A Framework for Transforming Structured Analysis and Design Artifacts to UML

Terrence P. Fries
Coastal Carolina University
P. O. Box 261954
Conway, South Carolina 29528 USA
1+843+349+2676
tfries@coastal.edu

ABSTRACT
The Unified Modeling Language (UML) has become the de facto standard for modeling the architecture and behavior of an object-oriented software system. However, many legacy systems have been documented using non-object structured analysis and structured design. This paper proposes a framework to convert a data flow diagram and an entity relationship diagram into UML artifact, including a use case diagram, sequence diagrams, and a class diagram. The new UML model can be used by analysts or computer aided software engineering tools to implement new object-oriented system. The set of rules for transformation is tested on a structured analysis and design example.

Categories and Subject Descriptors
D.2.2 [**Programming Languages**]: Design Tools and Techniques – *object-oriented design methods, structured programming.*

General Terms
Documentation, Performance, Design, Standardization.

Keywords
UML, structured analysis and design, class diagram, sequence diagram, data flow diagram, entity relationship diagram.

1. INTRODUCTION
The Unified Modeling Language (UML) has become the de facto standard for modeling the architecture and behavior of an object-oriented software system [3,10]. However, many legacy systems have been documented using non-object oriented techniques, primarily structured analysis and structured design (SASD) [5, 7, 19, 20]. The most common artifacts of SASD are the data flow diagram (DFD) and the entity relationship diagram (ERD).

The goal of program understanding is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner [14]. There are many problems in understanding a legacy system. A legacy system may be written in an out-of-date programming language and may contain numerous modification. Frequently, modifications are "kludges" (clumsy or inelegant solutions to a problem) that make the program impossible to comprehend. In addition, newer systems that were documented with SASD techniques rather then UML may suffer from the same problems. Software engineers require UML documentation to aid in the maintenance of these systems. UML diagrams have been shown to be an effective aid in program understanding, although, with limiting factors such as an ill-defined syntax and semantics, spatial layout, and domain knowledge [15].

As legacy systems are converted to use modern object-oriented languages and techniques it is necessary to convert the existing SASD model to UML. Documentation is essential in the understanding of legacy systems. The new UML model can act as the basis for design and implementation of the new system, as well as acting as documentation for the future evolution of the new system. In addition, computer aided software engineering tools can use the UML format for quality assurance and automatic code generation.

Some have attempted to direct conversion of structured code to object-oriented code without considering the design model. For example, Newcomb [9] automated the conversion of a legacy COBOL system to C++. Unfortunately, this does not produce true object-oriented code, since there is no underlying object-oriented design. Instead, the system is composed of procedural code in C++. In addition, there is no resulting documentation of the new system available for future maintenance. Accurate design documentation is an essential component of true object-oriented design.

Several researchers have proposed integrating SASD artifacts with those of UML [1, 6, 17, 18]. They claim that neither methodology alone adequately represents the complete system. Others have proposed an enhanced data flow diagram, called data flow net (DF net), to complement object-oriented methods with functional decomposition [12, 13]. While these proposals for a hybrid methodology may provide accurate documentation of the design, they have an inherent difference in the implementation they produce. Unfortunately, structured artifacts do not readily produce objects, rather, they yield functional, or procedural, based code. Therefore, the non-object oriented components cannot be translated into object-oriented code.

Researchers have proposed some preliminary methods to transform SASD data flow diagrams and entity relationship

SIGDOC'06, October 18-20, 2006, Myrtle Beach, South Carolina, USA.

diagrams to UML artifacts, however, they are either limited in capability or flawed. Bloomfield [2] proposes a method to convert DFDs to UML class diagrams, does not produce UML sequence diagrams which are critical to implementing a software system. Shiroiwa creates a meta model using UML class diagrams and the Object Constraint Language (OCL) provided by the UML 2.0 specification [11]. In this method, the DFD is encoded into the meta model which is then converted into a use case diagram and sequence diagrams. However, this requires an extra step of converting the DFD into the meta model. It also does not address the creation of a class diagram which is crucial to the design workflow.

Tran [16] proposes an approach to transforming DFDs and ERDs from SASD into a UML use case diagram, sequence diagrams, and a class diagram. This proposal provides a starting point with generalizations, but never provides formal rules that could be used to automate the transformation. It also contains several problem areas that produce erroneous UML artifacts. The proposal makes the mistake of treating external entities and data stores as equivalent entities, when, in fact, each has its own procedures and limitations. For example, a DFD data flow from a data store to a process is transformed into a sequence diagram in which the data store initiates a message to the process without any stimulus. This is contrary to the behavior of any normal data store. In practice, a data store can only send information to a process after it has received a query message. Another problem is that actors are added to use case diagram with no external entity counterpart in the DFD. The method requires that some of the transformations be done manually and requires decisions and examination by a person. As a result, the method cannot be automated.

