Chapter 3

Checking consistency of specification\(^1\)

Specification based testing is an approach that uses specification, represented as a model, to generate test cases. The quality of test cases thus generated is highly dependent on the quality of specification\([\text{Bei02}]\). The Unified Modeling Language (UML) is a general-purpose visual modelling language that is used to specify, visualize, construct and document the artifacts of a software system\([\text{JRB01}]\). The primary issue related to quality is consistency of specification captured using the model. Need for consistency checking between UML diagrams are discussed. Rules are defined for checking intra-model and inter-model consistency between UML diagrams are presented and a transformational approach to consistency checking is proposed alongwith a prototype tool.

\(^{1}\)A part of the work stated here is reported in
3.1 Introduction

The objective of software development being delivery of quality software to the customer, the need to capture requirements without errors is of primary importance. Inconsistency management has been defined in Finkelstein et al.[FST96] as 'the process by which inconsistencies between software models are handled so as to support the goals of the stakeholders concerned'. A number of techniques and tools have been developed for capturing requirements to suit the needs of different domains and applications.

The area of software specification has two classes of techniques: formal techniques (e.g. Z which emphasizes formality at the cost of ease of use and understandability) and rigour based techniques (e.g. UML, which emphasizes ease of use and understandability, at the cost of formality). As a rigour based technique lacks mathematical foundation, it is free to be interpreted differently by different people, leading to inconsistency. The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of a software system and is used at different levels and stages of the software development life cycle. UML models presented by different diagrams view a system from different perspectives or from different abstraction levels[SKS]. Thus, the various UML models of the same system are not independent specifications but strongly overlapping i.e. they depend on each other in many ways. Therefore, a change in one model affects another. The usage of several diagrams within the UML to capture specification leads to inconsistency that may arise due to mismatch between different diagrams. This has given rise to the need for checking consistency of requirements captured using different diagrams.

Consistency management is a process comprising activities like specification and management of consistency handling policy, consistency checking, detection of overlaps, diagnosis and consistency handling and tracking [ZK01]. A model
is consistent when it conforms to the semantics of all the domains involved in
the development process. e.g. application domain, modelling domain, language
domain[SC02]. Inconsistency arises when some model expression violates a prin-
ciple like same name for attribute and method. The Unified Modeling Language
(UML) defines well formedness rules as constraints on the modeling primitives
to prevent errors in modelling. However, they do not suffice in detecting incon-
sistencies, especially those across diagrams. Therefore, additional constraints are
required to ensure consistency and completeness to help detect and prevent errors
during specification of a system.

There are different dimensions to consistency checking[Sha06]: intra-model
Vs inter-model, structural/static Vs behavioural/dynamic, horizontal Vs vertical.
Intra-model consistency is concerned with the issue of consistency between dia-
agrams of the same type e.g. consistency between use case diagrams. Inter-model
consistency looks at the way each model is designed with reference to the other
models so as to be meaningful. Structural/static consistency concerns consistency
among the structural elements used in representing requirements. Behavioural/-
dynamic consistency concerns checking consistency during run time. Horizontal
consistency checking involves checking consistency between diagrams at the same
level of abstraction whereas vertical consistency involves checking consistency be-
tween diagrams at different levels.

Example:
An example for inconsistency in UML diagrams is shown in Figure 3.1 and Fig-
ure 3.2. Figure 3.1 shows a sequence diagram for the scenario, Customer places a
order. Also, the classes involved in the scenario are shown using the class diagram
in Figure 3.2.

A customer places an order. The order is checked with the items in the in-
ventory and a notification is given. The customer then confirms the order after
which the inventory is updated. A comparison of elements between Figure 3.1 and Figure 3.2 show that the sequence diagram does not match the operations stated in the class diagram. Rule 6 states that a class in a class diagram must have an operation that corresponds to a message in a sequence diagram. In the example, the message, confirm_order() is not specified in the class diagram. This is a case of inconsistency between the sequence diagram and class diagram. There is a need to add the method confirm_order() in the class diagram.

