A Grammar-Based Approach to Class Diagram Validation

Faizan Javed
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL 35294-1170, USA
javedf@cis.uab.edu

Marjan Mernik
Faculty of Electrical Engineering and Computer Science
University of Maribor
Smetanova 17
2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

Barrett R. Bryant, Jeff Gray
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL 35294-1170, USA
{bryant, gray}@cis.uab.edu

ABSTRACT
The UML has grown in popularity as the standard modeling language for describing software applications. However, UML lacks the formalism of a rigid semantics, which can lead to ambiguities in understanding the specifications. We propose a grammar-based approach to validating class diagrams and illustrate this technique using a simple case-study. Our technique involves converting UML representations into an equivalent grammar form, and then using existing language transformation and development tools to assist in the validation process. A string comparison metric is also used which provides feedback, allowing the user to modify the original class diagram according to the functionality desired.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *Model Checking, Validation.*

General Terms
Algorithms, Design, Reliability, Languages, Verification.

1. INTRODUCTION
The software industry continues to experience rapid growth with a side effect of rising costs. A common critique of today's software projects is that developers spend too much time on the coding process and begin the implementation phase prematurely. Throughout the development lifecycle, the system is tested according to the requirements to uncover any anomalies. In the event of any deviation from the original specifications, it is more costly in terms of time and money if the problem is rectified after the coding stage rather than at some point before it [11].

The Unified Modeling Language (UML) [1] was created to address some of these issues faced in the software design process. UML provides the ability to model both the static and dynamic aspects of an application. Using UML, classes that model a system can be described using class diagrams, and collaborations between classes to perform the use cases can be modeled by UML dynamic diagrams, such as sequence diagrams or activity diagrams.

Static validation can be used to check whether a model conforms to a valid syntax. Techniques supporting static validation can also check whether a model includes some related snapshots (i.e., system states consisting of objects possessing attribute values and links) desired by the end-user, but perhaps missing from the current model. We believe that the latter problem can occur as a system becomes more intricate; in this situation, it can become hard for a developer to detect whether a state envisaged by the user is included in the model. As an alternative, dynamic validation can be used to check if the dynamic features of the model exhibit the behavior required by the end-user. For performing static and dynamic validation of UML models, we propose a grammar-based approach to validating UML models with an initial focus on static validation.

Grammar-based systems [5] make use of a grammar or sentences generated by a grammar to solve problems outside the domain of programming languages. The rationale of this approach is that tools for working with grammars already exist; the problem-solving process can be expedited if a suitable representation of the problem in the form of a grammar can be found. The approach also involves the use of a compiler-generator tool called LISA [2] that is used to construct a Domain-Specific Language (DSL), which are languages designed specifically for a particular domain. The defining properties of DSL's are that they are small, more focused than general-purpose programming languages (e.g., Java), and usually declarative [4]. The technique also uses XSLT [3], a translation language for XML documents.

The rest of the paper is organized as follows: Section 2 describes the related work in this area. Section 3 gives an overview of the technologies used in the overall architecture of the approach. Section 4 presents the approach applied to a video store case study, while section 5 introduces some novel ideas in the context of this work. The paper concludes with summary remarks and a mention of future research directions in Section 6.

2. RELATED WORK
Various distinctive approaches have been taken to validate UML models. In [7], OCL expressions are employed to generate complex snapshots representing system states automatically at a particular point in time. Another approach makes use of the semantic model given by Abstract State Machines [9] to validate both the static and dynamic aspects of a UML model [8]. A

4th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools, 21 May 2005, Saint-Louis ISBN 1-59593-130-9

formal methods based framework has been proposed in [11], where the system specification is first described by a graphical intuitive model before being made precise and validated by use of a formal specification tool. Story Driven Modeling [12] is a software development process which converts textual use case scenarios into storyboards (i.e., sequences of UML interaction diagrams) from which the modeler derives class diagrams and UML based behavior specifications and JUnit tests. These are then turned into Java implementations by a code generator, and the JUnit tests are run to check whether the method behavior conforms to the use case scenario. An approach involving validation of UML models using theorem proving is discussed in [16], while [17] exploits use-cases by transforming them into planning problems to derive a sequence of messages. Relational logic has also been used to generate system snapshots, as in the case of the Alloy constraint analyzer [18].