This paper proposes a framework for the transformation of SASD data flow diagrams and entity relationship diagrams into UML 2.0 artifacts. Section 2 presents the transformation framework. Section 3 illustrates and confirms the correctness of the framework by applying it to a test case. Section 4 summarizes the research and discusses future work.

2. TRANSFORMATION FRAMEWORK

Structured analysis and structured design produces four types of artifacts to document functional, or process, models: data flow diagrams (DFD), structure charts, task diagrams, and state models. In addition, SASD uses the entity relationship diagram (ERD) to model data [5, 17, 20].

Structure charts are used for procedural programs to illustrate the partitioning of a program into named modules, or functions. These are not appropriate for transformation to UML since they represent the architectural hierarchy of a procedural-based system. The only functional information they provide are to identify the tasks, which can also be derived from the DFDs.

Task diagrams show detailed interactions between independent threads of execution. Since they show only implementation details of procedures, they are not included in this transformation framework.

The state model describes the states and events in a system using a diagram or table. There are many different types of state diagrams and tables. The most common type of state diagram is a duplicate of the UML state diagram. Therefore, transformation of the state diagram is unnecessary and the state model does not require consideration here.

This research addresses the two primary types of diagrams common to most SASD designs: data flow diagram (DFD) and entity relationship diagram (ERD). The DFD provides functional information about the processes, or tasks, the system performs and the flow of data between processes in the system. The ERD illustrates the data model by showing the data entities and their relationships. For purposes of this paper, all SASD diagrams will use the Yourdon /DeMarco notation. Four types of UML artifacts can be created from the DFD and ERD artifacts: use case diagram, sequence diagrams, state machine diagram, and class diagram.

2.1 Creation of the Use Case Diagram

Processes in a level 1 DFD represent the major tasks the system performs and can be mapped directly into the UML equivalent use cases. DFD external entities are equivalent to UML actors. A data flow between an external entity and a process can be mapped into a line of communication between an actor and a use case which correspond to the DFD external entity and process, respectively. Using this framework, the level 1 DFD in Figure 1 can be transformed into the UML use case diagram shown in Figure 2.

The following rules are used to map the DFD into a use case diagram:

R1.1 For each external entity in the DFD, create an actor in the use case diagram.

R1.2 For each process in the DFD, create a use case.

R1.3 For each data flow between an external entity and a process, create an association, or line of communication, between the corresponding actor and use case.

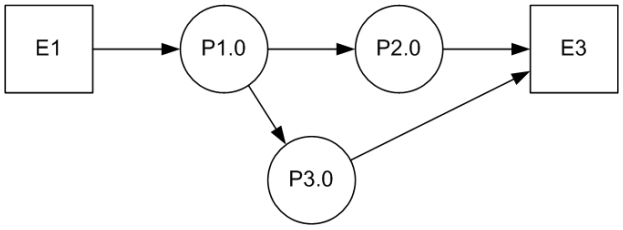

Figure 1. Level 1 DFD.

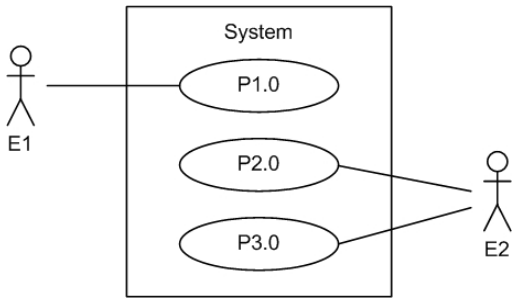

Figure 2. Use case diagram transformed from the DFD.

2.2 Creation of the Sequence Diagrams

In UML, a sequence diagram is drawn for each use case. Since each use case corresponds to a process in the DFD, the flows into and out of the process can be used to develop a single sequence diagram for that use case. A process must have at least one flow in and one flow out. Each of the flow configurations for a particular process can be transformed into a sequence diagram component and then the components are combined. Data flows into a DFD process can originate from an external entity, a data store, or another process. Similarly, a data flow out of a process can go to an external entity, a data store, or another process. As a result, data flows can connect components in five different configurations:

- external entity to process
- data store to process
- process to external entity
- process to data store
- process to process

A data flow connecting components in any other configuration is invalid. Each valid configuration can be transformed into a component of a sequence diagram using a predefined set of rules.

