DEVELOPMENT OF A LIBRARY FOR AUTOMATED VERIFICATION OF UML MODELS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MAKBULE FİLİZ ÇELİK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

APRIL 2006

Approval of the Graduate School of Informatics

_____
Assoc. Prof. Dr. Nazife Baykal
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____
Assoc. Prof. Dr. Yasemin Yardımcı
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____
Dr. Altan Koçyiğit
Supervisor

Examining Committee Members

Prof. Dr. Semih Bilgen	(METU, EE)	_____
Dr. Altan Koçyiğit	(METU, IS)	_____
Assoc. Prof. Dr. Onur Demirörs	(METU, IS)	_____
Dr. Ali Arifoğlu	(METU, IS)	_____
Dr. Çiğdem Gencel	(METU, IS)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name and Surname: Makbule Filiz ÇELİK
Signature:

ABSTRACT

DEVELOPMENT OF A LIBRARY FOR AUTOMATED VERIFICATION OF UML MODELS

Çelik, Makbule Filiz
MSc., Department of Information Systems
Supervisor: Dr. Altan Koçyiğit

April 2006, 64 Pages

Software designs are mostly modeled as Unified Modeling Language (UML) diagrams when object oriented software development is concerned. Some popular topics in the industry such as Model Driven Development, generating test cases automatically in the early phases of software development, automated generation of code from design model etc. use the benefits of UML designs. All of these topics have something in common which is the need for accuracy against the meta-model not to face problems in the latter phases of the development process. Support on the full checking of the design models is necessary for the detection of design inconsistencies.

This thesis presents an approach for automated verification of UML design models and explains the implementation of the library called UMLChecker.

Keywords: UML, verification, UML model checking, WFR

ÖZ

UML MODELLERİNİN OTOMATİK OLARAK DOĞRULANMASINI SAĞLAYAN BİR KÜTÜPHANE GELİŞTİRİLMESİ

Çelik, Makbule Filiz
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Dr. Altan Koçyiğit

Nisan 2006, 64 Sayfa

Nesne yönelimli yazılım geliştirme konusunda The Unified Modeling Language (UML) en fazla tercih edilen ve en yaygın kullanıma sahip olan modelleme yöntemidir. Model Yönelimli Geliştirme, yazılım geliştirme sürecinde otomatik olarak test verisi üretme ve modelden koda otomatik geçiş gibi bir çok konu son yıllarda popüler olmuştur. Bu çalışmalar UML kullanımı üzerinde yoğunlaşmaktadır. UML tasarımlarının faydalı olması için öncelikle doğru ve geçerli olması gerekmektedir, aksi durumda yazılım geliştirmenin ilerleyen safhalarında çeşitli sorunlara neden olacaktır. Modellerdeki olası tasarım hatalarının ve çelişkilerin bulunabilmesi için modellerin tam olarak kontrolü konusunda çalışmalara ihtiyaç vardır.

Bu tezde, UML ile tasarlanmış modellerin doğruluk ve tutarlılık kontrolünü yapan bir yöntem ve bu yöntemi kullanan UMLChecker isimli bir kütüphanenin geliştirilmesi anlatılmaktadır.
Anahtar Kelimeler: UML, tutarlılık kontrolü, UML model kontrolü, WFR

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Altan Koçyiğit, for giving me the chance to work with him. Without his patience, encouragement, support and valuable comments, I would not have completed this thesis.

I would like to thank the examining committee members for their helpful comments about this thesis. I also would like to thank Çağatay Göncü and Oktay Türetken for their support during the study I have done for validation of my thesis.

Finally, I thank my family for their support during the time I spent working for this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE

LIST OF ABBREVIATIONS AND ACRONYMS

AEM	:	Association-end Multiplicity
AMP	:	All Message Paths
API	:	Application Programming Interface
COLL	:	Collection Coverage
COND	:	Condition Coverage
CS	:	Class Attribute
DOM	:	Document Object Model
EML	:	Each Message On Link
FP	:	Full Predicate Coverage
GN	:	Generalization
JAXP	:	Java API for XML Processing
HTML	:	Hyper Text Markup Language
MDD	:	Model Driven Development
MOF	:	Meta Object Facility
MSC	:	Message Sequence Charts
OCL	:	Object Constraint Language
OMDAG	:	Object-Method Directed Acyclic Graph
OMG	:	Object Management Group
PVS	:	Prototype Verification Systems
SAX	:	Simple API for XML
SDK	:	Software Development Kit
SDL	:	Specification and Description Language
SeDiTeC	:	Testing Based on Sequence Diagrams
TLPVS	:	Temporal Logic Prototype Verification System

UML	:	The Unified Modeling Language
WFR	:	Well Formedness Rules
W3C	:	The World Wide Web Consortium
XMI	:	XML Metadata Interchange
XML	:	eXtensible Markup Language

CHAPTER 1

INTRODUCTION

Modeling is being used for many years in software development methodologies. They offer a way to break down the system into smaller parts so that each part can be more understandable and easy to implement, without loosing the view of the big picture.

Although there are many modeling tools and techniques, when object-oriented software development is concerned, UML (The Unified Modeling Language) [1] is the most preferred one for modeling systems. The main reason behind this preference lies in the concept of standardization. The first version of UML was mostly a unification of the methods of Booch, Rumbaugh and Jacobson [1]. But it has been widely accepted since then and became an Object Management Group (OMG) [2] standard for the visual design of software applications.

In the recent years, more and more organizations have paid attention to Model Driven Development (MDD) [3]. MDD approach deals with the complexity of development of large software systems by creating more abstract level models and facilitating the transformations between these models. Important parts of these models are used to generate the implementation automatically. It is important to notice that, if any of these models contains errors, then these errors are also reflected to the implementation. Model validation tools are necessary to take the full advantage of Model Driven Development approach.

Another popular topic in the industry is generating test cases automatically in the early phases of software development [4]. The growing complexity of software systems causes more and more difficulty in testing the systems. Therefore, many researchers work on methods to generate automated test cases and test data from model diagrams. Using an accurate model for developing test cases is essential not to face problems in the latter phases of the development process.

UML covers different phases in software development such as requirements analysis, design and implementation. The automated link from the UML design models to generation of the code may sometimes be broken because of the unrecognized design errors. One way to catch the design errors in models is to check the model against the well-formedness rules (WFR)[5]. The UML specification describes a set of WFR that apply to the UML meta-model. WFRs are a set of constraints that a well-formed design should satisfy. A design described by UML is considered well-formed if none of the WFRs are violated.

Although a lot of work has already been carried out on UML semantics, there are still some points lacking. The main features of UML CASE Tools are designing UML diagrams, generating source code from these diagrams and reverse engineering[1]. Although current tools for drawing UML diagrams produce source code from the diagrams, they do not support all of the well-formedness rules. For example, user can call a method with inappropriate parameters, which should be avoided. Since the code is not generated from a well-formed design, there may be inconsistencies between what the code does and what actually it should do, which may cause problems in the latter phases of the development.

It is very important to understand that UML is a modeling language, not a method [6]. That is why the semantics of UML can be interpreted in many ways. It is difficult to manually check and detect the inconsistencies in UML models.

[1] Obtaining the diagrams from the source code

Although UML drawing tools, such as Rational Rose[2] or Together[3] can help modelers on some semantically well-defined relationships, they do not offer a full checking of the model since the UML semantics may be informal most of the time. More support is needed for the detection of design inconsistencies. An approach, which tests the UML models, is highly required.

1.1 Objectives of the thesis

Early detection of design flaws saves lots of time in the forthcoming phases of the development process. Design errors should also be eliminated if the aim is to generate code automatically from the design itself. This thesis describes an approach to trace the design model and find the possible defects.

In this thesis, we have developed a library, UMLChecker, which enables modelers to verify their diagrams automatically, as a whole. We have implemented a majority of the WFRs listed in the UML specification.

1.2 Scope of the thesis

One of the disadvantages of UML drawing tools is their expensiveness. The software we have developed is free to use and can be called from any program, if needed. It can be enriched with other functionalities to check the model against different constraints. Developers can embed the UMLChecker into their applications.

UMLChecker is independent from the CASE Tool used in designing the UML model. It assumes that the design model will be input in the XMI format[7], which is an explicit interchange format for UML based tools. Most UML drawing tools have the facility to convert the model into XMI format. Also, there are many free

---

[2] IBM, Rational Rose. (n.d.). Retrieved April 16, 2006, from http://www-306.ibm.com/software/rational/

[3] Borland Software Corporation. (n.d.). Retrieved April 16, 2006, from www.borland.com

software which do the same task. Therefore we did not consider converting the UML into XMI in the scope of this thesis but made the assumption that a UML model can somehow be converted into the XMI format, which would then be used by our UMLChecker.

All the different types of diagrams; use case, interaction, state, class, component and deployment diagrams, in the design model may be transformed into XMI format and then checked by the UMLChecker as a whole. In our library, we have checked the model for use case, class, state, sequence and collaboration diagrams. Each type of diagram is checked within itself at first. Then, the relations between different types of diagrams are inspected. For instance, at first a class diagram is examined and all the classes are confirmed to be well-formed, then the sequence diagram is examined. If a method of a class is used in the sequence diagram, but not mentioned in the class diagram, the model will fail to be well-formed.

UMLChecker is implemented in the Java Programming Language. This language is preferred because of its support for object-oriented software development. Also, Java is platform independent that is the generated code can be run on different operating systems.

1.3 Thesis organization

The rest of the thesis is structured as follows. Chapter 2 presents background information on UML and automated verification of UML models. Chapter 2 also includes the previous research in the area of automated UML testing. Chapter 3 provides details of the UMLChecker; architecture, implementation details and a sample application using UMLChecker library. And finally, Chapter 4 discusses the conclusions and possible directions for future work.

CHAPTER 2

BACKGROUND INFORMATION

In this part of the thesis, background information related to our subject is given. In section 2.1 UML is discussed, section 2.2 describes the XMI data interchange format and in section 2.2 validation and verification methods are discussed.

2.1 The Unified Modeling Language

The OMG specification states that "*The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.* " [2]

UML has become the de facto and de jure standard diagramming notation for object-oriented modeling. Grady Booch and Jim Rumbaugh have started UML in 1994 to combine the diagramming notations from their popular methods – the Booch and the OMT (Object Modeling Technique) methods. Later, Ivar Jacobson, the creator of the Objectory Method, has joined them and the group came to be known as the *three amigos*. With time, many others have contributed UML to take its current version [8]. In 1997, the UML was adopted as a standard by the OMG (Object Management Group).

2.1.1 The Model

The UML metamodel is a model describing the UML language including classes, attributes, associations, packages, collaborations, use cases, actors, messages, states, and all the other details used in UML. In Figure 1, the package structure of the UML metamodel is summarized.

The UML metamodel is published in the UML specification. We have used the latest specification of UML, which is UML 2.0.

UML 2.0 metamodel defines the syntax, a set of well-formedness rules and informally defined semantics. Syntax is defined by means of UML diagrams. WFR are expressed in terms of OCL expressions. And, finally, semantics are described informally using natural language.

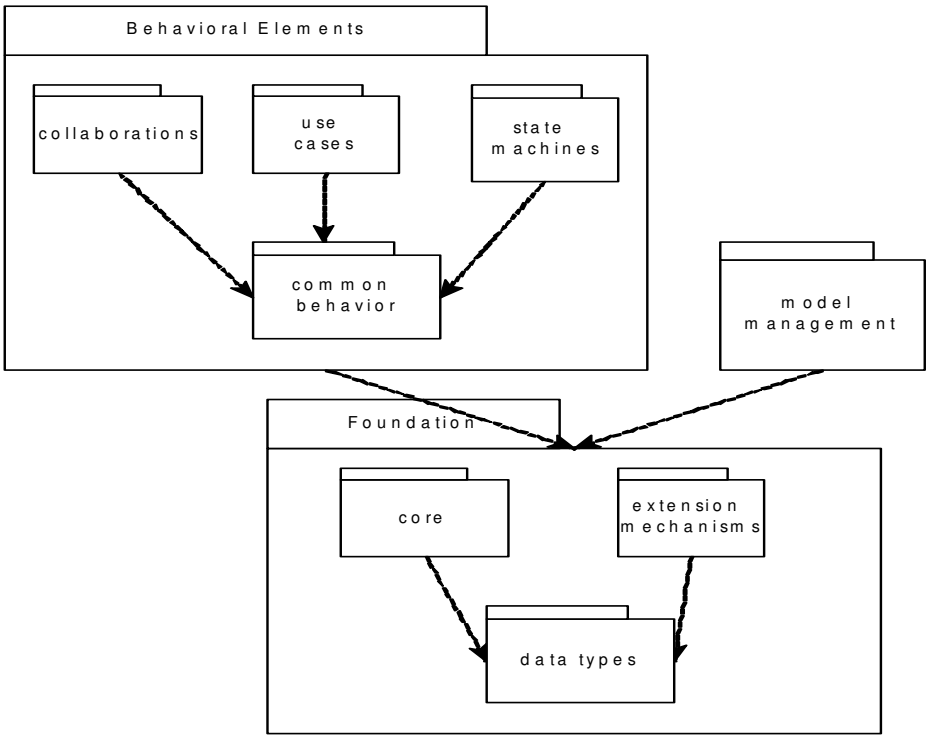


Figure 1 - UML metamodel package structure

UML metamodel is defined by four layers[4]. Every layer is an instance of the layer above and it adds new functionality and semantics to the above layer. The levels are as follows:

1. Meta-metamodel layer (M3): defines a language to use in the next layers
2. Metamodel layer (M2): contains instances of elements in the M3 and defines a language for specifying models. UML is located in this layer.
3. Model layer (M1): is an instance of the metamodel and is used by the developers to specify system design models.
4. User Object layer (M0): contains instances of system design model elements.

2.1.2 Diagrams

In the UML, no one diagram can capture the different elements of a system as a whole. Therefore, UML is made up of nine types of diagrams that can be used to model from different perspectives. The nine UML diagrams are explained in the coming sections.

