Odyssey-VCS: a Flexible Version Control System for UML Model Elements

Hamilton Oliveira, Leonardo Murta, Cláudia Werner

COPPE/UFRJ – Systems Engineering and Computer Science Program
Federal University of Rio de Janeiro – P.O. Box 68511
21945-970 Rio de Janeiro, Brazil
{hamilton, murta, werner}@cos.ufrj.br

Abstract. Many current version control systems use a simple data model that is barely sufficient to manipulate source-code. This simple data model is not sufficient to provide versioning capabilities for software modeling environments, which are strongly focused on analysis and architectural design artifacts. In this work, we introduce a flexible version control system for UML model elements. This version control system, named Odyssey-VCS, deals with the complex data model used by UML-based CASE tools. Moreover, it allows the configuration of both the unit of versioning and unit of comparison for each specific project, respecting the different needs of the diverse development scenarios.

1 Introduction

Computer Aided Software Engineering (CASE) tools can be classified into two main groups [24]: lower CASE tools and upper CASE tools. Lower CASE tools are mostly concerned about implementation and testing issues; whereas upper CASE tools deal with higher abstraction levels, entailing requirements, analysis, and design artifacts. Besides the necessity of Software Configuration Management (SCM) support for every kind of CASE tool [21], this support has been typically focused on lower CASE tools, providing a huge infrastructure over the last decades to leverage evolution of source-code artifacts.

However, due to the increasing software development complexity, SCM support is also needed by upper CASE tools. Model-driven development is emerging as a promising technique for complexity control. Model-driven approaches focus on the definition of high level models and apply subsequent transformations to obtain implementation artifacts. Nevertheless, the current SCM infrastructure does not properly support the evolution of model-based artifacts.

A first thought would be to adapt the existing SCM techniques, formerly applied to source-code, to this new context. However, current SCM infrastructures are not suited to the coarse grained artifacts used by upper CASE tools. For example, most current SCM systems are based on file system structures, while upper CASE tools are based on higher level structures. The mapping of these complex structures used by upper CASE tools to file structures is dangerous due to concept mismatch.

SCM 2005, September 5-6, 2005, Lisbon, Portugal.

Moreover, most SCM standards [10, 11] recommend the selective identification of Configuration Items (CI) that depend on individual characteristics of the software development projects. Nearly all state-of-the-practice SCM systems have a fixed identification of CI: the file. Due to this fact, every artifact that needs versioning information should be stored into an individual file. However, in some circumstances it is neither desirable nor possible to map every high level analysis and design artifact into an individual file.

Aiming to diminish the effects of these problems, we propose a novel approach to support UML-based upper CASE tools in evolving their artifacts. This approach, named Odyssey-VCS, consists of a Version Control System (VCS) for UML model elements that can be tailored to the specific needs of each software development project, as recommended by SCM standards. The main goal of Odyssey-VCS is to aid architects in the concurrent modeling of software systems using heterogeneous UML-based upper CASE tools.

Odyssey-VCS maintains a per-project behavior descriptor that informs how each UML model element type should be dealt. This behavior descriptor determines when evolution information is needed for a UML model element, considering this element as a CI. This evolution information comprises a unique version identification and auxiliary contextual information, such as who changed the element, when it was changed, and why it has been changed. Moreover, this behavior descriptor also indicates which elements are considered atomic for conflict detection purpose. Odyssey-VCS raises a conflict flag when two or more developers concurrently change an element that is considered atomic.

During the design of our approach, we provided our own solutions to overcome some challenges described in the SCM literature [5], such as: (1) data model that deals with complex CIs; (2) homogeneous versioning for different types of CIs; (3) distributed and heterogeneous workspaces; and (4) concurrent engineering with high level models. Moreover, a guiding philosophy of our work is to adopt standardized solutions and successful technologies used in other VCSs.

The rest of this paper is organized as follows. Section 2 details the problem to ground the ensuing discussion. Section 3 presents an overview of Odyssey-VCS, which is followed by a discussion of its internal mechanisms in Section 4. Section 5 shows a straightforward example that demonstrates how the challenges formerly discussed are addressed. Section 6 discusses related work, and we conclude the paper in Section 7 with an outlook at our future work.

2 Problem Statement

VCSs that use a data model based on file system structures usually consider files and directories as their CIs. This approach leads to three types of CIs: composite, textual and binary. The first type, composite CI, is represented by directories and can aggregate textual, binary or other composite CIs. The second type of CI, represented by text files, is the most important type because it can be internally manipulated by the VCS in order to execute basic version control operations, such as diff, patch and merge.

The third type, binary files, is only controlled, but not internally manipulated by the VCS since their internal structures are usually opaque to this tool.

For this reason, text files are seen by these VCSs as white box artifacts, and binary files are seen as black box artifacts. In fact, the use of automatic merge facilities, even for text files, is known to be an error prone activity [12]. They usually provide generic merging algorithms that do not take into account the specific syntactic structure of the text file. In spite of this limitation, almost all VCSs use text file mergers without differentiating text file contents. For example, the same merge algorithm used with a flat text file is also used with a Prolog file, a Java file, a LaTeX file or even an XML file.

