Abstract

Agile software development practices encourage the evolutionary paradigm of software construction. The result is a series of changes made to the code and designs starting from an initial version. However, rarely do designs keep pace with the changes in code - rather they are often outdated with the first few releases. The problem is only compounded by the various design representations that together comprise the design (state diagrams, class diagrams etc.). This adversely affects development since it is best to begin the process of evolution with the latest design representation that is consistent with the code since it provides a high level view of the system that is easy to understand. Such evolutionary practices therefore present three main challenges from a development environment perspective: (1) Consistency amongst different design representations (2) Traceability of design into code and vice versa (3) Versioning of designs along with code to maintain a consistent view of the system.

What makes the design analysis process complex? The design of the system is represented by several UML [1] diagrams. Each kind of UML diagram represents one view of the system. The set of design diagrams is to be correlated and relations between the entities of design diagrams are to be checked. In the process of checking for consistency we have to trace through different design diagrams according to relation between the entities of design diagrams and verify the consistency rules for each relation. Consistency amongst design diagrams does not imply consistency in code. We solve this problem by using a relational meta-model that ties together various design & code entities and an algorithm to check consistency over this relational meta-model [2]. Managing the various versions of design diagrams and code in a repository such the consistency relation between various versions of code and design are maintained is our focus for this paper. We are interested in creating a design evolution environment (DEE) that will seamlessly support design evolution and traceability. Such an environment will encourage incremental development in the correct manner - starting from design and keeping design consistent with code.

The rest of this paper is organized as follows. In Section 2 we motivate our work through a case study and give a formal description of the problem. Section 3 introduces a relational meta-model around which our notion of consistency is defined and defines the consistency rules. Section 4 de-
scribes our versioning algorithm and analysis of correctness and complexity. Section 5 introduces the layered model of SCM and our implementation of multi level versioning using this layered model on the eclipse platform. The results and current status is discussed in Section 6, a comparison with other efforts in this area is discussed in section 7 and we conclude in Section 8. Please follow the steps outlined below when submitting your

2. Problem definition

A software development project typically starts the development phase with a prototype design that we term version-0 from which the coding is done. While developing code we may need to alter the design resulting in a design version-1. In the meanwhile the code files will have progressed a few versions such as from version-1 to version-6. Let us assume that we now have a functional system. Now we do re-factoring and performance improvement by analyzing the working design i.e. version-1. We will get the re-factored design i.e. version-2 according to which we have to change code files. This will add one or two more versions to code files. We illustrate this notion with a specific case study concerning a banking ATM system.

Figure 1. Software Evolution with Versioning

Figure 1 shows the evolution of an ATM system designed in UML and built using Java (http://www.math-cs.gordon.edu/courses/cs211/ATMExample/). A design version represents all UML design diagrams that combine to give a working ATM system. Similarly the code version represents version of the working package that contains all 22 java files organized as transaction, physical, banking and ATM packages. Here we face the following problems during the course of it’s evolution: (1) Unable to identify which code and design versions are compatible/consistent with each other. (2) Ensuring that the packages remain compatible with each other while upgrading one or more packages of the working system to a new version. To do this we must trace the design version of the packages and check its compatibility with other packages.

These issues can be solved by a versioning scheme that maintains two levels of versions - one at design level and other at code level. Instead of checking for compatibility each time, verification can be provided by the version number itself. This kind of versioning will considerably speed up testing, deployment and other phases of the software engineering lifecycle.

While versioning is usually done on individual files in the repository, we need versioning at the design level as well to hide the complexity of the versioning problem of all the files that correspond to a given design version. Versioning should be done for each design entity in all model entities like class diagram, use case diagram, sequence diagram, collaboration diagram, classes, use cases and association, generalization relation between classes. This problem can be solved by a layered model of SCM with two levels of versioning one at design level and other at implementation level.