2.2.1 External entity to process configuration

When a data flow into a process originates from an external entity, the external entity can be transformed into a UML actor, the process becomes an object representing the system, and the data flow becomes the parameter of a message sent to the process as shown in Figure 3. The object representing the system is frequently a manager object that coordinates the other objects in the system, equivalent to an interface. While it may appear unusual that an actor is initiating a call to an object operation, the message is actually sent by a user interface with which the actor interacts. In earlier stages of development, the user interface is implicit. It is not until later in the design process that details of the user interface are added to the sequence diagram.

The rules for transforming an entity to process configuration into a sequence diagram component are:

R2.1 Transform the external entity in the DFD into an actor lifeline in the sequence diagram.

R2.2 Transform the process in the DFD into an object lifeline which represents the System in the sequence diagram.

R2.3 Add a message from the actor to the object. Transform the data flow into an object and use this as the parameter in the message.

2.2.2 Data store to process configuration

The few early attempts at transforming a data flow from a data store to a process, treated the data store the same as an external entity [16]. This produced a transformation as shown in Figure 4 in which the data store sends a message to the system without a prior request. There are two reasons for the error in this. First, unlike an external entity, a data store cannot autonomously decide to send a message to a process without a trigger from the process. In some DFDs, the query is explicitly shown and, in others, the query is implicit. Second, a data store object is a subordinate object to the system manager object and UML 2.0 standards require that subordinate objects be placed to the right of the dominant object in a sequence diagram [3, 10]. An accurate depiction of the sequence diagram explicitly shows the system initiating a query and the object representing the data store responding as shown in Figure 5. The object representing the data store handles the details of actual communication with the external file or database to hide its format.

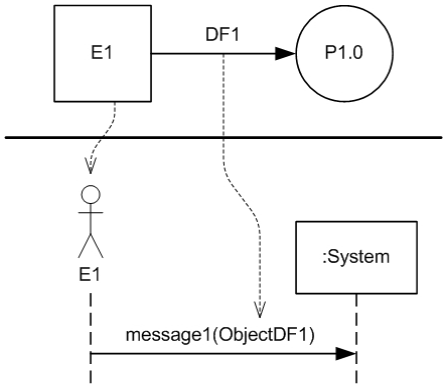

Figure 3. Transformation of external entity to process DFD.

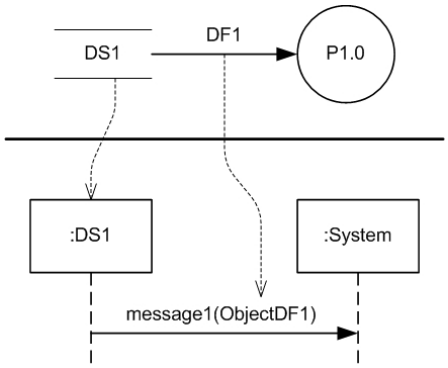

Figure 4. Incorrect transformation of data store to process.

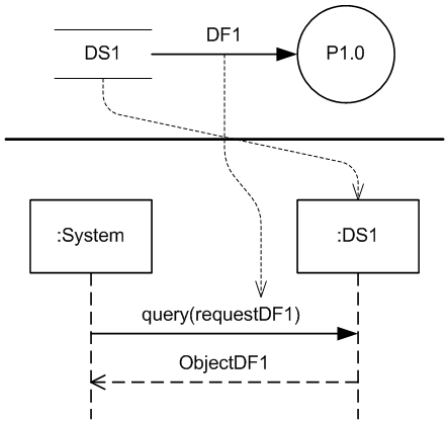

Figure 5. Transformation of data store to process DFD.

The correct rules for this transformation are:

R3.1 Transform the data store in the DFD into an object lifeline in the sequence diagram.

R3.2 Transform the process in the DFD into an object lifeline in the sequence diagram.

R3.3 Add a query message from the system to the object regardless of whether the query is implicit or explicit in the DFD. Transform the data flow into an object and use this as the parameter in the query.

R4.4 Add a return message from the data store object to the system that returns the object equivalent of the data flow.

2.2.3 Process to external entity configuration

A data flow configuration from a process to an external entity can be transformed in a manner similar to the first case in Section 2.2.1. Using this configuration, the process becomes the system manager, the external entity is transformed into an actor, and the data flow is converted into the parameter in the message as shown in Figure 6. The set of rules governing this transformation are:

R4.1 Transform the process in the DFD into an object lifeline in the sequence diagram.

R4.2 Transform the data store in the DFD into an object lifeline in the sequence diagram.

R4.3 Add a message from the actor to the object. Transform the data flow into an object and use this as the parameter in the message.