Moreover, a common algorithm used by many tools to discover the type of a file is based on control characters. This algorithm searches for a zero ASCII byte inside the file. If this byte is found, the file is marked as a binary; otherwise, the file is marked as a text. This algorithm is used, for example, by MS Visual SourceSafe [23]. Beyond other problems related to the non existence of zero ASCII byte control code in some binary files, all different kinds of text files will be marked as a flat text file. Nevertheless, few VCSs such as Rational ClearCase provide special support for different file types (eg. mdl, doc, xml, etc.) and allow the usage of external mergers [26].

When a file is marked as a flat text file, the VCS considers a line as the unit of comparison[1] (UC). As shown in Fig. 1.a, the UC used in a flat text file is mapped into a paragraph. This is a well fitted mapping because a paragraph has enough cohesion and relative low coupling with other paragraphs, and a defined structure composed by a topic sentence and some supporting sentences. However, this is not the case in other situations as follows:

- A Prolog file usually has complex facts and rules written using more than one line. In this case, UC should be Prolog predicates.
- A Java file, as any other object-oriented language, has complex structures such as packages, classes, methods, and attributes, with methods and attributes as possible candidates to be UC.
- A LaTeX file does not use carriage return and linefeed as delimiters to a paragraph structure. A blank line is needed to identify a paragraph. As a consequence, UC should be the whole structure between blank lines.
- Finally, an XML file is a document composed of elements that may have other elements and attributes by themselves. In this context, a reasonable UC would be elements or attributes. An element or an attribute may be composed of many lines and changes in any of these different lines should be considered as changes in the same UC.

The use of line as UC is especially applicable in situations when a single line has high cohesion and low coupling with other lines. Lines should not be used as UC in files that employ data models with different abstraction structures. In the case of Java files, if a line metaphor is used, UC is mapped to a non existing Java structure (Fig.

[1] We define unit of comparison as an atomic element used for conflict computation. Conflicts occur when two or more developers concurrently work on any part of the same unit of comparison.

1.b). A Java method may be implemented by more than one line; whereas a line may comprise more than one Java command ended by a semicolon. In addition, a command is usually too excessively coupled to other commands to be considered a UC. Hence, the indicated structure should be java attributes and methods.

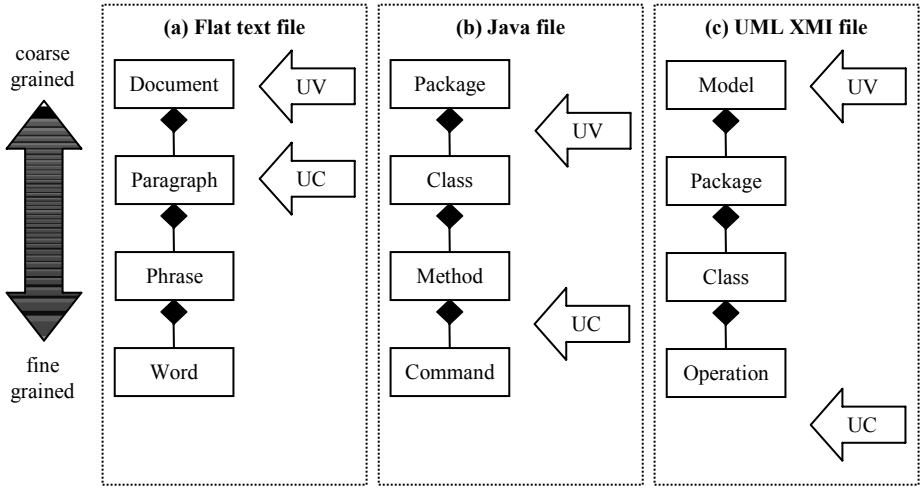

Fig. 1. Current UC applied to flat text files, Java files and UML XMI files.

As discussed before, some VCSs, such as Rational ClearCase, provide support for different abstraction structures through a pluggable merge facility. This strategy, however, is not enough to provide this kind of support because the UC concept only helps to set a boundary between file parts. In these systems, the whole file is considered to be a unit of versioning[2] (UV). For example, in a flat text file, UC is a paragraph and the UV is the whole document, as shown in Fig. 1.a.

This problem is more significant when the file type does not fit the abstraction structure of the file system data model. For example, Java may have one or more classes per file and these classes must be part of the same package, consequently, the UV map to a non existing java element, as shown in Fig. 1.b. In this scenario, a package may have more than one UV distributed through different files, and a class may share the same UV with other classes in the same file.

Java files are not the worst case. A convention may be established to recommend the construction of only one class per file. In this way, UV would be mapped to the class abstraction. Therefore, each class in the system would have their evolution controlled individually. On the other hand, object oriented data model can also be made persistent through files. One of the most common approaches to map an object-oriented data model to a file is to use markup languages. In this case, the whole object network is mapped into a singular file: an XML file. For instance, when Rational ClearCase marks a file as XML, UC is changed and the merge and diff tools act in a

[2] We define unit of versioning as an atomic element associated to versioning information. A new version of the element is created when any part of it is modified.

special way to provide more control over parallel development. However, UV remains being the whole XML file.

In our specific case, UML-based upper CASE tools use an object-oriented data model named Meta Object Facility (MOF) [18] and persist their models using XML Metadata Interchange (XMI) format [20]. In this scenario, the whole object network, composed of thousands of analysis and design artifacts, is persisted into a single XMI file. Fig. 1.c shows a fragment of an UML class diagram mapped to an XMI file. UC is smaller than an operation even if Rational ClearCase XML merge is used. This occurs because a UML operation is described via many XML elements, which represent visibility, return type, arguments, etc., and the parallel development over different XML sub-elements of the same UML operation would not conflict.