Our work differs from previous approaches in that we represent the class diagram as a context-free grammar (CFG), and scenarios as a series of object instantiations of the class diagram. If a scenario is successfully parsed by the CFG, it indicates that the particular scenario can be represented by the class diagram.

3. THE VALIDATION PROCESS

Figure 1 gives an overview of the grammar-driven approach to UML validation. The remainder of this section will elaborate on the various stages of the approach and provide a brief introduction to the tools and technologies used at each step.

3.1 CONVERTING TO A GRAMMAR

In Stage 1, a UML class diagram for a particular application is converted into an equivalent XML representation. Stage 2 uses XSLT and LISA to perform transformations on the XML representation. XSLT is a language that can transform an XML document into any other text-based format [3]. LISA is an interactive environment for programming language development where users can specify, generate, compile-on-the-fly and execute programs in a newly specified language [2]. The XSLT portion of the process transforms the XML file to generate an equivalent CFG representation of the class diagram. Note that we are not dealing explicitly with objects when using the CFG representation. The CFG describes the structure of the class diagram albeit in a textual form, and does not contain type information of the attributes. Thus, this CFG can be viewed as a DSL for representing the domain of the original class diagram. A LISA specification file is created, which makes use of the generated DSL grammar to create a parser for the language. All the steps required to convert the class diagram to a DSL are automated.

3.2 CALCULATING VARIABILITY

After Stage 2, a textual representation of the original class diagram in the form of a grammar has been obtained. Stage 3 shows that all possible instantiations of this CFG correspond to valid strings in this language, which in turn correspond to all valid object diagram configurations of the UML diagram. Because the language that the CFG describes could be infinite, it is not feasible to perform static validation at this step by enumerating all possible strings of the CFG generated, and then testing if the use case string exists in this set. Hence, using this naïve approach of generating all possible use cases can be laborious and in some cases, impossible to achieve. A key observation here is that to generate all possible use cases, all possible combinations of the production rules in the CFG need to be generated. If the CFG includes recursive rules, then these rules need to be exercised only once because multiple runs through the rules will not append any new meaning to the use cases. Taking this into account we can calculate exactly how many different use cases need to be generated. The calculation is similar to the variability calculation of feature diagrams [13], with additional rules to handle recursion. The number of all possible different use cases is calculated by the rules in Table 1, where *A* stands for non-terminals, *a* is a terminal symbol, ***B*** denotes a non-terminal or terminal symbol, and *Var* stands for *Variability*:

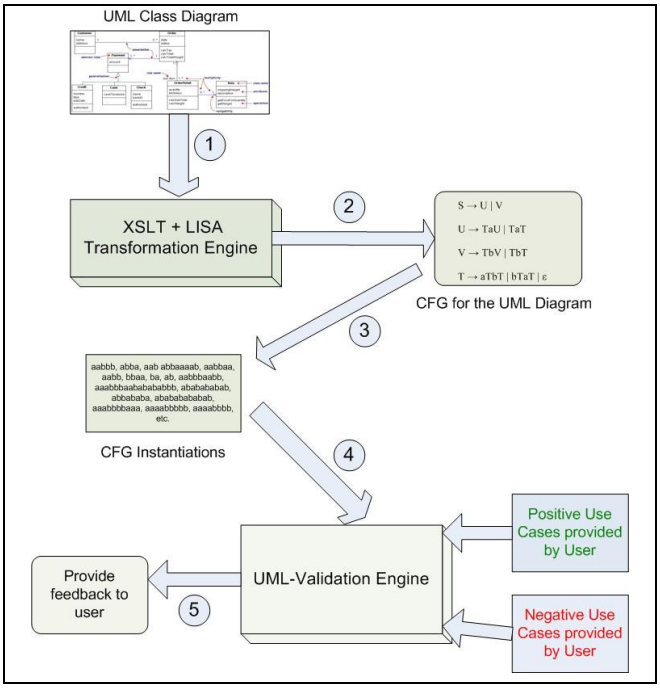

Figure 1. Architecture of the Validation Approach

Calculating the variability and eliciting all possible use cases is an automated process achieved by performing an exhaustive search to derive all string instances reachable from the start symbol. A concrete example will be detailed in section 4.1.