UML defines constraints on models as well formedness rules. These well formedness rules are applicable at the intra-model level and not at the inter-model level. Current work in literature handles inconsistency between UML diagrams based on the interaction specified between models in the UML superstructure specification[uml07]. As navigation between metamodels of all diagrams is not present, it is not possible to check for consistency of specification in a wholesome manner (e.g. no relation between use case and sequence diagram). In this work, interactions between the metamodels is introduced by annotating model elements. A set of rules are written based on the introduced interactions to check for consistency of specification. Also, a prototype tool is built to validate the approach. The approach shows the possibility of increasing consistency among UML diagrams sig-
nificantly thereby ensuring completeness and consistency of specifications leading to customer satisfaction.

In this chapter, rules to manage inconsistencies that can occur when capturing specifications using the UML are presented. Section 3.2 elucidates different consistency dimensions in the context of UML. Section 3.3 explains how the Object Constraint Language (OCL) aids in expressing constraints on UML modelling elements. A survey of work in the area of consistency management with focus on UML models is presented in Section 3.4. This work looks at two aspects of consistency management, namely, intra-model and inter-model consistency checking (Section 3.5).

Focus is on checking structural consistency, between UML models using a transformational approach discussed in Section 3.6. In this work, consistency checking activity is approached as a two pronged approach: first, rules for inconsistency checking are defined using OCL terminology; second, repository is created and rules are implemented as SQL queries, triggers and views discussed in Section 3.8. The architecture of the prototype tool and its usage is also discussed. Finally, Section 6.6 summarizes the contribution of this chapter and introduces issues to be tackled in the next chapter.

### 3.2 UML and Consistency Checking

The Unified Modeling Language (UML) is used as a standard for modeling object oriented software. UML helps in modeling different aspects of a system through the use of various diagrams, like use case diagrams, activity diagrams, sequence diagrams, class diagrams and component diagrams. Each aspect of a system is represented using a particular type of UML diagram and a set of such diagrams is termed as a model.
UML diagrams represent two different views of a system model, namely, static (structural) view and dynamic (behavioural) view [BRJ05]. Static view emphasizes the structure of the system while the dynamic view tries to project the runtime interactions of the system. Static diagrams include class diagram, object diagram, component diagram, deployment diagram, package diagram, composite structure diagram and profile diagram. Dynamic diagrams include use case diagram, activity diagram, sequence diagram, statechart diagram, collaboration diagram, interaction overview diagram and timing diagram.

The advantage of using UML is the fact that different diagrams can be used to model varying aspects of a system. i.e. the diagram that best represents an aspect of the system to be captured can be used. This same aspect leads to difficulties too, in the form of maintaining completeness and consistency within and between UML diagrams. This problem is further magnified as software systems are built today in a manner that requires collaborative and collective effort of people physically distributed across many locations. The phases of requirements gathering, analysis and design involves more than one UML model. Any error in these initial phases of software development will be carried over through the rest of the software development phases and will require major effort to be corrected. The following are the factors that have been found to lead to the problem of inconsistency:

- Omission
- Lack of Standardization
- Multiplicity of Stakeholders
- Addition of new features during system evolution
- Interdependency of diagrams
- Overlapping of model elements
Erroneous models have a huge impact on the development process in terms of added cost, time and effort. Therefore, the need for consistency management arises.

In this work, a subset of the UML models that are widely used has been selected with the objective of studying the points of inconsistency and proposing ways to overcome them. The models chosen are, both static and dynamic, namely, use case, activity, sequence, class and state diagrams. Being the widely used set of diagrams, the objective was to study the points of inconsistency and ways to overcome them.

The problem can be specifically broken down into two issues:

1. How do we ensure that the model captures the specification of a system?

2. How do we check that the models capturing specification are consistent, both, within and between themselves?

The first question requires that each use case diagram capturing requirements be elaborated by one or more diagrams that capture the dynamic view. For example, each use case in a use case diagram must be elaborated by one or more activity diagrams. In case of the second question, the objective is to ensure intra-model and inter-model consistency. An example for intra-model consistency in the case of use case diagrams can be that two uses cases with the same name must refer to the same requirement. An example for inter-model consistency is ‘a function belonging to a class in an activity diagram must have a corresponding method in a class in the class diagram’.