2.1.2.1 Use Case Diagram

Use cases are used to describe the functional requirements, which indicate what the system will do [8]. The use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." Actors may be anything with a behavior such as a person, an organization or a computer system. Use cases are scenarios that describe the actors using the system to accomplish a goal [9]. The use case diagram shows which actors interact with each use case. Each use case must have a name, a number and a description.

In Figure 2, semantic elements of a use case diagram can be seen. In Figure 3 a sample use case diagram is given [8].

---

[4] Object Management Group. (n.d.). Retrieved April 16, 2006, from http://www.uml.org/

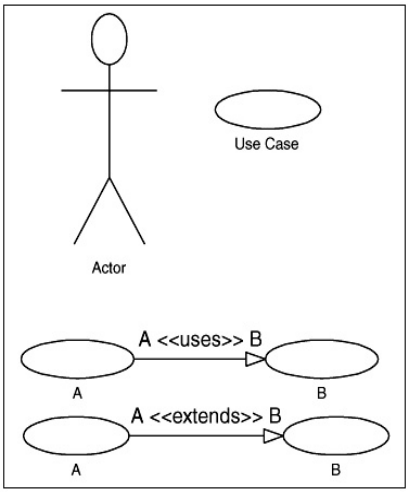


Figure 2 – Semantic elements of Use Case Diagrams

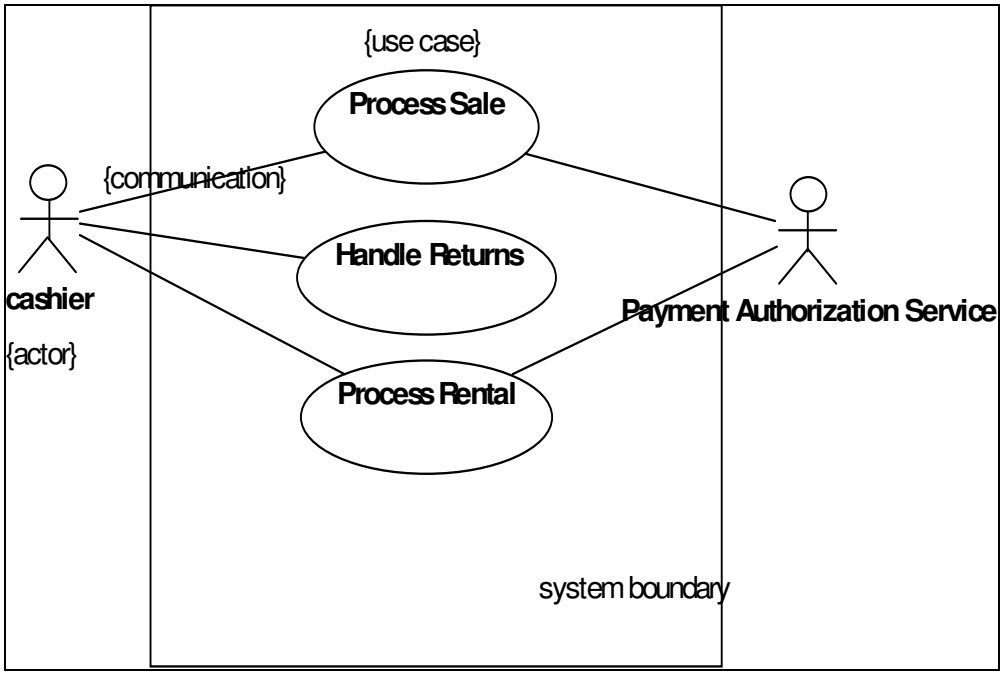


Figure 3 - Example Use Case Diagram

2.1.2.2 Class diagram

The class diagram defines a detailed design of the system. The class diagram classifies the domain concepts defined in the use case into a set of interrelated classes. The relationship or association between the classes can be either an "is-a" or "has-a" relationship. Each class in the class diagram may be capable of providing certain functionalities. These functionalities provided by the class are termed "methods" of the class. Apart from this, each class may have certain "attributes" that uniquely identify the class.

Classes define objects each having states and behaviors. Attributes and associations describe states. Associations represent the relationships between the instances of classes. Attributes are similar to associations. From a conceptual perspective, there are no differences. However, difference comes into play when specification and implementation is concerned. Operations describe behaviors. Methods are the implementations of operations.

Figure 4 shows a sample class diagram[5].

2.1.2.3 Object diagram

The object diagram is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time.

In Figure 5, an example object diagram can be seen[6].

---

[5] UML 2 Class Diagrams. (n.d.). Retrieved April 16, 2006, from http://www.agilemodeling.com/artifacts/classDiagram.htm

[6] Object Diagrams in UML. (n.d.). Retrieved April 16, 2006, from http://www.developer.com/design/article.php/2223551

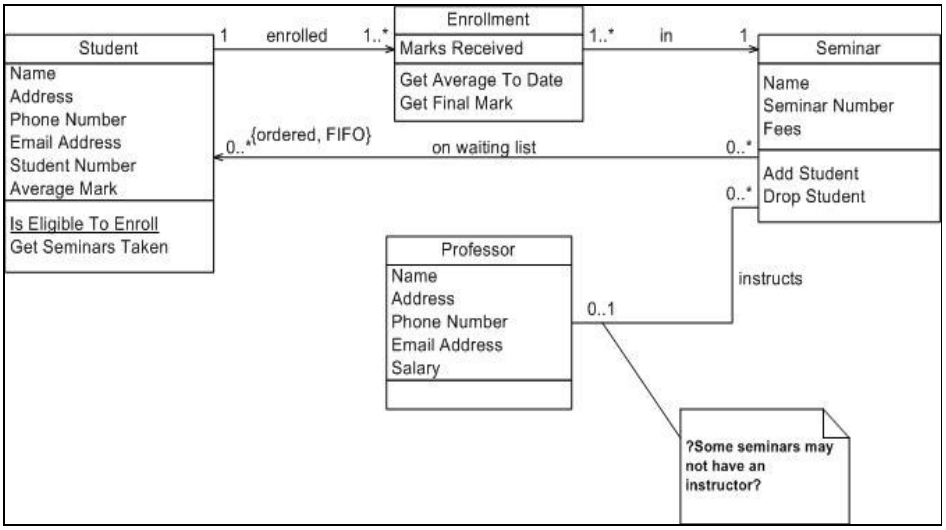


Figure 4 - Example Class Diagram

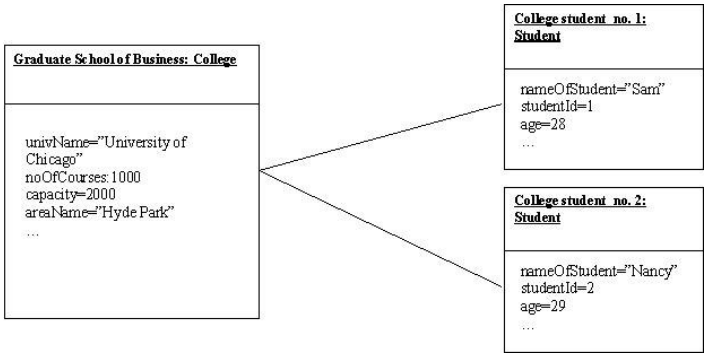


Figure 5 - Example of Object Diagram

2.1.2.4 State diagram

A state diagram, as the name suggests, represents the different states that objects in the system undergo during their life cycle. Objects in the system change states in response to events. Events are any external stimuli that may have effect on the objects. When an object recognizes an event, it takes an action considering its current state. Additionally, a state diagram also captures the transition of the

object's state from an initial state to a final state in response to events affecting the system.

Figure 6 depicts a sample state diagram[7].

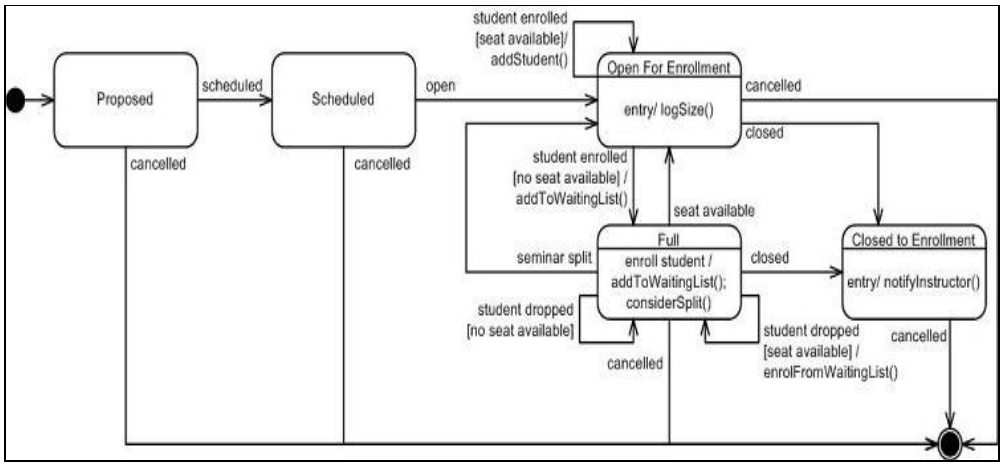


Figure 6 - Example of state diagram

2.1.2.5 Activity diagram

The process flows in the system are captured in the activity diagram. Similar to a state diagram, an activity diagram also consists of activities, actions, transitions, initial and final states, and guard conditions. Unlike the state diagram, the states in the activity diagram represent the states of executing the computation, not the state of the objects. Activity diagrams show the flow of activities but not the objects which are actually performing the activities.

A sample activity diagram can be seen in Figure 7[8].

---

[7] UML 2 State Machine Diagrams. (n.d.). Retrieved April 16, 2006, from http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm

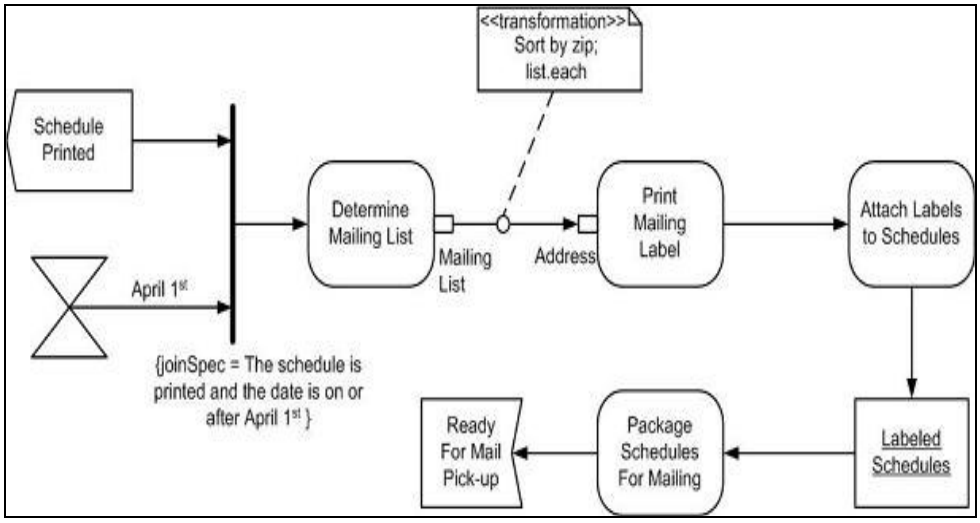


Figure 7 - Example Activity Diagram

2.1.2.6 Sequence diagram

A sequence diagram represents the interaction between different objects in the system in a two-dimensional graphical form. The important aspect of a sequence diagram is that it is time-ordered. Time is represented in the vertical dimension of the diagram. The horizontal dimension represents the roles of the classes, which resemble the objects. Time factor makes sure that the exact sequence of the interactions between the objects is represented step by step. Different objects in the sequence diagram interact with each other by passing "messages". A message is shown as an arrow from the lifeline of an object to that of another. In Figure 8, an example sequence diagram is given[9].

---

[8] UML 2 Activity Diagrams. (n.d.). Retrieved April 16, 2006, from http://www.agilemodeling.com/artifacts/activityDiagram.htm

[9] UML 2 Sequence Diagrams. (n.d.). Retrieved April 16, 2006, from http://www.agilemodeling.com/artifacts/sequenceDiagram.htm

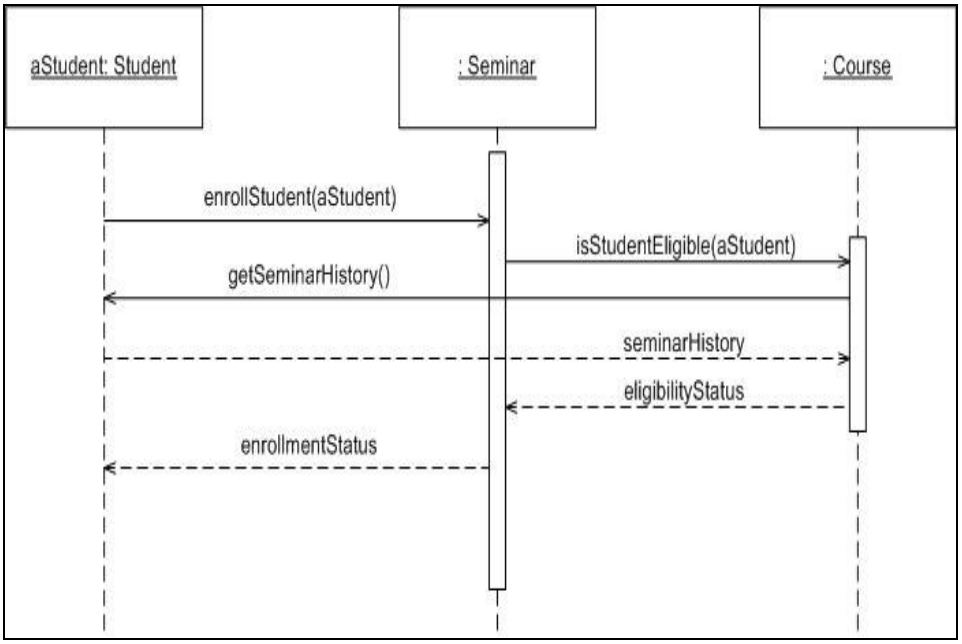


Figure 8 - Example Sequence Diagram

2.1.2.7 Collaboration diagram

A collaboration diagram groups together the interactions between different objects. The interactions are listed as numbered interactions that help to trace the sequence of the interactions. The collaboration diagram helps to identify all the possible interactions that each object has with other objects.

