

## **TAGDUR: A Tool for Producing UML Sequence, Deployment, and Component Diagrams Through Reengineering of Legacy Systems**

Richard Millham, Jianjun Pu, Hongji Yang  
De Montfort University, England  
[Richard.Millham@shaw.ca](mailto:Richard.Millham@shaw.ca) & [hyang@dmu.ac.uk](mailto:hyang@dmu.ac.uk)

**Abstract:** *A further introduction of TAGDUR, a reengineering tool that first transforms a procedural legacy system into an object-oriented, event-driven system and then models and*

*documents this transformed system through a series of UML diagrams. This paper focuses on TAGDUR's generation of sequence, deployment, and component diagrams.*

**Keywords:** *UML (Unified Modeling Language), Reengineering, WSL*

This paper is a second installment in a series [4] that introduces TAGDUR (Transformation and Automatic Generation of Documentation in UML through Reengineering). TAGDUR is a reengineering tool that transforms a legacy system's outmoded architecture to a more modern one and then represents this transformed system through a series of UML (Unified Modeling Language) diagrams in order to overcome a legacy system's frequent lack of documentation. The architectural transformation is from the legacy system's original procedurally-structured to an object-oriented, event-driven architecture. Once this transformation is complete, TAGDUR documents the structure and behavior of the transformed system through a series of UML diagrams including class, activity, deployment, sequence, and component diagrams.

This paper gives a brief overview of the problems posed by many legacy systems, a general description of TAGDUR's design, and how TAGDUR generates three types of UML diagrams: sequence, deployment, and component.

### **1.1 Overview of Problem**

Early computer systems, faced with severe memory constraints, were designed to be both procedurally structured and driven.

In many cases, these systems were designed as standalone systems. As time passed and business needs changed, a need arose to integrate these disparate systems together and remodel them to

accommodate a multi-tiered, Web-based platform. In order to accommodate this remodeled platform, the original sequential-driven, procedurally structured legacy system must be transformed to an object-oriented, event-driven system. Object orientation, because it encapsulates variables and procedures into modules, is well suited to this new Web architecture where pieces of software must be encapsulated into component modules with clearly defined interfaces. A transfer to a Web-based architecture requires a real-time, event-driven response rather than the legacy system's original procedural invocation.

Object orientation offers additional advantages. Object-oriented systems have lower maintenance costs than procedural software. Because object oriented systems have encapsulated software modules, modules from different systems can more easily be integrated than software that is procedurally structured with global variables.

Often, the high cost of development and switchover costs precludes developing replacement systems for these legacy systems.

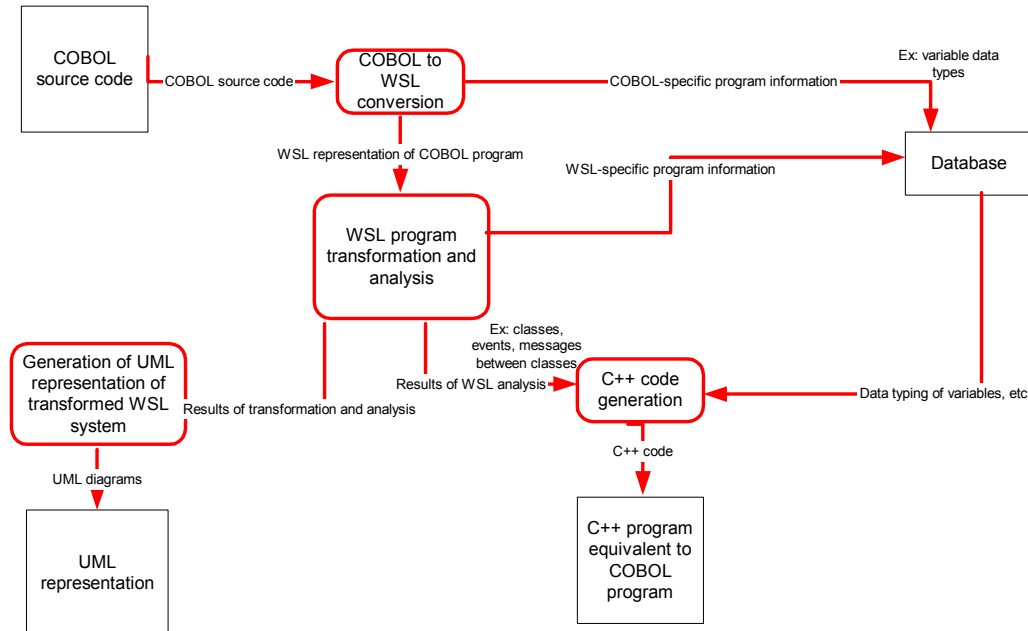
However, in order to integrate disparate systems, developers must fully understand the systems that are being integrated, whether these systems have been reengineered or not. Developers must understand the structure of the system, the data and control flows, and the execution of external events. Documentation containing this information is often missing or obsolete. Consequently, any reengineering process must