Var (A ::= $\mathbf{B}_1$ … $\mathbf{B}_n$)	= Var($\mathbf{B}_1$) * … * Var($\mathbf{B}_n$)	
Var (A ::= $\mathbf{B}_1$ | … | $\mathbf{B}_n$)	= Var($\mathbf{B}_1$) + … + Var($\mathbf{B}_n$)	
Var (A ::= a)	= 1	(single non-terminal)
Var (A ::= eps)	= 1	(empty production)
Var (A ::= A**B**)	= Var(**B**)	(left recursion)
Var (A ::= **B**A)	= Var(**B**)	(right recursion)

Table 1. Rules for Variability Calculation

3.3 THE FEEDBACK SYSTEM

In Step 4, the user presents a set of positive strings in the language that correspond to scenarios in the use case model. The user may also provide a set of negative strings. In UML, use cases capture the requirements of the customer and describe the functionality of

the system. The user-supplied strings are tested against the parser generated for the DSL by LISA. Ideally, only all the positive strings should be parsed successfully. If a positive string is not parsed, it indicates that a use case desired by the user can not be represented by the current class diagram. As additional feedback (Stage 5), the approach provides a list of strings that are most similar to the string provided. This metric is calculated using the Levenshtein distance [6]. The Levenshtein distance, also known as the *Edit Distance*, measures the similarity between two strings. The distance is the number of deletions, insertions, or substitutions required to transform a *source* string into a *target* string. The greater the Levenshtein distance, the more dissimilar the strings are. In the next section, we illustrate the validation process using a Video Store case study.

4. CASE-STUDY: A VIDEO STORE

This section presents a brief description of a video store case study. A video store consists of a video and customer database. The video database contains information on all video titles currently on file, and the user (customer) database contains information on all current members of the video store, as well as all videos currently rented by each customer. A customer can walk into the video store and either rent a movie, or become a member of the store. The customer is served by the owner of the store (or an employee). The store owner can also add new titles to the video database. Figure 2 gives the use case diagram for this example.

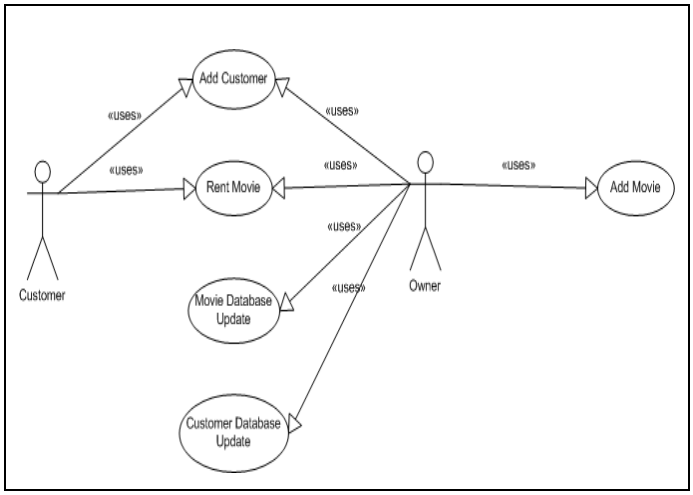

Figure 2. Use case diagram for the Video Store System

In UML, the functionality of the system is represented by use cases that interact with the system actors. In the video store example, the actors are the customer and owner. Each use case can be refined to an activity diagram. Activity diagrams focus on work performed during the activities in a use case instance or in an object. Due to space limitations, we only present the activity diagram for the *rent movie* use case (see Figure 3). The use case and activity diagrams are used by the user in forming the input test cases for the feedback component of the process. The activity diagrams are not used by the automated component of the validation process, but they are used by the user when constructing appropriate use cases while analyzing the CFG. The activity diagram also helps the user better formulate the desired functionality of the system.

4.1 Validating Static Behavior

In UML, class diagrams model the static structure of a system – the classes and their relationships. However, class diagrams do not explain how these structures cooperate to manage their tasks and provide the functionality of the system. The video store system is composed of the classes *VideoStore*, *User* and *Movies* (see Figure 4). The association *Rentals* describes the videos rented by a customer.

