Appeared in Proceedings of the 7th IASTED International Conference on Software Engineering and Application (SEA), 2003. (Preprinted-Version)

TAGDUR: A Tool for Producing UML Diagrams Through Reengineering of Legacy Systems

Richard Millham, Hongji Yang
De Montfort University, England
Richard.Millham@shaw.ca & hyang@dmu.ac.uk

Abstract: *Introducing 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.*

Keywords: TAGDUR, Reengineering, Program Transformation, UML

In this paper, we introduce TAGDUR (**T**ransformation and **A**utomatic **G**eneration of **D**ocumentation in **U**ML through **R**eengineering). TAGDUR is a reengineering tool designed to address the problems frequently found in many legacy systems such as lack of documentation and the need to transform the present structure to a more modern architecture. TAGDUR first transforms a procedurally-structured system to an object-oriented, event-driven architecture. Then TAGDUR generates documentation of the structure and behavior of this transformed system through a series of UML (Unified Modeling Language) diagrams. These diagrams include class, deployment, sequence, and activity diagrams.

This paper gives a brief overview of the problems posed by many legacy systems, a description of TAGDUR's overall design and workings, and an overview of how TAGDUR generates two types of its diagrams: class and activity.

1. Overview of Problem

During the early days of computing, computer systems were designed to fit a particular platform and were designed to address the hardware limitations of that particular platform. As an example, most legacy computer platforms were single processor and had severe memory constraints. As a result, the legacy systems built for these systems were designed to operate in a strictly sequential manner and were procedural-driven.

These legacy systems, in many cases, were designed in an ad-hoc manner and were designed as stand-alone systems. As time passed and business needs changed, the need became apparent to integrate these disparate systems together and remodel them to accommodate a multi-tiered, Web-based platform. This remodeling of legacy systems from a sequential-driven, procedural structured to an event-driven, component-based architecture requires a transformation of the original system to an object-oriented, event-driven system. Object orientation, because it encapsulates methods and their variables into modules, is well-suited to a multi-tiered Web-based architecture where pieces of software must be defined in encapsulated modules with cleanly-defined interfaces. This particular legacy system is procedural-driven and is batch-oriented; a move to a Web-based architecture requires a real-time, event-driven response rather than a procedural invocation.

Object orientation promises other advantages as well. Because object orientation has encapsulated modules of software with cleanly-defined interfaces, software, and hence their objects, from different systems can be more easily integrated than if this software was structured into procedures with global variables. Furthermore, object orientation has lower maintenance costs than procedural software.

Developing systems to address these business needs and replace these legacy systems is prevented by the high cost of development. One solution is to reengineer the existing legacy system by transforming the legacy system to an object-oriented, event-driven architecture. Once this transformation is completed, the transformed system can be adapted to meet new business needs such as system integration.

However in order to integrate disparate systems, developers must fully understand the systems being integrated, whether they have been reengineered or not. They must not only understand the software structure of the system but be able to follow the data and control flow and the execution of events. Documentation detailing this information for most legacy systems is either missing or

out-of-date. Consequently, any reengineering efforts must first have some facility to generate documentation regarding the structure and dynamics of this system.

Our tool, TAGDUR (**T**ransformation and **A**utomatic **G**eneration of **D**ocumentation in **U**ML through **R**eengineering), was designed to try and overcome this lack of documentation problem. By utilizing information acquired during the transformational process and by parsing the code of the transformed system, this tool generates UML diagrams of the transformed system.