incorporate some capability within it to generate documentation pertaining to the structure and dynamics of this system.

Our tool, TAGDUR, was designed with the purpose of trying to address the problem of lack of system documentation. By utilizing

information obtained during the transformation process and by parsing the code of the transformed system, our tool is able to utilize this acquired information in its generation of UML diagrams of the transformed system.

## 2. Tool Design



Overview of TAGDUR Tool Design

The COBOL legacy system is converted into WSL using a set of COBOL to WSL conversion rules that were developed using, for its basis, Martin Ward's [11] paper, *The Syntax and Semantics of the Wide Spectrum Language*, which defined the basis of the Wide Spectrum Language. Because WSL lacks a data typing capability, programming language-specific information, such as variable data types of the original legacy system, can not be represented in WSL but, instead, are stored in the database for future use, such as during the C++ code generation phase.

Once this COBOL to WSL conversion phase has been completed, TAGDUR, via a series of transformation steps, transforms the original procedurally structured and sequential legacy system to an object-oriented, event-driven system. The first of these steps is to use our clustering technique to identify objects along with their attributes and operations. Our technique groups closely coupled procedures and variables into classes. [2] The second

transformation step is to evaluate tasks, using several different algorithms as outlined in [3] and at procedural, program block, and individual code line level of granularity, for their degree of task independence. A task is defined as an atomic unit of work; tasks can be defined at several different levels of granularity such as program block, procedure, and individual code line. Our task evaluation technique consists of analyzing two normally sequential for the presence of data and control dependencies between them. If a dependency exists, these tasks are deemed to be sequential; if no dependency exists, these tasks are deemed to be able to execute independently. The last transformation step is to identify possible events from source code. The event identification step involves constructing a control graph of procedure calls, I/O calls, system interrupts, and error invocations by parsing source code. Nodes of this control graph which involve interactions with other objects are modeled as events. An example, raising an error is modeled as an event occurrence between the object where the error occurred and the object, usually a System object,

which handles the error. Once the events are identified, each event is evaluated in terms of its synchronicity. Synchronicity is determined by evaluating, at the individual code line level of granularity, whether the task where the event occurred and the task immediately successive to this task share any control or data dependencies. If a dependency exists, the event is deemed to be synchronous; otherwise, if no dependency exists, the event is deemed to be asynchronous.

Once the transformation process completes, TAGDUR documents this transformed system by representing it through a series of UML diagrams. These UML diagrams are represented in UXF textual format. UXF (Uml eXchange Format) is a XML-based model interchange for UML models developed by Junichi Suzuki and Yoshikazu Yamamoto. [8] Diagrams represented in UXF can be imported into a UXF-compatible graphical tool for viewing.

After the transformation process finishes, the WSL intermediate representation is restructured into classes. Each variable that is associated with a class is modeled as an attribute of the class while each procedure that is associated with a class is modeled as an operation of the class.