3. Layered model of Software Configuration management (SCM)

Figure 2. Layered Model of SCM

Configuration management has a wide range of concepts as mentioned in [3] some of which interact with each other. Concepts can be grouped under the categories of Development, Deployment, Model, Process and Storage. These concepts can be represented in four layers according to their functionality and interactions as shown in Figure 2. We term the four layers Application, Modeling, Link and Storage layers. As in any layered architecture, each layer communicates directly with only the layer immediately below it. Each entity in the model needs to be edited or used in application layer and needs to be stored in storage with different contexts, so it uses the Link layer function to store
them in repository. User works on the model layer entities through application layer and each type of model entity has a different access procedure in the link layer and storage mechanism in storage layer. Every item that is needed for the system is represented as a Model entity, so versioning of the model entity is very important to reduce complexity.

3.1 User Actions Through the Layered Model

The system model has various entities like UML diagrams that represent the design and source code that implements the design. The UML diagrams are stored in a repository in their storage format which is in turn managed by the context management in the link layer and associated with a system model entity and is given to the work space for editing. Similarly the source code is associated with a system model entity (class). It is to be given to the workspace for a developer and its storage should be managed within the context of the programming language.

The traditional change request, lifecycle model SCM tools directly work with repository and user. Instead we can have the concept of an application that selects the model entity and then uses the change request or lifecycle model interface in the Link layer to perform the operation. Here the user is isolated from the actual repository. He can directly select a system model entity and change its state from development to testing, testing to approved etc.

If this layered model is followed by the SCM product vendors, the products can seamlessly interact with each other and facilitate the development community by hiding storage and organizing complexity below model level. The next section describes the versioning in this layered model of SCM.

3.2 Versioning in the Layered Model of SCM

Versioning is done in the repository at the level of files, but we need versioning at the design level to hide the complexity of versions of all the files that correspond to a design entity. Versioning should be done for each design entity in the model like class diagram, use case diagram, sequence diagram, collaboration diagram, classes, use cases and relationships between classes. The versioned design entity must be stored in the repository using the context for that kind of entity.

As per the layered model proposed above, versioning is taken care of in the model layer. The model layer represents all entities for each project, subsystem representations (UML diagrams) at a module level and fine-grained entities like individual classes. Versioning of these entities individually is straightforward. But maintaining the relation between the UML diagram versions and classes’ versions is essential for a meaningful procedure that guarantees consistency.

To maintain a meaningful flow in the version of various design objects the relations between the design objects and their derived objects (code, binaries) should also be represented in the configuration management system. For maintaining these relationships the OO model of SCM [10] is appropriate. As described in [10], classes form items and class diagram forms aggregate items. When a new version of a class is created the new item instance corresponds to the new class version created and the derivation procedure from one version to other is used for the creation of new version identity or number.

When a new kind of entity is to be added to the model, it should first be defined and then its relations with various other items (both inward and outward dependencies) should be defined next along with the derivation procedure. For example, if we want to add test cases to each class, then first, format of the test must be defined, and the derivation procedure for deriving from its old version of the test must be defined. It should be linked to the class item. If we cannot define the derivation procedure we can abstract it by creating an empty file or copying the old file.

Complex design diagrams like package diagram, use case diagram, sequence diagram are associated with one or more classes. If versions of a class belonging to it are changed, a new version of that diagram will be created.

4. Relational meta-model & consistency checking

Our approach to consistency is centered on a relational meta-model in which we describe relations between entities belonging to the four perspectives of a software system as shown in Figure 3.

We have intentionally omitted activity diagrams and collaboration diagrams since the activity diagram only helps in creating sequence diagrams and its other functionalities are available in state diagram itself. Collaboration diagram represents the information present in sequence diagram as operating context of objects rather than time based messages of interactions. So these two diagrams are not mandatory in our meta-model, but user can use it for his reference. The relational model entities are grouped to various perspectives such as requirements perspective, development perspective, source perspective and deployment perspective.

The requirements perspective is related to the requirements classification and consists of use cases and use case sequence diagram representing the sequence of activities of an individual use case. The development perspective represents OO development components suitable for bottom up development. The source perspective here represents enti-
ties needed for implementation of the design using a pure OO language like Java. As is to be expected, these two perspectives are highly strongly related to each other. The deployment perspective gives details of components and their deployment information. The relation between component and class diagram provides deployment to code traceability. This, along with the relation between the use case sequence diagram and method sequence diagram provides end to end requirements to deployment traceability.