Another bad aspect related to UC in this scenario is regarding the proximity principle. UML uses an N-dimension structure composed by different diagrams to model software, and this N-dimension structure is mapped to a single XMI file, which is a one-dimension structure. This problem leverages the difficulty of implementing generic conflict detection algorithms. For example, two classes connected via inheritance association are considered "near" in a UML model. However, if more than one developer changes these classes concurrently, VCS would probably not be able to detect a conflict because the classes are apart from each other in the XMI file.

The problems are even worse when analyzing the UV. UV in this case is the whole UML model and it is not possible to distinguish versions of its parts, such as classes or use cases. The upper CASE tools will only be able to select versions of the whole model to work on. Therefore, the developers could not ask the VCS who changed a specific use case two days ago or what the existing versions of a given class are.

3 Odyssey-VCS Overview

Aiming to diminish the effects of the problems presented in Section 2, we introduce Odyssey-VCS, a flexible VCS for UML model elements. In this section we discuss the high-level features of Odyssey-VCS and show how these features help to overcome the challenges presented in Section 1.

3.1 Complex Data Model

Every upper CASE tool splits modeling elements into two categories: semantic and syntactic elements. Semantic elements symbolize conceptual elements and contain all information related to these elements, while syntactic elements are representations of semantic elements inside a diagram and their data are diagram dependent, like color, position, and size. In the context of the proposed approach, CIs are semantic elements of UML-based upper CASE tools. To be more precise, any subtype of *ModelElement* in the UML meta-model is a candidate to be CI in our approach. For this reason, Odyssey-VCS is able to version even the relationships among UML model elements, since relationships are also model elements. Examples of these model elements are: use cases, actors, classes, class associations, operations, attributes, components, etc.

Due to the complexity of this data model, it is not desirable to have a single versioning behavior for every CI type. Moreover, most SCM standards recommend the definition of a per-project SCM plan [10] and an important section of the SCM plan is the CI identification. The CI identification section of the SCM plan describes all CIs that should be placed under SCM. However, the current VCSs do not work with fine-grained CIs. As a consequence of this problem, all artifacts are put under version control, resulting in an extra overhead to the overall process, since some artifacts are not supposed to be controlled.

Our approach allows a fine-grained definition of CIs. For example, a class may be defined as an atomic CI for a given project; whereas operations and attributes may be controlled in another project. This flexibility provided by Odyssey-VCS allows more precise definition of CIs, adhering to the recommendations of existing SCM standards.

3.2 Homogeneous Versioning

As discussed before, the software development process deals with different kinds of artifacts, such as: use case descriptions, use case diagrams, class diagrams, sequence diagrams, code, test plans, test data, etc. All these artifacts must be controlled in a consistent way to provide snapshots of the system in different moments of development and maintenance. These snapshots when applied to a formal revision are called baselines or system configurations.

A baseline can be seen as a version of the whole system. Each model element has its own version, but the aggregation of these model elements has another version: the baseline version. A problem related to most VCSs is the way they manage baselines. Baselines are not seen as composite versions, but a structure with a completely different semantic. It is a trouble in situations where only one baseline level is not sufficient. Therefore, most current VCSs do not use version and baseline in a homogeneous way due to a lack of composite CIs.[3] A baseline is a special kind of CI, a composite one. It aggregates other CIs and its version is related to the versions of its parts. If a new version is created for some part of a composite CI, a new version should also be created for the whole CI.

In our approach both baselines and versions are dealt in the same way. If a CI is not composed of other CIs, the notion of version is the conventional one. However, if there is a composition relationship between CIs, the version of one CI depends on the version of the other. This situation is comparable to the usage of baselines. The main difference between the conventional approach and our approach is that in our case the baseline is not another type of element, but a CI, too. For example, a UML model has packages composed of classes and classes composed of attributes and operations. In this scenario, a package can be seen as a baseline of all its classes, and a class can be seen as a baseline of its attributes and operations. If one attribute is changed, a new version of the class that encapsulates this attribute is also created, because the class has been indirectly changed. Due to the new version of the class, the package that contains this class will also receive a new version.

[3] As an exception we can cite Subversion [3], which treats directories as composite CIs.

This homogeneous way of treating baselines and versions allows future queries over a specific package or class and complete reconstruction of any previous state, with the correct set of attributes and operations. For instance, it is possible to ask for the most recent version of the root CI of a system (i.e. UML model). Due to the uniform metaphor for versions and baseline, it is easy to transform this element into a part of a bigger system because the whole system is seen by Odyssey-VCS as an ordinary CI.

A possible drawback of this approach is the risk of an explosion of versions of composite CIs. However, file-based VCSs that use this approach deal with this problem by applying hard-links to sub-CIs that have not changed. We also use this technique to avoid waste of storage space.

3.3 Distributed and Heterogeneous Workspaces

The usage of a universal format is a key feature to support heterogeneous workspaces, maintained by different upper CASE tools. XMI is the most adopted format for both commercial and academic UML-based upper CASE tools. For this reason, Odyssey-VCS approach uses XMI as the protocol of communication between upper CASE tools and the VCS. These tools can connect to Odyssey-VCS through the Internet and query for a specific version of a CI, modify it and send back to Odyssey-VCS.

