CHAPTER V

DETAILED-DESIGN OF LMS
5.1 INTRODUCTION

Despite of being the de facto model for software design, UML lacks in preciseness and formality of a formal language. In article 2.2.3, we have discussed that use of different UML diagrams (Use Case, Sequence, Activity, State Chart, Deployment etc.) at different levels of abstraction should be supported with a proof of consistency among them. This chapter presents a software design methodology for detailed design of an LMS, which combines use of a semi-formal modeling technique (UML) and a formal modeling language (VDM-SL), to support rigorous analysis and verification of the design with the requirement specification (Sengupta & Dasgupta, 2014). In conventional development process, software detailed design (hence onward mentioned as only design) is a seamless transformation of the software requirements, as described in the SRS document, into a form suitable for coding and implementation. A software design artifact demonstrates how to fulfill the requirements in the implementation of that software item. UML is the most accepted method for software design among the practitioners. UML diagrams are also useful in communication among different stakeholders but at the same time being semi-formal in nature they lack from formal syntax and preciseness in the notations. This makes it challenging to verify the design artifact against the requirements. Conversely, a formal specification language like VDM-SL has the advantage of preciseness and unambiguous modeling, but unable to provide ease of understanding like UML. The integrated use of a graphical method like UML and a formal method like VDM can be significantly beneficiary in the software design due to their complimentary features. However, the use of formal methods in the design entails that formal methods must have also been used for the requirement specifications. Since, the RE methodology described in chapter III uses combination of Use case diagrams and VDM-SL specification to model the requirements; we presume here that the requirement artifact satisfies the clause.
5.2 PROPOSED METHODOLOGY FOR SOFTWARE DESIGN

The proposed methodology of software design combines semi-formal and formal methods with an objective to take the advantages of both UML and VDM-SL by enhancing the relationship between the two methodologies, using a proposed framework [Figure 5.1]. This methodology starts with elaborating the Use case diagrams of RE phase into UML class, Activity, and Sequence Diagrams. The VDM-SL specification of RE phase is extended for a more detailed-description in design phase. In the design verification process, we first check for consistency among different UML diagrams of the design artifact and then verify them for traceability to the requirements (specified in the RE phase). Since comparison of NL written requirements for traceability is difficult to perform, instead of using the NL statements, we represent the requirements more formally using a lightweight formal method - Conceptual Graph (CG). We refer to the CGs for software requirements as Requirement Conceptual Graph (RCG) and CGs for design elements as Design Conceptual Graph (DCG). The traceability is checked by testing whether a DCG has its origin in a RCG. We propose an algorithm to accomplish this task. The entire methodology is illustrated in the context of design and development of an LMS.

5.3 PROPOSED FRAMEWORK

![Figure 5.1: The proposed framework for detail design](image-url)
A systematic approach in software design brings confidence to the developers about the correct implementation of the system. It is therefore important to use different SE methods in the design in an integrated and collaborative way. Figure 5.1 depicts the framework for the proposed methodology that reflects the interconnections between the used methods and artifacts. The framework provides insights on how the semi-formal and formal methods are integrated for the working of the proposed methodology. It promotes a systemic approach to guide the NL written requirements into a correct and implementable software design. The framework has two building blocks: a requirement model and a design model. The requirement model (in RE phase) represents the individual requirements with help of Use-Case diagram and corresponding VDM-SL specification. Since software requirements are generally specified in NL statements, we convert the requirement in NL statements into RCG and use them for the verification of the design model. This process is already discussed in details in chapter III. Here, our methodology takes the output of the requirement model that comes in the form of Use-case and VDM-SL specification, which is already verified against the requirements specified in the SRS. The focus is now rather on the design model and its verification.