To check whether a model is consistent (intra-model consistency), the following questions need to be answered:

1. Is a diagram well defined?
2. Is a diagram consistent with other diagrams in the model?

To check whether models are consistent among themselves (inter-model consistency), questions that need to be answered are:

1. Is the class diagram consistent with the sequence diagram?
2. Is the class diagram consistent with the activity diagram?
3. Is the state diagram consistent with the class diagram?

UML being a semi-formal modelling language, it is not possible to define all aspects of a specification. To aid specifying constraints on modelling elements, the Object Constraint Language (OCL) is used which is the focus of the next section.

### 3.3 UML and the Object Constraint Language

The Object Constraint Language (OCL)[ocl06], is a formal language used to describe expressions on UML models. UML diagrams are not expressive enough to provide all aspects of a specification. Hence, the need to describe additional constraints about objects in a model. OCL is used as a specification language along with UML to augment the expressiveness of UML. Constraints are expressed as an OCL rule of the form:

```
context <model element>
invariant <constraint name> :
OCL expression
```

OCL can be used as a query language, to specify invariants, to describe pre and post conditions on operations and to describe guards. Consider the example given in Figure 3.3. The class 'Person' with attributes and method is shown. Suppose, it is required that an employee must be above 18 to be employed by the company,
then, this constraint on the age of the person cannot be represented through the class represented in the class diagram. Therefore, there is need for a language to describe additional constraints about the objects of the class 'Person'. The OCL invariant to specify the constraint is given below:

```
context Person
invariant checkAge:
  self.age > 18
```

OCL also provides constraint and object query expressions on any Meta Object Facility (MOF) model or meta-model which otherwise cannot be expressed by the diagrammatic notation. Inter-model diagram navigation involves following links from one object to locate another object or a collection of objects. OCL can be used for the purpose of navigation between model elements. A constraint on a model can be defined as a restriction on one or more values of a model. In this work, OCL constraints are used to define consistency constraints on a model.

### 3.4 Related Work on Consistency Checking

A set of UML models is consistent when there are no conflicts between two arbitrary models in the set, which are presented by different UML diagrams[Shi06]. Conflict between models can be identified in many ways, e.g. method belonging to a class shown in the sequence diagram but not in the class diagram, same name for attribute and method of a class.
One of the important concerns of system modelling has been consistency. Different approaches on how to detect inconsistencies at the intra-model and inter-model level have been proposed. The broad classification is a) Direct Approach and b) Transformational Approach. Direct approach involves comparing design models directly [KK03, HHKT02, LLH01] whereas the transformational approach involves transforming design models from one to another or into some intermediate model, for comparison[TE00, Egy01, HS04, KC04]. Examples of intermediate models used in the transformational approach include graphical models, formal methods, petrinets and knowledge representation based techniques.

Direct Approach
Constraints defined on UML using OCL is used for consistency checking by [KK03]. Constraints written in OCL is subjected to syntactic and view dependent analysis using the Dresden OCL Parse Toolkit. The analysis is based on information about the metamodel (entities, attributes and links). Hnatkowska et al[HHKT02] define a consistency checking pattern consisting of the specification phase, algorithmization phase and checking phase. They define two types of constraints, general and specific on three layers, namely, system model layer, model layer and diagram layer. Li et al[LLH01] present a method to check static consistency between Use Case(UC) and Conceptual Model(CM) by defining semantics using pre and post conditions and state invariants.

Transformational Approach
In the transformational approach, conversion to graphical models is the basis for work done by [CPC+04, Rou03, TE00]. Formal approaches involves use of a formal notation like Z. Kim and Carrington[KC04] use Object Z to define integrity consistency constraints of UML model elements at the language level. Derrick et
al[Der02] develop a framework based on the Open Distributed Processing (ODP) standardization initiative. Consistency checking is done across viewpoints (class, state and sequence diagrams) using Object Z and LOTOS. Petrinets are used as the intermediate models for consistency checking by [HS04, SM00, BDM02, ZS02].

