Migrating Functional Requirements in SSUCD Use Cases to a More Formal Representation

Mohamed El-Attar
Information and Computer Science Department
King Fahd University of Petroleum and Minerals
P.O. 5066, Al Dhahran 31261, Kingdom of Saudi Arabia
melattar@kfupm.edu.sa

James Miller
STEAM Laboratory
Department of Electrical and Computer Engineering
University of Alberta, Edmonton, Alberta, Canada
jm@ece.ualberta.ca

***Abstract*- Use case modeling is a popular technique to elicit and model functional requirements of a software development project. In a use case driven development methodology, use cases are used as a basis to guide the development of UML design models. In this paper, we provide a model transformation approach to transform use cases descriptions written in a nearly unstructured form to a more formal representation. A more formal representation, which is machine-readable, can be used to systematically generate other UML design models, in particular UML activity diagrams. The main advantage of using this model transformation approach is to avoid potential errors introduced by modelers if they were to develop the UML design models while depending solely on their skill and experience. The proposed model transformation approach is applied to a library system to demonstrate its applicability and to validate its correctness and effectiveness.**

Keywords – Use Cases; SSUCD; SUCD; Model Transformation.

I. INTRODUCTION

Use case diagrams [3, 6] have become the de-facto modeling tool to elicit and model functional requirements for object-oriented software development projects. In a use case driven development methodology, the use case model is developed at the analysis phase used to drive the development of other UML (Unified Modeling Language) [12] design artifacts at the design phase. This is process is far from straightforward sine naturally there is a gap between the analysis and design phases. If the development of UML design artifacts based on use case models is dependent solely on human skill, experience and judgment, then there will be a great risk of developing design artifacts that have a design view which is inconsistent with the analytical view as presented by the use case model. As a result, system architects may construct a design that provides different functionality than that required (i.e., developing the 'wrong' system), leading to costly reworks and schedule overruns, in addition to the intangible cost of unsatisfied customers.

Model transformation provides a more rigorous approach towards developing UML design artifacts based on use case models. Model transformation greatly reduces the human factor during the development process thus increasing the likelihood of developing a system that satisfies its prescribed functional requirements. To this end, this paper presents a model transformation approach that transforms use cases written in a form named SSUCD (**S**imple **S**tructured **U**se **C**ase **D**escriptions) [2] to another more formal representation named SUCD (**S**tructured **U**se **C**ase **D**escriptions) [5]. SUCD is a language first introduced in [5] that is used to structure use case descriptions by embedding enough structure within the use case descriptions to facilitate the transformation of workflows in use case descriptions into UML activity diagrams. Use cases are ideally written by business analysts. In [1], an experiment was conducted which revealed that the language SUCD was too difficult to be used by its potential users (business analysts). The experiment indicated that when using SUCD, the majority of defects detected in the models developed were due to syntax errors resulting from using of the SUCD language. Consequently, the authors of the SUCD language developed a simplified version of SUCD, which is SSUCD [2]. SSUCD was intentionally designed to accessible to business analysts. The usability of SSUCD was empirically evaluated in [1]. The results of the experiment indicate that users of SSUCD develop higher quality use case models. SSUCD was also intentionally designed to help business analysts develop use case models that are consistent to combat the issue of developing inconsistent use case models when not utilizing any structure.

The remainder of the paper is organized as follows: Section 2 briefly outlines the related work and provides an introduction to the SSUCD and SUCD languages. The proposed model transformation technique is detailed in Section 3. In Section 4, a library system case study is used to evaluate the correctness and effectiveness of the proposed transformation technique. Finally, Section 5 concludes and discusses future work.