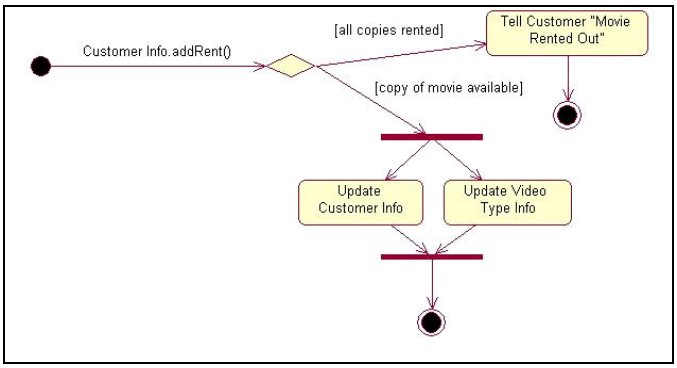

Figure 3. Activity Diagram for the rent movie use case.

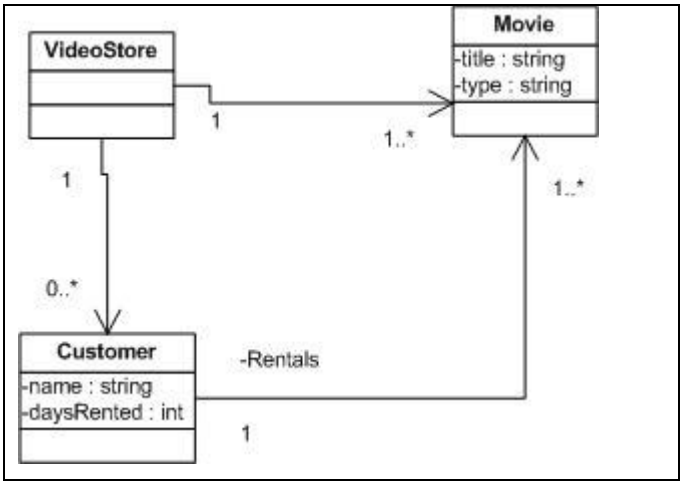

Figure 4. Class Diagram for the VideoStore Case Study.

Initially, the class diagram is converted into an XML representation. During Stage 2 of the validation process, the XML representation of the class diagram is converted into a CFG representation, as shown in Table 2.

1. VideoStore → MOVIES CUSTOMERS | CUSTOMERS MOVIES | MOVIES | CUSTOMERS
2. MOVIES → MOVIES MOVIE | MOVIE
3. MOVIE → title type
4. CUSTOMERS → CUSTOMERS CUSTOMER | eps
5. CUSTOMER → name days RENTALS
6. RENTALS → RENTALS RENTAL | RENTAL
7. RENTAL → MOVIE1
8. MOVIE1 → title type

Table 2. Video Store Class Diagram Represented as a CFG

A CFG consists of a start symbol, a set of production rules, terminals, and non-terminals. The CFG in Table 2 contains seven productions with the start symbol indicated by *VideoStore*.

Terminals are represented in lower-case letters, and upper-case letters correspond to non-terminals. This CFG describes a particular language, and instantiations of this CFG correspond to use cases of the video store system. Note that the relation *Rentals* is translated to a separate non-terminal *Movie1,* which goes to the terminal *title type* (the terminal set can be any subset of the list of attributes of the class *Movie*), describing the movie title(s) being rented by the user. This translation scheme helps prevent cyclical relationships to occur in the CFG. An example demonstrating the robustness of this approach when confronted with such situations is shown in section 4.3.

The user can provide positive use case scenarios as strings from this language to the parser generated by LISA. If a string can be parsed, it signifies that the use case scenario represented by that string can be generated by the UML class diagram. To obtain all possible use cases for the VideoStore class diagram, we compute the variability of the VideoStore CFG as given in Table 3:

```
Var(VideoStore ::= MOVIES CUSTOMERS | CUSTOMERS
                   MOVIES | MOVIES | CUSTOMERS)
          = (Var(MOVIES) * Var(CUSTOMERS) ) * 2 +
            Var(MOVIES) + Var(CUSTOMERS)
          = (2 * 3) * 2 + 2 + 3 = 17

Var(MOVIES ::= MOVIES MOVIE | MOVIE)
          = Var(MOVIES MOVIE) + Var(MOVIE)
          = Var(MOVIE) + Var(MOVIE)
          = 1 + 1 = 2

Var(MOVIE ::= title type)
          = Var(title) * Var(type)
          = 1 * 1 = 1

Var(CUSTOMERS ::= CUSTOMERS CUSTOMER | eps)
          =Var(CUSTOMERS CUSTOMER) +
           Var(eps)
          = Var(CUSTOMER) + 1
          = 2 + 1 = 3

Var(CUSTOMER ::= name days RENTALS)
          = Var(name) * Var(days) * Var(RENTALS)
          = 1 * 1 * 2 = 2

Var(RENTALS ::= RENTALS RENTAL | RENTAL)
          = Var(RENTALS RENTAL) + Var(RENTAL)
          = Var(RENTAL) + Var(RENTAL)
          = 1 + 1 = 2

Var(RENTAL ::= MOVIE1) = Var(MOVIE1) = 1

Var(MOVIE1 ::= title type)
          = Var(title) * Var(type)
          = 1 * 1 = 1
```