Collaboration diagrams are actually a type of class diagrams. They include the roles of the classifiers and associations, which lacks in class diagrams.

Figure 9 shows a sample collaboration diagram[10].

[10] UML 2 Communication Diagrams. (n.d.). Retrieved April 16, 2006, from http://www.agilemodeling.com/artifacts/communicationDiagram.htm

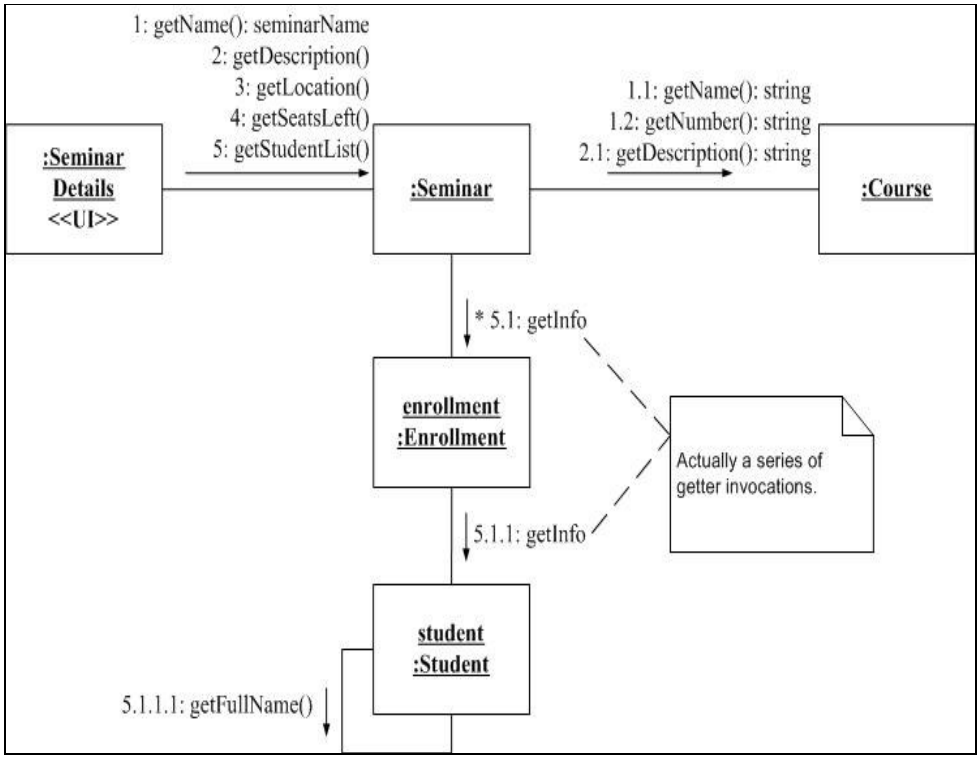


Figure 9 - Example Collaboration diagram

2.1.2.8 Component diagram

The component diagram represents the high-level parts that make up the system. This diagram depicts, at a high level, what components form part of the system and how they are interrelated. A component diagram depicts the components culled after the system has undergone the development or construction phase. A sample component diagram is seen in Figure 10[11].

[11] UML 2 Component Diagrams. (n.d.). Retrieved April 16, 2006, from http://www.agilemodeling.com/artifacts/componentDiagram.htm

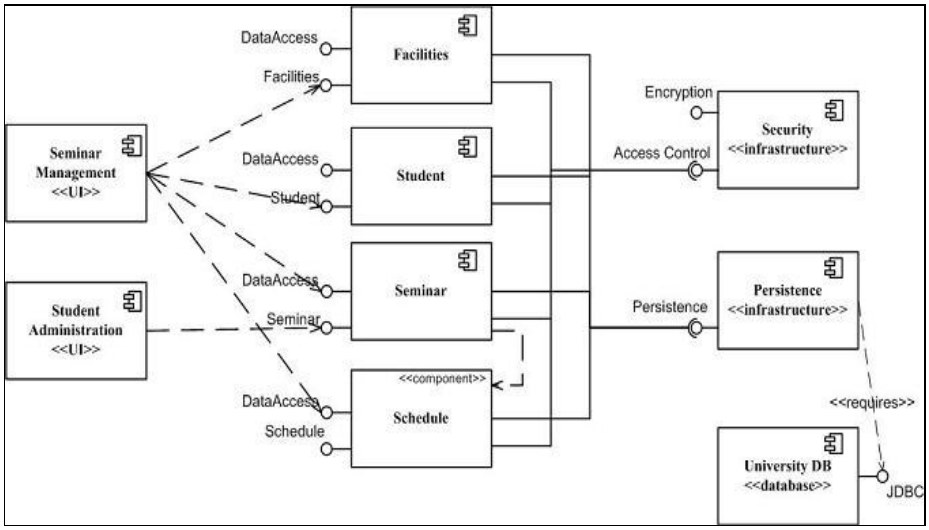


Figure 10 - Example Component Diagram

2.1.2.9 Deployment diagram

The deployment diagram captures the configuration of the runtime elements of the application. This diagram is by far most useful when a system is built and ready to be deployed. In Figure 11, a sample deployment diagram can be seen[12].

2.1.3 Categorizing the diagrams

These diagrams may be grouped as being static, dynamic and implementation:

- Static: The static characteristic of a system is essentially the structural aspect of the system. The static characteristics define what parts the system is made up of.
    - Use Case and class diagrams are in this group
- Dynamic: The behavioral features of a system; for example, the ways a system behaves in response to certain events or actions are the dynamic

[12] UML 2 Deployment Diagrams. (n.d.). Retrieved April 16, 2006, from http://www.agilemodeling.com/artifacts/deploymentDiagram.htm

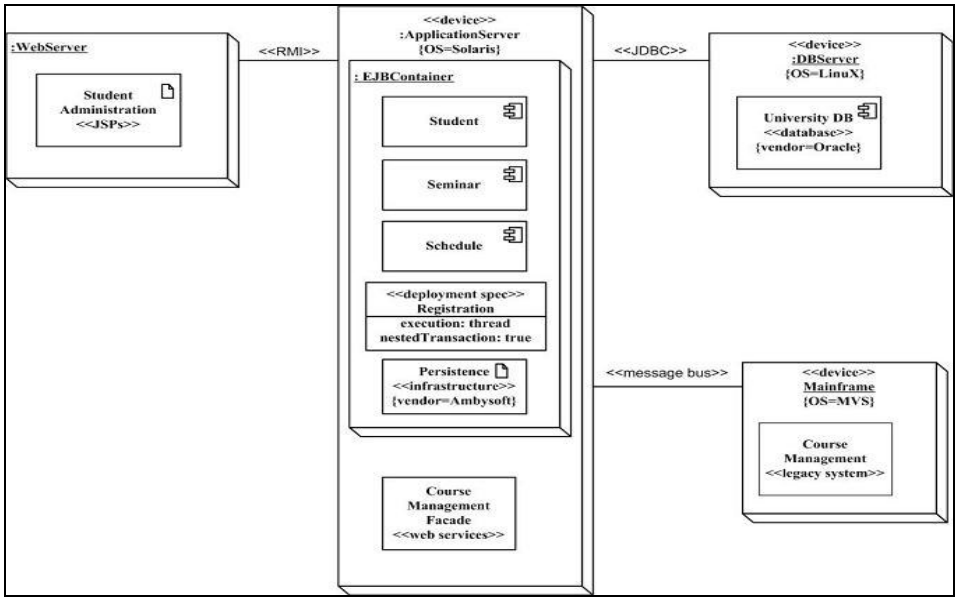


Figure 11 - Example Deployment diagram

characteristics of a system.

- Object, state, activity, sequence and collaboration diagrams are in this group

- Implementation: The implementation characteristic of a system is an entirely new feature that describes the different elements required for deploying a system.
  - Component and deployment diagrams are in this group

Considering that the UML diagrams can be used in different stages in the life cycle of a system, the views in the UML show how a system can be viewed from a software life cycle perspective. This categorization enables us to understand where exactly the UML diagrams fit in as well as their applicability. The views in the UML Model are:

- Design View: The design view of a system is the structural view of the system. This gives an idea of what a given system is made up of. Class diagrams and object diagrams form the design view of the system.

- Process View: The dynamic behavior of a system can be seen using the process view. The different diagrams such as the state diagram, activity diagram, sequence diagram, and collaboration diagram are used in this view.
- Component View: Next, you have the component view that shows the grouped modules of a given system modeled using the component diagram.
- Deployment View: The deployment diagram of UML is used to identify the deployment modules for a given system. This is the deployment view of the
- Use case View: Finally, we have the use case view. Use case diagrams of UML are used to view a system from this perspective as a set of discrete activities or transactions.

2.2 XMI as data exchange interface

The XML Metadata Interchange (XMI) [7] is a standard for software tools to exchange models. The main purpose of XMI is to enable easy interchange of metadata among modeling tools and between modeling tools and metadata repositories in distributed heterogeneous environments. XMI integrates three key industry standards:

- XML - eXtensible Markup Language, a W3C standard;
- UML - Unified Modeling Language, an OMG modeling standard;
- MOF - Meta Object Facility and OMG modeling and metadata repository standard.

The integration of these three standards into XMI marries the best of OMG and W3C metadata and modeling technologies allowing developers of distributed systems share object models and other meta data over the Internet. XMI, together with MOF and UML form the core of the OMG repository architecture that integrates object oriented modeling and design tools between each other and with a MOF based extensible repository framework. This architecture allows tools to

share metadata programmatically. This allows the widest degree of latitude for tool, repository and object framework developers and lowers the barrier to entry for implementing OMG metadata standards.

2.3 Validation and Verification

Binder [10] says that model validation attempts to establish confidence in the sufficiency of a formal abstract component with respect to its informal behavioral requirements. He also states that verification attempts to show that the implementation is correct with respect to its representation, without executing it.

There are some studies on verification of model designs using Prototype Verification Systems (PVS) [11]. PVS is a specification language integrated with support tools and a theorem prover[13]. It is intended to capture the state-of-the-art in mechanized formal methods and to be sufficiently rugged that it can be used for significant applications. PVS is currently an evolving technology as new capabilities are embedded into the system.

In the following sections, the previous work related to validation and verification of UML designs, testing UML designs and checking the consistency of UML designs is given.

2.3.1 Verification and Validation of UML Designs

Precise UML Development Environment (PrUDE) is one of the projects that support UML verification, basing on the PVS [12]. The PrUDE project aims to develop a software verification and validation environment supporting model-checking and proof-checking. Model-checking and proof-checking are based on the PVS toolkit. UML is used as the graphical notation for PrUDE. The interface

[13] PVS Specification and Verification System. (n.d.). Retrieved January 25, 2006, from http://pvs.csl.sri.com/

of PrUDE with UML is based on XMI ([12] [13]). PrUDE takes an XMI file generated from the UML specification as input, then parses the XMI file, and then generates PVS files representing the formal semantics of the UML diagrams. The tool then checks the model for well-formedness, type-correctness by PVS type-checker and conformance with business rules by PVS theorem prover. PrUDE also generates test cases from valid UML diagrams [13]. Currently, PrUDE supports only UML statechart diagrams. Future release is planned to include several other diagrams such as class and sequence diagrams[14].

Arons et al [14] also proposes a PVS-based verification system called TLPVS, which aims the formal verification of linear temporal logic properties of systems. TLPVS includes a set of theories defining a temporal logic, a number of proof rules for proving soundness and response properties, and strategies which aid in conducting the proofs [15]. Similar to PrUDE, UML is used as the notation for diagrams and XMI is used for transforming UML diagrams into the PVS formats. TLPVS does not support the model checking against conformance with business rules and generation of test cases, which is possible with PrUDE. On the other hand, it allows verification of more complex properties such as liveness, asserting that under certain conditions a given event will occur, which lacks in PrUDE [12] [14]. Only the class diagrams and state machine diagrams are considered for TLPVS [15].

Giese et al claim that when it comes to the verification of complex systems, theorem proving is limited in terms of model checking [16].

Gallardo et al have presented a study on model checking of UML diagrams [17]. They recommend some tips for verification of statecharts and sequence diagrams to check the complex UML diagrams. Their work focuses on using existing tools

---

[14] PrUDE1.2 Documentation. (n.d.). Retrieved April 16, 2006, from http://www.isot.ece.uvic.ca/prude/manual.html

for SDL and MSC since they believe that these tools will continue to develop parallel to the development of UML.

Although, there are studies on the verification and validation of UML diagrams, they mainly focus on class and state diagrams. In our study, we have extended the domain of diagrams including use case, sequence and collaboration diagrams.

2.3.2 Testing UML Designs

There are various studies on detecting the UML model faults, which can be summarized as follows.

Andrews et al describe a technique for testing executable forms of UML models and propose test adequacy criteria for UML model elements [18]. They propose three test criteria for classes, associations and generalizations: Association-end multiplicity (AEM), generalization (GN) and class attribute (CS) criterion. For testing of collaboration diagrams, they propose five test criteria: Condition coverage (cond), full predicate coverage (FP), each message on link (EML), all message paths (AMP) and collection coverage (coll) [18]. These test adequacy criteria for class and collaboration diagrams are used in combination with each other to detect the faults in the model and evaluate the correctness of the model.

Briand and Labiche [19], propose a system testing methodology called Testing Object-Oriented Systems with the Unified Modeling Language, TOTEM. In their approach, Briand et al have derived requirements by analyzing the UML designs.

Abdurrazik and Offutt have developed test criteria for collaboration diagrams. The approach was to test the model itself and then use the model for generating test cases automatically [20].