2. Tool Design

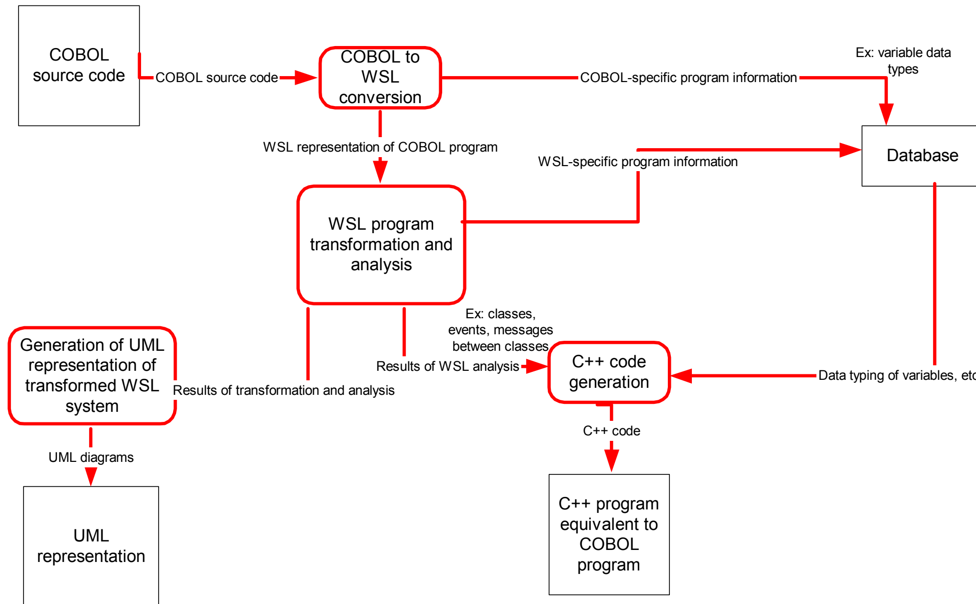

Fig. 1: Overview of TAGDUR Tool Design

We first convert the COBOL legacy system into WSL using a set of COBOL to WSL conversion rules. These rules were formulated using Martin Ward's [10] paper, *The Syntax and Semantics of the Wide Spectrum Language,* which defined the basis of the Wide Spectrum Language. Because WSL is a type less language, programming language-specific information, such as variable data types of the original legacy system, can not be represented in WSL but are stored in the database for future use, such as during the C++ code generation process.

After the conversion from COBOL to WSL has finished, TAGDUR transform the original procedural-structured and sequential legacy system to an object-oriented, event-driven system. This transformation occurs using a series of transformation steps. The first step is to identify potential classes, including their attributes and operations through our own clustering technique. This technique groups highly inter-related procedures and variables into classes. [3] The next transformation step is to evaluate tasks, at both the procedure and individual code line level of granularity, for their degree of task independence. This evaluation is accomplished using several different algorithms as outlined in [4]. A task is defined as an atomic unit of work; tasks can be defined at several different levels of granularity such as program, procedure, and individual code line. Our task evaluation technique involves analyzing two normally sequential tasks for 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 in parallel. The final transformation step is to identify possible events from source code. This event identification step is accomplished by constructing a control graph of procedure calls, I/O calls, system interrupts, and error invocations by parsing the source code. The nodes of this control graph that represent interaction with other objects are modeled as events. An example, an interrupt is modeled as an event occurrence between the object where the interrupt occurred and the System object, which handles interrupts. Once the events are identified, each event is evaluated in terms of its synchronicity. This evaluation is accomplished by evaluating, at the individual code line level of granularity, the task where the event occurred and the task immediately successive to this task in order to determine if any control or data dependencies exist among them. 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 is complete, TAGDUR generates documentation of the transformed system by representing it through a series of UML diagrams. These UML diagrams are represented in UXF textual format. A developer can import these UML diagrams into a UXF-compatible graphical tool for viewing. UXF (UML eXchange Format) is a XML-based model interchange for UML models developed by Junichi Suzuki and Yoshikazu Yamamoto. [7]

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

The information that was obtained during the transformation process is utilized during the automatic generation of documentation through UML diagrams. An example, classes identified during the class identification process form the basis of the UML class diagram. Events identified during the event identification transformation process are modeled as events in the UML activity diagram. The identification of independent tasks during the determination of independent tasks transformation step is utilized when modeling parallel or sequential control flows in UML activity diagrams.

2.1 Rationale Behind Tool Design Decisions

UML diagrams were selected as the method to document the transformed system for several reasons. UML provides documentation of the system from multiple perspectives; UML provides use-case diagrams, which address the modeling needs of end-users, but also provides statecharts and class diagrams, which are most useful to developers. The diagrams that form UML, diagrams such as statecharts, have been used by the software community for years to model and to understand software systems. UML is an accepted worldwide standard with significant tool support. Furthermore, UML is independent of any programming language and platform. Although use case diagram generation is not a current feature of TAGDUR, TAGDUR embraces class and activity, which are similar to statechart, diagrams in its UML modeling.