3.4 Concurrent Engineering

Odyssey-VCS is based on an optimistic strategy for concurrency control. The optimistic strategy lets developers change the same model in parallel, and merge the changes when the models are checked-in, as shown in Fig. 2. This strategy leverages parallel work, but increases the complexity of merge algorithms, as detailed later in Section 4.2.

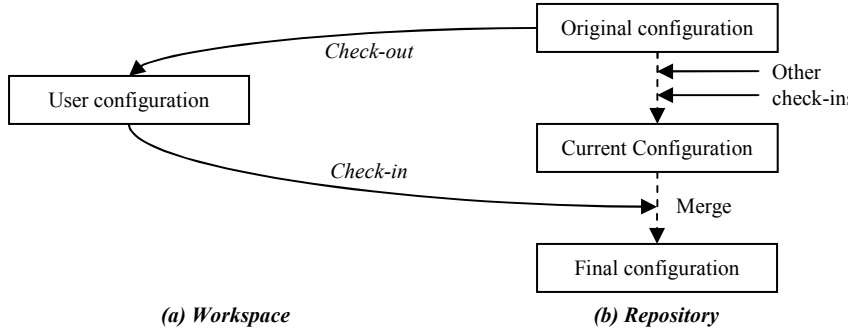

Fig. 2. Optimistic strategy for concurrency control

The original configuration shown in Fig. 2.b is the starting point of a new development cycle. The User configuration is created when a given developer checks-out the

original configuration and performs some changes. During this period of time, other developers also work on the original configuration, merging their work into the current configuration. Finally, the user configuration is also merged into the current configuration, creating the final configuration.

When a conflict is detected during the merge procedure, the whole check-in is rolled-back and the developer receives a message containing a detailed conflict description and the original, user and current configurations. After performing a manual merge, which can be supported by external tools, the developer resubmits the UML model to the repository.

4 Odyssey-VCS internals

The Odyssey-VCS architecture, which was implemented in Java from the scratch to avoid dependencies to existing file-based VCSs, is composed of three major layers: client, transport, and server. The most important element of the *client layer*, presented in Fig. 3.a, is the upper CASE tool. We are assuming that this tool uses UML as modeling notation and is able to externalize UML models using XMI. The integration between the upper CASE tool and the Odyssey-VCS infrastructure can be done via two alternative mechanisms: Odyssey-VCS plug-in and Odyssey-VCS client tool. Some upper CASE tools offer an extension infrastructure that allows the addition of external tools. In this case, Odyssey-VCS plug-in can be used, providing a seamless integration. For instance, we adopted this mechanism to integrate Odyssey-VCS with Odyssey environment [25]. However, some upper CASE tools have a poorly documented extension infrastructure, or do not even have it. In these situations, it is possible to use the Odyssey-VCS client tool. This tool opens an XMI file previously saved by the upper CASE tool and allows the execution of Odyssey-VCS commands. This mechanism was used to integrate Odyssey-VCS with Poseidon [1].

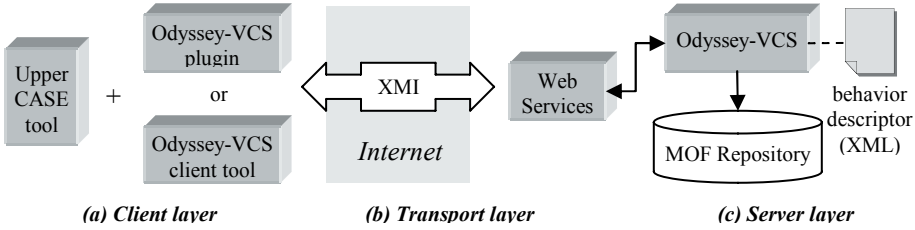

Fig. 3. Odyssey-VCS Overall Architecture

The *transport layer*, presented in Fig. 3.b, is responsible to allow distributed development of UML models over the Internet. The current implementation of this layer uses local calls or Web Services [2] as a transport protocol. However, it can be replaced by other protocols, such as WebDAV, RMI, sockets, etc. Moreover, before being sent over the Internet, the XMI files pass through a compression layer (zlib) to increase the overall throughput of the transport layer. This kind of approach, in place of using deltas, is also being adopted by other SCM tools [6].

Finally, the *server layer* processes the XMI files, applying different versioning behaviors depending on the project needs. The checked-in XMI file is transformed into an object network, which is merged with existing objects and stored into a MOF Repository named MDR [13] for further querying and retrieval. Both behavior configuration and merge algorithm are presented in the next sections.