Table 3. Variability Calculation for the VideoStore CFG

The variability value obtained from the start rule (rule 1 in Figure 5) indicates the total number of unique use cases, which are 17 in our example. We enumerate 6 of these use cases:

1. Insert one movie in the database:
   title type
2. Insert one or more movies in the database:
   title type title type
3. Insert customer with a rent transaction:
   name days title type
4. Insert movies and a customer without a rent transaction:
   title type title type name days
5. Insert one movie and a customer with a rent transaction:
   title type name days title type
6. Insert multiple movies and a customer with a rent transaction:
   title type title type name days title type

As an example, a possible use case description for *Add Movie* is:

1. Request movie title
2. Request movie type
3. Insert the movie in movie database

An example scenario for this would be:

Titanic reg

where *reg* stands for *regular*, the type of the movie. This would also be one of the positive samples, i.e., one of the functionalities required by the user.

Similarly, the *Rent Movie* use case is:

1. Request user name
2. Request total days for rent
3. Request movie name
4. Request movie type.
5. Update movie database
6. Update customer database

A scenario for this use case is:

Mike 5
TheRing horror
Shrek child

The *Rent Movie* scenario can be parsed by production 4. Class diagrams lack an explicit order for processing or instantiating their various entities while CFG's implicitly define a sequence of events beginning from the *start* symbol. Thus, in our representation of class diagrams as CFG's, the CFG's are initiated by enumerating all possible permutations of the class non-terminals, which in the case study are *MOVIES* and *CUSTOMERS*. This allows a more extensive and definite representation of the class diagrams for validation purposes, and expedites the testing process by allowing the user to concentrate on analyzing strings corresponding to particular use case scenarios only, rather than presenting a complete system-wide scenario every time..

4.2 REFACTORING THE CLASS DIAGRAM

An example of a negative sample would be a string representing a *Delete Movie* use case scenario, which is not one of the desired functionalities. If such a string can be parsed by the CFG, it would indicate an error in the class diagram; thus, the class diagram

would need to be revised. Correspondingly, if a positive sample fails, then it would mean that a required functionality cannot be obtained from the class diagram, and the class diagram needs to be reconsidered. As an example, it can be observed that the *Add Customer* use case cannot be satisfied by the class diagram in Figure 4. In other words, the derived CFG is not able to generate any strings corresponding to the *Add Customer* scenario. Upon noticing this lack of functionality, the UML class diagram can be refactored as follows: note that class *Customer* contains the attributes *name* and *daysRented*. A new class *Rental* is added, and attribute *daysRented* is shifted to class *Rental*. After this refactoring, the *Add Customer* use case is possible. Figure 5 shows the refactored class diagram.

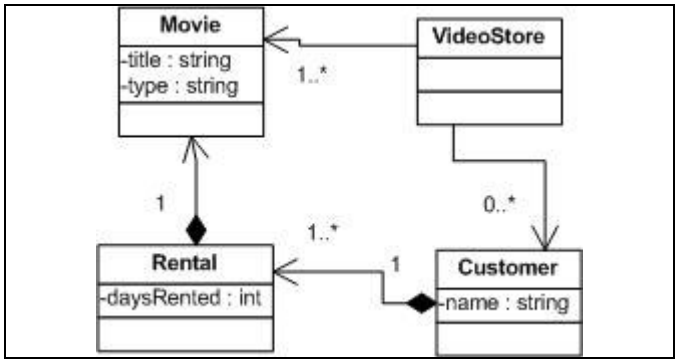

Figure 5. Refactored Class Diagram for the Video Store.