4.1. Consistency checking in relational meta-model

In order to check consistency amongst different entities, we introduce two classes of consistency rules - one set dealing with a single entity by itself and the other dealing with cross entity dependencies. The rules for consistency within a single entity such as a state diagram are termed well-formedness rules. There is a set of well-formedness rules for every UML diagram and design entity. The UML diagrams are described on XML Metadata Interchange (XMI) representation. These rules described on the XMI representation of class diagram are discussed in detail in next sub-section along with versioning scheme.

4.2. Modeling the Class Diagram

Class diagram models class structure and contents using design elements such as classes, attributes and methods. It also displays relationships such as inheritance, associations and aggregations.

Class diagram:

- **UML: Class Set (Defines)**
  - **UML: class (Defines)**
    - *UML: Association (Defines)*
    - *UML: Attributes (Defines)*
    - *UML: Methods (Defines)*
    - *UML: action sequence type: method (Refers)*
    - **UML: State-Machines (Refers)**

For an entity x the defined set and referred set is as follows:

- **Dx** = \{definedentities\} = \{class, attribute, method\}
- **Rx** = \{referredentities\} = \{class, methodsequencediagramsandstatemachines\}
- **ED** = \{externaldefinitionfromimports\} = \{jdk1.5library\}

The component is said to be internally consistent if its R set is a subset of its D set and ED set. The rule for an individual entity to be consistent therefore is:

- **R ⊆ D ∪ ED**

Similar to the Class Diagram XMI, all the other diagrams are also represented in XMI with defines and refers properties and few wellformedness constraints. For details, we refer the reader to [2].

4.3 Consistency between Entities

The consistency across entities is ensured by the following rule, if an entity X requires some other entity Y, the entity Y should be defined some where in the design, else the design is inconsistent due to X having an undefined reference Y. This error has to be resolved.

As defined earlier, for every entity in relational meta-model we can construct sets of defined entities and referred entities. For entities such as design diagrams the defined and referred entities are identified from the XMI representation of the diagram. The source supplies source entities for class, state and methods.

If there are 'N' design diagrams for a system, each design diagram will have a defined set and referred set. So there will be 'N'
defined sets \(D_1, D_2, \ldots, D_n\) and \(N\) referred sets \(R_1, R_2, \ldots, R_n\). We will also have external definitions used.

The system is considered to be consistent if and only if any referred item is defined somewhere in the system. In other words, the union of all the referred sets must be a proper subset of the union of all the defined sets. Mathematically:

\[
i \bigcup R_i \subseteq \{ \bigcup D_i \} \bigcup ED
\]

This condition is both a necessary and sufficient condition for consistency. Some examples: If we don’t have sate diagram for a class, then we can think of inserting a dummy state diagram with only one state ‘alive’ on which all methods of class except constructor is allowed and after every message it remains in the “alive” state except for destructor. This way the condition of a state diagram for each class is met in relational meta-model. Similarly if we don’t have sequence diagram for a method like a get method for some attribute, we can create a method sequence diagram with that object and no interactions.

5 Versioning Scheme for Design and Code Consistency

In our model, a simple entity is one that has no children in the XMI tree. Examples include class attributes which may be references to other classes. The design notion and implementation notion are equivalent for such entities.

But when it comes to a compound entity (one with children) such as a class diagram, a component or an entire system, implementation artifacts do not correspond one on one to design artifacts even though they are based on a design. So in our versioning scheme every entity in our repository is referred to by a combination of two version numbers - design version number and implementation version number. For each design version we have multiple implementation versions. For every entity we maintain the following information:

\[
\begin{align*}
NAME &= \text{name of the entity} \\
TYPE &= \text{type of entity} \\
ID &= \text{uniquely generated hash value} \\
D &= \{\text{defined entities}\} \\
R &= \{\text{referred entities}\} \\
ED &= \{\text{external definition from imports}\} \\
MV &= \text{Model version number (MV number)} \\
FV &= \text{File version number (FV number)} \\
CHILDREN &= \{X_{1MV,FV}, X_{2MV,FV}, \ldots, X_{nMV,FV}\}
\end{align*}
\]

The \(D\) and \(R\) set for a component \(C\) can be generated using their children as follows:

\[
D = \bigcup D_{X_{iMV,FV}}
\]

\[
R = \bigcup R_{X_{iMV,FV}}
\]

where \(X_i\) is a child of the component.