4.1 Behavior Configuration

When a UML model element is checked-in, a specific action should be performed. However, as a flexible approach, Odyssey-VCS reads a behavior descriptor to decide what to do. This behavior descriptor informs which elements are UC and UV. For example, the scrap of a behavior descriptor shown in Fig. 4 is indicating that no versioning information should be stored for attributes and operations. On the other hand, classes are considered as CIs (UV=true), meaning that every version should be registered. Moreover, classes are also considered atomic elements (UC=true). Due to that, Odyssey-VCS will notify a conflict when two or more people edit the same class, even if they are working in different parts of the class.

```
<type name="org.omg.uml.foundation.core.UmlClass">
        <UC>true</UC>
        <UV>true</UV>
</type>
<type name="org.omg.uml.foundation.core.Attribute">
        <UC>true</UC>
        <UV>false</UV>
</type>
<type name="org.omg.uml.foundation.core.Operation">
        <UC>true</UC>
        <UV>false</UV>
</type>
```

Fig. 4. Scrap of a behavior descriptor XML file

It is important to notice that every software development project has its own behavior descriptor. This allows the customization of Odyssey-VCS to the specific needs of projects. For instance, if a project does not define class as UC, no conflict is detected when two people work on different parts of it. On the other hand, if operation is set as UV, every change on operations is registered together with versioning information.

Another important aspect is the interplay between UV and non-UV elements. Odyssey-VCS stores all physical versions of every element, but only stores logical versioning information of UV elements. For this reason, it is possible to correctly retrieve the context relationships of a UV element, even if it is related to non-UV elements.

4.2 Merge Algorithm

A built-in merge algorithm is also provided together with the flexible versioning infrastructure. This merge algorithm takes into account the configurations shown in Fig.

2. After analyzing the presence/absence of elements in these configurations, and the internal values of these elements after a check-in, we reached a complex scenario that is summarized in Table 1. The configurations and relations among them, used in Table 1, are defined as follows:

- O: Original configuration;
- U: User configuration;
- C: Current configuration;
- F: Final configuration;
- e_X: Element "e" in configuration "X"; and
- $e_X \equiv e_Y$: True if element "e" is identical in both configurations "X" and "Y".

Table 1. Odyssey-VCS merge algorithm

Case	$e \in O$	$e \in C$	$e \in U$	$e_O \equiv e_C$	$e_O \equiv e_U$	Action
1	T	T	T	T	T	Add e_C (or e_U) into F
2	T	T	T	T	F	Add e_U into F
3	T	T	T	F	T	Add e_C into F
4	T	T	T	F	F	Notify a conflict: "concurrent changes over the same element"
5	T	T	F	T	N/A	None (do not add "e" into F)
6	T	T	F	F	N/A	Notify a conflict: "concurrent removal and change over the same element"
7	T	F	T	N/A	T	None (do not add "e" into F)
8	T	F	T	N/A	F	Notify a conflict: "concurrent removal and change over the same element"
9	T	F	F	N/A	N/A	None (do not add "e" into F)
10	F	T	T	N/A	N/A	N/A
11	F	T	F	N/A	N/A	Add e_C into F
12	F	F	T	N/A	N/A	Add e_U into F
13	F	F	F	N/A	N/A	N/A

Table 1 shows, for every possible scenario, which action should be taken by Odyssey-VCS. For example, case 3 shows a scenario where a given element (use case, for example) exists in all configurations ($e \in O$, $e \in C$, and $e \in U$), was changed in the current configuration ($e_O \not\equiv e_C$), but was not touched in the user configuration ($\mathbf{e_O \equiv e_U}$). In this case, Odyssey-VCS promotes the element from the current configuration to the final configuration. On the other hand, case 6 shows a scenario where a given element (an operation, for example) was removed from the user configuration ($e \notin U$) but exists in all other configurations ($e \in O$, $e \in C$). However, the element was changed by other users ($e_O \not\equiv e_C$). As a result to this scenario, Odyssey-VCS notifies a conflict, arguing that the same element was removed and changed by different developers.

The results of the merge algorithm are consistent to the UML structure (guaranteed by MDR repository), but may be inconsistent with UML well-formedness rules. The current release of Odyssey-VCS does not apply well-formedness rules consistency check. However, this validation can be obtained by external tools.

5 Example

In this section we present an intentionally simple usage example that aims to illustrate how our approach provides flexibility and concurrent access during the evolution of UML models. The target system of this usage example is a hotel network control system presented in the literature.

Initially, suppose that Odyssey-VCS is configured as described in Table 2. It is important to notice that *Class*, a composite element that encloses *Attributes* and *Operations*, is configured as UC. In other words, this means that should two or more developers interact concomitantly over the same class, a conflict will happen. This conflict occurs even if they are working on different parts of the class. Moreover, *Actor* is not configured as UV. This means that no versioning information will be stored regarding this element.

Table 2. Odyssey-VCS configuration

Element	Unit of Versioning (UV)	Unit of Comparison (UC)
Model	True	False
Package	True	False
Class	True	**True**
Attribute	True	True
Operation	True	True
Use Case	True	True
Actor	**False**	True

After configuring the Odyssey-VCS system, an initial version of the model has been committed to the repository. This initial version, shown in Fig. 5, comprises six classes, all in the same package, four use cases, and one actor. This model is used as the basis for further development.