Step 5 of the process provides feedback to the user in the event that a string is not in the language of the CFG. In this case, the user is provided a set of strings that are similar (to a certain extent) to the string provided by the user in the hope that one of the strings in the feedback set would be what the user really desires. The total size of the feedback string set and the similarity degree of the strings are parameters provided by the user. As an example, consider the following string, which is rejected by the CFG in Figure 5:

TheRing

The closest match for this string is rule 3. Although this string will fail to be parsed, the user might be provided with the feedback set given in Table 4.

Similarity: 60%
Total number of strings: 4
1) TheRing a
2) TheRing aa
3) TheRing aaa
4) TheRing aaaa

Table 4. A Sample Feedback Set Provided by the System

The similarity measure of 60% requires that the total number of strings in the set is at least 60% similar to the original string. As an example, compare the following two strings: *TheRing,* and *TheRing a*. These two strings differ by only 1 character.

4.3 CYCLICAL RELATIONS

Naïve approaches to generating CFG's from UML diagrams can encounter complications like an infinite, non-terminating recursive grammar. Our approach is able to derive a correct infinite terminating recursive grammar in the presence of such relationships, and we demonstrate this by proposing a small modification to the *VideoStore* case study example.

Assume that in the class diagram of Figure 4, a 1-to-many relation *Rental2* exists between *Movie* and *Customers*. This would make sense if multiple copies of a movie exist, and can be rented by many customers. An example of a naïve CFG produced from this class diagram would be:

1. VideoStore → MOVIES CUSTOMERS | CUSTOMERS MOVIES | MOVIES | CUSTOMERS
2. MOVIES → MOVIES MOVIE | MOVIE
3. MOVIE → title type RENTALS2
4. RENTALS2 → RENTALS2 RENTAL2 | RENTAL2
5. RENTAL2 → CUSTOMER
6. CUSTOMERS → CUSTOMERS CUSTOMER | CUSTOMER
7. CUSTOMER → name days RENTALS
8. RENTALS → RENTALS RENTAL | RENTAL
9. RENTAL → MOVIE

Production sets (3, 4, 5) and (7, 8, 9) indicate a cyclical non-terminating relationship in the grammar. In this situation, use case strings will fail to be parsed because the grammar is non-terminating and infinite. This problem arises whenever a class, modeled by a non-terminal, is referred to by another class via a relation, which is also modeled by a non-terminal. We propose using a new non-terminal to model the destination class pointed to by the source class in a relation. This prevents the problem of inheriting the relation non-terminals of the destination class by the source class. Using this technique, the CFG for the modified *VideoStore* example changes to the CFG in Table 5.

1. VideoStore → MOVIES CUSTOMERS | CUSTOMERS MOVIES | MOVIES | CUSTOMERS
2. MOVIES → MOVIES MOVIE | MOVIE
3. MOVIE → title type RENTALS2
4. RENTALS2 → RENTALS2 RENTAL2 | RENTAL2
5. RENTAL2 → CUSTOMER1
6. CUSTOMER1 → name days
7. CUSTOMERS → CUSTOMERS CUSTOMER | CUSTOMER
8. CUSTOMER → name days RENTALS
9. RENTALS → RENTALS RENTAL | RENTAL
10. RENTAL → MOVIE1
11. MOVIE1 → title type

Table 5. VideoStore CFG to Handle Cyclical Relations

An example of a string in this CFG would be:
1. theRing horror
2. Jack 5 Ann 5 Mike 10
3. jurassicPark child
4. John 1 Jane 5
5. Bruce 10
6. theRingTwo horror

Sentences 1-4 correspond to two instances of the *Rentals2* relation, and sentences 5 and 6 are an instance of the *Rental* relation.

4.4 MULTIPLE INHERITANCE

Figure 6 shows a class diagram example using diamond inheritance, a special case of multiple inheritance. Class *Person* has attributes *id* and *name*, and classes *Student* and *Assistant* are subclasses of *Person* with attributes *Grades* and *Salary* respectively. Class *Student Assistant* inherits from both *Student* and *Assistant*. Thus, class *Student Assistant* has base class *Person* appearing more than once as an ancestor. Our translation scheme generates the CFG given in Table 6 for this example (note that for brevity, we have not included the *start* production which enumerates all possible class non-terminals).