Egyed in [Egy01] proposes the VIEWINTEGRA approach where source diagrams are transformed into the diagram type of the target. This is done so that transformations of the source diagrams are conceptually close to target diagrams they need to be compared with. Then consistency comparison is done to compare the transformation of the source diagram with the target diagram. For example, if a sequence diagram is to be compared with a class diagram, then the sequence diagram would be transformed to an 'interpreted' class diagram which is then compared with the target class diagram or vice versa. The class, sequence, collaboration, object and statechart diagrams are taken into consideration.

Despite lot of work done in the area, there are still problems that need to be solved:

- UML superstructure specification[uml07] document defines rules for intra-model consistency while constraints for elements across models have not been specified.

- Even though using constructs of UML(meta-model) for checking consistency reduces the cost and time involved in the process, the relationship between the models is such that it does not allow for direct checking between all models. Similarly, though the OCL can be used to specify elements and relations, it is difficult to use the same in specifying the consistency rules between diagrams.

- The transformational approach converts the UML models into a secondary model and use an intermediate means like trees and graphs for verification.
The advantage provided by the transformational approach is in having the same intermediate model for all the diagrams (e.g., Petrinets). Also, once the rules for transforming a model are decided, it becomes easier by using automation techniques. Also, models can be forward and reverse engineered. The disadvantage of using the transformational approach is having to implement the transformation in addition to consistency checking resulting in larger overheads.

- Common CASE (Computer Aided Software Engineering) tools do not support consistency checking completely.

### 3.5 Relationship among UML Models

Based on interviews with UML practitioners, clients and a web survey, Dobing et al. and Nugroho et al.[DP06, DP08, NC08] observe that use case diagrams and class diagrams have highest usage levels followed by sequence, activity and state diagrams. The class, sequence, activity and state diagrams provide additional information of requirements not captured by use case diagrams. Based on the same, it was decided to concentrate effort on checking consistency among five diagrams, namely, use case, activity, sequence, class and state diagrams. Using the UML superstructure specification as well as work by Dobing et al and Nugroho et al, the relationship between the models was studied with consistency checking as the goal. The strength of inter-relationships between diagrams is shown in Table 3.1 based on [SKS, uml07].

The strength of the relationship between the five diagrams (Table 3.1) is classified as 'Strong', 'Medium' and 'Weak'. 'Strong' indicates that both diagrams contain the same model elements to a large extent. 'Medium (A)' indicates that annotation is required or additional information is required without which the
Table 3.1: Strength of inter-relationships among models

<table>
<thead>
<tr>
<th></th>
<th>Use Case</th>
<th>Activity</th>
<th>Sequence</th>
<th>State</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case</td>
<td>X</td>
<td>Medium (A)</td>
<td>Medium (A)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Activity</td>
<td>Medium (A)</td>
<td>X</td>
<td>-</td>
<td>Weak</td>
<td>Strong</td>
</tr>
<tr>
<td>Sequence</td>
<td>Medium (A)</td>
<td>-</td>
<td>X</td>
<td>Weak</td>
<td>Strong</td>
</tr>
<tr>
<td>State</td>
<td>Weak</td>
<td>Weak</td>
<td>Weak</td>
<td>X</td>
<td>Strong</td>
</tr>
<tr>
<td>Class</td>
<td>Medium (A)</td>
<td>Weak</td>
<td>Weak</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

relation will be weak or non existent. For example, lets take the relationship between use case and class diagram. A class diagram must be annotated with the use case it belongs to, for otherwise, the relationship does not exist. A 'weak' indicator means that the diagrams have very little common information. Again, the relationships need not necessarily be bidirectional. For example, in Table 3.1, the relationship between State - Class is defined as strong, whereas, the relationship between Class - State is not existent. Based on Table 3.1, a diagram showing relationship among the models was evolved which is depicted in Figure 3.4.