WSL was chosen for an intermediate language for several reasons. WSL is programming and platform independent. Consequently, the transformations and modelling that TAGDUR performs on a WSL-represented system could be performed regardless of whether the original legacy system was in COBOL or C. The original legacy systems need only to be converted into WSL first.

WSL has other advantages as well. WSL has excellent tool support, in the FermaT transformation system [9] which allows transformations and code simplification to be carried out automatically. It has the capability of enabling proof-of-correctness testing. WSL is programming and platform independent. WSL was also specifically designed to be easy to analyse and transform.

In this legacy system, finding system artefacts to use as a basis for modelling UML diagrams of the system is difficult. Any documentation of the system often does not exist. The original developers and end-users, who would be most knowledgeable about the design of the system, have long since left the organization. Often these systems have been left in light maintenance mode for many years; consequently, current maintainers and end-users have a minimal knowledge of the system. Because the source code, along with the associated data files, are the only available system artefacts, any UML diagrams that are generated to model this system must be based primarily on source code.

2.3 Advantages of TAGDUR

Some existing tools, such as RIGI, are capable of transforming a system and then generating visual documentation of the static structure of this system. RIGI was developed by Hausi Muller and his team at the University of Victoria, Canada. While RIGI documents the static view of the system, it does not have a corresponding capability of documenting the dynamic view of the system. [6]

Other existing tools, such as Argo UML, provide a dynamic view of the system through UML sequence diagrams. Sequence diagrams are useful in depicting the messages passed between interacting objects; however, sequence diagrams do not properly model the internal control logic and data manipulation within objects. Activity diagrams, on the other hand, can model the

internal activities, control logic, and data manipulation of objects. Consequently, our tool generates activity diagrams in addition to sequence diagrams.

TAGDUR, our tool, has additional features as well. The tool has the capability of generating code for a C++ equivalent program of the transformed system. Future additions to this tool will include the ability to data reengineer the underlying databases or file systems of the system. Another future addition will be the ability to generate test scripts and cases from activity diagrams; these test cases provide a means of regression testing the various iterations of the transformed system as it is being redeveloped in order to integrate this system with other related systems.

3. Generation of UML Diagrams

TAGDUR generates several types of UML diagrams of the transformed system, including class, sequence, deployment, and activity diagrams. This paper focuses on the generation of class and activity diagrams, representing the static and dynamic views of the system respectively. This paper indicates how TAGDUR utilizes information gained during the transformation process to generate class and activity diagrams.

3.1 Class Diagrams

Class diagrams represent the static structure of the system. Class diagrams convey information about classes used in the system such as their properties, their interfaces, and how these classes interact with one another. [2] Associations are relationships between two classes.

After our reverse engineering process transforms the legacy system from its procedural structure to that of an object-oriented one, our tool extracts the class diagram from the transformed system. Class definitions from the system are modeled as classes in the UML class diagram; variables encapsulated within a class become class attributes and procedures associated with a class become methods in the UML class diagram.

Classes in this system are grouped into UML packages. A package, in this legacy system example, corresponds to the original COBOL copybook. The assumption is that the original programmers divided up the system into modules, in this case COBOL copybooks, according to some logical criteria. This logical modularization of the system is preserved in the form of packages in UML diagrams.

Our tool models accesses between classes of other's attributes or methods as static associations between classes. Each end of an association contains a multiplicity; the multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Depending on the ratio of classes accessing the attributes/method to the classes accessed, the multiplicity of these association ends may be modeled as many-to-one, one-to-one, etc. [5] An example, if Class A accesses multiple methods and attributes of Class B, this association end would be modeled as many-to-one. If Class A access only one attribute of Class B, then this association end would be modeled as one-to-one multiplicity.