Fraikin and Leonhardt present a tool SeDiTeC, which uses UML sequence diagrams to generate automated test cases [21]. In this project, although the main aim is to focus on automated test data generation, sequence diagrams are checked according to some predefined criteria in order to produce correct test data. Fraikin and Leonhardt [21] define the concept of "testable sequence diagram". According to this definition, a sequence diagram has to fulfill the criteria listed in Table 1 to be testable.

When these criteria are concerned, it can be seen that they are a very minimal subset of the well-formedness rules in the UML specification. Our study may be used to facilitate the current studies on automated test generation since automated test generation requires the UML model to be well formed.

Table 1 - The criteria for sequence diagrams to be testable

The criteria for sequence diagrams to be defined as testable
1. There is exactly one actor that represents the test tool. 2. There is at least one object. 3. For every object in the diagram the corresponding interface or class is specified. 4. Every object, upon which a non-static method is called, has a unique name. 5. The first method call of the diagram is initiated by the actor. 6. Every method call is associated with a declared method in the target class. 7. The method call sequence represents a single thread of execution.

In their study, Dinh-Trong et al [22] describe how a UML model can be executed and identify the structural and behavioral design characteristics that need to be observed during testing. In their approach, the UML design is first converted to an

executable form and then the tests are exercised and finally possible faults are reported.

Pilskalns et al [23] propose a graph-based approach, which uses both class and sequence diagrams. Sequence diagrams are transformed into Object-Method Directed Acyclic Graph (OMDAG). OMDAG is, then, used to derive test execution paths, which are recorded in a table called Object-Method Execution Table (OMET). Pilskalns et al [23] have successfully managed to define the test inputs in their study but the execution of the model using the derived inputs is not achieved.

Gogolla et al [24] have studied on validating UML class diagrams using snapshots, in their model USE. Snapshots are object diagrams, which represent system states consisting of objects possessing attribute values and links. Test cases are used to demonstrate that snapshots can be constructed to obey constraints in the model. This tool is limited since it uses the class diagrams only. Information from some other types of UML diagrams cannot be integrated for model execution.

2.3.3 UML Consistency Checking

Although, there are some studies on consistency checking of UML models, they mainly focus on state machines.

Latella et al. [25], encode UML state machines in the format of hierarchical automata, which are then compiled to SPIN [26]. Unfortunately, they do not support several important features of UML state machines such as do activities, entry and exit actions, completion events and transitions, history, and choice states, and it appears non-trivial to add some of these features in their framework.

Lilius and Porres employ a custom format to describe the semantics of UML state machines, their vUML tool, however, shows several deviations from the semantics, in particular, as regards completion events and transition selection [27]. In any case, the translations to such formats are non-trivial, and it is not obvious how to adapt them to changes in the UML semantics.

Knapp and Merz [28], describe a tool called HUGO, where model-checking technology is applied to relate UML state machines and interaction diagrams. The state machine view is considered as the “model” and the interaction view is accepted as the “property”. HUGO verifies the consistency of the state machines against the specifications. It can check whether there are deadlocks. HUGO also has a code generation module that produces JAVA code that behaves as prescribed by the state machines. HUGO supports a limited set of WFR when checking the consistency of the model.

Chiorean et al [29] underline that correctness against the UML definition has to be a prerequisite for every UML model. By correctness, they mean that the model should comply with the modeling language. They claim that to test the model’s correctness, it should be evaluated against the WFR. In their model, the model, which is in the XMI format, is loaded at first. Then, the WFR are loaded which are listed in a text file. After these steps have been accomplished, the model is checked. Chiorean et al [29] have used UML 1.4, which is now an older version. They also have not accomplished all of the WFR.

CHAPTER 3

UMLCHECKER

This chapter gives the implementation details of the library UMLChecker. In section 3.1 the methodology used for verifying the UML designs is given. Section 3.2 describes the architecture of the library. In Section 3.3 implementation details of the UMLChecker is explained in detail and finally in Section 3.4, UMLChecker is validated.

3.1. Verification of the UML Designs

The process of verifying the consistency of UML designs is summarized in Figure 12. There are two inputs to the system: the UML design model in XMI format and a set of well formedness rules, which the model will be checked against. After these inputs are given to UMLChecker, the consistency check is executed and the possible errors in the design are reported to the user or the caller program.

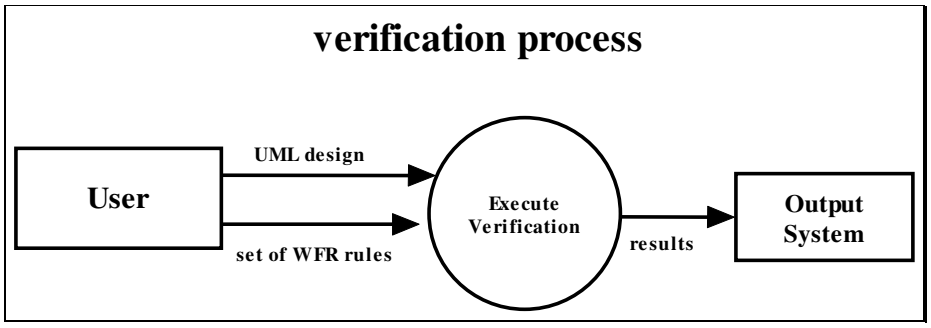


Figure 12 - Overview of the UML Design verification process

The details of the test input are explained in section 3.1.1 and the testing process is explained in section 3.1.2.

3.1.1. Inputs

The two types of inputs that the UMLChecker needs for checking the consistency of a UML diagram are explained in this section.

3.1.1.1. Well-Formedness Rules (WFR)

The prepared UML design will be checked according to the well-formedness rules specified in the UML specification. The rules, which the model will be checked against, should be loaded into UMLChecker when the program is run, initially.

Each WFR rule is separately implemented as a class, the details of which are explained in section 3.3. Users have two options for selecting which WFR rules to check the model against, which are also explained in detail in section 3.3.

3.1.1.2. UML Model in the XMI Format

The other and the most crucial input for the system is the UML design model. The model should be converted to XMI format before processing. Most of the modeling tools have plug-ins for converting the UML design to XMI format (such as Rational Rose[15], Metamill[16] etc.). Also, there are various free software which do the same task. Therefore we did not consider converting the UML into XMI in the scope of this thesis but made the assumption that a UML model can somehow be converted into the XMI format, which would then be used by our UMLChecker.

---

[15] IBM Rational Rose. (n.d.). Retrieved April 16, 2006, from http://www-128.ibm.com/developerworks/ webservices/ library/co-cow21.html

[16] Metamill. (n.d.). Retrieved April 16, 2006, from http://www.metamill.com/

Users can upload more than one XMI file representing the UML diagram. An example of the XMI input file is given while explaining the implementation details in section 3.2.

3.1.2. Verification Methodology

Verification is performed using the use case diagrams, class diagrams, collaboration diagrams and sequence diagrams. By the verification of the UML designs, we make sure that the diagrams are correct syntactically. Although UML drawing tools have some controls for the correctness, they are not enough for a UML design to be called well-formed.

After the WFR and the UML design in the XMI format is uploaded to UMLChecker, the verification of the design starts.

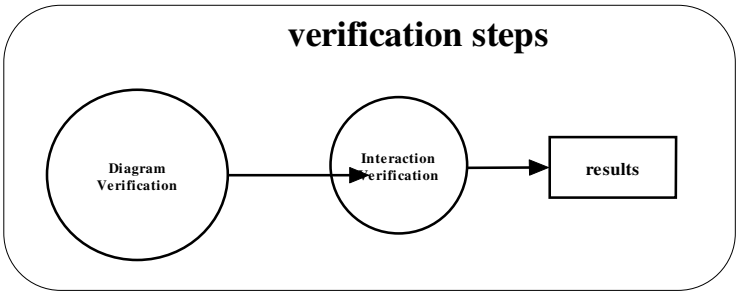


Figure 13 – Verification of the model

As seen in Figure 13, the model is checked within each diagram initially. For example, at the first step the class diagrams are checked against the well formedness rules. UMLChecker verifies whether each type of diagrams are designed correctly in the first place. If there are any errors found, the UMLChecker reports errors and continues to execute the check (i.e., it does not stop). After each diagram type is verified within itself, the UMLChecker makes the consistency checks considering the interactions between the different types of diagrams. For example, if a method, which does not exist in the class diagram, is

used in the sequence diagram, the UMLChecker recognizes this inconsistency and reports it to the user.

3.2. UMLChecker Architecture

UMLChecker is implemented as a stand-alone package so that it may be called from any program. In Figure 14, the basic architecture of the UMLChecker is shown.

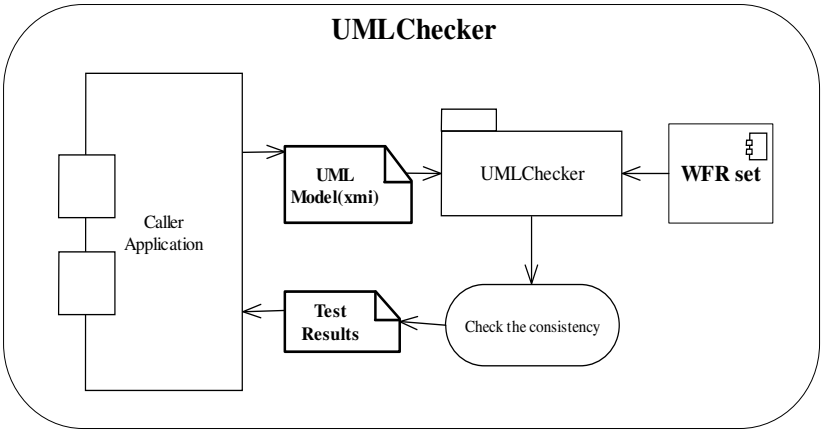


Figure 14 - Basic Architecture of the UMLChecker

UMLChecker is not an executable program; therefore an application that would evoke UMLChecker is needed. This caller application firstly inputs the UML model, which should be in XMI format. Then the application would call the UMLChecker that includes the well-formedness rules within. UMLChecker checks the UML model against the rules and reports the inconsistencies back to the caller application. The results could be displayed to the user within the caller application. Screenshots from a sample application that uses UMLChecker are displayed in Appendix B.

UMLChecker checks for the rules, which are listed in Appendix A.

3.3. UMLChecker Implementation Details

Our approach for checking the consistency of a UML model may be summarized as follows:

1. Load the elements from the XMI file.
2. Load the implemented well-formedness rules.
3. Identify the set of rules that should be applied to a specific element.
4. Check that the set of rules defined for each element is applied.

The details of this approach are explained in the following sections.

3.3.1. Tools and Libraries used by UMLChecker

We have developed the UMLChecker library using Eclipse SDK 3.1, using the Java 2 Platform Standard Edition Development Kit 5.0. In this thesis, we have used Metamill 4.2 for converting the model to XMI format. Additionally, two libraries are used to parse the XMI document, which are explained in the next two sections.

3.3.1.1. javax.xml.parsers.*

This package provides classes allowing the processing of XML documents. It supports two types of pluggable parsers: SAX (Simple API for XML) and DOM (Document Object Model) [17]. We have used the Document Object Model as will be explained in section 3.3.1.2.

We have used two classes of this package:

- DocumentBuilder: Defines the API to obtain DOM Document instances from an XML document.

---

[17] javax.xml.parsers. (n.d.). Retrieved April 15, 2006, from http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/package-summary.html

- DocumentBuilderFactory: Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.

3.3.1.2. org.w3c.dom.*

The Document Object Model (DOM) is an application programming interface (API) for valid HTML and well-formed XML documents. This package provides the interfaces for the Document Object Model, which is a component API of the Java API for XML Processing (JAXP). The Document Object Model Level 2 Core API allows programs to dynamically access and to update the content and structure of documents

It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term "document" is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data[18].

3.3.2. UMLChecker Architecture

In Figure 15, the architecture that lies beneath the UMLChecker is given. Each component is explained in detail in the following sections.

3.3.2.1. CheckerFacade

When exactly one object is needed to coordinate actions across the systems, the singleton and Façade design patterns are applied to restrict access and instantiation of a class to one object [30].

---

[18] Document Object Model (DOM) Level 2 Core Specification. (n.d.). Retrieved April 15, 2006, from http://www.w3.org/TR/DOM-Level-2-Core/

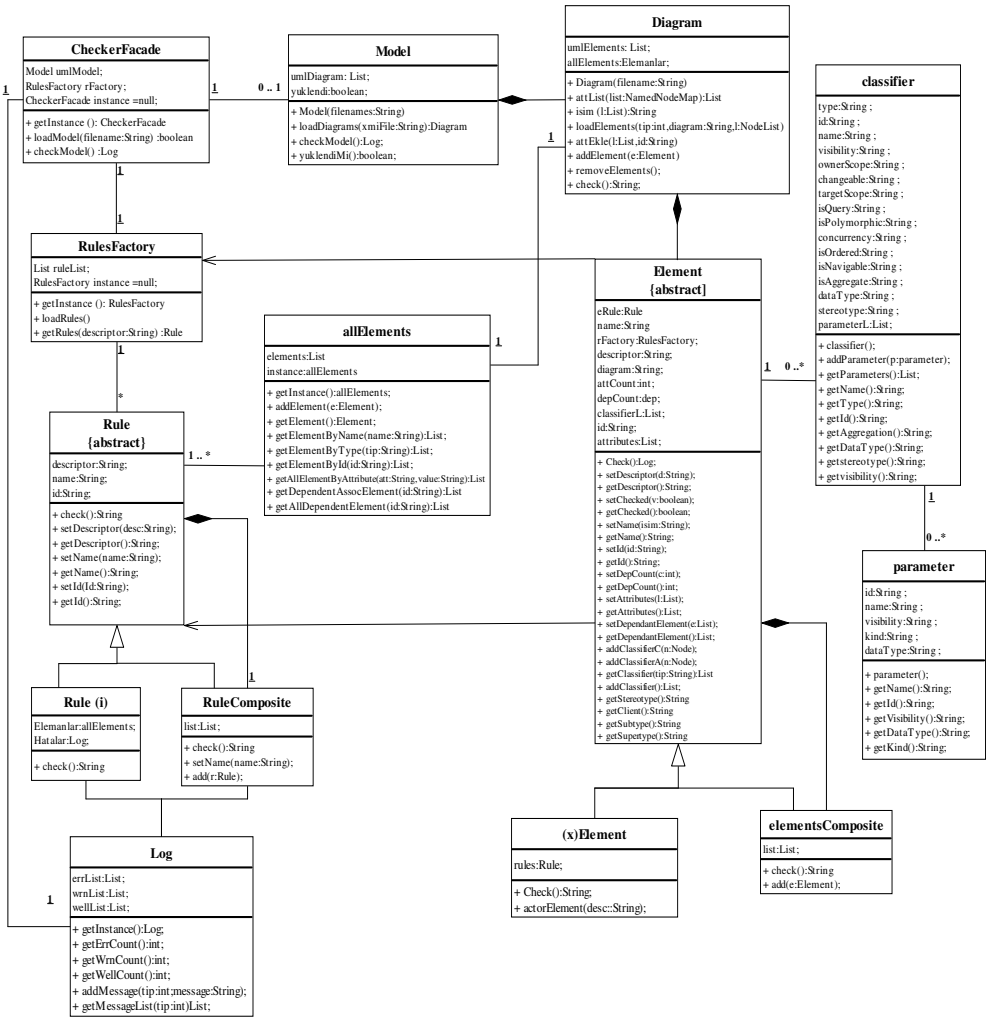


Figure 15 - Architecture of UMLChecker

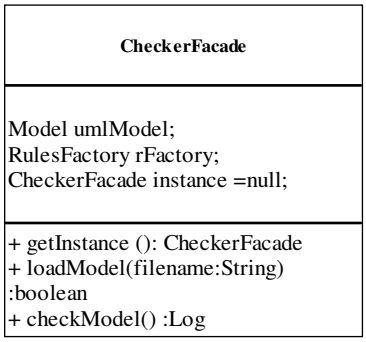


Figure 16 - CheckerFacade class

CheckerFacade is the main class, which is used as the interface between the caller application and the UMLChecker package.

