Aspects of Lyee Configuration Management

Volker GRUHN, Raschid IJIOUI, Dirk PETERS, Robert QUECK, Clemens SCHÄFER
Chair for Applied Telematics / e-Business,[*]
Universität Leipzig, Klostergasse 3, D-04109 Leipzig, Germany

Abstract.
Based on the static structure of Lyee programs, a configuration management concept for Lyee software projects is presented. After describing a modularization concept which is vital for configuration management, the configuration management concept itself is elaborated by showing the involved elements and procedures. Finally, the structure of the actual configuration management tool is presented and discussed.

1 Introduction

Software configuration management (SCM) has been defined as the discipline of controlling the evolution of complex software systems [6]. As a vital task in professional software development, it also has to be performed in software projects developed using the Lyee methodology.

We start our considerations with the motivation of SCM for Lyee. Then, we summarize the static structure of Lyee programs that has been presented in [1]. Based upon this, we motivate and elaborate on a modularization concept that enables us to split Lyee projects into smaller parts. Next, the elements and procedures of the Lyee software configuration management system (*LyeeSCM*) are depicted. Its relevant technical aspects are discussed briefly before aspects about the realization of the tool are shown.

2 Motivation for Lyee Software Configuration Management

Developing a SCM concept for Lyee is a relevant task, even though Lyee is often supposed to be completely different from other software engineering approaches. Informally speaking, SCM mainly addresses two aspects of software development: iterations in development phases - especially during implementation - and concurrent work. For example we can imagine a typical software project in an iterative phase, where programmers correct the errors in source code. Then, a SCM tool would be used to store the corrected versions of the software parts and combine them into the complete program. As the same programmers may work on the same source files at the same time, the concurrent nature of their work is obvious. Thus, the SCM is responsible for coordinating changes, distributing them around the development site, and avoiding or releasing conflicts that might occur.

[*]The chair for Applied Telematics / e-Business is endowed by Deutsche Telekom AG.

Even though there is no iteration for error correction in Lyee projects since the generated source code is always correct, there are possible iterations in the Lyee software process. For example, a ready-made Lyee software may be presented to the customer, who might see that he made wrong assumptions about his requirements. Hence, the Lyee software has to be adapted to meet the customer's corrected needs. This introduces similar iterations into the software process, so the situation is not different from the conventional situation described above. Hence, the need for SCM in Lyee projects is obvious.

In terms of concurrent development, Lyee projects are not different from conventional ones. In larger projects, the number of programmers who have to work simultaneously on the same code will definitely be greater than one. Thus, we have the problem of keeping the common data (the programmers' input) consistent around the programming site - a typical situation where configuration management is needed. Hence, we can subsume that for Lyee projects there is the same need for SCM as for software development projects performed in a conventional manner.

3 Static Program Structure

To set up an SCM system, it is important to identify *configuration items*. This means the entities forming a piece of software and their relationships with each other have to be identified. In case of Lyee, this implies that the relationships between the parts of a Lyee program have to be clearly defined, because then and only then the entities forming a program can be mapped onto the configuration items needed for configuration management. This work has mostly been done in [1] by describing the static structure of Lyee programs. Thus, it is sufficient to only present the essence of this structure here in order to base our further considerations upon it.

In [1], the parts of Lyee programs that are of structural interest for SCM are modeled using sets and their relationships. This is done by defining all elements of Lyee programs that are programmers' input and not provided by the Lyee framework as tuples which consist of other Lyee program elements or sets of Lyee program elements. The aim of these definitions is to capture the static structure of Lyee programs; it is not intended to provide a dynamic model that could represent a Lyee program at runtime. As the parts of Lyee programs which are of structural relevance for SCM purposes are emphasized, the definitions remain incomplete to a certain extent. In all cases, there are additional attributes without structural relevance which were left out to keep the definitions simple. These omitted attributes are subsumed in sets named X. Figure 1 shows one typical definition (here: for the Process Route Diagram PRD). In figure 2, the associations between all elements of the definitions are shown in the form of a UML class diagram.

Definition (Process Route Diagram)

A *process route diagram* is defined as a tuple $d = (\nu, B, b_0, p_0, X)$ where ν is the *name* of the process route diagram. $B = \{b_1, ..., b_n\}, n \in \mathbb{N}$, is a set of *base structures*. $b_0 \in B$ is an *initial base structure*, p_0 an *initial pallet*.