II. BACKGROUND AND RELATED WORK

There exist two tools that automate the generation of activity diagrams from use case models such as "Catalyze Suite" [8] and "TopTeam Analyst" [9]. Both tools produce diagrams similar to UML activity diagrams, which their developers refer to as 'Flow diagrams'. However, such tools and methods depend on the utilization of use case descriptions with no structure, meaning that the source use case descriptions used are vulnerable to inconsistencies. If inconsistent use case models are used as a source to generate UML activity diagrams, therefore these inconsistencies will propagate onwards to the UML activity diagrams, which in turn will propagate to the implementation source code where the cost of fixing such inconsistency error escalates significantly. Therefore, tools that generate UML activity diagrams should be geared towards using the SSUCD language to ensure that the source use case models used are consistent. This approach presented in this paper uses use cases written in the SSUCD form to contribute towards the

 ISBN: 978-1-61208-165-6

overall goal of generating UML activity diagrams that provide a consistent and correct view of the system's functional requirements.

III. A BRIEF BACKGROUND TO SSUCD AND SUCD

The model transformation approach proposed in this paper depends on using SSUCD use case descriptions as input and produced SUCD use case descriptions as output. As a prelude to outlining the model transformation mapping rules and algorithms shown in Section 3, it is necessary to briefly introduce the SSUCD and SUCD languages; the components used in the model transformation. To this end, this section provides a brief introduction to SSUCD and SUCD using a use case description of a system outlined in [5]. The use case is concerned with the functionality of borrowing a book from a library. SSUCD and SUCD use cases do not mandate any particular template to be used. SSUCD and SUCD use cases however require a minimal set of fields to be present in a use case description. The fields required are the (a) Use Case Name section, (b) the Associated Actors section, (c) the Description section, and (d) the Extension Points section. SUCD use cases, being more formal that SSUCD, do contain further subsections within some sections of its template. For example, in the Extension Points section, SUCD use cases case outline Public Extension Points and Private Extension Points. A detailed description of the SSUCD and SUCD languages are out of the scope of this paper. For detailed descriptions of the SSUCD and SUCD languages as well as their formal syntax, our interested readers are referred to [2] and [5], respectively. However, to illustrate the difference between using both languages, Figures 1 and 2 show the textual description of the "Borrow Book" use case using SSUCD and SUCD, respectively.

Use Case Name:
Borrow Book

Brief Description:
This use case is initiated by a Member to allow that member to borrow a book. A Librarian is then involved to carry out the transaction.

Preconditions:
The book must exist

Basic Flow:
The use case begins when a member brings a book they would like to borrow. Information about the book is then retrieved from the database by entering the book's name or barcode. The member then provides their library card for the librarian to scan. The librarian needs to authenticate first before scanning the book's barcode. The librarian then updates the member's record with the newly borrowed book. The book's status is then changed in the database and set as 'Borrowed'.

Alternative Flow:
When the librarian scans the book's barcode, if the barcode cannot be scanned, then the book's barcode is entered manually.

Postconditions:
The number of borrowed books in the member's record is increased by one

Extension Points:
Balance overdue

Fig. 1. The description of the Borrow Book use case described in SSUCD

Use Case Name:
Borrow Book

Brief Description:
This use case is initiated by a Member to allow that member to borrow a book. A Librarian is then involved to carry out the transaction.

Preconditions:
The book must exist

Basic Flow:

{BEGIN Use Case}

{BEGIN bring book to borrow}
• Member -> Brings the book he/she would like to borrow
• PERFORM Retrieve book information (2)
• Member -> Provides library card
• Librarian -> Scans member's card
{END bring book to borrow}

{BEGIN authenticate librarian}
• INCLUDE Authenticate Librarian (1)
{END authenticate librarian}

{BEGIN scan book}
• Librarian -> Scan's book's barcode
RESUME {update member's record} {update book's status} (5)
{END scan book}

{BEGIN update member's record}
• Librarian -> Updates the Member's record with the newly borrowed book
RESUME {END}
{END update member's record}

{BEGIN update book's status}
• SYSTEM -> Changes the book's status in the database to 'Borrowed'
{END update book's status}

{END Use Case}

Alternative Flows:
FLOW Basic Flow (3)
AT {scan book} (4)
• Librarian -> Scans the book's barcode
IF barcode cannot be scanned
{BEGIN enter barcode manually}
• Librarian -> Enters the book's barcode number manually

 ISBN: 978-1-61208-165-6

{END enter barcode manually}
CONTINUE {update member's record} {update book's status}
Subflows:

SUBFLOW Retrieve book information
{BEGIN enter and retrieve book information}
• Librarian -> enters the book's name or barcode
• SYSTEM -> retrieve the given book's information from database
{END enter and retrieve book information}

Postconditions:
The number of borrowed books in the member's record is increased by one

PUBLIC EXTENSION POINT
Balance overdue

Fig. 2. The description of the Borrow Book use case described in SUCD

It can be easily deduced from Figures 1 and 2 that SUCD use case descriptions contain far more structure that SSUCD use cases. This is the chief motivation behind this work. Due to the complexity of this transformation problem, if the transformation is performed manually then there will be a great risk of developing SUCD use cases that are inconsistent with their corresponding source SSUCD use cases.

IV. TRANSFORMING USE CASES FROM SSUCD TO SUCD

This section describes the preparation activities requisite for the transformation process to take place. In this section, we also present the model transformation rules and algorithms. Automation support is important to ensure the syntactical correctness of the models used and created and to ensure the speed and accuracy of the application of the transformation process and therefore the transformation rules and algorithms were coded using two popular tools within the model transformation research community.

A. Generating the Metamodels of SSUCD and SUCD

A model transformation process uses a source model to produce a target model based on the transformation rules and algorithm. As a prerequisite to the execution of the transformation, the source model needs to conform to a metamodel. The conformance of the source model to a metamodel ensures the syntactical correctness of the source model. Similarly, the target model also needs to conform to metamodel to ensure the syntactical correctness of the produced model. The derivation of the metamodels for both source and target models were reverse engineered from the E-BNF rules for both SSUCD and SUCD, respectively. The E-BNF rules for SSUCD and SUCD are defined in [2] and [5], respectively. The metamodels were reverse engineering using a tool named ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) [7]. ANTLR is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages [7]. The high-level metamodels for SUCD and SSUCD are shown in Figures 3 and 4, respectively.

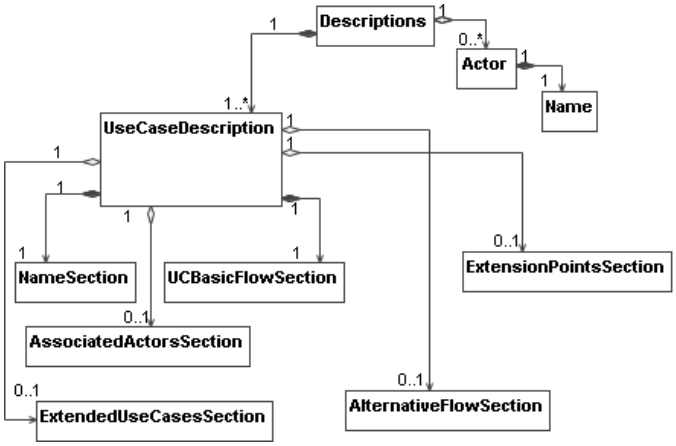

Fig. 3. The high-level components of the SUCD metamodel.

As shown in Figures 3 and 4, the SSUCD and SUCD metamodels formats the use case descriptions into several sections represented as objects. Each section describes a certain important aspect of the use case. For example, “AssociatedActorsSection” object is used to describe the list of actors associated with the given use case. It can be shown that the SSUCD metamodel is a simplified version of SUCD as the metamodel of SSUCD is a subset of the SUCD metamodel. ANTLR generated a compiler that can parse SSUCD use case descriptions. The compilation process results in the generation of an object model that represents the given SSUCD use cases.

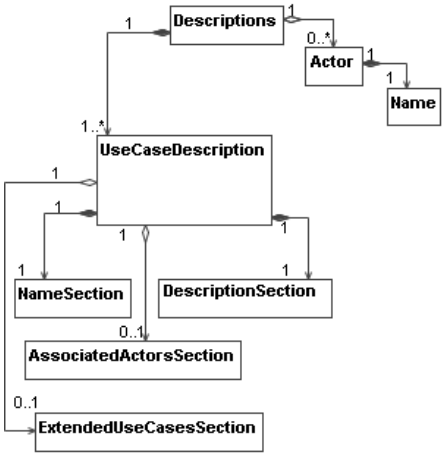

Fig. 4. The high-level components of the SSUCD metamodel.