Information, which was obtained during the transformation process, is used in the production of UML diagrams. An example, classes that have been identified during the class identification process become classes in the class diagram. The independent task evaluation process determines the sequencing order of tasks; each task is given a sequence number, in ascending order of execution. If two or more tasks may execute in parallel, these tasks are assigned the same sequence number. The sequence number of the task enclosing an event, such as a file I/O operation, is modelled as the sequence number of the message in the sequence diagram. Events that are identified during the event identification process are portrayed as messages between objects in the sequence diagram.

## 2.2 Rationale behind Tool Design Decisions

The UML modeling notation was selected as the method to model the transformed reason for several reasons. UML, through its series of diagrams, provides several different perspectives of the system. UML contains use case diagrams, which model business processes from the end-user perspective, but also contains activity and

class diagrams, which describe both the static structure and behaviour of the system in terms that are most useful to developers. UML is a world-accepted modelling standard with significant tool support. UML is also platform and programming language independent. Although use case diagram generation is not a present feature of UML, TAGDUR provides class and activity diagrams.

WSL has many advantages which make it ideal as an intermediate language. WSL was designed to be easy to analyse and transform. WSL is supported by several tools, including the Fermat transformation system which provides automatic transformation and code simplification. WSL is programming and platform independent. As a result, the transformations and modeling that TAGDUR performs on a WSL-represented system can be accomplished regardless of whether the original legacy system was in COBOL or C. The original legacy systems need only to be converted into WSL first. [10]

Fermat provides the ability to convert source code of other languages, such as IBM 370 assembly language, into WSL and then convert this WSL code into other programming languages such as C or COBOL.[9] TAGDUR can utilize Fermat's language conversion features by having Fermat translate assembly language into WSL and then transform this WSL representation into an object-oriented architecture which then can be translated into C++.

Finding system artefacts, on which to base the modelling of UML diagrams of the system is difficult for many legacy systems. Documentation of the system often does not exist. The original developers and end-users, who are often an important source of information about the system, have long since left the organization. Because these systems often have been left in light maintenance mode for many years, current maintainers and end-users have minimal knowledge of the system. Because the source code, along with the associated data files, are only available system artefacts, any generation of UML diagrams, which are used to model the system, must be based primarily on source code.

## 2.4 Advantages of TAGDUR

Rumbaugh et al identifies three viewpoints necessary for understanding a software systems: the objects manipulated by the computation (the data model), the manipulations themselves (functional model), and how the manipulations are organized and synchronized (the dynamic model). [6] We wish to propose another view required for system understanding, the architectural view, which represents the physical components of the system and the relationships among them.

TAGDUR addresses all four of these views through its generation of UML diagrams: the static, behavioral, dynamic, and architectural views through the class, activity, sequence, and deployment or component diagrams respectively. The first paper in this series focused on the generation of UML class and activity diagrams, which represent the static and behavioral view of the system. [6] This paper focuses on representing the dynamic and architectural views through UML diagrams.

Sequence diagrams, which represent the dynamic view, are useful in depicting the messages passed between interacting objects. The developer needs to understand the interactions among objects as depicted in a sequence diagram in order to properly design the interfaces of these objects. If the developers plan to distribute objects among several different platforms, such as in a multi-tiered Web architecture, the developer would need to understand the object interactions in order to group frequently-interacting objects on the same platform in order to minimize communication costs.

The architectural view is required in order to understand how the system is related to external physical entities, such as a printer, and the dynamic configuration of program files. This architectural view is need if a physical entity, such as a database file, changes and the developer needs to quickly ascertain which program files access this database file in order to change the relevant code within them. In a large legacy system, such as the particular telecommunications legacy system used in our study, there are 106 source code files. It is necessary to depict the relationships among source code files, as in which source code files load which other source code files as libraries, in a component diagram because the number of potentially loadable libraries is often too numerous to keep track of manually.