Figure 1: Sample Definition of a Process Route Diagram

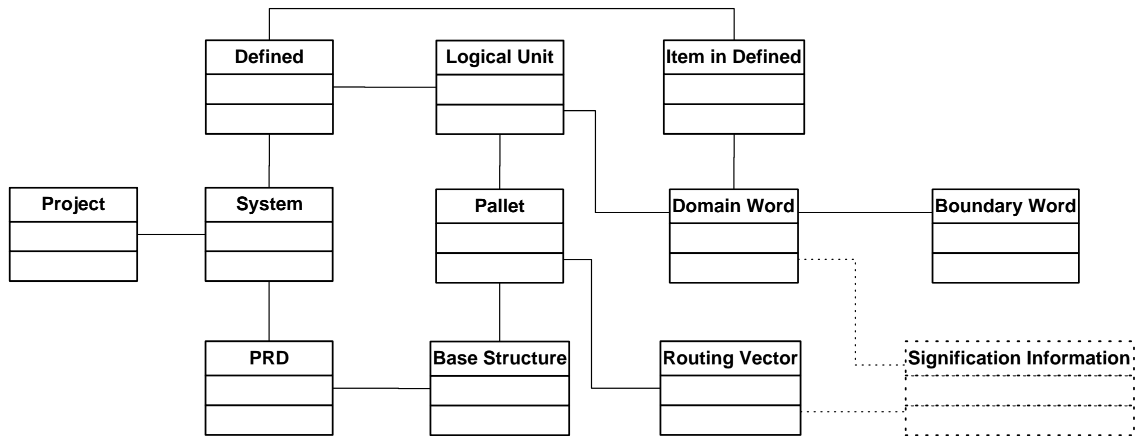

Figure 2: UML Class Diagram of Static Program Structure

3.1 Boundary Software

In the definitions mentioned before, boundary software has not been explicitly modeled. Since boundary software has to be regarded as an important part of any Lyee program, the need for representing it within the definitions seems obvious. Unfortunately, the structure of boundary software is not as clearly derivable as the structure of Lyee programs in general. This is due to the fact that the structure of boundary software is highly dependent on the target system the software is being developed for. If, for example, Visual Basic was used as the target language, the boundary software for the designed screens would have to be provided in Visual Basic files. If the target language was Java, these files and their structure would be different.

As a result [1] concludes that the boundary software can be regarded as part of the set X of either the defined definition or the system definition. Thus, incorporating boundary software into the system is more a matter of the implementation of the static structure than of its definitions, which is sufficient to go ahead in the context of this paper.

3.2 Completeness of Program Structure

In the presentation of the definitions, there are elements that are not covered by the static structure definitions. This is due to the fact that definitions are only provided for those elements comprising the “programmers’ input” of a Lyee program. There are no definitions for the elements which are provided by the Lyee framework. If we keep in mind that the elements *Tense Control Function*, *Predicate Vector Information*, and all *Action Vectors* (except some routing vectors captured by the definitions) are provided by the Lyee framework, it is feasible that all elements of Lyee programs are covered by the definitions, although not all elements of Lyee programs were defined. This means that it is possible to build a structural representation of a Lyee program which is sufficient for the needs of configuration management based on the definitions presented above.

3.3 Mapping the Lyee Program Structure onto Software Objects

In terms of SCM, software products consist of software objects that are related to each other. Thus, we now have to determine how the structure defined above can be mapped onto a struc-

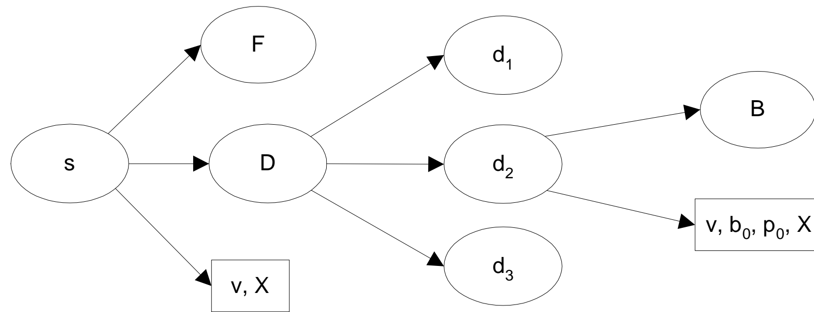

Figure 3: Pallet and Logical Unit as NUCM Artifacts

ture that can be processed by a SCM system. From now on, we will use the Network-Unified Configuration Management system (NUCM) [2], a testbed for SCM systems that provides the functionality necessary to implement a SCM system, as a basis for our considerations and the later implementation. It should be noted that in NUCM software objects are called *atoms* if they are atomic or *collections* if they are composite.