Transformation Mapping Rules and Algorithms

The transformation and mapping rules and algorithms prescribe the process through which a source model is transformed into a target model. In the scope of model-driven engineering, the transformation mapping rules and algorithms are defined in terms of model, which in turn must conform to a metamodel. The ATL (Atlas Transformation Language) metamodel was chosen as the metamodel for model transformation problem considered in this paper. ATL was selected since it provides two methods to describe

 ISBN: 978-1-61208-165-6

transformation rules: (a) using "matching rules" (declarative programming); and (b) using "called rules" (imperative programming). For the transformation problem at hand it was necessary to use both types of programming methods provided by ATL as complex transformation algorithms can be too difficult to program declaratively only [10]. Figures 5→8 outline the mapping rules and algorithms of the proposed model transformation technique per description section.

Use Case Name Section	
SSUCD	**SUCD**
The "Use Case Name" section starts with the label "Use Case Name:" ***MAPPING1:*** *Every use case in the model must have a name and therefore this section must exist in every use case description*	The "Use Case Name Section" starts with the label" Use Case Name:" ***MAPPING1:*** *Every use case in the model must have a name and therefore this must exist in every use case description.*
If the use case is abstract then this section is followed by the keyword "ABSTRACT" Use case name as-is in free flow NL ***MAPPING2:*** *Use case name must be unique in the entire model. No two use cases can have the same name.*	If the use case is abstract, this section then is followed by the keyword "ABSTRACT" Use case name as-is in free flow NL ***MAPPING2:*** *Use case name must be unique in the entire model. No two use cases can have the same name.*
Transformation Rule: The use case names must be exactly the same in both SSUCD and SUCD.	
If the use case is implementing an abstract use case, the keyword "IMPLEMENTS" is shown followed by the name of the abstract use case. Any additional abstract use cases which the given use case implements, is stated by using a comma followed by the name of the other abstract use cases. For example: IMPLEMENTS UseCaseA, UseCaseB, UseCaseC ***MAPPING3:*** *Use cases that are implemented must exist in the target model.* ***MAPPING4:*** *Use cases that are implemented must be abstract. In other words, they should have the keyword "ABSTRACT" in their "Use Case Name" section.*	If the use case is implementing an abstract use case, the keyword "IMPLEMENTS" is shown followed by the name of the abstract use case. Any additional abstract use cases which the given use case implements, is stated by using a comma followed by the name of the other abstract use cases. For example: IMPLEMENTS UseCaseA, UseCaseB, UseCaseC ***MAPPING3:*** *Use cases that are implemented must exist in the target model.* ***MAPPING4:*** *Use cases that are implemented must be abstract. In other words, they should have the keyword "ABSTRACT" in their "Use Case Name" section.*
Transformation Rule: The names of the implemented use cases in both SSUCD and SUCD must match. Both SSUCD and SUCD must include the keyword ABSTRACT.	
If the use case is specializing a concrete use case, the keyword "SPECIALIZES" is shown followed by the name of the concrete use case. Any additional concrete use cases which the given use case specializes, is stated by using a comma followed by the name of the other concrete use cases. For example: SPECIALIZES UseCaseA, UseCaseB, UseCaseC ***MAPPING5:*** *Use cases that are specialized must exist in the target model.*	If the use case is specializing a concrete use case, the keyword "SPECIALIZES" is shown followed by the name of the concrete use case. Any additional concrete use cases which the given use case specializes, is stated by using a comma followed by the name of the other concrete use cases. For example: SPECIALIZES UseCaseA, UseCaseB, UseCaseC ***MAPPING5:*** *Use cases that are specialized must exist in the target model.*
MAPPING6: *Use cases that are specialized must NOT be abstract. In other words, they should NOT have the keyword "ABSTRACT" in their "Use Case Name" section.*	***MAPPING6:*** *Use cases that are specialized must NOT be abstract. In other words, they should NOT have the keyword "ABSTRACT" in their "Use Case Name" section.*
Transformation Rule: The names of the parent use cases in both SSUCD and SUCD must match. Both SSUCD and SUCD must NOT include the keyword ABSTRACT.	