The relationship between model elements shows that the strength of inter-relationship between diagrams varies. Checking consistency using a direct approach [CPC+04] is possible in case of intra-model consistency checking as well as between static and dynamic models that share elements in the metamodel. Hence, using a direct approach is not best suited as it is not possible to check for consistency between all model elements. Transformational approach involves transforming UML models to either a) graphical model like Petrinets and using the primitives of the graphical model to check for consistency or b) formal method and using a model checker to ensure consistency. Both ways, the objective is to use the capabilities of the model/formal method in checking consistency. In this work, the objective is to use models, static and dynamic, both strongly and weakly connected(e.g. use case and activity diagram) to capture requirements of a system by annotating the models. The advantage of a transformational approach is that
inconsistencies are easier to find in case of diagrams of the same type [SKS]. As UML consists of different types of diagrams, a transformational approach is best suited to check for consistency by transforming the diagrams into a common form.

3.6 Transformational Approach to Consistency Checking

Transformational approach to checking consistency between models can be done in two ways as shown in Figure 3.5. In the first case, one model is transformed to another and checked for consistency as shown in Figure 3.5(b). Figure 3.5(c) shows the second case which involves transforming both models to a common model before checking for consistency.

Consistency constraints are conditions imposed on the use of notations, vari-
(a) Is model A consistent with model B?

(b) Model transformation followed by consistency checking

(c) Model transformation followed by consistency checking

Figure 3.5: Transformational Approach to Consistency Checking

ables and names, types and symbols so that a set of well-formed diagrams can be regarded as a meaningful model[BBG05]. Conditions can be imposed on the structural as well as semantic aspects of the diagrams. To enable automated checking of a model in an effective, efficient and simple manner, consistency constraints are written as rules.

Demuth et al discuss the use of UML and its constraint language, OCL, for the design of integrity constraints for relational databases in [DH99]. They provide a methodology for translating invariants written in OCL to SQL. Rule based approach to consistency checking is used by Chiorean et al[CPC+04] and Zapata et al[ZGG07]. Chiorean et al use OCL to check for syntactic consistency among class diagrams. For this, they use Well Formed Rules(WFRs) written in OCL specified in the UML superstructure document. Zapata et al use rule based approach to look at the issue of inter-model inconsistency. They define rules to check consistency between UML use case and class diagrams designed using ArgoUML
and stored in XMI format. Mapping is done between use case and class diagrams by the condition that the noun in each use case become a class and the verb, a method of the class. Rules written using OCL are then translated to XQuery and XPath expressions used for validation. Alexander Egyed [Egy06] defines rules at the instance level for consistency checking between sequence, class and state diagrams. Consistency checking is done instantly as the model is developed thereby providing immediate feedback to the analyst. However, the evaluation time across size of the model remains a constant.

Different work, thus provide solutions to ease the consistency checking process. However, there are limitations. Chiorean et al propose using WFRs to check consistency. However, the WFRs are for class diagrams and hence consistency is ensured at the intra-model level only. Zapata et al present a novel approach to checking inter-model consistency between use case and class diagrams by mapping the noun in the use case to class name and verb in the use case to method name of the class. The approach requires that method name in a class be directly linked to a use case, introducing an additional constraint of having a use case for every method belonging to a class. Besides, the noun in the use case is the name of a class in the class diagram. The drawback of the technique is that for every class and method to be defined, there is need to define a use case in the use case diagram. The technique introduced by Zapata et al requires that use case diagrams be defined at the lowest level of granularity which is tedious and cumbersome. Egyed [Egy06] introduces an instant consistency checking approach providing the analyst with immediate feedback in case the model is inconsistent. In this approach, there is no way to predict which model elements are accessed for a consistency rule in advance. Also, there is need to study the need for instant consistency checking as the model is being developed versus applying consistency checking on the basis of need. The VIEWINTEGRA approach does not have the overhead of a third
party language in the process of transformation. However, in order to check for consistency between different diagrams, a minimum of ten transformation methods is required. In case pair-wise comparison between diagrams is to be done, then 55 transformation methods are required to check consistency between eleven models. The drawback of the method is the overhead involved in transformation and the accuracy of the transformation between diagrams.