MV uniquely identifies the \(D\) and \(R\) sets of an entity. The model repository in the SCM provides new MV if there are some changes in \(D\) or \(R\) set of the entity. FV uniquely identifies the children set of the entity with child’s MV details. The repository provides a new FV if there are some changes in the children set.

5.1 Algorithm for consistent versioning

The algorithm to generate version numbers should feed off the consistency rules. We can apply the algorithm whenever the entity is first entered into the repository and subsequently whenever it is checked in.

Entity creation in repository:

Algorithm 1

1: Name = new-entity;
2: Type = type;
3: \(ID = generate\) – unique – \(ID()\);
4: \(D = \emptyset\); \{null set\}
5: \(R = \emptyset\);
6: \(CHILDREN = \emptyset\);
7: \(ED = \emptyset\);
8: \(MV = 0\);
9: \(FV = 0\);

This method creates a new entity in repository with the appropriate type and initializes the information with null value and versions numbers as zero. The versions of the entities are generated for each check-in using the following algorithm.

The check-in algorithm first checks for well-formedness rules for the entity. If the entity is well-formed then we proceed to the version generation algorithm. The version algorithm extracts the \(D\) set and \(R\) set and checks it against all the MVs of the entity in repository. If some of the MV match, then we check the children set and change the file version FV accordingly. If no model version matches create new model version and set FV to zero.

5.2 Correctness of the Versioning Algorithm

When two different versions of an entity in the repository have same MVs but different FVs, then their \(D\) and \(R\) sets are the same but some children (internal elements) are different in the two. This implies that the two have the same dependencies to other modules and provide the same functionality to other modules and so these two versions can be
interchanged in a system without affecting consistency of the system.

**CLAIM:** Two different versions of an entity having same MVs but different FVs can replace each other without affecting consistency of the system.

**Proof:** An entity E with model version number MV and file version number FV is represented by $E_{MV,FV}$. The children, ID and other properties of that element E is represented by $CHILDREN_E, ID_E$. Let us consider an entity S having ‘n’ children. Then

$$CHILDREN_{S_{MV,FV}} = \{X_{1_{MV,FV}}, X_{2_{MV,FV}} ,\cdots, X_{n_{MV,FV}} \}$$

$$D_{S_{MV,FV}} = i \cup D_{X_{i_{MV,FV}}}$$

$$R_{S_{MV,FV}} = i \cup R_{X_{i_{MV,FV}}}$$

$$R_{S_{MV,FV}} \subseteq D_{S_{MV,FV}} \cup E_{D_{S_{MV,FV}}}$$

Since entity is consistent.

Let us consider a contradiction that the entity S with MV' and FV' has ‘n’ children having the same MV but with a different FV. This version of the system has to be inconsistent with the entity S having version MV and FV. Then

$$CHILDREN_{S'_{MV',FV'}} = \{X_{1_{MV',FV'}}, X_{2_{MV',FV'}}, \cdots, X_{n_{MV',FV'}} \}$$

**D set calculation:**

$$D_{S'_{MV',FV'}} = i \cup D_{X_{i_{MV',FV'}}}$$

$$D_{S'_{MV',FV'}} = i \cup D_{X_{i_{MV',FV'}}}$$

since D set is same if MV is equal

$$D_{S_{MV},FV} = D_{S_{MV',FV}}$$

**R set calculation:**

$$R_{S'_{MV',FV'}} = i \cup R_{X_{i_{MV,FV'}}}$$

$$R_{S'_{MV',FV'}} = i \cup R_{X_{i_{MV,FV'}}}$$

since R set is same if MV is equal

$$R_{S_{MV,FV}} = R_{S_{MV',FV'}}$$

Since D set and R set are equal then their MV has to be equal but

$$MV = MV'$$

which is a contradiction.