Fig. 5. Transforming the "Name Section"

Figure 5 outlines the mapping rules for the "Name Section". The purpose of "Name Section" is mainly to specify the name of the use case. The "Name Section" is also used to specify if the use case is abstract or concrete. Moreover, the "Name Section" is also used to specify if the use case is *generalizing* or *specializing* another use case. The transformation process of the "Name Section" can be fully automated since this section is very similar in the SSUCD and SUCD forms.

Associated Actors Section	
SSUCD	**SUCD**
The "Associated Actors" section starts with label "Associated Actors:" ***MAPPING7:*** *If the use case does not have any actors associated with it then this section is removed entirely.*	The "Associated Actors" section starts with label "Associated Actors:" ***MAPPING7:*** *If the use case does not have any actors associated with it then this section is removed entirely.*

 ISBN: 978-1-61208-165-6

Any associated actors are then listed (comma separated) in a new line as such: ActorA, ActorB, ActorC. ***MAPPING8:*** *Actors listed in this section must exist in the model. In other words, there must be actor descriptions with the stated actor names in the "Actor Name" section.*	Any associated actors are then listed (comma separated) in a new line as such: ActorA, ActorB, ActorC. ***MAPPING8:*** *Actors listed in this section must exist in the model. In other words, there must be actor descriptions with the stated actor names in the "Actor Name" section.*
Transformation Rule: The names of the actors listed in both SSUCD and SUCD must match.	

Fig. 6. Transforming the "Associated Actors Section"

Figure 5 outlines the mapping rules for the "Associated Actors Section". The purpose of "Associated Actors Section" is mainly to specify the names of any actors involved with the use case. Once again the "Associated Actors Sections" of the SSUCD and SUCD forms are very similar hence the transformation is straightforward and fully automated.

Description Section	
SSUCD	**SUCD**
The "Description" section starts with label "Description:" ***MAPPING9:*** *Every use case must have a description. Therefore, every use case must have a "Description" section.* The "Description" section in SSUCD is populated with the free flow Natural Language. The only structure involved in this section is the use of the "INCLUDE" keyword. The "INCLUDE" keyword is used to indicated other use cases which the given use case includes. The "INCLUDE" keyword is embedded within the free flow text. It is used by showing the keyword "INCLUDE" followed by two angled brackets (< >). The name of the included use case is stated between the angled brackets. For example: Description: Free-flow text, free-flow text..... INCLUDE <UseCaseA> free-flow text, free-flow text... ***MAPPING10:*** *The name of stated inclusion use case (the included use case) must exist in the mo*	The "Description" section starts with label "Description:" ***MAPPING10:*** *Every use case must have a description. Therefore, every use case must have a "Description" section.* The "Description" section then must have a "Basic Flow:" label. The heart of "Description" section basically consists of "headers" which contain "actions". An "action" basically consists of a bullet point, followed by the name of the actor performing the action then followed by an arrow → then followed by the action description written in natural language. For example: • Librarian → Enter member's
Transformation Rule: The conversion of this section will be semi-automated. There are only two rules to consider when converting this section. First, the actor names used in SUCD must be listed in the "Associated Actors" section of SSUCD (apart from the SYSTEM actor). Secondly, the "include" statement in SSUCD use cases stated as such INCLUDE <UseCaseA> must be mentioned at least one once in SUCD and stated as such: • INCLUDE <UseCaseA>.	

Fig 7. Transforming the "Description Section"

Figure 7 outlines the mapping rules for the "Description Section". The purpose of "Description Section" is mainly to describe the behavior of the use case. In the SSUCD form, the description is provided in an unstructured natural language form. The only exception being the specification of an *included* use case where the keyword INCLUDE is used to specify the inclusion use case. Meanwhile, in the SUCD form the description section is far more structured. Each statement is specified individually along with the actor that is responsible for performing the actor. If the performer is the system itself, then the keyword SYSTEM is used. Hence, the transformation process of the "Description Section" cannot be fully automated and it requires human cognition to partition the description in the SSUCD form to bullet points in the SUCD form.

