comprises of parameter declaration, variable declaration, cursor definitions with SQL, IF constructs, LOOP constructs and exception blocks.

Source analyzer defined rules for creating aspect blocks from the input PLSQL. The source is sliced into various aspects such as global data variables, input processing logic, output generation logic, database interaction SQL, and environment dependency calls.

Stereotypes were defined for significant aspects in the language. These were used to represent the extracted aspect blocks in UML class diagram notation. Following diagram shows the stereotypes defined and their relations.

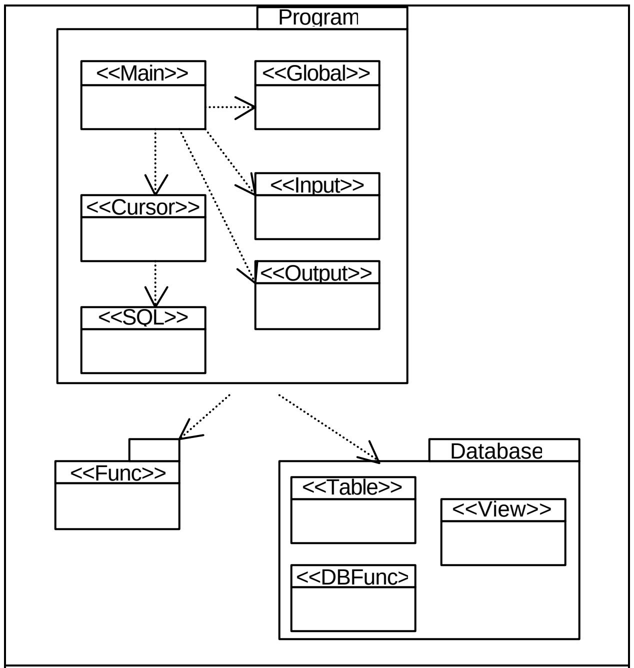

Figure 6: PLSQL aspect stereotypes

The translator generated Java equivalents for significant aspects of PLSQL. It also generated intermediate representation for SQLs using SQLJ constructs.

Future Work

Automating UML domain model generation for a given application, experimenting with multiple languages and architectures, design metrics collection are potential extensions to this research.

References

1. UML 1.4 Specification
2. Model Driven Architecture (MDA) Document number ormsc/2001-07-01 ORMSC July 9, 2001
3. ANTLR (www.antlr.org)
4. Object-Oriented Software Engineering by Ivar Jacobson et al.
5. Software Reengineering, Robert S Arnold