The initial descriptions of the requirements at the requirement model by use-case diagrams are elaborated in the design phase with help of Class diagrams, Sequence diagrams, and Activity diagrams, each of them reflects different perspectives of the requirements. The core idea behind the transition from the requirement model to the design model is the elaboration of the requirement concepts (“what to be done”) towards solution of the problem (“how to be done”). Since such transition involves many assumptions on the domain, an early identification and decomposition of the structure of the domain objects to be used must be explicitly specified. We use a domain ontology to serve this purpose; it represents the basic hierarchical structure of the components in the form of object, process, and entity. This ontology works as a common agreement about the semantics of the components used by the formal and semi-formal methods in the RE and Design phase. Figure 5.2 shows a partial view of the hierarchical ontology for the domain of LMS.
CHAPTER V - DETAILED DESIGN OF LMS

The VDM-SL design specification is the extension of the VDM-SL requirement specification where the operations and data types of the latter are elaborated in accordance with the ontology. Figure 5.3 shows a partial view of elaboration of the VDM-SL specification in the design model.

5.3.1 Sample requirements for an LMS

We illustrate the proposed methodology with a case study of LMS development. We consider only some basic functional requirements to keep the illustration simple.

“Authors should create contents. Teachers should create courses from the available contents. The system agent should manage the content and the course in the repository.”
5.3.2 Requirement model

First, using the RE methodology discussed in chapter III, the requirements are modeled using use case diagram, VDM-SL specification and Conceptual Graph. A sample requirement model for the requirement- *Authors should create contents* is shown in Figure 5.4.

![Requirement specification diagram](image)

Figure 5.4: Requirement specification

5.3.3 Design model

In the design model, we construct different UML diagrams from the use case representation of the requirements. Although many researchers have contributed to the automatic derivation of different UML constructs (class, sequence, and activity diagrams) from use case requirements, unfortunately most of them are either domain specific or with limited capabilities (Yue et al., 2006 and Li, 1999 and Li, 2000). Therefore, our proposed approach relies on manual methods of translating use cases into class, and activity diagrams, which is more conventional for most of the analysts, designers, and developers. However, checking consistency between different UML diagrams is essential to ensure correctness of the design model. In addition, since UML diagrams and VDM-SL specification are disjoint in nature, consistency must also be checked between these two constructs of the design artifact.
5.4 CONSISTENCY RULES

In order to ensure traceability of requirements in different phases of software lifecycle, it is also important to ensure consistency between different UML diagrams used in those phases (Sengupta & Bhattacharya, 2006). Unfortunately, the ability of UML to depict multiple views of the system also comes along with an unavoidable risk of inconsistency. Usually, consistency checking ensures correctness by revealing problems and misuse of UML. If such problems are discovered early in the design process, it is easier and more cost effective to fix than discovered at a later stage. Since the proposed methodology involves different modeling styles like UML, VDM-SL and ontology, such consistency checking also guides the translation of the design into a programming codes, in a precise and unambiguous manner. Therefore, we propose some consistency rules to check syntactic consistency among the UML diagrams, VDM-SL specification and the ontology.

First, we formally define different UML constructs

- **Formalizing Use Case Diagram (UCD):**

  \[ \text{UCD}=\{\text{<Actor>,<UseCase>,<Rel>}\} \]

  \[ \text{Actor}=\{\text{actor}_i \mid 1 \leq i \leq n \} \]

  \[ \text{UseCase}=\{\text{usecase}_i \mid 1 \leq i \leq n \} \]

  \[ \text{Rel}=\{\text{Assoc, Include, Extend, Gen}\} \]

  \[ \text{Assoc}(\text{actor}_i, \text{usecase}_j) \]

  \[ \text{Include}(\text{usecase}_i, \text{usecase}_j) \mid i \neq j \]

  \[ \text{Gen}(\text{actor}_i, \text{actor}_j) \mid i \neq j \]

- **Formalizing Activity Diagram (ACD):**

  \[ \text{ACD}=\{\text{<ActionStates>, <Decision>, <Fork>, <Containment>}\} \]

  \[ \text{ActionStates}=\{\text{AS}_i \mid 1 \leq i \leq n \} \]
Decision = \{DS_i\} 1 \leq i \leq n
Fork = \{Join|Split\}
Containment = \{ AS_i , V \} 1 \leq i \leq n
V = \{var_i\} 1 \leq i \leq n