In this work, the third approach (Figure 3.5(c)) shown in Figure 3.5 is adopted. Relational databases are used as the common model to check for inconsistency. Relational databases provide mechanisms for managing data. There are several advantages to storing UML models within a repository. First, relational databases guarantee a high level of interoperability. Also, a shared database ensures cooperation of designers and developers in the development of a system. Secondly, storage of designs in a repository helps in reuse. Third, reliability and scalability of data is ensured when a database is used. Navigation in RDBMS is provided through mechanisms like defining relations in the form of tables, view and joins. Other advantages provided by relational databases applicable for use in consistency checking is use of query language, SQL and support of views and scalability. UML is used to model small to large applications. Therefore, the amount of data can become voluminous making it difficult to handle. Integrity constraints can be used to enforce accuracy of data pertaining to an application. Also, rules written using SQL can be used to query the database.

A relational database is used to store data extracted from the model of an application stored in XMI format. Rules are defined based on the interactions between elements in the metamodel shown in Figure 3.4. The UML metamodel being clearly defined, the number of rules to be defined is static. Also, rules help in ascertaining the level of consistency to be achieved dependent on attributes of the project like size and complexity. The rules have been formulated by studying
the UML diagrams and looking at the constraints between them and are written using the Object Constraint Language (OCL). Rules can be translated from OCL to SQL queries, triggers and views.

For consistency to be checked between various models of UML, there is need to link models by annotating them. Also, certain guidelines have to be followed to ensure that requirements specified are elaborated.

### 3.7 Design Guidelines

A model M, can be said to be consistent with reference to constraint, C, if application of the constraint to the model evaluates to true; else, the model is inconsistent. Consistency constraints help define how specification using UML models should be constructed.

The advantage in applying such constraints is that anomalies in the model can be prevented and detected. As mentioned in Section 3.1, consistency checking of UML models can be implemented at two levels: intra-model and inter-model. This work focuses on intra-model and inter-model level and looks at the issue of structural consistency between the use case, activity, collaboration, class and state diagrams.

Design guidelines are rules that hold throughout the process of modeling. The relationships required to be established between models to enforce consistency is shown in Figure 3.4. A study of the metamodels depicted in Figure 3.6 shows that all relations cannot be enforced due to the lack of common elements among models. Hence, design guidelines were introduced to enforce these relations which are vital to the integrity of the database.

The following guidelines are required to be followed when building the models.
Figure 3.6: Modeling elements in the UML Metamodel
Each use case in a use case diagram must have a corresponding activity diagram.

Each use case in a use case diagram must have a corresponding sequence diagram.

A class must be annotated with the name of the use case it is part of.

Each class in a class diagram must have a state diagram.

Figure 3.7: Metamodel incorporating design guidelines

The metamodel developed based on the metamodels of the use case, activity, sequence, class and state diagrams incorporating the above guidelines is shown in Figure 3.7. Rules can be defined based on the interactions in the metamodel. Well formedness Rules (WFR) have been specified in the UML superstructure document to ensure consistency when developing UML diagrams. These rules are confined to diagrams of a particular type e.g. rules for use case diagrams. UML does not specify rules to be followed at the inter-model level. In this section, additional rules are defined at the intra-model level. Also, based on the UML metamodels and extending the idea of WFRs to inter-model level, a set of rules are formulated to check consistency between the use case, activity, sequence, class and state diagram.
3.7.1 Intra-model consistency checking rules

As mentioned earlier, intra-model consistency concerns checking for inconsistency that may exist within a model. For example, consider the use case model of a system shown in Figure 3.8, consisting of two use case diagrams specifying requirements of two actors 'Administrator' and 'Manager'. Part of the use case diagrams are shown in Figure 3.8(a) and Figure 3.8(b).

![Use case diagram A](image1)

![Use case diagram B](image2)

Figure 3.8: An example of intra-model inconsistency

The diagrams show inconsistency with reference to Rule 10: No two use cases in a use case diagram can have the same name, unless it refers to the same requirement and Rule 8: A use case cannot include use cases that directly or indirectly include it. This is a case of intra-model inconsistency showing conflicts within the use case model. Other rules used to check for intra-model consistency in case of use case and activity diagrams are listed in Appendix A.0.4.
3.7.2 Inter-model consistency checking rules

Inter-model consistency concerns checking for inconsistency that may exist between diagrams belonging to two or more models. For example, consider the class diagram and activity diagram shown in Figure 3.9, elaborating on the use case 'Place order' by customer. Part of the activity diagram is shown in Figure 3.9(a) and the classes in the class diagram are shown in Figure 3.9(b).