The attributes and the operations of the class **CheckerFacade** (presented in Figure 16) are explained as follows:

- *umlModel:* used to store the models loaded in the XMI format.
- *rFactory:* used to store the rules that should be checked.
- *getInstance()*: as explained above, **CheckerFacade** class is a singleton class. This method is used to get the instance of the **CheckerFacade** that is previously instantiated. If there is not such an instance, a new one is created by this method.
- *loadModel(filename):* this method creates a new **Model** element as *umlModel*, mentioned above, and loads the model from the UML diagrams in XMI format.

  The filenames are given as an argument for this method. This argument should be a list consisting of the names of the files.
- *checkModel():* this method is used to check the model, which is loaded by the *loadModel()* method, against the WFR rules.

  The method returns an instance of a **Log** class, which contains errors, warnings and success messages about the consistency check.

3.3.2.2. RulesFactory

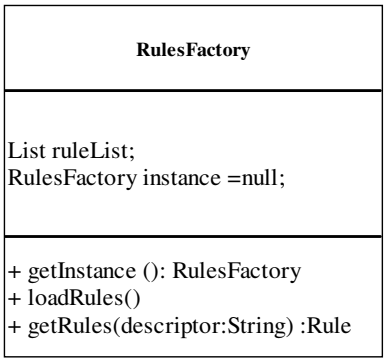


Figure 17 - RulesFactory Class

RulesFactory class loads all the rules used in tests. It is designed and implemented as a singleton class because it is needed to be instantiated for once.

The attributes and the operations of the class **RulesFactory** (presented in Figure 17) are explained as follows:

- *ruleList:* used to store all the rules which are implemented in the package.
- *getInstance()*: since **RulesFactory** class is a singleton class, this method is used to get the instance which is previously instantiated. If there is not such an instance, a new one is created by this method.
- *loadRules():* this method creates all the rules that are implemented. After creating the rules, their description must be set. Descriptors are used to identify which rules will apply for which kind of UML elements. Finally, created rules are added to the member "*ruleList*".

  getRules(descriptor): this method returns all the rules in "*ruleList*" for the specified descriptor.

3.3.2.3. Rule

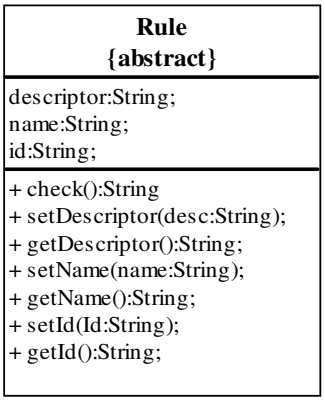


Figure 18 - Rule Class

Rule class in UMLChecker package is designed and implemented as an abstract class. All the rule classes explained in section 3.3.1.5 extends this class and overrides the *check()* method accordingly.

In Figure 18, the attributes and the operations of the class **Rule** are presented.

3.3.2.4. Rule 1 ... 129

For a UML design to be well-formed it should conform to all the rules listed in the UML specification. In this study, we have selected a representative subset of these rules and implemented them. The rules in this subset are listed and explained in Appendix A.

Each rule is implemented as a class, which extends the **Rule** class. Rules override the *check()* method of the **Rule** class such that it controls the model against the well-formedness rule for which it is created.

3.3.2.5. RuleComposite

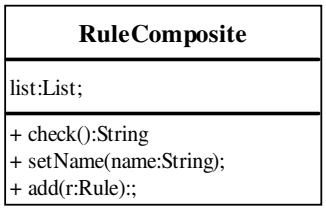


Figure 19 - RuleComposite class

RuleComposite extends the **Rule** class and is used to contain different kinds of rules within. In Figure 19, the attributes and the operations of the class **Rule** are presented.

3.3.2.6. Model

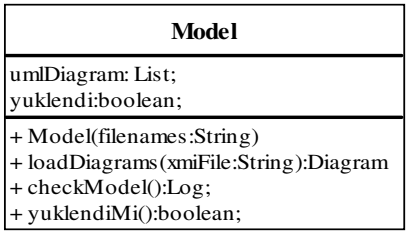


Figure 20 - Model class

The UML model which is in the form of XMI file are loaded into **Model** class and checked.

The attributes and the operations of the class **Model** (presented in Figure 20) are explained as follows:

- *umlDiagram: umlDiagram* is a list which holds the different kinds of UML diagrams such as use case diagram and class diagram.
- *model()*: the names of the files which contain the UML data in the xmi format are inputs for this method. For each file, the contents are loaded into *umlDiagram* list, using the *loadDiagram()* method.
- *loadDiagram(filename):* this method creates a new **Diagram** class which will contain the XMI data, and returns this diagram as a result.
- *checkModel():*each diagram in the *umlDiagram* list are checked and the results are returned to the caller.

3.3.2.7. Diagram

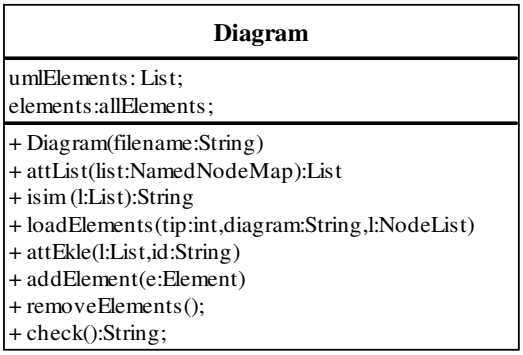


Figure 21 - Diagram Class

Each UML diagram in the UML model is stored as an instance of **Diagram** class.

The attributes and the operations of the class **Diagram** (presented in Figure 21) are explained as follows:

- *umlElements: umlElements* is a list which holds all the elements in a diagram (e.g. actors, use cases, associations, generalizations).
- *elements:* many of the well-formedness rules control the relations between the different elements within a diagram. Therefore each element should have access to other elements in order to check itself. Each element, which is added to *umlElements* list, is also added to *elements* list, which is an instance of **allElements** class, which will be explained in the following section.
- *Diagram()*: the name of the file which contain the UML diagram in the xmi format is an input for this method. The file is parsed using **DocumentBuilderFactory.** The libraries used from parsing the model are explained in sections 3.3.1.1 and 3.3.1.2.

  After the XMI data is parsed into Document Object Model, each type of element in the model is inserted into the *umlElements* list using *loadElements* method.
- *loadElements():* type of the element (e.g. actor) and the parent node of the Document Object Model are inputs for this method. Each parent node is parsed to find the classifiers, constraint, attributes, and operations etc. of the element, if applicable. When an element is fully extracted from the Document Object Model, it is added to the *umlElements* and *allElements* lists.
- *checkModel():* each element in the *umlElements* list are checked and the results are return to the caller.

3.3.2.8. Element

Element class in UMLChecker package is designed and implemented as an abstract class. All the element classes explained in section 3.3.1.10 extend this class and override the *check()* method.

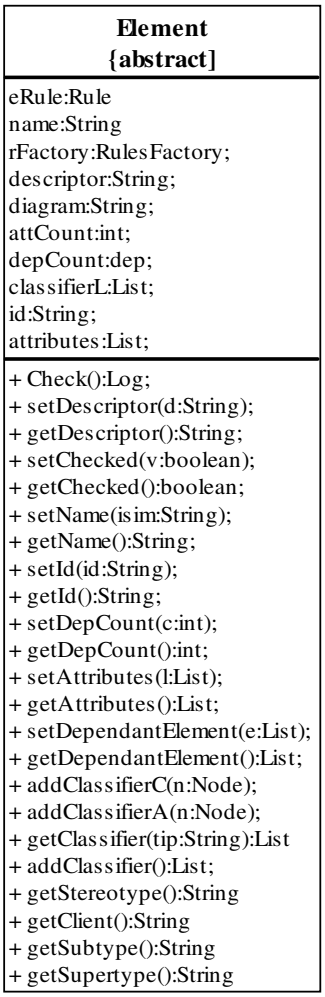


Figure 22 - Element class

The attributes and the operations of the class **Element** (presented in Figure 22) are explained as follows:

- *eRule: is a list which contains all the rules which apply to the element.*
- *classifierL:* many elements have classifiers within itself. For example, a class has attributes and an association has association ends. The classifiers of an element are set when loading the elements in the **Diagram** class with the *loadElements()* method.
- *attributes:* each element may have properties such as isLeaf, isVisible etc. Attributes list contain all these properties of an element, which are set

while loading the elements in the **Diagram** class with the *loadElements()* method.

- *Check():* element is checked against all the rules added in *eRule* list and the results are reported back to the caller.

3.3.2.9. allElements

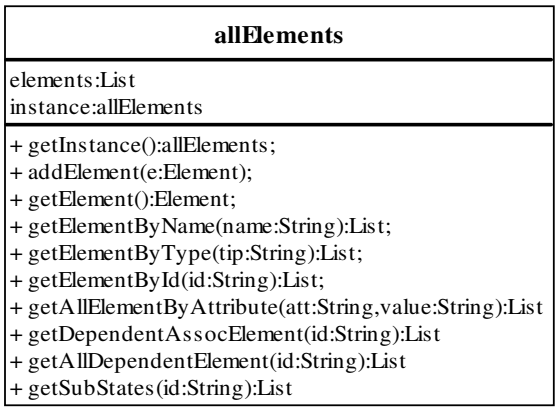


Figure 23 - allElements class

allElements class loads all the elements which are loaded from the XMI files. It is designed and implemented as a singleton class because it is needed to be instantiated for once.

The attributes and the operations of the class **allElements** (presented in Figure 23) are explained as follows:

- *elements:* used to store all the elements which exist in the UML model.
- *getInstance()*: since **allElements** class is a singleton class, this method is used to get the instance which is previously instantiated.
- *getElementByName():* returns a list including all the elements having the same name with the input parameter.
- *getElementByType():* returns a list including all the elements having the same type with the input parameter.

- *getElementById():* returns a list including all the elements having the same id with the input parameter. When a UML diagram is exported into xmi format, each element is given a unique id and all the references are stored with this id. Since it is the only attribute which may be unique among the elements, the id attribute is used while processing the elements.
- *getAllElementByAttribute():* returns a list including all the elements having the same attribute with the input parameter.
- *getDependentAssocElement():* an id is an input for this method. It returns all the associations of an element with the given id.
- *geAlltDependentElement ():* an id is an input for this method. It returns all the other elements which are associated to an element with the given id.

3.3.2.10. Element classes

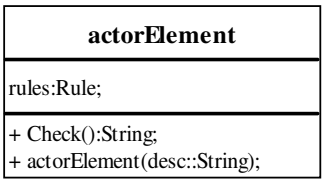


Figure 24 - An example element class: actorElement class

Each different type of element is implemented as a separate class. Some of these elements are:

- Actor
- Association
- Class
- Generalization
- Stereotype
- Use case
- Pseudo state
- Composite state

In the *loadElements()* method of **Diagram** class, each element is created according to its type and added to the element list.

Elements are checked against the WFR rules by the *check()* method. Firstly, the instance of the **RulesFactory** is obtained and then all the rules, which have the same descriptor with the element to be checked, are obtained. Thus, each element has the information for what rules to check itself against. In Figure 24, an example of an element class, **actorElement**, is presented.

3.3.2.11. Classifier

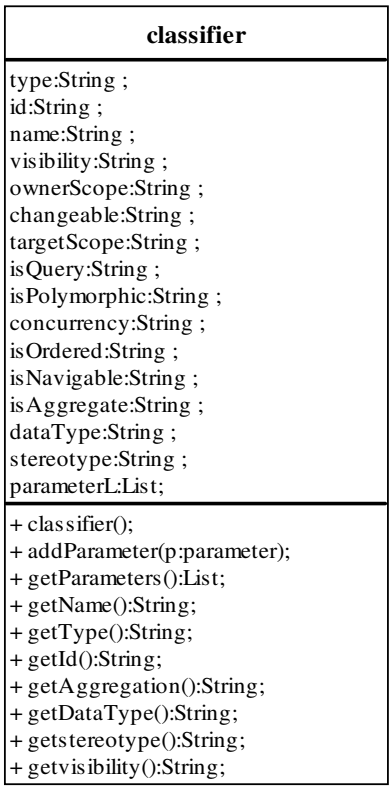


Figure 25 - classifier element

Elements have classifiers within itself, which are frequently used while checking the element against the well-formedness rules. The **classifier** class is used to contain all the classifiers within an element.