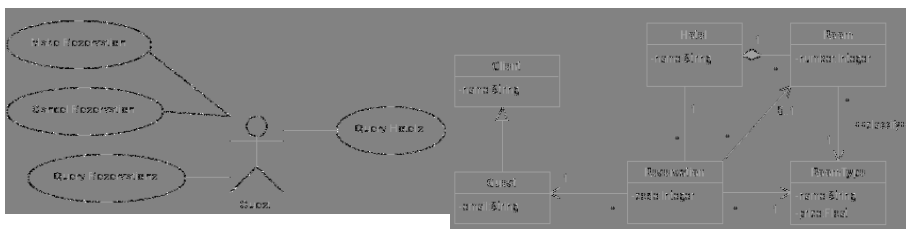

Fig. 5. Hotel network control system use case and class models

In the following, two developers, namely John and Mary, are using different upper CASE tools, respectively Poseidon and Odyssey, to concurrently change the initial version of the hotel network control system model. John wants to rename the *email* attribute of the *Guest* class to *telephone*, add the *gender* attribute into the *Client* class and add a new use case named *Modify Reservation*. On the other hand, Mary wants to change the type of the *price* attribute of the *RoomType* class from *Float* to *Currency* and include two new operations in the *Guest* class: *getEmail():String* and *setEmail(email:String):void*.

The changes performed by John, shown in Fig. 6.a, were committed first into the repository. No conflicts were detected and the commit was successfully merged into the current version of the repository. After John's commit, the current version of *Guest* class in the repository has no more the *email* attribute. However, the version of the *Guest* class in the Mary workspace is out-of-date, still containing the *email* attribute, as shown in Fig. 6.b.

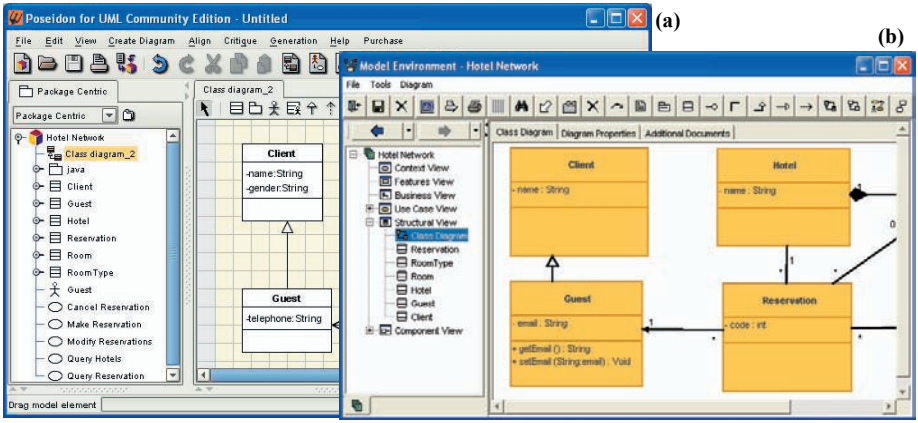

Fig. 6. Poseidon (a) and Odyssey (b) working over the same UML model.

When Mary tries to commit her version, all UML model elements, but *Guest* class, are correctly merged. Even the sub-elements of the *Guest* class were individually merged successfully. However, besides their syntactical correctness, they are semantically incompatible, because the operations *getEmail():String* and *setEmail(email:String):void* were created to manipulate the *email* attribute, which was renamed to *telephone* by John. Fortunately, *Class* type was defined as UC. Due to that, a conflict is raised, as shown in Fig. 7.

Odyssey-VCS provides all necessary information to allow Mary fixing the conflict. This information comprises, in addition to Mary's local version of the model, the original and the current versions of the repository. After manually fixing the conflict, Mary is finally able to commit her changes into the repository. Fig. 8.b shows the final state of the repository, which has the original version of the model (version 1), John's commit (version 2) and Mary's commit (version 3).

Fig. 8.a shows the third version of the model in Mary's workspace, after a new check-out. This third version contemplates the original intention of both John and Mary. It is worth to notice that every version that is relative to a type defined as UV in Table 2 has contextual information when stored in the repository. This contextual information, which encloses date, responsible, and additional comments, can be further used by other developers. However, *Actor* was not defined as UV. For this reason, its versions are neither shown in Fig. 8 nor computed by Odyssey-VCS.