![Activity Diagram](a) Part of activity diagram for use case 'Place Order'

![Class Diagram](b) Part of class diagram

Figure 3.9: An example of inter-model inconsistency
The diagrams show inconsistency with reference to Rule 32: Each class and activity in an activity diagram must have a corresponding class and method in the class diagram. The method `chk_Inventory()` in the class diagram does not find a corresponding method in the class diagram. This is a case of inter-model inconsistency showing conflicts in requirements captured across UML models. A list of rules to check for inconsistency between models is listed in Appendix A.0.5.

3.8 Applying the Transformational Approach

The set of rules defined in Sections 3.7.1 and 3.7.2 are used for checking consistency between use case, activity, sequence, class and state diagrams based on the metamodel. Rules are classified as Warning, Moderate and Severe, dependent on the risk attached to a consistency rule. This classification of rules is done by the user. The inputs to the system are diagrams in XMI format. VP-UML, a Computer Aided Software Engineering (CASE) tool was used to capture requirements of a system. The UML diagrams are stored as XMI documents. Saxon parser\(^2\) is used to parse XMI documents. Metamodel in Figure 3.7 is translated to tables. OCL constraints are translated to SQL triggers, views and procedures which are used to enforce consistency based on work done by [DH99]. The rule based system, checks for consistency and generates a report listing the instances where rules are violated. Further sections explain the steps in applying the rule based approach in detail.

3.8.1 Translation of UML model to Database Schema

The UML repository is based on the meta-model and is implemented using an RDBMS, MySQL. The meta-model is mapped to a relational database schema.

Data integrity is enforced by converting OCL invariants to assertions, views and triggers. The steps involved in translating a UML meta-model represented as a class diagram to a database schema is given below [DH99]:

1. **Mapping association**: Each class in the association relationship is represented as a table. Associations are implemented using foreign key references. In case of one-to-one relationship, a foreign key is defined in either class. For one-to-many relationship, a foreign key is defined in the class on the many side. In case of many-to-many relationship, an additional table is defined and the combination of primary key from each of the tables is taken.

2. **Mapping inheritance**: In case of inheritance between classes, there are three ways for mapping inheritance to a relational database.
   
   a. One table per hierarchy
   
   b. One table per concrete class
   
   c. One table per class

   Any or a combination of the above can be used for mapping inheritance relationship between classes to a relational database.

3. **Mapping aggregation**: In case of aggregation between classes, the aggregated class’s attributes are put into the same table as the aggregating class.

### 3.8.2 Code Generation

Structured Query Language (SQL) is used for accessing and manipulating databases. Methods for automatic translation of OCL expressions to SQL is discussed by Demuth et al [DH99, DHL01] and Mohanty et al [LBMS05] i.e. methods to translate OCL to SQL queries, views and triggers. SQL is used to check if a database satisfies the constraints as well as disallow insertion of data that does not satisfy constraints in [DH99, DHL01] through the OCL2SQL tool. An example of the
Example:

(i) **Views:** A view is a virtual table that consists of columns from one or more tables and is a query stored as an object in the database. It derives its data from one or more tables. A view can be referenced like a table using queries and serves as a security mechanism by hiding data wherein users cannot view or access data in the underlying tables.

The class 'Classifier' from the UML meta-model has attributes name, id and a boolean flag which indicates whether the classifier isActor or not. This class is mapped to a single table with columns(attributes) name, id and isActor respectively. An OCL constraint on the class 'Actor' is 'All actors should have a name' i.e. not empty. The OCL expression is given below:

```ocl
classifier
inv actorNotNull: (isActor=true) implies (self.name → notEmpty())
```

The translation of this invariant into corresponding SQL view involves evaluating all tuples of the table Classifier where the type of classifier is actor and name is not empty.

```sql
create view actorNotNull as
(select * from Classifier
where not ((isActor=true) and (name is not null));
```