The information gained during the transformation process of this system is used in modelling these multiplicities. During the object identification process, TAGDUR constructs two matrices: one matrix is a procedural usage grid which records the number of times procedure A is called by procedure B and the other matrix is a variable usage grid which records the number of times variable A is accessed within procedure B. These matrixes are used during the object identification process where highly coupled procedures and variables are grouped into classes. These matrices are also used when modeling UML diagrams. Variables or procedures that form attributes/operations of one class, class A, but are accessed by procedures that form operations of another class, class B, are modelled as an association between classes A and B. These two usage matrices are used when modeled the multiplicity of these associations. Using both procedural and variable usage matrices, the number of times that all variables and procedures of object A are accessed by procedures of object B form the type of multiplicity relationship between classes A and B. An example, if the procedures of object B access the variables and procedures of object A ten times, the association between object A and B is modeled as an association of 1:n multiplicity.

3.2 Activity Diagrams

Activity diagrams describe the internal behaviour of a class method as a sequence of steps. These sequence of steps model the dynamic, or behavioural, view of a system in contrast to class diagrams, which model the static, or structural, view of the system.

An activity in UML represents a step in the execution of a business process. Activities are linked by connections, called transitions, which connect an activity to its next activity. The transitions between activities may be guarded by mutually exclusive Boolean conditions. These conditions determine which control flow(s) and activities are selected. [5]

Activity diagrams may contain action states. Action states are states that model atomic actions or operations. Activity diagrams may also contain events. [1]

Activity diagrams can be partitioned into object swimlanes that determine where an activity is placed in the swimlane of the object where the activity occurs. [1]

Tasks that have been determined to be able to execute in parallel by the independent task evaluation step of the transformation process are modelled as parallel activities and flows in the activity diagram while tasks that have been determined to be able to execute sequentially only are modeled as sequential activities and flows. In activity diagrams, synchronization bars are used to synchronise the divergence of sequential activities into parallel tasks or the merging of parallel tasks to a sequential task. These enable the control flow to transition to several parallel activities simultaneously and to ensure that all parallel tasks complete before proceeding to execute the next sequential task. [5]

Our activity diagrams are code-based. Each activity represents an atomic WSL statement.
Because the WSL code lines of a procedure form steps in the execution of this procedure and because individual WSL code lines form an atomic unit of execution, basing activities on individual WSL code lines is a logical basis for the nodes of an activity diagram. Conditions within WSL control constructs, such as WSL's if-then statements, form conditions within the guards that govern the flow of control to activities enclosed by the condition blocks of this WSL control construct.

One might question why TAGDUR chooses to generate activity diagrams from WSL code rather than simply allow the developers to view the WSL or generated C++ code of the transformed system. Activity diagrams were chosen for many reasons. UML is widely understood by many developers while WSL and, to a much lesser extent, C++ has less of a universal understanding. Furthermore, activity diagrams clearly represent the interaction among objects and the occurrences of events among activities; this representation would be much less apparent than if the developer were simply perusing the code of the transformed system. Although our activity diagram is based on WSL code and individual WSL code lines are used to distinguish action states in the activity diagram, this lack of understanding is mitigated by attached comments which describe the WSL code line being modeled. An example, a file I/O event is described in terms of its type (File I/O), sub-type (Read), and destination, source, and index variables. The decision to base our activity diagram on WSL, rather than C++, was due to several reasons, including the fact that WSL is programming and platform independent. An example, a file I/O operation is represented through one type of WSL statement while representing the same operation in C++ may take several C++ statements depending on the type of file being accessed. Consequently, many implementation-specific details that would clutter an activity diagram based on C++ code are avoided if this same activity diagram is based on WSL code instead.

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

Fig 2 :

WSL Code Sample:

```
X : = Y + 1
If X > 4 Then
        Fetch D1, Y2, Z2
Else
        Call B.UpdateRec(X)
Fi
J := D1 + X
M := N + 2        /* potentially parallel operation */
K := 3            /* potentially parallel operation */
```

The following diagram models the following WSL code sample as action states in an activity

diagram. Each action state is labeled by the WSL statement whose entry action the state represents. Each action state is placed in the object swimlane whose object produces the action. An example, the invocation of Class B's UpdateRec method is represented as an action state in the Object B swimlane. Potential parallel operations, such as "M := N + 2" and "K:= 3", are modeled as parallel flows emanating from a fork synchronization bar. An if-then-else WSL construct is modeled in the activity diagram as a branch, with mutually exclusive guard conditions, to two action states. Using these guard conditions, the control flow in the activity diagram is governed in a similar manner to the system that it represents. Potentially parallel executing WSL code lines are modeled as parallel control flows in the activity diagram.