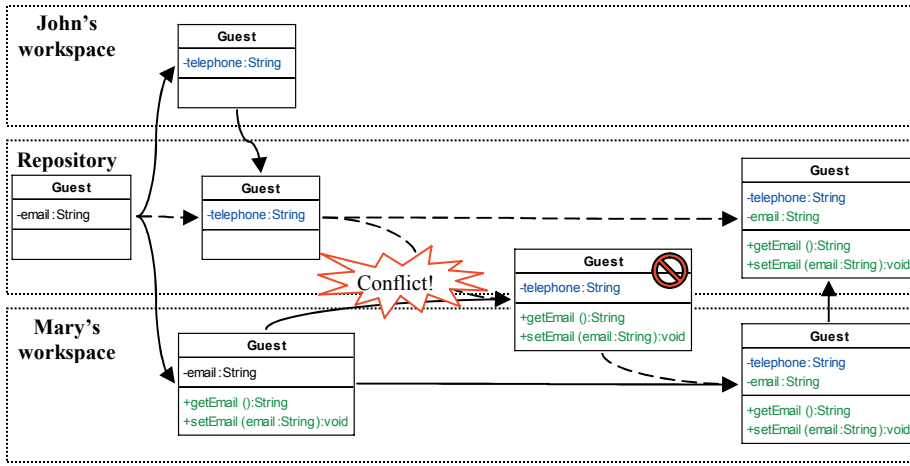

Fig. 7. Merge and conflict detection scenario

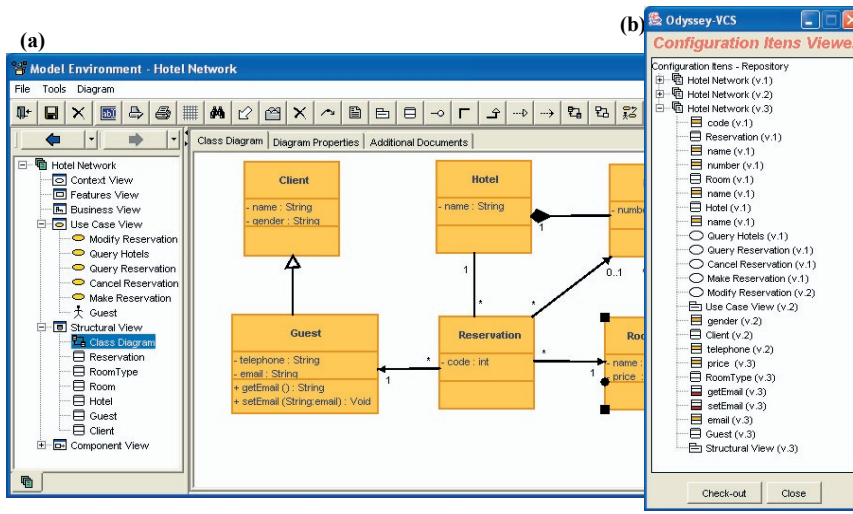

Fig. 8. Odyssey (a) performing check-out through Odyssey-VCS plugin (b)

6 Related Work

Most commercial and open-source VCS are based on file system data model [3, 7, 23, 26]. As previously discussed, these VCS have several limitations to manipulate artifacts with complex internal data model. However, these solutions are generic and mature, being suitable for versioning of text-based artifacts. We do not see these approaches as direct competitors of our approach. Our approach can be used to version

UML model elements while these approaches can be used to version source-code in the same software development project.

There are also other approaches that employ other data models, such as entity-relationship or even object oriented. However, these approaches work at source code level and are focused on a specific programming language. For instance, Goldstein et al. [8], Habermann et al. [9], and Render et al. [22] support versioning of Smalltalk, C and Pascal source code, respectively. Our approach can also be seen as complementary to these approaches since we are focused on UML model elements and these approaches are focused on source-code.

Some few approaches use non file system data model to version analysis and design artifacts. For instance, Ohst et al. [17] propose an approach for versioning analysis and design artifacts via syntax trees stored in XML files. Working at fine grained UML artifacts they can correctly manage structural changes on these artifacts. However, the usage of XML does not mean adherence to modeling standards. Their XML format does not follow XMI specification for UML, leading to incompatibilities with existing UML-based upper CASE tools. Moreover, Nguyen et al. [16] use a hyper-versioning system to apply version control over complex artifacts, including UML analysis and design artifacts. This work has a strong focus on versioning relationships among the elements. However, it is also based on a proprietary UML data model, reducing compatibility with existing upper CASE tools.

Finally, OMG is working on a specification for MOF versioning [19]. Besides the nonexistence of a final specification version up to now, it is possible to notice that Odyssey-VCS is pretty adherent to the specification philosophy. Similar to the specification, Odyssey-VCS has its own versioning meta-model. Moreover, it stores the versioned elements into separate per-version extents with associated history of changes. Probably, it will be straightforward to adhere to the final version of the specification in the future.

7 Conclusion

In this paper we presented an approach for version control of UML model elements. Our approach differs from the existing approaches in the following aspects. First, we provide support for flexibility during CI identification, allowing the configuration of UC and UV for UML model elements. Second, our approach is based on well adopted standards, raising the compatibility with existing upper CASE tools. Finally we have focused on current challenges of SCM to avoid reinventing the wheel regarding already solved problems. Besides these main aspects, our approach also provides a built-in merge algorithm, supporting concurrent development.

The existence of a fine-grained VCS for UML model elements can be seen as the basis for upcoming work. For instance, Dantas et al. [4] have proposed an approach for traceability link detection via data mining over UML model elements stored in the Odyssey-VCS repository. Moreover, another work is being performed to transform Odyssey-VCS into a change-oriented VCS [15]. All these tools are part of a broader

infrastructure named Odyssey-SCM [14], which aims to provide SCM functionalities for component-based development environments.

Our approach, however, is currently tightly coupled to the UML meta-model. An important future work is the generalization to the MOF meta-model layer, allowing versioning of any MOF compliant meta-model. Additionally, the current version of Odyssey-VCS does not use deltas to compose versions. On one hand, the negative impact of this decision is higher network traffic. On the other hand, we do not need to compute a version based on prior versions and deltas, which saves some CPU cycles. We also use zlib to help reducing the transport overhead due to the absence of deltas. While performance and scalability were not a major concern for this first prototype, we intend to work on these issues for the next releases. Currently, we are performing some benchmarks to measure the performance of Odyssey-VCS when the size of the repository increases. After that, we intend to run some case studies in real software development scenarios.