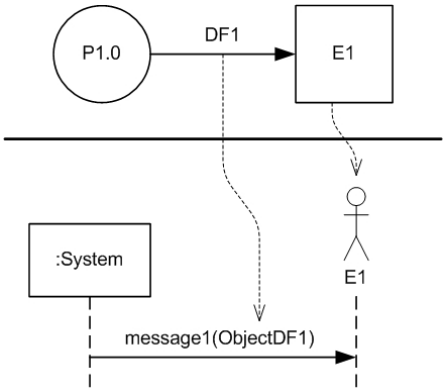

Figure 6. Transformation of process to external entity DFD.

2.2.4 Process to data store configuration

In the case of a data flow from a process to a data store, the configuration can be transformed as shown in Figure 7. In this transformation, the process and data store are transformed into the system manager and an object representing the data store, respectively. The data flow becomes the parameter in the message.

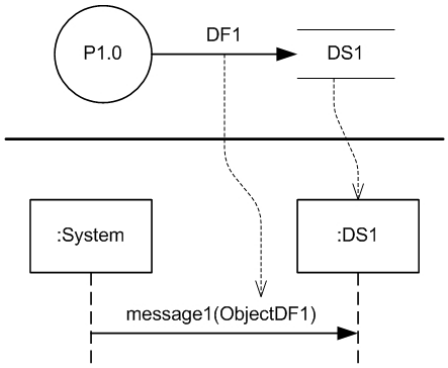

Figure 7. Transformation of process to data store DFD.

2.2.5 Creating the sequence diagram

Any valid DFD will not contain a process with a single flow in or out as shown in the examples above. The transformations described above should be performed for each data flow into or out of a particular process and the results of each combined.

To create a sequence diagram for each use case:

R5.1 Locate the corresponding process in the DFD

R.5.2 For each data flow in or out of the process, use the appropriate transformation rules to add actors and/or objects, if necessary, and to add messages to the sequence diagram.

2.3 Creation of the State Machine Diagram

The final data flow configuration to be address is from one process to another process. Since each process is transformed into a use case and each sequence diagram represents a use case, a single sequence diagram cannot be used to represent this configuration. When one process sends a data flow to another, it indicates an order dependency which are best represented in in UML by a state machine diagram. A UML state machine is used for modeling the history of a single reactive object. In this case, the reactive object is the entire system. The transformation of a process to process data flow is shown in Figure 8.

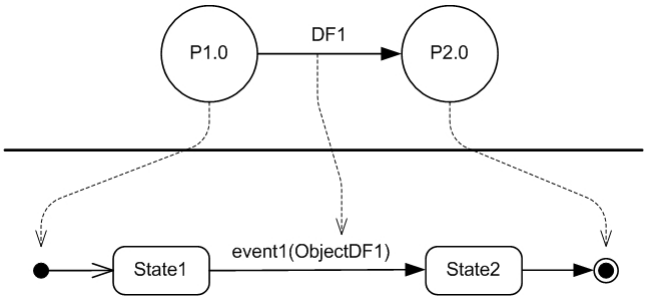

Figure 8. Transformation of process to process DFD.

2.4 Creation of the Class Diagram

There are several commonly used notations for representing an ERD. For this paper, the Chen-style ERD notation will be used

[4]. Both UML class diagrams and ERDs represent entities and their relationships. This provides an intuitive transformation of the ERD to a class diagram: ERD entities become UML classes and ERD attributes become attributes of the corresponding UML class. Cardinalities on the ERD can be used in conjunction with the modalities to produce multiplicities for the UML class diagram. For example, a cardinality of 1 becomes a multiplicity of 1 or 0..1 for mandatory and optional modalities, respectively. Similarly, a cardinality of N becomes 1..* or 0..* for mandatory and optional modalities, respectively.

UML has the additional feature of representing entity behaviors. While it one could use the relationship as an operation in one of the classes, unfortunately, there is no commonly accepted standard for naming the relationship in an ERD. As a result, it is impossible to predict for which class the name of the relationship is an appropriate operation. The transformation of ann ERD to a class diagram is shown in Figure 9.

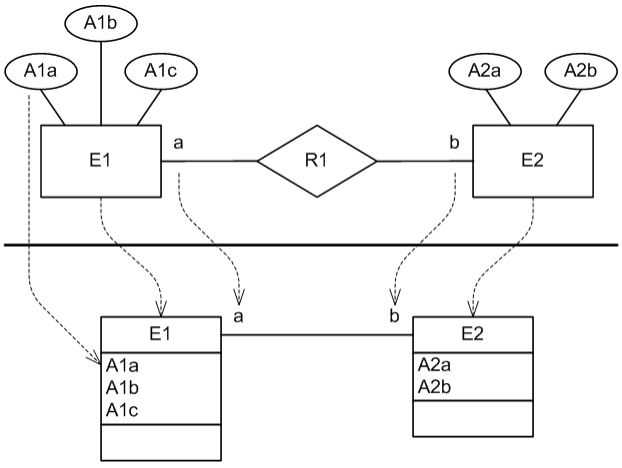

Figure 9. Transformation of ERD to class diagram.