(ii) **Triggers:** A trigger is a statement that gets executed automatically when a database is modified. A trigger defined on a class executes whenever the specified event occurs and the corresponding condition is satisfied.
Consider the rule where every activity diagram must be associated with a use case in the use case diagram. Hence, when an activity diagram is added, there must be a corresponding use case that it is associated with. This rule when converted to trigger is represented below:

```sql
create trigger Insert_AD
before insert on activitydiag ad
fore each row
begin
if ad.fk_ass is null
    dbms_output.put_line(Each activity diagram must be associated with a use case)
end if;
end
```

### 3.8.3 Consistency Checking - The Process

The consistency checking process is shown, as a UML activity diagram, in Figure 3.10. The customer provides the requirements of a system. The requirements analyst, then extracts the specification and captures it using UML diagrams. The diagrams stored in XMI format are extracted by the consistency checking tool and stored in a model repository. The analyst then schedules execution of rule either by partial batch, full batch or user selected evaluation. Full batch involves execution of the complete set of rules whereas partial batch involves executing a class of rules like intra model rules pertaining to a particular diagram. User selected evaluation involves executing one or more rules based on user selection. Rules are executed and in case inconsistency is detected, details stored in the error file. The process continues until all rules are executed.
Figure 3.10: Consistency Checking Process
3.8.4 Prototype tool

A prototype tool is built to support the consistency checking process between UML diagrams. The architecture of the tool is shown in Figure 3.11.

The UML model that is input to the tool as XMI format is parsed and stored in the model repository. The user can decide to select a set of rules for execution or perform batch execution (all rules). Result of consistency checking, stored as an error file, is given to the user indicating the rule checked, the model element that has failed the rule and corresponding severity.

The process of checking the consistency of a UML model with regards to a set of rules is divided into two phases. The first phase is the extraction of data from XMI files, creation of the model repository and compilation of the rules in SQL creating the rule repository. Figure 3.12 depicts the selection of the application folder, for extraction into the model repository. This phase must complete successfully in order to allow the start of the second phase, namely, consistency evaluation.

The second phase involves the execution of consistency checking rules. As
mentioned earlier, there are two ways of performing consistency check: complete
and user selection as shown in Figure 3.13. Complete involves executing all rules
whereas user selection requires the user to select one or more rules for execution.

Once rules are executed, results of execution are reported with a list of rules
that failed with respect to the model elements. Detailed information on the re-
results(Figure 3.14) of consistency checking is provided showing the classification,
model element failed as well as the severity. The results show the inconsistent
parts of the specification to the analyst.

Figure 3.14: List of errors after application of rules

3.9 Summary

This chapter presented a transformational approach to consistency checking between UML diagrams, both at the intra and inter-model level. A study on the relation between UML and consistency checking as well as OCL is presented. Based on the study of relationship between UML models, a set of design guidelines were introduced to be followed when capturing specification using UML. A transformational approach was preferred due to the advantage of having a common intermediate form. A set of rules, both at the intra-model and inter-model level was evolved manually based on the relationship between UML models.

Existing work on transforming OCL to SQL [DH99, DHL01, LBMS05] is used as the basis for transformation. However, the rule transformation is based on the meta-model derived when capturing requirements following the design guidelines as there is no such constraint specified in the UML superstructure specification. Design guidelines involves annotating diagrams with extra details. For example,
annotating use case with corresponding activity diagram name.

In this work, a subset of diagrams and rules have been presented. The use case, activity, sequence, class and state diagrams have been considered and a set of rules both at intra-model and inter-model level have been presented. The objective was to demonstrate the ability of the approach in handling structural consistency between static and dynamic models. Additional rules can be written to make the process of consistency checking more comprehensive. Also, at present only five models have been considered on the premise that these five models are mostly used. However, additional models can be included by defining constraints making the approach adaptive to suit needs of the customer.

The approach can be integrated using current CASE tools as part of default consistency checking that tools should perform to help software designers in capturing requirements using UML. Also, the feedback provided by the method points out errors in design which aid designers in providing consistent models. Once it is ensured that the UML models are consistent, the next phase of testing can be done with confidence knowing that the models are consistent.