Moreover, another limitation is the use of a predefined merge algorithm. Future releases of Odyssey-VCS should allow the replacement of the built-in merge algorithm for project-specific merge algorithms. Another future work is the construction of a tool that allows visual merge of UML models. The current version of Odyssey-VCS only notifies the conflict, providing all information for the merge. However, the merge itself is not supported by Odyssey-VCS, requiring the user to do it directly in the XMI file or using existing UML-based upper CASE tools.

Acknowledgments

Our thanks to the members of the Software Reuse Group at COPPE/UFRJ, especially Cristine Dantas and Luiz Gustavo Lopes, who helped in many discussions regarding the architecture of the approach. Moreover, we would like to thank CNPq and CAPES for the financial support.

References

1. Boger, M., Sturm, T., Schildhauer, E., and Graham, E.: Poseidon for UML user guide. Gentleware AG (2000)
2. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D.: Web Services Architecture - W3C Working Group Note. World Wide Web Consortium (W3C). In: http://www.w3.org/TR/ws-arch, Accessed in: 25/Jul/2005
3. Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M.: Version Control with Subversion. O'Reilly (2004)
4. Dantas, C. R., Murta, L. G. P., and Werner, C. M. L.: Consistent Evolution of UML Models by Automatic Detection of Change Traces. International Workshop on Principles of Software Evolution (IWPSE), Lisbon, Portugal, September (2005)
5. Estublier, J.: Software Configuration Management: a Roadmap. International Conference on Software Engineering, The Future of Software Engineering, Limerick, Ireland, June (2000) 279-289

6. Estublier, J., Leblang, D., Clemm, G., Conradi, R., Tichy, W., van der Hoek, A., and Wiborg-Weber, D.: Impact of the research community on the field of software configuration management: summary of an impact project report. ACM SIGSOFT Software Engineering Notes, Vol. 27, no. 5, September (2002) 31-39
7. Fogel, K. and Bar, M.: Open Source Development with CVS. The Coriolis Group, Scottsdale, Arizona (2001)
8. Goldstein, I. P. and Bobrow, D. G.: A Layered Approach to Software Design. In: Barstow, D. R., Shrobe, H. E., and Sandewall, E. (eds.): Interactive Programming Environments. McGraw-Hill, New York, NY (1984) 387-413
9. Habermann, A. N. and Notkin, D.: Gandalf: Software Development Environments. Transactions on Software Engineering, Vol. 12, no. 12, December (1986) 1117-1127
10. IEEE: Std 1042 - IEEE Guide to Software Configuration Management. Institute of Electrical and Electronics Engineers (1987)
11. ISO: ISO 10007, Quality Management - Guidelines for Configuration Management. International Organization for Standardization (1995)
12. Leon, A.: A Guide to Software Configuration Management. Artech House Publishers, Norwood, MA (2000)
13. Matula, M.: NetBeans Metadata Repository. NetBeans Community. In: http://mdr.netbeans.org, Accessed in: 25/Jul/2005
14. Murta, L. G. P., Dantas, C. R., Oliveira, H. L. R., Lopes, L. G. B., and Werner, C. M. L.: Odyssey-SCM. In: http://reuse.cos.ufrj.br/odyssey/scm, Accessed in: 25/Jul/2005
15. Murta, L. G. P., Oliveira, H. L. R., Dantas, C. R., Lopes, L. G. B., and Werner, C. M. L.: Towards Component-based Software Maintenance via Software Configuration Management Techniques. Workshop on Modern Software Maintenance (WMSWM), Brasília, Brazil, October (2004)
16. Nguyen, T. N., Munson, E. V., and Boyland, J. T.: The molhado hypertext versioning system. Conference on Hypertext and Hypermedia, Santa Cruz, USA, August (2004) 185-194
17. Ohst, D. and Kelter, U.: A Fine-grained Version and Configuration Model in Analysis and Design. International Conference on Software Maintenance (ICSM), Montreal, Canada, October (2002) 521-527
18. OMG: Meta Object Facility (MOF) Specification, version 1.4. Object Management Group. In: http://www.omg.org/technology/documents/formal/mof.htm, Accessed in: 25/Jul/2005
19. OMG: MOF 2.0 Versioning and Development Lifecycle RFP. In: http://www.omg.org/cgi-bin/doc?ad/02-06-23, Accessed in: 25/Jul/2005
20. OMG: XML Metadata Interchange (XMI) Specification, Version 2.0. Object Management Group. In: http://www.omg.org/technology/documents/formal/xmi.htm, Accessed in: 25/Jul/2005
21. Pressman, R. S.: Software Engineering: A Practitioner's Approach. McGraw-Hill (1997)
22. Render, H. and Campbell, R.: An Object-oriented Model of Software Configuration Management. International Workshop on Software Configuration Management, Trondheim, Norway, June (1991) 127-139
23. Roche, T. and Whipple, L. C.: Essential SourceSafe. Hentzenwerke Publishing (2001)
24. Voelcker, J.: Automating Software: Proceed with Caution. IEEE Spectrum, Vol. 25, no. 7, July (1988) 25-27
25. Werner, C. M. L., Mangan, M. A. S., Murta, L. G. P., Souza, R. P., Mattoso, M., Braga, R. M. M., and Borges, M. R. S.: OdysseyShare: an Environment for Collaborative Component-Based Development. IEEE Conference on Information Reuse and Integration (IRI), Las Vegas, USA, October (2003) 61-68
26. White, B. A.: Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction. Addison-Wesley (2000)