An ERD may also show specialization, or subtype, relationship as an "is-a" relationship which can be transformed into the UML class diagram as a generalization as shown in Figure 10. However, one must be careful that the ERD "is-a" relationship is actually a generalization and not a role. It is poor object-oriented design to use inheritance for a role-played-by relationship because an object of the derived class cannot change class during its lifetime. For example, subtypes of Manager and Programmer of the supertype Employee do not allow a Programmer object to change to a Manager object if the person is promoted. In this case, Manager and Programmer are roles played by an Employee.

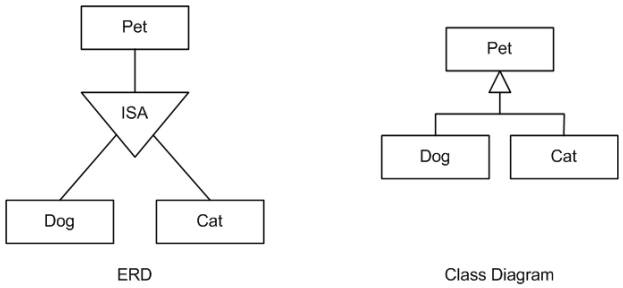

Figure 10. ERD subtype to UML generalization

The set of rules to transform an ERD into a class diagram are:

R6.1 Transform each entity in the ERD into a class. Add the entity's attributes to the class attribute compartment.

R6.2 Transform for each relationship in the ERD, add an association between the corresponding classes. Convert the ERD cardinalities and modalities for the relationship to multiplicities in the class diagram.

R6.3 Convert subtype relations in the ERD to generalizations in the class diagram. Caution: ensure that the subtype relation is actually an ISA relation.

3. CASE STUDY

A case study is used to illustrate the applicability of the proposed framework for transforming SASD artifacts to UML. The case study uses the ubiquitous Hoosier Burger inventory control system [8]. This example was chosen because it provides both a DFD and an ERD whose accuracy is widely accepted. The Hoosier Burger DFD and ERD are shown in Figures 11 and 12, respectively.

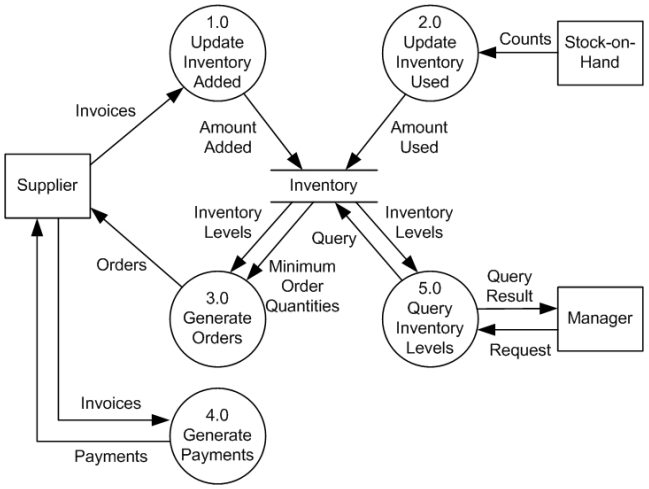

Figure 11. Hoosier Burger inventory control system DFD.

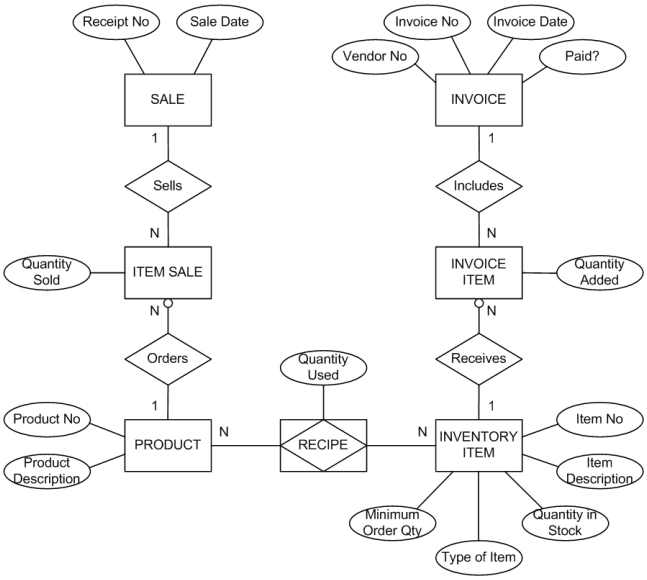

Figure 12. Hoosier Burger inventory control system ERD.