When two versions of entity in repository have different MV their D and R sets are not same. This change has to come from some child (internal elements). So there is no consideration about the FV if the MV changes. This implies that the two versions have different dependencies on other modules but provide same functionalities to other modules. If the two versions are interchanged in a system, then its consistency will get adversely affected. This can also be proved similarly.

5.3. Implementation on the Eclipse Framework

Our implementation of the versioning scheme is on the Eclipse platform that works in the upper three layers of layered model described earlier. Eclipse has good separation of responsibilities implemented by different frameworks. Application layer functionalities such as editors and viewers are implemented on Eclipse Graphical Editing Framework (GEF) [9]. The modeling functionalities are implemented in Eclipse Modeling framework (EMF) [9].

We are using external CVS as the storage layer and employing the versioning capabilities of CVS for our minor/file version. EMF.edit frame works as the link layer that access data from CVS and links to EMF. Traceability links are also specified in this layer. Our relational meta-model will be implemented by EMF that serves as the model layer. Our major version representing design versioning will be implemented as part of the identifier of the entities in EMF and consistency check will be carried out for these identifiers. Using GEF on the top, we have implemented our editors that will serve as our application layer.
6 Current Status and Results

We applied our versioning scheme on an open source project called Mondrian from SourceForge. Mondrian is an OLAP server written in Java. It enables you to interactively analyze very large datasets stored in SQL databases without writing SQL. We considered 11 versions of it for analysis.

Figure 4 shows the summary of each check-in operation of Mondrian as gleaned from the CVS history of this project. The repository starts with the number of design changes equal to number of entities (initial check-in). Thereafter it tracks changes made at each check in operation. From version 0.5, 0.6 through 1.0 there are very few design changes but at version 1.1.0 there are many design changes done in refactoring for improvements in performance, and stability. After that, till version 2.0.1 all the changes are small enhancements. From the graph, we can see the influence of high-level design changes which does correspond roughly to the roadmap published at http://mondrian.sourceforge.net/roadmap.html.

The Table 1 shows a comparison of the version numbers generated by CVS for the original source files and the version numbers created in a normal SCM tool like CVS when BOTH design files and code files are checked in simultaneously. The last column depicts the version numbers generated based on design changes to create branches and tie model versions to design versions.

It can be clearly seen that having independently generated version numbers for the code and design files would need a version management system to correlate the two. Our algorithm accomplishes precisely this task and makes it simpler for human consumption by tagging correlated files with appropriate version numbers as shown in the last column of the table.

7. Related work

In [5],[4] G. Antoniol and his co-authors, present an approach to check the compliance of OO design with respect to source code and supports its evolution. Their process works on design artifacts expressed in the OMT notation and accepts C++ source code. It recovers an "as is" design from the code, compares recovered design with the actual design and helps the user to deal with inconsistencies. In [11] T.Mens and his co-authors propose to use description logic to specify and detect inconsistencies between UML models. In description logic we have to specify rules for inconsistencies separately. So this will require different rule definitions for each diagram. [13] Leverage’s the Object Constraint Language (OCL) to capture the evolution of UML models. Specific constraints describe how one model is different from another. Their technique formalizes the evolution of UML models, but it is unclear whether it could be extended to other modeling techniques. Their approach operates at a lower level of abstraction (i.e., classes, objects) and does not explicitly address the issue of versions, variants, and optionality. In their paper [8] the authors discuss the process of transforming UML representations to make it suitable for model checking. A model checker is an automatic tool that is able to compare the requirements and design descriptions of a given system. Karsten Diethers and Michael Huhn [12] present a tool for model checking UML designs. Their approach operates by converting UML to finite state automata, checking for errors in the automata and then reporting errors back in UML. The conversion rules are many and hard to change. This approach will not work with incomplete design or high level design. In [6] titled "Model Checking and Code Generation for UML State Machines and Collaborations" by Alexander Knapp and Stephan Merz, they discuss model checking and code generation from UML state and collaboration diagrams. In their paper [7] Lange and Chaudron propose a definition of completeness of a UML model and present