Extension Points Section	
SSUCD	**SUCD**
The "Extension Points" section starts with label "Extension Points:" ***MAPPING11:*** *If the use case does not have any public extension points this section is removed entirely.* The name of the extension points are then listed while separated with commas as follows: Extension Points: <EP1>, <EP2>, <EP3>…	Public Extension Points Section: The "Public Extension Points" section starts with label "Public Extension Points:" ***MAPPING11:*** *If the use case does not have any public extension points this section is removed entirely.* The name of the extension points are then listed while separated with commas as follows: Public Extension Points: <EP1>, <EP2>, <EP3>…
Transformation Rule: The names of the publication extension points listed in both SSUCD and SUCD must match.	
For an extension use case, it states the it extends another use case using the following structure: Extended Use Cases: Base UC Name: <UseCaseA> Extension Point: <EP_Name> IF <Condition> ***MAPPING12:*** *The name of*	For an extension use case, it states the it extends another use case as well as it states the extension behavior using the following structure: PUBLIC EXTENSION POINT BEHAVIOR EXTENDING {UseCaseA : EP_Name} ***MAPPING12:*** *The name of stated use case being*

stated use case being extended must exist.	*extended must exist in the model.*
MAPPING13: *The name of the stated extension point must be listed in the "Extension Points" section of the extended use case.*	***MAPPING13:*** *The name of the stated extension point must be listed in the "Extension Points" section of the extended use case.*
Transformation Rule: The use case name stated in SSUCD as Base UC Name: <UseCaseA>, must be the same as that stated in SUCD as EXTENDING {UseCaseA: …etc. The extension point name stated in SSUCD as Extension Point: <EP_Name>, must be the same as that stated in SUCD as EXTENDING {UseCaseA : EP_Name}.	

Fig 8. Transforming the "Extension Points Section"

Figure 8 outlines the mapping rules for the "Extension Points Section". The purpose of "Extension Points Section" is to state the extension points of a use case. The transformation process of the "Extension Points Section" may also be fully automated.

V. Library System Case Study

The library system discussed in this section was previously presented in [5], which the research work that introduced SUCD. This library system was specifically to evaluate the correctness of the proposed model transformation technique as the work presented in [5] outlines a set of use cases and how they are transformed into UML activity diagrams. In order to perform the evaluation, the SUCD use cases were rewritten as SSUCD use cases by extracting only the information required by the SSUCD structure. The entire set SUCD use cases and their corresponding SSUCD use cases used in this case study are available in [5]. For illustrative purposes, an example of a SUCD use case presented in [5] and its reverse-engineering SSUCD version are shown in Figures 1 and 2 (see Section 2), respectively.

A. Applying the Model Transformation

Using the reserve engineering SSUCD use cases as the source, the textual descriptions were analyzed by ANTLR to generate a representative object models that conform to the metamodel previously produced (see Section 3.1). The generated object models were used as input by ATL to apply the model transformation algorithms and mapping rules previously encoded. ATL then generates a set of object models that represent the SUCD equivalent of the SSUCD use cases used as input. As encoded in ATL, the generated object models representing the SUCD use cases are set to conform to the target metamodel (see Figure 3). A simple tool was used to read the generated object models representing the SUCD use cases to produce the text files presenting the SUCD use cases in a textual form.

B. Verifying the Correctness of the Produced SUCD Use Case Descriptions

The correctness of the produced SUCD use case descriptions was verified through two distinct means. The first approach involved the use of the *Diff* tool to check for differences between the produced SUCD use cases and the SUCD use cases already shown in [5]. Although some minor differences were found in the layout (white spaces and empty lines), the textual content of the use case descriptions were confirmed to be the same.