Using the transformation rules, the use case diagram in Figure 13 was created. Figure 14 shows the DFD portion for process 1.0 (Update Inventory Added), the resulting sequence diagram for resulting use case, and the path of transformation for the various components. The sequence diagram for the Update Inventory Used use case in Figure 15 is developed in the same manner.

In developing the sequence diagram for the Generate Orders use cases as shown in Figures 16, a decision had to be made about which data flow from the data store to the process takes place first. While the Inventory Levels data flow is above the other in the DFD, it does not necessarily imply order. There is no guarantee that the DFD author intends a particular ordering of data flows by their vertical or horizontal position because position is often chosen to reduce complexity in the diagram. The sequence diagram for the Generate Payments use case is shown in Figure 17.

The Query Inventory Levels use case requires a decision as to which entity initiates the sequence, the Inventory data store or the Manager object, since data flows in and out of both. Since a data store cannot initiate an operation, the Manager object must be the first to act. With this decision, the transformation provides the sequence diagram in Figure 18.

The class diagram developed with transformation rules is shown in Figure 19.

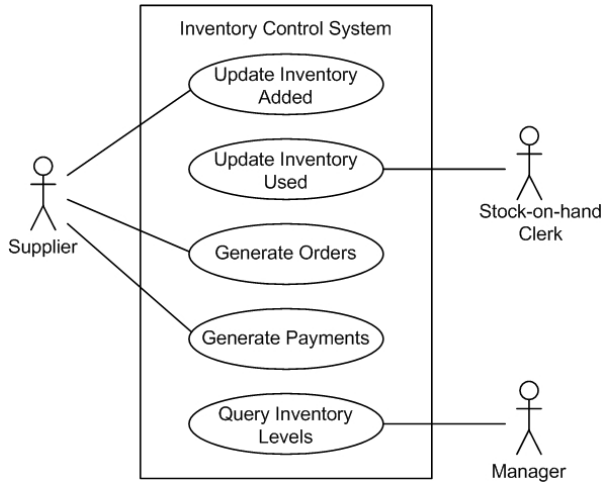

Figure 13. Use case diagram for Inventory Control System.

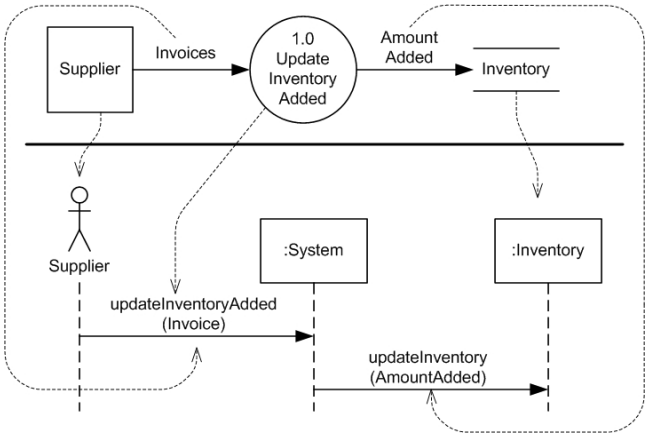

Figure 14. Sequence diagram for "Update Inventory Added" use case.

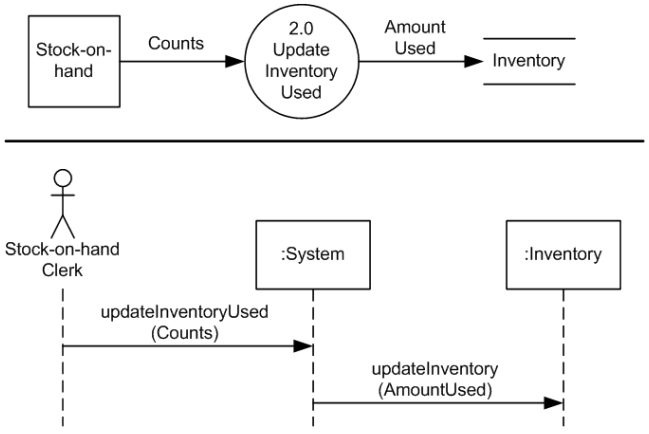

Figure 15. Sequence diagram for "Update Inventory Used."