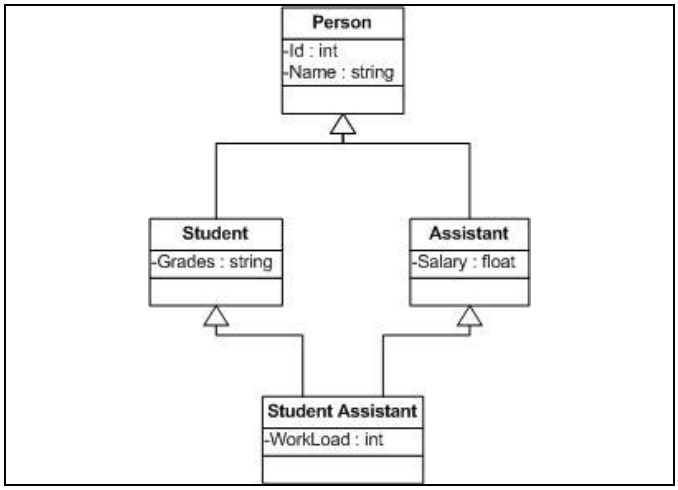

Figure 6. A Diamond Inheritance Example

1. STUDENTASSISTANT → workload SA1
2. SA1 → STUDENT ASSISTANT
3. STUDENT → grades PERSON
4. ASSISTANT → salary PERSON
5. PERSON → id name

Table 6. CFG for the Diamond Inheritance Example

A statement in this language, corresponding to an instance of the *Student Assistant* class, is:

"workload grades id name salary id name"

This statement exhibits the same problem found in class structures using diamond inheritance in that base class attributes *id* and *name* are duplicated. Thus, our method is able to cope with class diagrams implemented with inheritance hierarchies.

5. OTHER IDEAS

The main gist of our work involves obtaining a CFG representation of the class diagram, and validating the class diagram by using use case scenarios to test whether the current class diagram configuration can generate the particular scenario. An interesting research direction is to see whether we can obtain a class diagram structure just by using positive and negative use case scenarios. This process would involve learning a CFG from language samples, and then constructing the class diagram from the inferred CFG. Learning a CFG from positive and negative samples is known as grammar induction [14]. However, the largest class of languages that can be efficiently learned by provably converging algorithms are regular languages. Learning CFG's has proved to be a harder problem and is still considered a real challenge in the grammar induction community. Our work in this area [15] makes some contributions towards inferring CFG's for simple DSL's. For the VideoStore class diagram in Figure 4, when provided with the positive and negative use case scenarios, our grammar induction engine inferred the CFG in Table 7.

1. NT15 ::= NT11 NT7 NT15 | eps
2. NT11 ::= NT10 NT6
3. NT10 ::= NT5 NT10 | eps
4. NT7 ::= NT5 NT7 | eps
5. NT6 ::= name days
6. NT5 ::= title type

Table 7. Inferred CFG for the VideoStore Case Study

From the inferred CFG, the UML class diagram in Figure 7 can be constructed by hand, which is very similar to the original VideoStore class diagram. The non-terminals *NT5* and *NT6* correspond to the classes *Movie* and *Customer*, respectively. This is a simple example that illustrates some future ideas, and we believe that this idea warrants more in-depth research work.

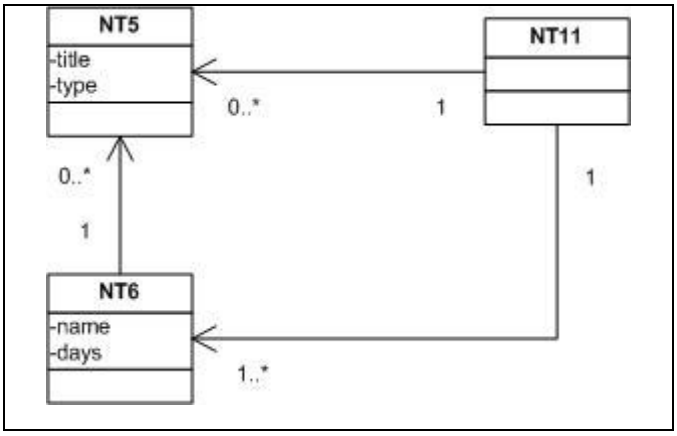

Figure 7. Inferred Class Diagram for the VideoStore CFG

6. CONCLUSION

This paper introduced a novel grammar-based approach to the UML static validation problem. The approach involves the representation of UML class diagrams as a DSL. An XSLT transformation is used to convert an XML representation of a model to the representative DSL. A parser for the DSL is generated by the LISA compiler generator tool. Positive and negative use cases are provided to the generated parser in the form of strings in the DSL. A string similarity measure is employed in order to provide feedback to the user regarding validation criterion. It is also shown that the approach can deal with issues like cyclical relations and multiple inheritance in class diagrams.