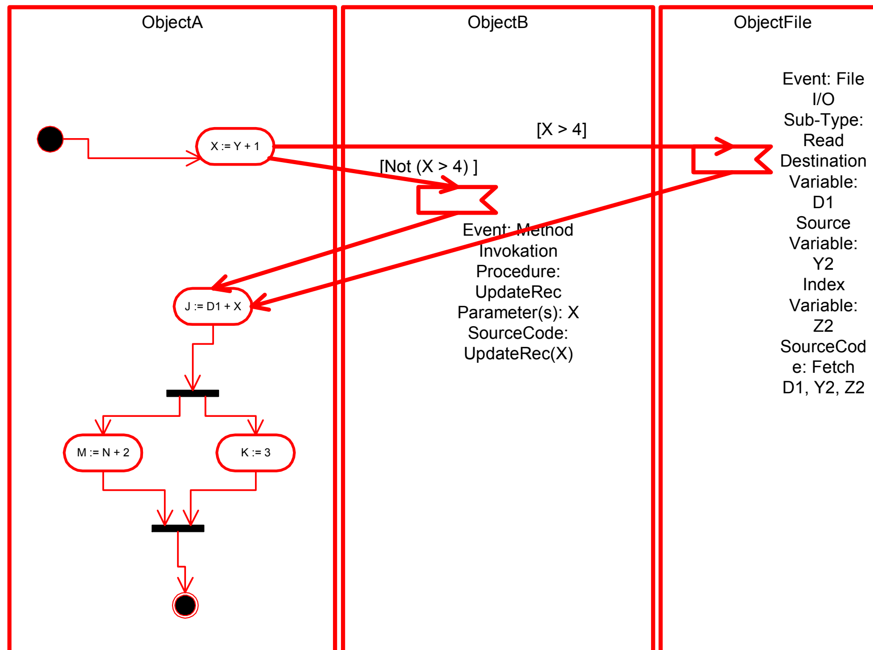

Fig. 3 : Activity Diagram Representation of the WSL Code Sample

4. Conclusion

TAGDUR is a reengineering tool designed to overcome two of the most daring problems of legacy systems; obsolete architecture and lack of documentation. TAGDUR addresses these two problems by first transforming the original procedural legacy architecture to an object oriented one and then TAGDUR models and documents this transformed system through a series of UML diagrams.

Once the transformation process completes, TAGDUR generates a representation of this transformed system as a C++ program. With documentation of this system being provided by TAGDUR as UML diagrams, developers can fully understand the system. With this understanding, developers can adapt this transformed system, now available as a C++ program automatically generated by TAGDUR, to meet new business needs such as system integration.

References

1) Alhir, Sinan Si UML in a Nutshell O'Reilly:Sebastapol, CA, USA, 1998
2) Bjorklund, Dag "The SMDL Statechart Description Language: Design, Semantics, and Implementation" Diploma Thesis, Abo Akademi University, Finland, 2001

3) 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

4) 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

5)Muller, Pierre-Alain Instant UML Wrox: Birmingham, UK, 2000

6) Storey, Margaret-Anne D., Hausi A. Müller, Kenny Wong "**Manipulating And Documenting Software Structures "** Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95) (Nice, France, October 16-20, 1995)

7) Suzuki, Junichi and Yoshikazu Yamamoto "Making UML models interoperable with UXF". Lecture Notes in Computer Science 1618, Springer-Verlag, Heidelberg, Germany

8) Ward, M., "The Syntax and Semantics of the Wide Spectrum Language", *Technical Report*, Durham University, England, 1992.

9) Ward, Martin "Specifications from Source Code -- Alchemists' Dream or Practical Reality?" 4th Reengineering Forum, September 19-21, 1994, Victoria, Canada