A first and straightforward approach leads to a mapping which transforms the tuples and sets in our definitions into collections in NUCM, whereas the remaining elements of our tuples become atoms. This is a valid approach as all information from the static program structure is mapped onto items that can be handled by NUCM. However, this approach is rather ineffective as it neither considers NUCM storing all entities in files nor the granularity of Lyee software. Hence, we have to identify configuration items in Lyee software just like in "traditionally-developed" software. In the latter, we have a boundary in the abstraction below which no decomposition of items takes place for versioning purposes. For the modularization concept (see below), we identified the PRDs as the level of abstraction on which we form modules of the software. Thus, it is reasonable to identify the configuration items on a level of abstraction below the PRDs. We can define the pallets to be the configuration entities. These considerations enable us to improve the first transformation mentioned before to get to a more practical one:

- A tuple from a definition always becomes a collection in the NUCM repository.
- All items of a tuple in a definition that are neither a set nor a tuple will be collected and stored together in one atom in the NUCM repository.
- Any set X in a definition will be added to the atom from the previous rule.
- All sets within tuples become collections in the NUCM repository.
- Pallets are stored together with all related data as one atom. Within these atoms, the complete hierarchy of items is stored. Thus, the structure of the pallets and its descendants can be reconstructed from the atom without storing the structure in collections and atoms.

Figure 3 shows a system and a process route diagram transformed into NUCM entities using this transformation. Collections are drawn as ovals, whereas atoms are drawn as rectangles.

4 Modularization Principle

Until now, we made observations with respect to the static structure of Lyee programs. This provided us with valuable information about establishing a configuration management concept. However, for this concept, the aspect of structural abstraction has to be regarded as well. As configuration management primarily addresses the concurrent development of software, it implies the need to split the software project into parts that can be developed concurrently. Unfortunately, Lyee does not support explicit structural abstraction. This means that we have to introduce this kind of abstraction in order to provide the basis for the applicability of the configuration management concept.

The basic idea that lies within common modularization approaches such as *static slicing* [7], *dynamic slicing* [3], or *concept analysis* [5] is the consideration of a "uses"-relation between functions and global variables. This relation provides vital information for modularization. If we regard the situation in Lyee, we do not have valuable information from the aspects related to functions like type inferencing or method signatures: Lyee software just is not structured in such a way that a program consists of a set of functions that call each other. However, something that Lyee and other approaches have in common is the use of global variables. In Lyee, a program's data storage is always performed using constructs similar to global variables: all data is stored and retrieved from the defineds and the defined items. Thus, we can investigate if it is feasible to base some kind of modularization of Lyee programs onto the usage of defineds and defined items in programs.

Now, we have to consider two aspects: Firstly, we have to find a correct concept for the identification of modules. This implies that the modules constitute the disjoint elements of a set and the full set has to represent the complete program. Secondly, this concept has to produce modules that are of a reasonable size developing them - in other words, the concept has to be practical. If we regard the structure of Lyee programs, all computation, data input and output is done on behalf of words. Considering the structure of Lyee software, words are defined within base structures. Thus, it is not possible to access words outside of the base structure they have been defined in. We can argue whether it is feasible to group base structures together to form modules or not. However, if we regard the complexity of functions that are modeled by base structures, we come to the conclusion that one single base structure captures a rather low level of functionality. Hence, it cannot be used in a sensible way as a model for development by concurring programmers. To overcome this problem, we raise the level of abstraction and combine multiple base structures into a process route diagram. It seems that the process route diagram has a functionality in itself that makes it possible to see the process route diagram as one entity with a certain functionality. This corresponds to the fact that in the Lyee documentation, the process route diagram is mentioned to capture the functionality of a so-called *business function* - a function that covers a typical use case.

Using these considerations, we can state that it is feasible to build modules of Lyee software on the granularity level of the process route diagrams. Now, the usage of the defineds can be helpful for our approach. We can try to group process route diagrams by their usage of defineds and defined items. Process route diagrams that use the same defineds could be grouped into one module. However, there is a problem with this approach: since a lot of data is stored in a database and almost the whole application accesses this data, it would not be possible to cut the application into modules anymore. However, from a practical point of view, there should be modularization in this case, too. Thus, we should be able to identify defineds and defined items that can be used by multiple modules. This means we have to distinguish between (initially) global variables that are local in the module after the modularization, and

variables that remain global after modularization.

4.1 Formal Concept

From these considerations, we now formalize the modularization concept more concretely. The idea is to recursively collect all defined items in a set that are used by a certain element, such as a process route diagram or a system for example. By intersecting these sets for different elements, we can judge if the two elements represent two valid modules, since the intersection of the sets has to be empty in that case. We initially define a function that addresses the defined items' visibility.

Definition: Visibility of Defined Items
The visibility function $v(x)$ of a defined item x is defined as follows:

$$v(x) := \begin{cases} \{x\} & \text{if the item is module-local} \\ \emptyset & \text{else} \end{cases}$$

This definition is trivial: $v(x)$ will be $\{x\}$ if the defined is marked as module-local, otherwise it evaluates to the empty set. If we assume that initially all defined items which belong to defineds of the type *screen* (i.e. that are dialog windows) are marked as local and that all defined items that belong to database defineds are marked as global, we get a sensible starting situation for all defined items in a program.