The future work involves application to more complex examples. Because a real application makes use of both dynamic and static views of a UML model, we also intend to extend this work to be able to validate dynamic aspects of a UML model via the use of state diagrams or activity diagrams. Our future work also involves validating OCL constraints as well as validating state charts from sequence diagrams, with a long term goal of inferring class diagrams from use cases and state charts from sequence diagrams.

REFERENCES

[1] *OMG Unified Modeling Language Specification, Version 1.5*. OMG, March 2003. OMG Document formal / 03-03-01, http://www.omg.org/uml.

[2] M. Mernik, M. Lenič, E. Avdičauševič, and V. Žumer, "LISA: An Interactive Environment for Programming Language Development," *11th International Conference on Compiler Construction*, Springer-Verlag LNCS 2304, Grenoble, France, April 2002, pp. 1-4.

[3] J. Clark, "XSL Transformations (XSLT) (Version 1)," *W3C Technical Report*, November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116 .

[4] M. Mernik, J. Heering, and T. Sloane, "When and How to Develop Domain-specific Languages," *CWI Technical Report*, SEN-E0309, 2003.

[5] M., Mernik, M. Črepinšek, T. Kosar, D. Rebernak, and V. Žumer, "Grammar-Based Systems: Definition and Examples," *Informatica*, vol. 28, no. 3, 2004, pp. 245-254.

[6] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," Doklady Akademi Nauk SSSR, 163(4):845-848, 1965(Russian). English translation in Soviet Physics Doklady, vol. 10, no. 8, 1966, pp. 707-710.

[7] M. Gogolla, J. Bohling, and M. Richters, "Validation of UML and OCL Models by Automatic Snapshot Generation," *6th International Conference on the Unified Modeling Language (UML)*, Springer-Verlag LNCS 2863, San Francisco, CA, October 2003, pp. 265-279.

[8] W. Shen, K. Compton, and J. Huggins, "A Validation Method for a UML Model Based on Abstract State Machines," *EUROCAST*, Canary Islands, Spain, February 2001, pp. 220-223.

[9] Y. Gurevich, "Evolving Algebras. 1993: Lipari Guide," *Specification and Validation Methods*, Oxford University Press, 1995, pp. 9-36.

[10] S. Dupuy-Chessa and L. du Bousquet. "Validation of UML models thanks to Z and Lustre," *International Symposium of Formal Methods Europe: Formal Methods for Increasing Software Productivity*, LNCS Vol. 2021, Springer-Verlag LNCS 2021, London, UK, 2001, pp. 242-258.

[11] S. Schach, *Object-oriented and Classical Software Engineering*, McGraw-Hill, 2005.

[12] I. Diethelm, L. Geiger, and A. Zündorf, "Systematic Story Driven Modeling, a case study," *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE 2004*, Edinburgh, Scotland, May 24–28, 2004.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," *Technical Report, CMU/SEI-90-TR-21, ADA 235785*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[14] The Grammar Induction Community Website: http://eurise.univ-st-etienne.fr/gi/

[15] M. Črepinšek, M. Mernik, B. R. Bryant, F. Javed and A. Sprague, "Inferring Context-Free Grammars for Domain-Specific Languages," *In Proceedings of Fifth Workshop on Language Description, Tools and Applications (LDTA 2005), J. Boyland, G. Hedin (Eds.),* pp. 64 - 81, 2005, Edinburgh, Scotland, UK.

[16] D. Muthiaye, "Real-time reactive System Development – a Formal Approach Based on UML and PVS.", *Ph.D. Thesis*, Department of Computer Science at Concordia University, Montreal, Canada, 2000.

[17] P. Frohlich and J. Link, "Automated Test Case Generation from Dynamic Models," *In Proc. 14th European Conf. Object-Oriented Programming (ECOOP 2000), E. Bertino, Ed.,* Springer, Berlin, LNCS 1850, 472-491.

[18] D. Jackson, I. Schechter, and I. Shlyakhter, "Alcoa: The Alloy Constraint Analyzer", *In Proc. Int. Conf. Software Engineering (ICSE 2000)*, ACM, New York, 730-733.