- **Formalizing Sequence Diagram (SD):**

SD = \{<Object>, <Event>, <Message>\}
Object = \{obj_i\} 1 \leq i \leq n
Event = \{e_i\} 1 \leq i \leq n
Message = \{m_i\} 1 \leq i \leq n

On the other side,
VDM-SL has the formal structure for system description:
System = \{<StateVariable>, <Function>, <Operation>, <Constraint>\}

Now, we define the following inter-consistency rules for the different UML constructs:

**Rule 1. Each action state (AS) in Activity Diagram (ACD) corresponds to a use-case (uc) in the Use-Case diagram (UCD).**

Justification:

An activity diagram exhibits the flow of activities within a system. It consists of activity states/action states, transitions and objects. Each activity/action state corresponds to a use-case in use case diagram.

**Formal Representation of Rule 1:**

\[ \forall a \in \text{ActionSates} \ ( \exists u \in \text{UseCases} \ (\text{correspondsTo} (a, u)) ) \]

**Rule 2. Each action state must access state variables mentioned as Constraint.**

Justification:
Considering each action state correspond to a use case and each use case is represented in the VDM-SL description with help of operations over the state variables, we can infer that an action state must access and alter the value of at least one state variable.

Formal Representation of Rule 2:

\[ \forall a \in \text{ActionStates} \; ( \exists s \in \text{StateVariables} \; (\text{access}(a, s)) ) \]

Rule 3. *Methods implementing an Action State must access its variables.*

Justification:

Considering method invoking as message passed between objects in Sequence Diagram (SD), and each method represented as an operation at the VDM-SL specification, we can infer that each method corresponding to an action state must access all the variables of that action state.

Formal Representation of Rule 3:

\[ \forall m \in \text{Message} \; ( \exists a \in \text{AS} , \exists s \in \text{StateVariables} \; (\text{correspondsTo} (a, m) \wedge \text{access}(a, s) \wedge \text{access}(m, s)) ) \]

Rule 4. *The sequence of invoking methods should match the order of their parent action states.*

Justification:

The actions defined in the ACD follow a particular sequence of flow. Since every action-state is related to one or more methods in SD, it necessary that these methods should conform to the sequence of the action states.

Formal Representation of Rule 4:

\[ \forall m_i, m_j \in \text{Message} \; ( \exists a_i, a_j \in \text{AS} \; (\text{correspondsTo} (a_i , m_i ) \wedge \text{correspondsTo} (a_j , m_j ) \wedge \text{precede}(m_i, m_j) = \text{precede} (e_i, e_j )) ) \]
Rule 5. All the variables used by design level VDM should have a correspondence to some variable of the requirement level VDM and the mapping should conform to the Ontology structure.

Justification:

The elaboration of the VDM-SL specification from requirement model to design model involves detailing of the state variables. In order to avoid any possible assumptions during this process, we support it by an ontology description. The ontology reveals the hierarchical structure of the components of the domain and thus can depict the origin and evolution of any state variable. Hence, it is essential that the mapping of a requirement level variable to a design-level variable should conform to the hierarchical decomposition of the ontology.

Formal Representation of Rule 5:

\[ \forall v_i \in \text{Req\_StateVariable}, v_j \in \text{Des\_StateVariable} \ (\exists v_a, v_b \in \text{ONT} (\text{correspondsTo}(v_i, v_a) \land \text{correspondsTo}(v_j, v_b) \land \text{parent}(v_i, v_j) = \text{parent}(v_a, v_b) ) ) \]

5.5 UML DESIGN OF THE SYSTEM

Next, Figure 5.5, 5.6, and 5.7 depicts different UML diagrams for our case study.