The attributes and operations defined in **classifier** class (presented in Figure 25) are explained as follows:

- The attributes defined in the classifiers include all the data needed to make the checks. Each attribute may be associated with different types of classifiers. For example, *IsPolyMorphic* is meaningful for the operations but irrelevant for the attributes, on the other hand, *id* is required for any kind of classifier.

3.3.2.12. Parameter

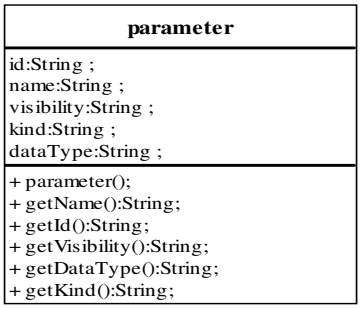


Figure 26 - parameter class

The operations may have parameters, which are used while checking the element against the well-formedness rules. The **parameter** class is used to contain all the parameters of an operation within an element.

The attributes (presented in Figure 26) include the name, visibility, data type and return type of the parameter.

3.3.2.13. Log

When each rule checks itself, it stores the resulting errors and warnings in the **Log** class, which is designed and implemented as a singleton object.

Log is an important class, in the sense that is used as an interface between the UMLChecker package and the caller application. The results of the checking process are stored in the **Log** class and returned to the caller.

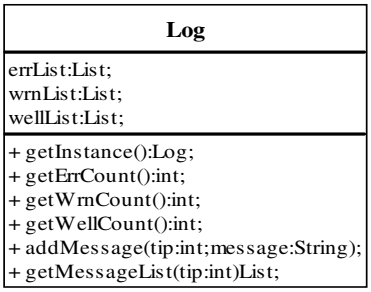


Figure 27 - Log class

The attributes and operations defined in **Log** class (presented in Figure 27) are explained as follows:

- *ErrList, wrnList, wellList;* These attributes are defined as lists containing the message which state the result of a rule applied to an element.
- *addMessage():* is used by the **Rule** class to add the messages to **Log** instance.
- *getMessage():* is used to get the results of the check.

3.3.3. Checking the Model against the Well-Formedness Rules

In this section how the UMLChecker operates will be explained. There are 88 rules implemented in the UMLChecker. Although the checks are different from each other, checking process is quite similar for all of the rules. Therefore, an example rule is selected and explained in detail.

In Figure 28, how the UMLChecker operates is shown as a sequence diagram for a sample rule (rule number 96: actors can only have associations to use cases and classes and these associations are binary).

As mentioned in previous sections, UMLChecker is implemented as a package that may be used freely by any program. When this is the case, the programmers using the UMLChecker should pursue the following steps:

- The caller application gets the instance of the class **CheckerFacade.**

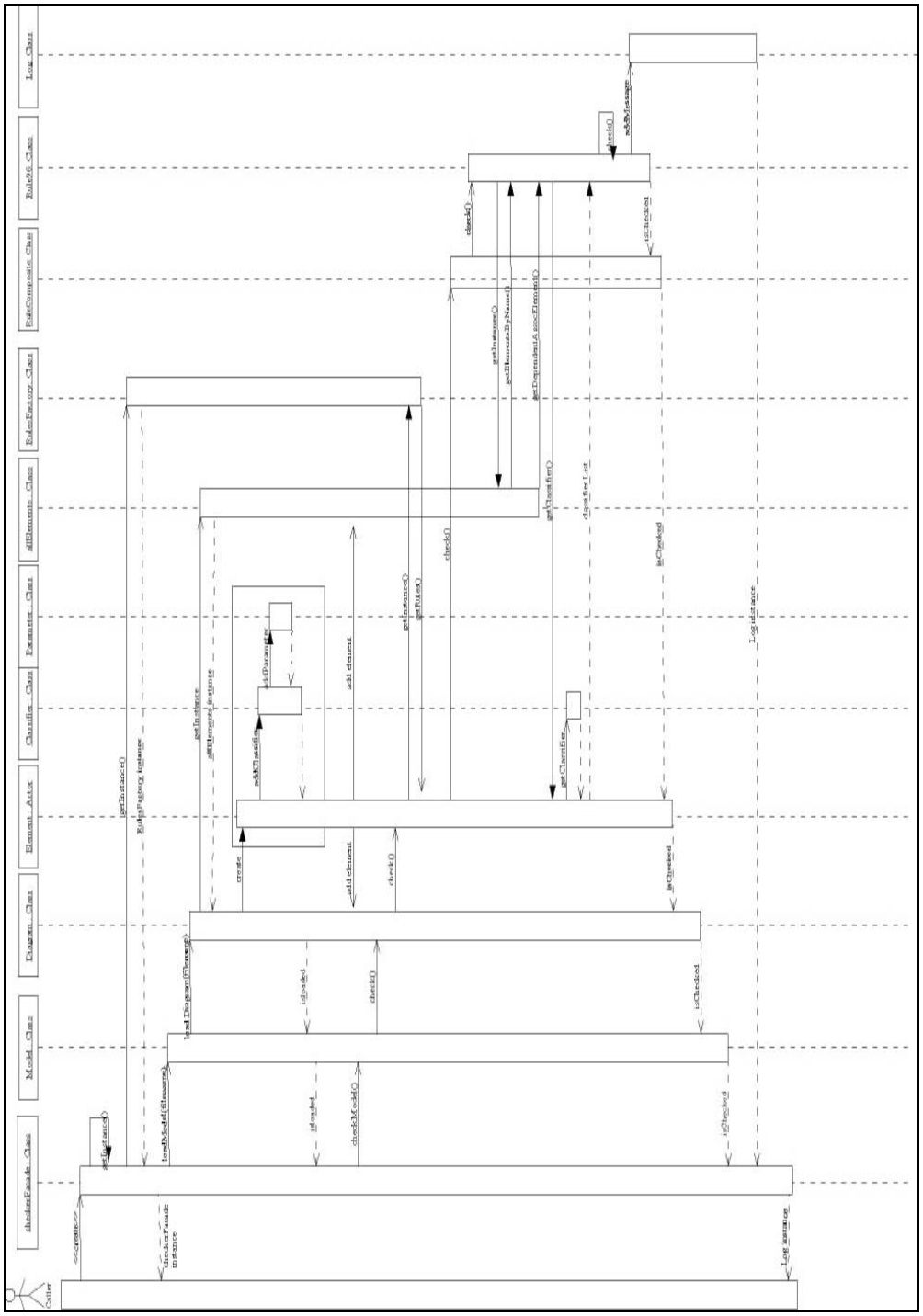


Figure 28 - Sequence Diagram for Rule96

- The caller application loads the model by inputting the filenames each containing a model diagram, as seen in Figure 29.

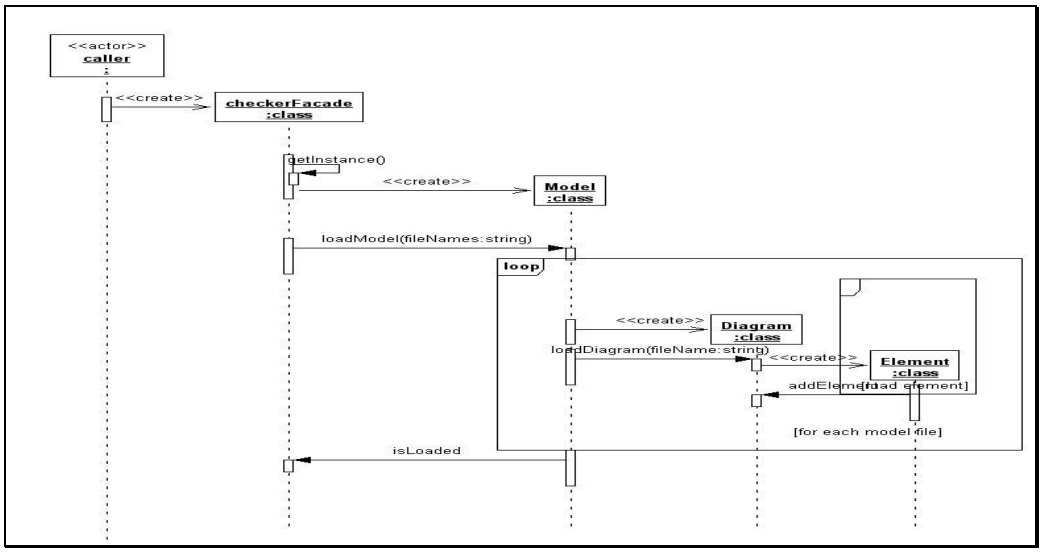


Figure 29 - Sequence Diagram for loading the rules

- The caller application checks the model by invoking *checkModel()* method of the class **CheckerFacade,** as seen in Figure 30**.**
- The results will be returned as a **Log** class to the caller application.

A sample application is also developed to show the use of UMLChecker within an application. Sample screens of this application may be seen in Appendix B.

The UMLChecker package is also suitable for adding or removing the well-formedness rules. When this is the case:

1. Users may want their UML diagram to be checked against all the rules defined in UMLChecker. If this is the case, *loadRules()* method of

RulesFactory class will load all the implemented WFR rules to the rule factory and these rules will be checked.

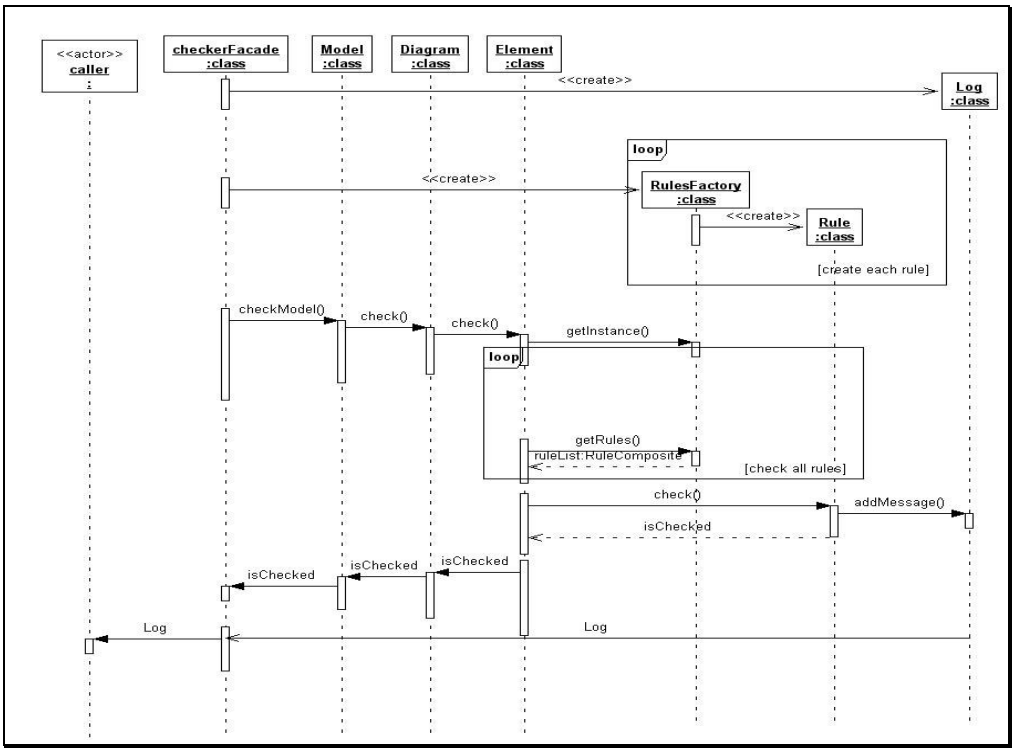


Figure 30 - Sequence Diagram for checking the rules

2. Considering the fact that UMLChecker is intended for use as a separate module by other researchers, we have found it more effective to upload the rules to UMLChecker. By this way, only the rules that needed to be fulfilled by the model can be checked. For example, if a researcher is interested only in class diagrams he/she may have the extensibility to check only the class diagrams of the model. In this case, users should override the *loadRules()* method of *RulesFactory* class. Loading the rules

is quite easy and straightforward. The code in Figure 31 explains how a user can define the set of rules that should be loaded into the UMLChecker:

```
1 Rule1 r1 = new Rule1();
2 r1.setDescriptor (ACTOR);
3 ruleList.add(r1);
```

Figure 31 - sample code to add a rule

- User should at first create the rule, which is to be checked. In the fist line, *Rule1,* which is the class that implements the WFR rule number one.
- Then the descriptor of the rule should be set. Descriptors are used to differentiate which element group may use which rules. In this example Rule1 is a rule associated with the actor elements in UML diagrams.
- Finally the created rule should be added to the rules factory. So that all the elements in the UML diagram may be aware of this rule and check itself against the rule.

If the user feels that the previously implemented rules are not enough and additional rules must be implemented, this may be achieved by a very similar code as in Figure 31. The users of UMLChecker library should do the following to add new rules:

- User should implement a rule e.g. *rule548,* by extending the **Rule** class. The following code segment shows how to create the new rule.

```
public class Rule548 extends Rule{
        public String Check(){
        // the code for checking new constraints is here
        }
}
```

- After implementing the rule, it should be created and added to the rule list as in the following code.

```
1 Rule548 r548 = new Rule548();
2 r548.setDescriptor (CLASS);
3 ruleList.add (r548);
```

3.4. Validation of UMLChecker

The Informatics Institute of METU and several other firms have been carrying out a large-scale project. Many people from different disciplines are contributing to this project. We have used the work products of this project to validate UMLChecker.

The design models of this project have been prepared by a group of people and documented accordingly. Another group of people have reviewed the artifacts including the design model. The review has been done manually; That is, no tools for automated review have been used. After each review, the design has been revised and the required corrections are performed. Therefore, the design has become more accurate iteration by iteration.