The second approach used to verify the correctness is to verify that the generated SUCD use case descriptions can be used as a source to generate representative UML activity diagrams using the approach presented in [5] to produce UML activity diagrams that match the UML activity diagrams shown in [5]. The generated UML activity diagrams along with the UML activity diagrams already shown in [5] were used as input by a tool named *UMLDiff* [11]. *UMLDiff* is an automated UML-aware structure differences algorithm which uses as input two object-oriented models then produces a report of the design evolution of the software system in the form of a change tree [11]. For given object models (representing the UML activity diagrams), the UMLDiff tool did not report any differences between the UML activity diagrams.

VI. Conclusion and Future Work

In this paper, we presented an approach that helps bridge the gap between the analysis and design phases in a use case-driven development process. The contribution of this paper is a model transformation technique that is almost fully automated, which can be used to transform use case descriptions written in the SSUCD form to use cases written in the SUCD form which may then be used to generate other types of UML design artifacts. The model transformation technique helps eliminate the human factor and thus eliminating human injected errors that may result from perform the transformation completely manually. The proposed approach was applied to use cases of a library system already presented in the literature. The correctness of the proposed technique was verified by differencing the textual descriptions of the generated SUCD use case descriptions with the SUCD use case descriptions presented in [5]. The second approach involved the use of a popular tool in the model differencing research community, named UMLDiff, to compare the UML activity diagrams produced with the generated SUCD use case descriptions against the UML activity diagrams already presented in [Seattle]. Both verification approaches indicate the correctness and the effectiveness of the proposed technique.

Future work will be directed towards extending the SSUCD and SUCD languages to allow for the specification of functional security requirements. The model transformation technique will also need to be enhanced to facilitate the transformation of the extended SSUCD and SUCD languages.

 ISBN: 978-1-61208-165-6

ACKNOWLEDGEMENTS

The author would like to acknowledge the support provided by the Deanship of Scientific Research (DSR) at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No. JF100008.

REFERENCES

[1] M. El-Attar and J. Miller, A Subject-Based Empirical Evaluation of SSUCD's Performance in Reducing Inconsistencies in Use Case Models, Journal of Empirical Software Engineering, vol.14, no. 5, pp. 477-512, (2009).

[2] M. El-Attar and J. Miller, Producing Robust Use Case Diagrams via Reverse Engineering of Use Case Descriptions, Journal of Software and Systems Modeling, vol. 7, no. 1, pp. 67-83 (2008).

[3] I. Jacobson, M. Ericsson, and A. Jacobson, *The Object Advantage*. ACM Press, 1995.

[4] M. El-Attar and J. Miller, A User-Centered Approach to Modeling BPEL Business Processes Using SUCD Use Cases. Journal of Software Development and Theory, Practice and Experimentation, vol. 1, no. 1, pp. 59-76 (2007)

[5] M. El-Attar and J. Miller, AGADUC: Towards a More Precise Presentation of Functional Requirements in Use Case Models, 4th ACIS International Conference on Software Engineering, Research, Management and Applications, Seattle, Washington, USA. pp.346-353, (2006).

[6] Object Management Group, UML Superstructure Specification (2005). http://www.omg.org/docs/formal/05-07-04.pdf, Version 2.0 formal/05-07-04. Accessed March 2011.

[7] T. Parr, The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers). Pragmatic Bookshelf, 2007.

[8] SteelTrace. "Catalyze Suite". Availabe [Online] www.steeltrace.com. Last Accessed March 2011.

[9] TechnoSoclutions. "Top Team Analsyt". Availabe [Online] http://www.technosolutions.com/topteam_requirements_management.html. Last Accessed March 2011.

[10] The Eclipse Foundation. ATL – A Model Transformation Technology. Available Online at (http://www.eclipse.org/atl/). Last accessed March 2011.

[11] Z. Xing and E. Strou, UMLDiff: An Algorithm for Object-Oriented Design Differencing, Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 54-65, 2005.

[12] OMG 2003, "UML Superstructure Specification", Object Management Group, http://www.omg.org/docs/ptc/03-08-02.pdf, 2003.

 ISBN: 978-1-61208-165-6