Use Case diagram:

![Use Case diagram](image-url)
CHAPTER V - DETAILED DESIGN OF LMS

Activity diagram:

![Activity diagram for the case study](image)

Figure 5.6: Activity diagram for the case study

Class diagram:

![Class diagram of the case study](image)

Figure 5.7: Class diagram of the case study
Sequence diagram:

![Sequence diagram of the case study](image)

**5.6 VDM-SL SPECIFICATION**

Next, we show a part of VDM-SL description from the case study.

```
System LMS
:
Operations
:
Upload (Cn:Content, Au:Author) &
rd ext Authorlist
wrt ext ContentList
Pre Au ∈ Authorlist
Post Contentlist = ContentList ∪ Cn
move (cn:Content) &
rd Content
wrt ext content_repository
Pre cn.storagepath= nil ∧ cn.bufferpath=nil
Post cn.storagepath = nil
store (cn:Content) &
rd Content
wrt ext content_repository
Pre cn.storagepath = nil ∧ cn.ContentID=nil
Post cn.ContentID ≠ nil
```

Figure 5.9: Part of the VDM-SL specification
5.7 DESIGN VERIFICATION

The design verification process has two sections. We first check the consistency between different design artifacts using our consistency rules. Then we check traceability of the design elements to the requirement model. Both these methods are illustrated with help of the running case study.

We identify different structural components from the design:

Use cases:

\[ UC = \{ \text{create\_content}, \text{arrange\_content}, \text{create\_course}, \text{arrange\_course} \} \]

Action states:

\[ AS = \{ \text{select\_content\_file}, \text{upload\_content\_file}, \text{content\_received}, \text{content\_repository}, \text{search\_content}, \text{build\_course}, \text{course\_repository} \} \]

State variables:

\[ SV = \{ \text{Client\_selectedfile}, \text{Server\_selectedfile}, \text{Content\_bufferlist}, \text{Contentlist}, \text{Course\_bufferlist}, \text{Course\_list} \} \]

Methods:

\[ M = \{ \text{Browse}, \text{Select}, \text{Upload}, \text{Post}, \text{Move}, \text{Content\_indexed}, \text{Course\_indexed}, \text{Display}, \text{Search}, \text{Set\_course\_info}, \text{Sequenced}, \text{Select} \} \]

In order to understand the inter-relationship between different design-constructs, Figure 5.10 illustrates the schematic view of how components of different UML diagrams are interconnected. Here, UC1 is a use case, which is mapped with e1, e2 event/action states of the ACD. Similarly, relationship between events and a set of methods \{m1, m2, m3, m4\} from CD, SD is shown. Finally, relationship between a set of variables \{v1, v2...v6\} from SD and VDM-SL description is shown.
5.7.1 Verifying consistency

We now verify the correctness of the design artifacts with the specified consistency rules. Since consistency checking of all the individual elements of the design artifacts would produce a very large number of records, we show one example per rule for the sake of simplicity. We use a mapping table [Table 5.1] to explore the interrelationship among the constituting elements of the different UML diagrams for the case study.

Verifying Rule 1: Each action state in the set \{e1, e2…e7\} has a corresponding origin in use-case set \{uc1, uc2…uc4\} [from Table 5.1].

Verifying Rule 2: Each action state in the ACD accesses at least one state variable [from Figure 5.6].

Verifying Rule 3: Method m5 accesses the variable \textit{contentlist}, which belongs to e4 while m5 corresponds to e4 [from Figure 5.10 and Table 5.1]

Verifying Rule 4: the method m3 precedes m5 in SD, which conforms to the sequence of their mapped activities e2 and e4 in ACD where e2 precedes e4 [from Figure 5.6 and Figure 5.8].
Verifying Rule 5: VDM-SL design level variables storagepath, bufferpath, localpath are decomposed parts of the variable content of the requirement model. This structure conforms to the ontology [from Figure 5.9 and Figure 5.2].

Table 5.1: Mapping between UML diagrams

<table>
<thead>
<tr>
<th>Usecase</th>
<th>Action states</th>
<th>Methods</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create_content</td>
<td>Select_content [e1]</td>
<td>Browse [m1]</td>
<td>Author</td>
</tr>
<tr>
<td>[UC1]</td>
<td></td>
<td>Select [m2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Upload_content [e2]</td>
<td>Upload [m3]</td>
<td></td>
</tr>
<tr>
<td>Arrange_content</td>
<td>Content_received [e3]</td>
<td>Post [m4]</td>
<td>Content</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indexed [m6]</td>
<td></td>
</tr>
<tr>
<td>Create_course</td>
<td>Search_content [e5]</td>
<td>Search [m7]</td>
<td>Teacher</td>
</tr>
<tr>
<td>[UC3]</td>
<td>Build_course [e6]</td>
<td>Courseinfo [m8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selectcontent [m9]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequenced [m10]</td>
<td></td>
</tr>
<tr>
<td>Arrange_course</td>
<td>Course_repository [e7]</td>
<td>Indexed [m11]</td>
<td>LMS agent</td>
</tr>
<tr>
<td>[UC4]</td>
<td></td>
<td>Displayed [m12]</td>
<td>Course</td>
</tr>
</tbody>
</table>
5.7.2 Verifying traceability