The errors found in the reviews contain not only the modeling errors but also the errors within the document such as unexplained items in the models or grammar and spelling errors. There were nearly 200 of errors concerning the design document. We have eliminated the irrelevant error types and considered only the modeling errors. The number of such errors was found to be 24.

In order to validate UMLChecker, we have chosen a sample diagram set from the design model of this project and exported them into the XMI format. After this, we have input the selected diagrams to the UMLChecker. We then compared the results obtained from the UMLChecker and those of the manual review.

The errors in the design model can be grouped as follows:

1. Errors relating to duplicate declaration, i.e. some operation names are declared twice.
2. Errors relating to unidentified references, i.e. some methods are used in sequence diagrams but they do not exist in the corresponding class.
3. Errors relating to general design flaws, i.e. an abstract class was inherited by a non-abstract class.

The results obtained from the UMLChecker are compared with those of manual review. The results are as follows:

1. UMLChecker recognized the first group of errors, mentioned above. And also some errors, which have missed in the manual review document, were recognized.
2. The UMLChecker has recognized a large group of errors in the second group but there were some missing errors. The main reason for

UMLChecker not to find the errors is the format of the XMI files including the model. Although the XMI is a standard format, the way the model is expressed with XMI may vary among the modeling tools. The UMLChecker was unable to parse some of the XMI models exported from the tool that the models were developed with. A wider-range scoped study should be carried out so that the UMLChecker may support various types of modeling tools.

3. UMLChecker could not recognize some of errors in the third group since they were not implemented. In the rule set of UMLChecker, there was not a control checking if an abstract class has been inherited by a non-abstract class, therefore some of the errors were missed.

To sum it up, the results of the comparison are summarized as follows:

- Out of 24 errors found manually for the selected diagrams, the UMLChecker found 17 errors.
- 2 errors were found which lacks in the manual review.
- The errors, which have not been recognized by the UMLChecker, were mostly because of the variations in the XMI format exported by different modeling tools.
- There were also some unrecognized errors by the UMLChecker, which were not in the rule set of the UMLChecker.

When the number of the errors found by the UMLChecker is concerned, it can be seen that the tool we have developed may be used for verifying the UML models, minimizing the need for manual reviewing of the model.

CHAPTER 4

CONCLUSION AND FUTURE WORK

Although various kinds of modeling tools and techniques exist in the industry, when object-oriented software development is concerned, UML is the widely used one mainly because of its standardization. UML has become popular recent years in many fields such as Model Driven Development, test case generation and code generation.

If the UML model contains errors, any progressive work done using the model as a basis would also be erroneous. Therefore, the use of UML in all the subject areas mentioned above requires the model to be accurate and consistent. Many research has already been done on UML semantics, however, there are still some points lacking. More support to detect the design inconsistencies in the earlier phases of development is needed. An approach for verifying the UML models is highly required.

Some researchers studied on this subject previously but the scope of their work was limited. Chiorean et al [29] have used an older version of UML specification and they also have not accomplished a majority of the WFR. Knapp and Merz [28], described a tool called HUGO, where model-checking technology is applied to relate UML state machines and interaction diagrams. Similarly, Pnueli et al considered only the class diagrams and state machine diagrams for TLPVS [15].

In this thesis, unlike previous studies, we support a wider range of domains of diagrams including use case, class, state, sequence and collaboration diagrams. We have also enlarged the number of rules applied for consistency checking. Another important distinguishing property of UMLChecker is being a reusable and pluggable library.

This thesis presents a library, UMLChecker, which enables modelers to verify their UML model as a whole. UML models to be verified are transformed into XMI format and then verification is performed. In this thesis, a set of rules is described which needs to be fulfilled for a model to be well formed.

As mentioned in the goals of this thesis:

- The UMLChecker is implemented as a library package so that developers can embed the UMLChecker into their applications.
- Unlike most of the UML CASE tools, the software we have developed is free to use and can be called from any program.
- It is suitable to be enriched with other functionalities to check the model against different constraints.

In this thesis, we had some restrictions while implementing the WFR. The nature of some rules was not suitable for an accurate consistency checking. For example, rule 10 (The type of the Parameters should be included in the Namespace of the Classifier) cannot be fully checked. If the type is en element defined in the model there would be no problem but otherwise, types such as integer, String, float etc., would not be found in the namespace and an error should be reported. Considering the fact that checking rules relating to the data types is suspicious, they do not report an error but a warning.

The second restriction is caused by capability of UML modeling tools. For example, methods are the implementations of operations. But, most usually method bodies are not shown at all on UML diagrams [1]. Therefore rules relating

to the implementation of operations, namely methods, were not implemented since they lacked in UML designs.

Another restriction for a wider use of UMLChecker may occur while parsing the model in the XMI format. UML tools have slight differences while converting the model to XMI format. In this thesis we have used Metamill 4.2 is for creating the XMI data from the UML model. As a future work, UMLChecker may be enhanced to support XMI formats generated from other UML tools. This would enable modelers to use the UML Case tool of their choice.

We have validated the UMLChecker with the design models of a large-scale project carried out in the Informatics Institute. We have compared the results of the manual review of the models with the results of the UMLChecker. We have seen that the UMLChecker recognized the majority of the errors. When the restrictions mentioned above are eliminated by a future study and UMLChecker is improved, it may be used for verifying the UML models in real projects, thus eliminating the need of manual reviewing of the model.

REFERENCES

[1] Rumbaugh J., Jacobson I., Booch G. (2000). The Unified Modeling Language Reference Manual. Addison Wesley.

[2] The Object Management Group. (2003). The Unified Modeling Language UML 1.5. Technical Report formal/03-03-01. The Object Management Group.

[3] Kleppe A., Warmer J., Bast W. (2003). MDA Explained: The Model Driven Architecture Practice and Promise. Addison Wesley.

[4] Offutt J., Abdurazik A. (1999, October). Generating tests from UML specifications. UML 1999 – the Unified Modeling Language. Beyond the Standard. 2nd International Conference Proceedings.

[5] The Object Management Group. (2005) The Unified Modeling 2.0 Infrastructure Specification formal/05-07-05. The Object Management Group.

[6] Fowler M., Scott K. (2000). UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language. Addison Wesley.

[7] XML Metadata Interchange. (2003). MOF 2.0/XMI Mapping Specification, V2.1, formal/05-09-01. The Object Management Group.

[8] Larman C. (2001). Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and the Unified Process. Prentice Hall.

[9] Cantor M. (1998). Object Oriented Project Management with UML. John Wiley & Sons.

[10] Binder R.V. (1999). Testing Object-Oriented Systems Models, Patterns, and Tools. Addison Wesley.

[11] Traore I. (2000, November). An Outline of PVS Semantics for UML Statecharts. Journal of Universal Computer Science (JUCS), Springer Pub. Co., Vol. 6, No. 11, 1088-1108.

[12] Traore I. (2001, August). An Integrated V&V Environment for Critical Systems Development. 5th IEEE International Symposium on Requirements Engineering Proceedings, 287.

[13] Traore I., Aredo D.B. (2004, November). Enhancing Structured Review with Model-based Verification. IEEE Transactions on Software Engineering, Vol. 30, No. 11, 736-753.

[14] Arons T., Hooman J., Kugler H., Pnueli A., van der Zwag M. (2004, October). Deductive Verification of UML models in TLPVS. UML 2004 – the Unified Modeling Language. Modeling Languages and Applications. 7th International Conference.

[15] Pnueli A., Arons T. (2004). TLPVS: a PVS based LTL verification system. Verification: Theory and Practice, vol.2772 of LNCS, 598-625.

[16] Giese H., Tichy M., Burmester S., Schafer W., Flake S. (2003, September). Towards the Compositional Verification of Real-Time UML Designs. ACM SIGSOFT Software Engineering Notes, vol.28, no.5.

[17] Gallardo M., Merino P., Pimentelis E. (2002, July). Debugging UML Designs with Model Checking. Journal of Object Technology, vol. 1, no. 2, 101-117.

[18] Andrews A., France R.B., Ghosh S., Craig G. (2003, April). Test Adequacy Criteria for UML Design Models. Journal of Software Testing, Verification and Reliability, vol. 13, no. 2, 95-127.

[19] Briand L., Labiche Y. (2002). A UML-based Approach to system testing. Software and system Modeling, vol. 1, no. 1, 10-42.

[20] Abdurazik A., Offutt J. (2000, October). Using UML collaboration diagrams for static checking and test generation. UML 2000 – the Unified Modeling Language. Advancing the Standard. 3rd International Conference.

[21] Fraikin F., Leonhardt T. (2002). SeDiTeC – Testing Based on Sequence Diagrams. 7th IEEE International Conference on Automated Software Engineering Proceedings, 261-266.

[22] Dinh-Trong T., Kawane N., Ghosh S., France R.B., Andrews A.A. (2005, June). A tool supported Approach to Testing UML Design Models. 10th IEEE International Conference on Engineering of complex Computer systems Proceedings.

[23] Pilskalns O., Andrews A., Ghosh S., France R.B. (2003, October). Rigorous Testing by Merging structural and Behavioral UML Representations. In Proceedings of the 6th International Conference on the Unified Modeling Language, 234-248.

[24] Gogolla M., Bohling J.,Richters M. (2003). Validation of UML and OCL models by automatic snapshot generation. Proceedings of the 6th International Conference on the Unified Modeling Language, 265-279.

[25] Latella D., Majzik I., Massink M. (1999). Automatic Verification of a Behavioral Subset of UML Statechart Diagrams Using the SPIN Model-Checker. Formal Aspects Comp, vol. 11, no. 6, 637-664.

[26] Holzmann G.J. (1997). The Model Checker SPIN. IEEE Transactions on software Engineering, vol. 5, no. 4.

[27] Lilius J., Paltor I.P. (1999). Formalizing UML State Machines for Model Checking. Proceedings of the 2nd International Conference on the Unified Modeling Language, vol. 1723 of LNCS, 430–445.

[28] Knapp A., Merz S. (2002). Model Checking and code Generation for UML State Machines and Collaborations. 5th workshop on Tools for System Design and Verification.

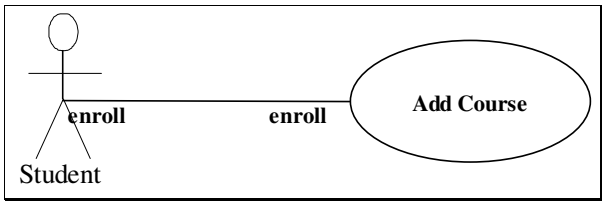
[29] Chiorean D., Carcu A., Pasca M., Botiza C., Chiorean H., Moldovan S. (2002). UML Model Checking. Informatica, vol.157, no. 1.

[30] Cooper J. W. (2002). Visual Basic Design Patterns. Addison Wesley.

APPENDICES

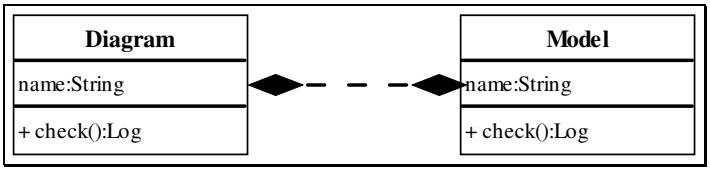
A. IMPLEMENTED WELL FORMEDNESS RULES[19]

1. The AssociationEnds must have a unique name within the Association.



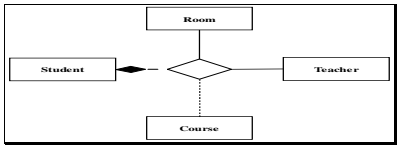
The association between the actor "Student" and the use case "Add Course" has two association ends both of which are named as "enroll". Because same name is given to both association ends, this diagram breaks the WFR.

2. At most one AssociationEnd may be an aggregation or composition.



The class "Diagram" is composed of the class "Model" and also, the class "Model" is composed of the class "Diagram". Since both of the association ends are composition, this diagram is an example of violation of this WFR.

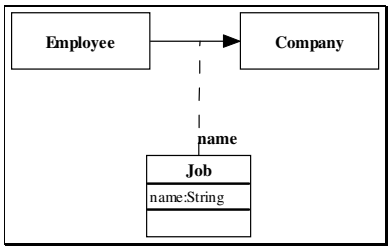
3. If an Association has 3 or more AssociationEnds then no AssociationEnd may be an aggregation or composition.



[19] UML Semantics. (n.d.). Retrieved January 16, 2006, from http://www-inf.int-evry.fr/COURS/UML/semantics/

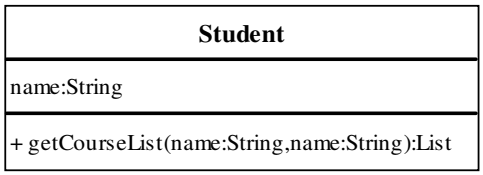
In this ternary relationship describing the course relationship, one association end is a composition, which breaks this WFR.

4. The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association.
5. The names of the AssociationEnds and the StructuralFeatures do not overlap.



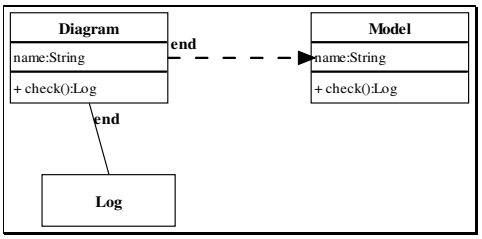
In this association class example, one association end is named as “name”, which is also the name of an attribute within the class “job”. This association class is not considered as well-formed since it breaks WFR.

6. An AssociationClass cannot be defined between itself and something else.
7. The Classifier of an AssociationEnd cannot be an Interface or a DataType unless the DataType is part of a composite aggregation.
8. All Parameters should have a unique name.

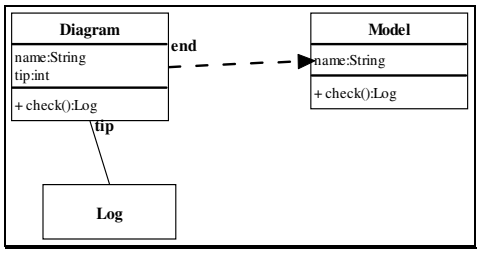


This class is an example for the violation of this WFR.