Table 1. The repositories update information of Javax package

<table>
<thead>
<tr>
<th>S.NO</th>
<th>CVS Repository version numbers of Javax package class diagram</th>
<th>CVS Repository version numbers of Javax package code files</th>
<th>Our version numbers of Javax package class diagram (model versions)</th>
<th>Our version numbers of Javax class package diagram (file versions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.1</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>1.2</td>
<td>1.0</td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
<td>1.3</td>
<td>1.0</td>
<td>1.3</td>
</tr>
<tr>
<td>4</td>
<td>1.1</td>
<td>1.4</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>5</td>
<td>1.1</td>
<td>1.5</td>
<td>1.0</td>
<td>2.2</td>
</tr>
<tr>
<td>6</td>
<td>1.1</td>
<td>1.7</td>
<td>1.0</td>
<td>2.3</td>
</tr>
<tr>
<td>7</td>
<td>1.1</td>
<td>1.8</td>
<td>1.0</td>
<td>2.4</td>
</tr>
<tr>
<td>8</td>
<td>1.1</td>
<td>1.9</td>
<td>1.0</td>
<td>2.5</td>
</tr>
<tr>
<td>9</td>
<td>1.1</td>
<td>1.10</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>10</td>
<td>1.1</td>
<td>1.10</td>
<td>1.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

The Table 1 shows a comparison of the version numbers generated by CVS for the original source files and the version numbers created in a normal SCM tool like CVS when BOTH design files and code files are checked in simultaneously. The last column depicts the version numbers generated based on design changes to create branches and tie model versions to design versions.

It can be clearly seen that having independently generated version numbers for the code and design files would need a version management system to correlate the two. Our algorithm accomplishes precisely this task and makes it simpler for human consumption by tagging correlated files with appropriate version numbers as shown in the last column of the table.

7. Related work

In [5],[4] G. Antoniol and his co-authors, present an approach to check the compliance of OO design with respect to source code and supports its evolution. Their process works on design artifacts expressed in the OMT notation and accepts C++ source code. It recovers an "as is" design from the code, compares recovered design with the actual design and helps the user to deal with inconsistencies. In [11] T.Mens and his co-authors propose to use description logic to specify and detect inconsistencies between UML models. In description logic we have to specify rules for inconsistencies separately. So this will require different rule definitions for each diagram. [13] Leverage’s the Object Constraint Language (OCL) to capture the evolution of UML models. Specific constraints describe how one model is different from another. Their technique formalizes the evolution of UML models, but it is unclear whether it could be extended to other modeling techniques. Their approach operates at a lower level of abstraction (i.e., classes, objects) and does not explicitly address the issue of versions, variants, and optionality. In their paper [8] the authors discuss the process of transforming UML representations to make it suitable for model checking. A model checker is an automatic tool that is able to compare the requirements and design descriptions of a given system. Karsten Diethers and Michael Huhn [12] present a tool for model checking UML designs. Their approach operates by converting UML to finite state automata, checking for errors in the automata and then reporting errors back in UML. The conversion rules are many and hard to change. This approach will not work with incomplete design or high level design. In [6] titled "Model Checking and Code Generation for UML State Machines and Collaborations" by Alexander Knapp and Stephan Merz, they discuss model checking and code generation from UML state and collaboration diagrams. In their paper [7] Lange and Chaudron propose a definition of completeness of a UML model and present
8. Conclusion and Future work

We have proposed a methodology to support seamless design evolution and traceability using a relational meta-model for both consistency checking and traceability linking of the entities in modeling layer of layered model of SCM. The proposed layered model of SCM represents a functionality classification. The model layer represents all items that need to exist for system development and relations between them may be different. The classes derived from the design diagram will be different for C++ and Java. In C++, each module with few classes can be represented in a single file but in Java each class must be represented in a different file. These kinds of issues must be handled by different derivation procedures. A single vendor cannot supply all kinds of derivation procedures, so vendors need to standardize the interfaces for adding new derivation procedures to SCM. In the proposed versioning scheme, all of the versioning complexity is handled by the model layer (design entity). The consistency between the versions of various entities must be maintained so that user can access, by simple queries the most recent tested version of entities belonging to the module. If this kind of easy method of selection is not provided users need to manually search versions of each class. So a good user interface for accessing versions of design objects is needed.

References