Although several existing tools provide one or more of these views, no tool provides all four of these views. One existing tool, RIGI, can transform a system and then generate visual documentation of its static structure. However, RIGI does not have the capability of documenting the dynamic, behavioural, or architectural view of the system. RIGI was developed by Hausi Muller and his team at the University of Victoria, Canada. [7]

### 3. Generation of UML Diagrams

TAGDUR generates several types of UML diagrams of the transformed system, including class, sequence, deployment, component, and activity diagrams. This paper focuses on the generation of sequence, deployment, and component diagrams, representing the dynamic and architectural views of the system respectively from the information gained during TAGDUR's transformation process of the system.

#### 3.1 Sequence Diagrams

Sequence diagrams are diagrams that depict the interactions, via message passing, among objects in a temporal context. The sequence diagram is divided up into vertical sections, called swimlanes. Each object in the system is assigned their own swimlane. Messages are depicted as being sent from the swimlane of their sending, or source, object to their receiving, or target, object.

Procedure calls are modeled as messages between the caller object that invokes the procedure and the called object that contains the procedure being invoked. Exceptions and interrupts are modeled as messages between the object where the interrupt/exception occurred and the System object, representing the underlying operating system, which handles the error or exception. File input/output calls, such as statements that read or write data from files, are modeled as messages between the object invoking the input-output method and the File object, which represents a generic database file.

Conditions within WSL control constructs, such as WSL's if-then statements, that enclose code that invoke a message form conditions within the guards that govern the passing of messages from one object to another. An example, given the WSL statement, **IF W<3 THEN Call**

**UpdateVal FI**, the condition **[W<3]** forms a guard to the method invocation message **UpdateVal**. If the WSL control construct that encloses code invoking a message

These messages may be modeled as synchronous or asynchronous. A message is deemed to be asynchronous if the task that is immediately successive to the task invoking the message may execute independently. If the immediately successive task can not execute independently, the message is deemed to be asynchronous. By evaluating tasks containing message invocations, the independent task evaluation process determines which tasks may execute independently from one another and, consequently, determines which messages contained in the tasks being evaluated are deemed to be synchronous or asynchronous.

The independent task evaluation process is responsible for determining the sequence numbers assigned to each message in the sequence diagram. The independent task evaluation process assigns a sequence number to tasks at both the procedural and individual codeline level of granularity.

Each message depicted in the sequence diagram is given a sequence number in the number: <procedure task sequence number>.<individual codeline task sequence number>. The procedure task sequence number is the sequence number of the procedure where the message is invoked and individual codeline task sequence is the sequence of the codeline where the message is invoked. The sequence indicates the order of execution in ascending order; messages with the same sequence number may be executed in parallel.

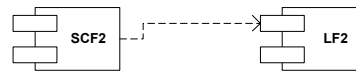
### 3.2 Component and Deployment Diagrams

Component diagrams depict the run-time relationships among software components of a system. These components may be simple files or libraries loaded dynamically. The relationships among software components are usually compilation dependencies. [5]

Deployment diagrams depict the physical arrangement of various hardware components and executable files of the system. [5]

Both the component and deployment diagrams are derived by parsing each source code file, in

turn, for the presence of keyword that indicate a relationship between the source code file being parsed and the physical entity/software component identified as the keyword. An example, during parsing source code file SCF1, the WSL statement “Put Sys01, X” is encountered. This WSL statement outputs the value of variable X to the physical data file denoted by Sys01. Consequently, this parsing determines that a relationship between source code file SCF1 and physical file SYS01 exists and hence, the deployment diagram models this relationship between the two physical nodes of SYS01 and SCF1. In a similar way, if source code file SCF2 contains a WSL statement that loads a library file, LF2, this loading is depicted in the component diagram as a dependency relationship between two software components SCF2 and LF2.



**Component Diagram Showing Compilation Dependency between library file, LF2, and source code file, SCF2**



**Deployment Diagram Showing the Interface relationship between source code file, SCF1, and file device, Sys01**

A small sample of WSL code is presented with a corresponding UML sequence diagram based on this code.