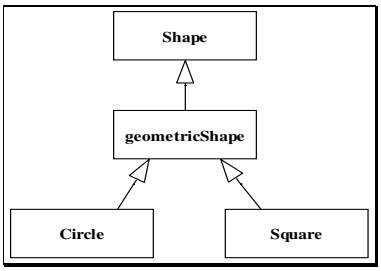
9. The type of the Parameters should be included in the Namespace of the Classifier.
10. A Class can only contain Classes, Associations, Generalizations, UseCases, Constraints, Dependencies, Collaborations, and Interfaces as a Namespace.
11. No BehavioralFeature of the same kind may have the same signature in a Classifier.
12. No Attributes may have the same name within a Classifier.
13. No opposite AssociationEnds may have the same name within a Classifier.



14. The name of an Attribute may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier.

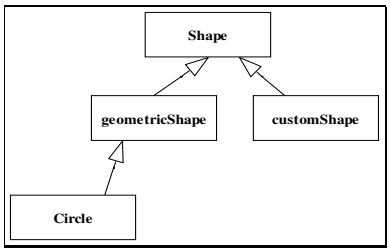


15. The name of an opposite AssociationEnd may not be the same as the name of an Attribute or a ModelElement contained in the Classifier.
16. A Constraint cannot be applied to itself.
17. A DataType can only contain Operations, which all must be queries.
18. A DataType cannot contain any other ModelElements.
19. A root cannot have any Generalizations.



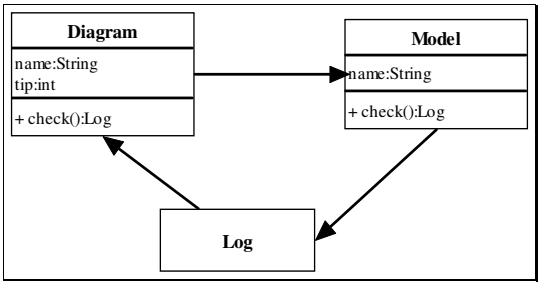
The class “geometricShape” is designed to be a root. Therefore, according to the WFR, it should not have any generalizations, but it has. Because of this, this class design is not well-formed.

20. No GeneralizableElement can have a supertype Generalization to an element which is a leaf.

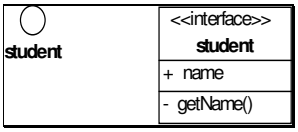


The class “geometricShape” is designed to be a leaf. Therefore, according to the WFR, it should not be a supertype for any generalizations, but it is. Because of this, this class design is not well-formed.

21. Circular inheritance is not allowed.

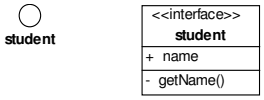


22. The supertype must be included in the Namespace of the GeneralizableElement.
23. A GeneralizableElement may only be a subclass of GeneralizableElement of the same kind.
24. An Interface can only contain Operations.



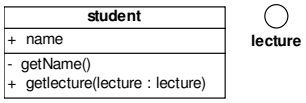
“student” interface has an attribute called “name” which breaks the rule.

25. An Interface cannot contain any Classifiers.
26. All Features defined in an Interface are public.



The interface “student” has an operation named “getName” which is defined as private, which breaks the rule.

27. If a contained element, which is not an Association or Generalization, has a name then the name must be unique in the Namespace.
28. All Associations must have a unique combination of name and associated Classifiers in the Namespace.
29. An Interface cannot be used as the type of a parameter.

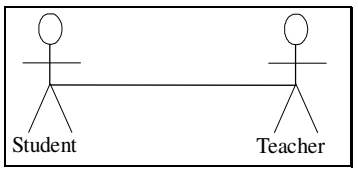


The "student" class has an operation "getLecture" which has a parameter of type "lecture" defined as an interface. A well-defined class should not include this parameter.

30. The connected type should be included in the current Namespace.
31. The argument ModelElement must conform to the parameter ModelElement in a Binding. In an instantiation it must be of the same kind.
32. A Constraint attached to a Stereotype must not conflict with Constraints on any inherited Stereotype, or associated with the base Class.
33. A Constraint attached to a stereotyped ModelElement must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the baseClass) of the ModelElement
34. A Constraint attached to a Stereotype will apply to all ModelElements classified by that Stereotype and must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the base Class) of the ModelElement
35. Stereotype names must not clash with any baseClass names
36. Stereotype names must not clash with the names of any inherited Stereotype.
37. Stereotype names must not clash in the (M2) meta-class namespace, nor with the names of any inherited Stereotype, nor with any baseClass names
38. The base Class name must be provided; icon is optional is specified in an implementation specific way
39. Tag names attached to a Stereotype must not clash with M2 meta-attribute namespace of the appropriate baseClass element, nor with Tag names of any inherited Stereotype

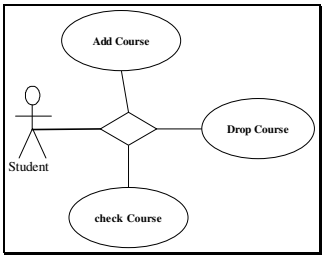
40. Tags associated with a ModelElement (directly via a property list or indirectly via a Stereotype) must not clash with any meta-attributes associated with the Model Element.
41. A model element must have at most one tagged value with a given tag name.
42. The types and order of actual arguments for an Action must match the parameters of the Request.
43. A CallAction must have exactly one target
44. The type of the dispatched Request should be Operation.
45. A CreateAction does not have a target expression.
46. A DestroyAction should not have arguments
47. The Links matches the declarations in the Classifiers.
48. If two Operations have the same signature they must be the same.
49. There are no name conflicts between the AttributeLinks and opposite LinkEnds.
50. There are not two Links of the same Association which connects the same set of Instances in the same way.
51. Each of the Classifiers must be a kind of Class.
52. The type of the ClassifierRole must conform to the type of the base AssociationEnd.
53. The type must be a kind of ClassifierRole.
54. The AssociationEndRoles must conform to the AssociationEnds of the base
55. The endpoints must be a kind of AssociationEndRoles.
56. The AssociationRoles connected to the ClassifierRole must match a subset of the Associations connected to the base Classifier.
57. Features of ClassifierRole must be a subset of those of the base Classifier.
58. A ClassifierRole does not have any Features of its own.
59. Classifiers and Associations of ClassifierRoles and AssociationRoles in Collaboration should be included in namespace owning the Collaboration.
60. All the constraining ModelElements should be included in the namespace owning the Collaboration.

61. If a ClassifierRole or an AssociationRole does not have a name then it should be the only one with a particular base.
62. A Collaboration may only contain ClassifierRoles and AssociationRoles.
63. Actors can only have Associations to UseCases and Classes and these Associations are binary.



In this example, the actor “student” has an association to the actor “teacher”, which is against this WFR.

64. Actors cannot contain any Classifiers.
65. UseCases can only have binary Associations.



In this example, the ternary association causes not to be well-defined.

66. UseCases can not have Associations to UseCases specifying the same entity.
67. A UseCase can only have ´uses´ or ´extends´ Generalizations.
68. A UseCase cannot contain any Classifiers.
69. A composite state can have at most one initial vertex
70. A composite state can have at most one deep history vertex
71. A composite state can have at most one shallow history vertex
72. There have to be at least two composite substates in a concurrent composite
73. Initial vertex can have one outgoing transition and no incoming transitions
74. A final pseudo state cannot have outgoing transitions
75. History vertices can have at most one outgoing transition
76. A join vertex must have at least two incoming transitions and exactly one outgoing transition

77. A fork vertex must have at least two outgoing transitions and exactly one incoming transition
78. A branch vertex must have one incoming transition segment and at least two outgoing transition segments with guards.
79. StateMachine is aggregated within either a classifier or a behavioral feature.
80. A top state is always a composite
81. A top state cannot have parents
82. The top state cannot be the source or target of a transition
83. There can be no history vertices in the top state
84. A fork segment should not have guards or triggers
85. A join segment should not have guards or triggers
86. A fork segment should always target a state
87. A join segment should always originate from a state
88. A branch segment must not have a trigger

B. SCREENSHOTS OF AN APPLICATION USING UMLCHECKER

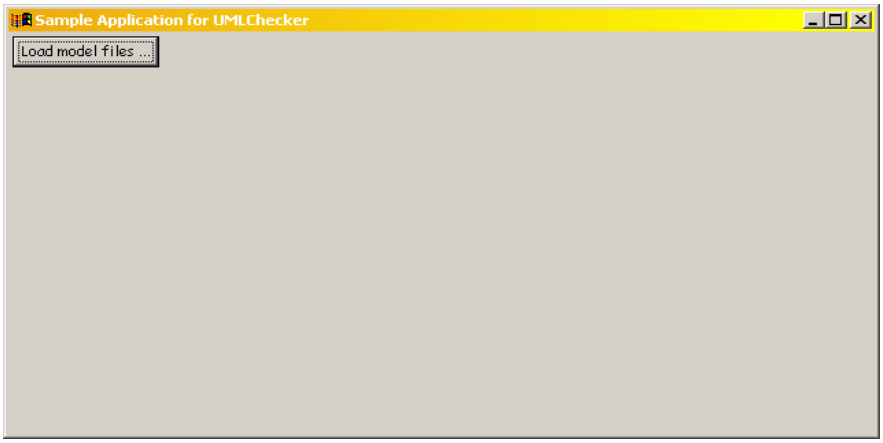


Figure 32 – Main screen for the sample application

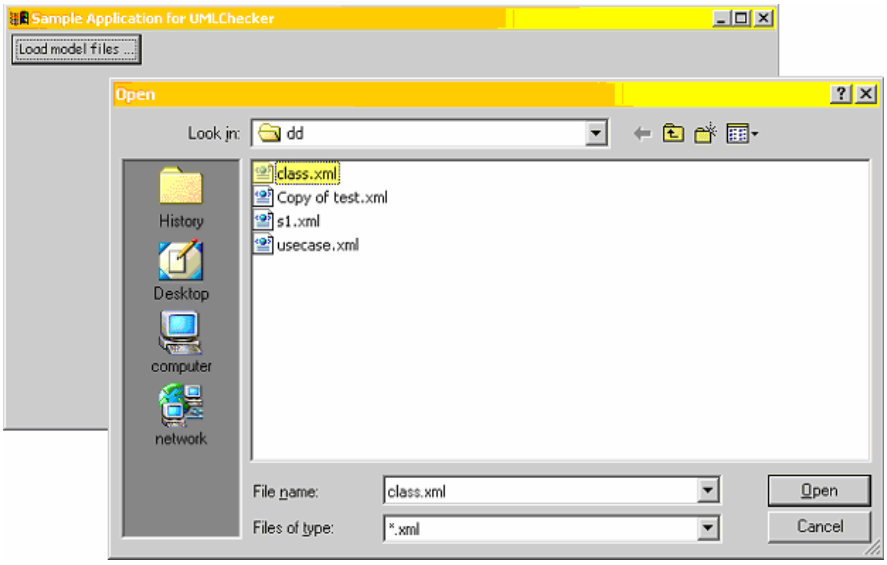


Figure 33 - Choosing the model files to be checked

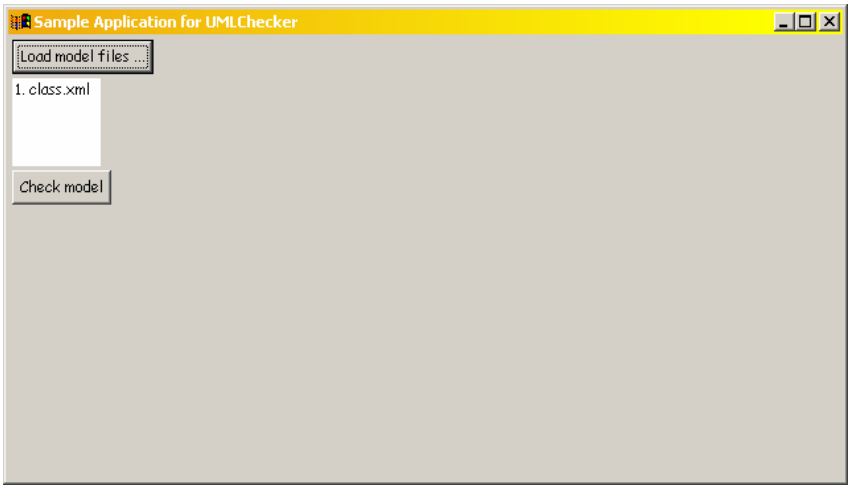


Figure 34 - Checking the model

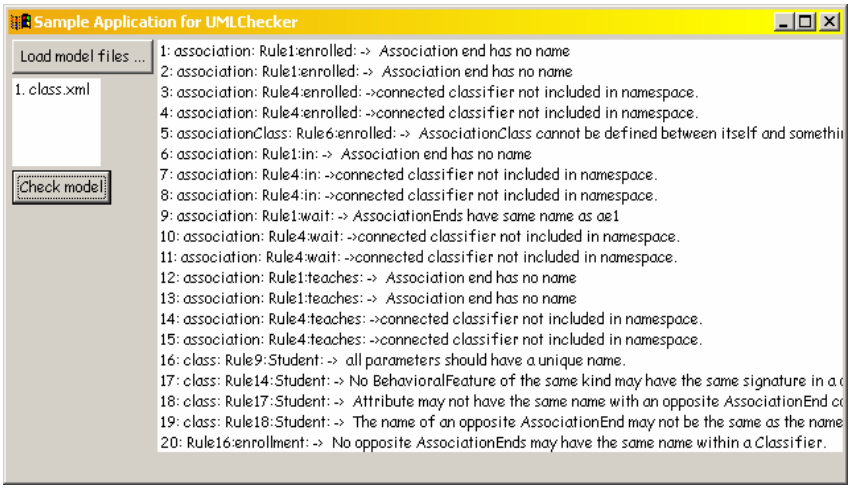


Figure 35 - The results listed as the result of the check