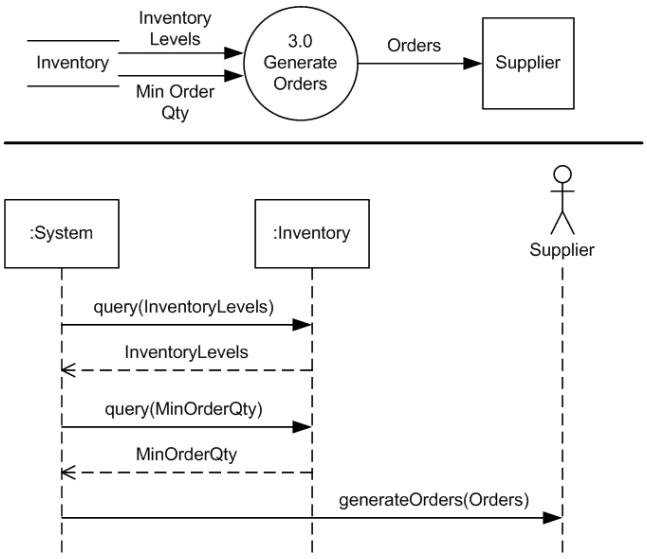

Figure 16. Sequence diagram for "Generate Orders."

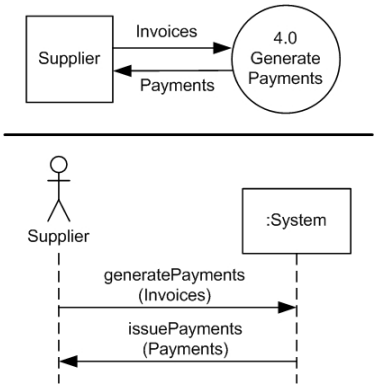

Figure 17. Sequence diagram for "Generate Payments."

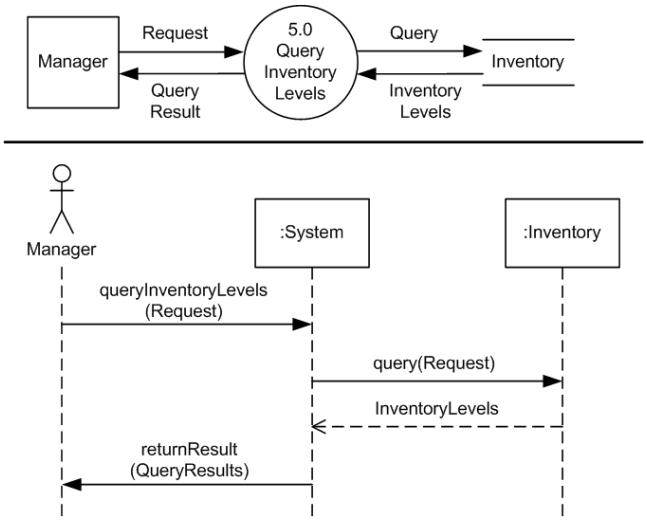

Figure 18. Sequence diagram for "Query Inventory Levels."

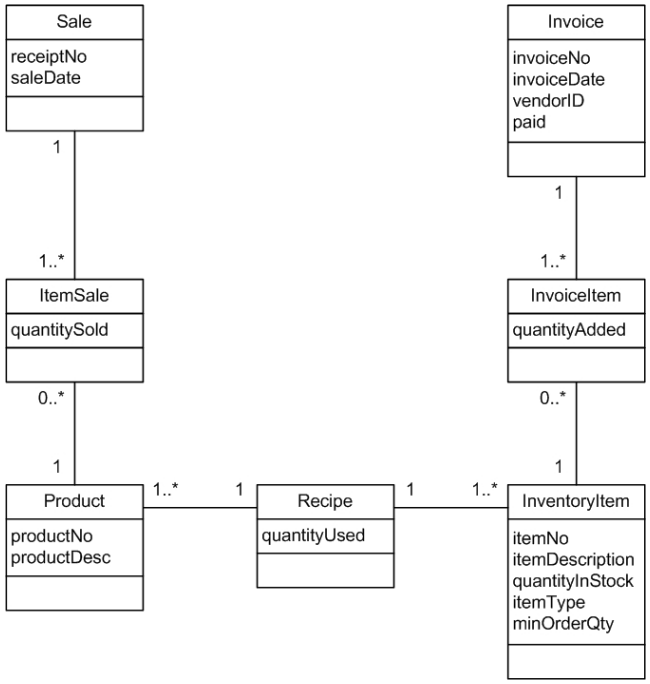

Figure 19. Class Diagram for Hoosier Burger

4. CONCLUSIONS

This paper has proposed a framework for transforming the structured analysis and design artifacts, DFD and ERD, into the UML artifacts. The set of transformation rules has been shown in the test case to create an accurate use case diagram, sequence diagrams, and class diagrams. The framework provides UML artifacts for documentation purposes. This documentation will aid software engineers in developing a new object-oriented system and in evolving the system over its lifetime.