Verifying consistency rules ensures partial correctness of the design; additionally, the design artifact should be checked for traceability with the requirement specification.

The proposed verification technique is analogous to the already discussed verification technique in RE phase, which is based on a three dimensional verification matrix [Figure 5.11]. However, here three dimensions are different, namely Methods, Classes and State variables of the design artifact. We load this datacube with value at each cell either 0 or 1, representing absence or presence of the variable to the corresponding class and method. In other words, C[i] M[j] V[k] =1 indicates that the k\textsuperscript{th} variable is used by the j\textsuperscript{th} method inside the i\textsuperscript{th} class within the design artifact. We next propose an algorithm for automatic construction of the Design Conceptual Graph (DCG) from this datacube and then compare it with the RCG to ensure traceability.

5.7.2.1 Verification technique

\textit{DCG construction}

Input: method \( m \)

Output: \( DCG \)

Initialization: \( DCG=<\text{null}> \)

Assumption: \( m=\text{ no. of classes, } n=\text{ no. of variables present in the datacube.} \)

Begin:

\textit{For each uc such that ismappedTo(uc,m)=true}

\textit{For } \( j=1 \text{ to } m \)
For $k=1$ to $p$

If $C[j]V[k] == 1$ then // $M[i] = m$ for all the cases

$DCG = DCG \cup C[j]$

End if

Next

Next

$DCG = DCG \cup uc$

Next

End

Once we construct the DCG, we can compare it with the RCG, which is already available from the RE phase to check the traceability of the design element to the requirement.

5.7.2.2 \textbf{Results}

Let us now check whether invoking $m5$ is justified against UC2. In other words, we will check if the design article representing “Agent moves content” is traceable to its origin to the requirement “Agent arranges content”.

In the case study, $move()$ uses two variables $storagepath$ and $bufferpath$ so value corresponding to Agent class and Content class would be 1. At the same time, we can map the method $m5$ with the use case UC2, from Table 5.1.

So we can write $<\text{invoking } m5> \equiv <\text{Agent, content, arrangecontent}>$.

More formally $DCG = arrange\_content(\text{Agent, content})$

Now, analyzing the semantic similarity, we find that the resultant DCG is equivalent to the RCG of the said requirement [Figure 5.12].

Figure 5.12: Requirement conceptual graph for “Agent arranges Content”
5.8 CONCLUSION

This chapter proposes a software design methodology that uses semi-formal and formal methods in an integrated way. The success of a design model depends on the correct detailing of the requirements. We have continued this work from the output of the RE phase that produces a requirement specification using combination of use-case, VDM-SL and conceptual graph. The proposed design model extends the use-case of the requirement specification to the Activity, Sequence and Class diagrams. Since the semi-formal UML diagrams and the formal VDM-SL specification are disjoint in nature, we propose a verification technique to check consistency between the design artifacts. We use an ontology description of the domain to avoid conflicting assumptions between requirement specification and design artifacts. We also propose a data cube based verification method for traceability of the design component with the requirement specification. The proposed methodology endeavors to bridge the semantic gap between requirement specification and design artifacts. Traceability between the NL written requirements and the design components could be established in an automated way. Furthermore, consistency between different UML diagrams and the formal description could be also checked in an automated way. The use of VDM as the formal method in the specification and the design, and availability of UML based CASE tools support this automation facility.

This chapter deals with the methodology related to the detailed design for functional requirements. However, interface requirements are also equally important for web-based applications like LMS. Next, we discuss on how formalism can be applied to model interface requirements to match the functional requirements.