#### WSL Code Sample:

```

Main() /* main calling program */
Begin
  Call A.CreateMessages()
End.

Class A
Begin
  Var X
  Var Z
  Proc CreateMessages()
  Begin
    If System.System_Error = 'Error' Then
      Call System.System_ErrorHandler()
    Else
      Call B.UpdateVal(X,Z)
      Put File, X, Z
    Fi
  End.

```

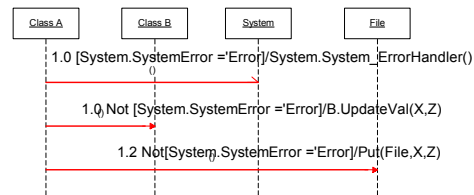
```

End
Class B
Begin
Proc UpdateVal(X,Z)
Begin
X := 8
Z := 1
End.
End
Class System
Begin
Var System_Error
Proc System_ErrorHandler()
Begin
/* handles errors */
End.
End
Class File
Begin
/* handles File I/O */
End

```

The following diagram models the WSL code sample as a sequence diagram. Each object is given its own swimlane. Messages, such as the method invocation Call B.UpdateVal(X,Z), is modeled as a message between the sending

object A (the object containing the method invocation) and the receiving object B (the object that handles the message). Potentially parallel-executing message tasks, such as method invocations of System\_ErrorHandler and UpdateVal, are given the same sequence number. Message tasks that must execute in sequence, such as UpdateVal(X,Z) and Put File, X,Z, are given different sequence numbers. Conditions, such as NOT[System.System\_Error = 'Error'], which enclose code that invoke a message, such as Call UpdateVal(X,Z), form guards to the message, UpdateVal(X,Z).



### Sequence Diagram Representation of the WSL Code Sample

### References

### 4. Conclusion

TAGDUR is a reengineering tool that was designed to address two of the most prominent problems of legacy systems: obsolete architecture and a lack of documentation. TAGDUR first transforms a legacy procedural architecture to an object oriented one and then generates documentation of this transformed system through a series of UML diagrams. This documentation allows the developer to fully understand the system and enables them to modify the transformed system, now available as a C++ program automatically generated by TAGDUR, to meet new business needs such as system integration

### Reference

- 1) Alhir, Sinan Si UML in a Nutshell O'Reilly:Sebastapol, CA, USA, 1998
- 2) Millham, Richard "An Investigation: Reengineering Sequential Procedure-Driven Software into Object-Oriented Event-Driven Software through UML Diagrams". Published in the Proceedings of the International Computer Software and Applications Conference, Oxford, 2002
- 3) Millham, Richard "Determining Granularity of Independent Tasks for Reengineering a Legacy System into an OO System" To be published in the Proceedings of the International Computer Software and Applications Conference, Dallas, Texas, 2003

- 4) Millham, Richard "TAGDUR: A Tool for Producing UML Diagrams Through Reengineering of Legacy Systems" Proceedings of the 7th IASTED International Conference on Software Engineering and Applications, Marina del la Rey, USA, 2003
- 5) Muller, Pierre-Alain Instant UML Wrox: Birmingham, UK, 2000
- 6) Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenson Object-Oriented Modeling and Design, Prentice-Hall, 1991
- 7) Storey, Margaret-Anne D., Hausi A. Müller, Kenny Wong "Manipulating And Documenting Software Structures ", ICSM '95 (Nice, France, October 16-20, 1995)
- 8) Suzuki, Junichi and Yoshikazu Yamamoto "Making UML models interoperable with UXF". Lecture Notes in Computer Science 1618, Springer-Verlag, Heidelberg, German
- 9) Ward, M. "The FermaT Assembler Re-Engineering Workbench", ICSM '(Florence, Italy, 2001)
- 10) Ward, Martin "Specifications from Source Code -- Alchemists' Dream or Practical Reality?" 4th Reengineering Forum, September 19-21, 1994, Victoria, Canada
- 11) Ward, M., "The Syntax and Semantics of the Wide Spectrum Language", *Technical Report*, Durham University, England, 1992.