The only problem encountered with the transformation rules is that occasionally a manual decision must be made, such as which of two flows should be first. This limits the ability to automate the set of rules. However, with user guidance or embedded artificial intelligence, this can be overcome.

Further research includes implementation of the embedded intelligence and the automation of this set of rules.

5. REFERENCES

[1] Alabiso, B. Transformation of data flow analysis models to object oriented design. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '88)* (San Diego, CA, Sept. 25-30, 1988). ACM Press, New York, NY, 2000, 335-353.

[2] Bloomfield, T. MDA, meta-modelling, and model transformation. In *Proceedings of the First European Conference on Model Driven Architecture (ECMDA '05) (LNCS 3748)* (Nuremberg, Germany, Nov. 7-10, 2005) Springer, Heidelberg, Germany, 2005.

[3] Booch, G., Jacobson, J., Rumbaugh, I. *Unified Modeling Language User Guide, 2nd Edition*. Addison Wesley, Upper Saddle River, NJ, 2005.

[4] Chen, P. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Systems., 1,* 1 (Mar. 1976), 9-36.

[5] DeMarco, T. *Structured Analysis and System Specification*. Yourdon Press, New York, NY, 1978.

[6] Fernandes, J. M. and Lilius, J. Functional and object-oriented views in embedded software modeling. In *Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS '04)* (Brno, Czech Republic, May 24-27, 2004). IEEE Press, New York, NY, 2004, 378-387.

[7] Gane, C. and Sarson, T. *Structured Systems Analysis and Design*. Improved Systems Technologies, Inc., New York, NY, 1977.

[8] Hoffer, J. A., George, J. F., and Valacich, J. S. Modern Systems Analysis and Design, 4th Edition. Prentice Hall, Englewood Cliffs, NJ, 2005.

[9] Newcombe, P. and Doblar, R. A. Automated transformation of legacy systems. *CrossTalk., 14,* 12 (Dec. 2001), 18-22.

[10] Rumbaugh, I., Jacobson, J., Booch, G. *Unified Modeling Language Reference Manual, 2nd Edition*. Addison Wesley, Upper Saddle River, NJ, 2004.

[11] Shiroiwa, M., Miura, T., and Shioya, I. Meta model approach for mediation. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC '03)* (Dallas, TX, Sept. 30-Oct. 3, 2003). IEEE Press, New York, NY, 2003, 480-485.

[12] Tan, H. B. K., Yang, Y., and Blan, L. Systematic transformation of functional analysis model in OO design and implementation. *IEEE Trans. on Software Engineering., 32,* 2 (Feb. 2006), 111-135.

[13] Tan, H. B. K. and Weihong, L. Systematic bridging the gap between requirements and OO design. In *Proceedings of the 17th IEEE International Conference Automated Software Engineering (ASE '02)* (Edinburgh, UK, Sept. 23-27, 2004). IEEE Press, New York, NY, 2002, 249-252.

[14] Tilley, S. The canonical activities of reverse engineering. *Annals of Software Engineering, 9*, 1-4 (2000), 249-271.

[15] Tilley, S., and Huang, S. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. *Proceedings of the 21st Annual International Conference on Documentation* (SIGDOC 2003) (San Francisco, CA, Oct. 12-15, 2003). 184-191.

[16] Tran, T. N., Khan, K. M., and Lan, Y.-C. A framework for transforming artifacts from data flow diagrams to UML. In *Proceedings of the 2004 IASTED International Conference on Software Engineering* (Innsbruck, Austria, Feb. 17-19, 2004). ACTA Press, Calgary, AB, Canada, 2004.

[17] Truscan, D., Fernandes, J. M., and Lilius, J. Tool support for DFD-UML model-based transformations. In *Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS '04)* (Brno, Czech Republic, May 24-27, 2004). IEEE Press, New York, NY, 2004, 388-397.

[18] von Konsky, B. R., Robey, M., and Nair, S. Integrating deisgn formalisms in software engineering education. In *Proceedings of the 17th IEEE Conference on Software Engineering Education and Training (CSEET '04)* (Norfolk, VA, March 1-3, 2004). IEEE Press, New York, NY, 2004, 78-83.

[19] Yourdon, E. *Modern Structured Analysis*. Yourdon Press, Upper Saddle River, NJ, 1989.

[20] Yourdon, E. and Constantine, L. L. *Structured Design: Fundamentals and Applications in Software Engineering*, 2nd Edition. Yourdon Press, Englewood Cliffs, NJ, 1989.