For the examination of the correlation of elements, we try to identify the subsequent or referenced items that are implicitly accessed when an element is selected. To reach this goal, we define a helper function.

Definition: Helper Function
For any element of the definitions of the static structure, the function *elementRel*(x) returns a set containing all referenced defined items that are not global. Thus, the function *elementRel* is defined as follows:

$$elementRel(x) := \begin{cases} \emptyset & \text{if } x \text{ is a routing vector} \\ setRel(x) & \text{if } x \text{ is a set} \\ tupleRel(x) & \text{if } x \text{ is a tuple} \\ v(x) & \text{if } x \text{ is a defined item} \\ \emptyset & \text{else} \end{cases}$$

This function refers to two other functions to work on sets and tuples. These will be defined next.

Let $t = (t_1, t_2, \ldots, t_n)$ be a tuple with n components t_1 to t_n, $n \in \mathbb{N}$. Then, the *tupleRel* function provides a set of all related elements and is defined as follows:

$$tupleRel(t) := \bigcup_{i=1}^{n} elementRel(t_i)$$

Analogously, we define the *setRel* function:

Let $s = \{s_1, s_2, \ldots, s_n\}$ be a set with n elements s_1 to s_n, $n \in \mathbb{N}$. Then define

$$setRel(s) := \bigcup_{i=1}^{n} elementRel(s_i)$$

Rationale If x is a set or a tuple, *elementRel* evaluates to the unification of all sets gained by evaluating *elementRel* to all components of the set or the tuple. This is achieved by the evaluation of *elementRel* to *tupleRel* or *setRel* in the case that x is a set or a tuple. If x is a defined item, *elementRel* evaluates to the visibility function $v(x)$. Thus, *elementRel* returns the defined item if the item is locally defined due to the definition of the visibility function. If x is a routing vector, the recursion stops. These functions can be used to set up the following splitting rules.

Splitting Projects
A project $p = (\nu, \{s_1, s_2, ..., s_n\}, X)$ can be split into n independent systems s_1 ... s_n, $n \in \mathbb{N}$.

Rationale This is trivial. From the definition it is clear, that

$$\forall_{i \neq j} : elementRel(s_i) \cap elementRel(s_j) = \emptyset$$

is valid for all systems. This is even independent of the marking of the defined items: since systems do not share defineds per definition, the resulting set can only be empty.

Splitting Systems
Suppose that we are given a system s consisting of multiple process route diagrams. For the system $s = (\nu, \{d_1, d_2, \ldots, d_n\}, F, X)$, $n \in \mathbb{N}$, any two process route diagrams d_i and d_j with $i \neq j$ can be developed separately if

$$elementRel(d_i) \cap elementRel(d_j) = \emptyset.$$

Rationale The intersection will evaluate to the empty set if the two process route diagrams share no defined items marked as local. This means they only share global defined items, if there are any. According to our considerations, the two modules can then be developed separately.

4.2 Remarks

Obviously, we can now cut a Lyee program into modules. This concept is a basic approach for identifying modules in Lyee programs. It shows that it is possible to judge the dependency of modules mainly by way of a basic definition of the visibility of defined items. For this point, we simply defined that defined items are either module-local or global in our considerations. Here, we could add more granularity. For example, it would be possible to define the visibility of a defined item depending on the data flow direction. This means that a defined item could be visible for read-access while being invisible for write access. Or, we could define the visibility of a defined item depending on the area from which it is accessed. For example, a defined item could be global if it is accessed from certain process route diagrams, but

remain module-local when accessed from any other process route diagrams. Then, we have the possibility of grouping process route diagrams together to form one complete module. However, for the moment we can restrict ourselves to the simple approach presented here as it contains the main concept - all other approaches just provide more sophisticated ways of composing modules, but these do not have consequences for the configuration management concept.

5 Functionality of Lyee SCM

After preparing the formal foundation for Lyee configuration management, we can now have a more detailed look at the elements and the procedures of the actual configuration management system. These parts will directly determine the architecture of the LyeeSCM tool discussed later.

5.1 Elements of the System

Initially, we list all logical elements that work together comprising the configuration management system. Mainly, these elements are necessary to provide persistence for the versioning information that occurs during the configuration management process.

a) *NUCM Repository*
The NUCM server is running on the server machine. We refer to this system as the *NUCM Repository*. It acts as a file container and stores files and their versions. The client/server architecture of NUCM enables us to design LyeeSCM as a client/server system, where we can use the NUCM server as-is for the server part of LyeeSCM.

b) *MetaRepresentation*
The *MetaRepresentation* structure is a dynamic data structure on the client machine which is only available at runtime of the LyeeSCM tool. This structure mainly contains the parts of the Lyee program that correspond to the static structure modeled in [1]. In the MetaRepresentation we also store version tag information, i.e. information on whether the software objects on the client machine are up-to-date or not.

c) *LyeeALL Database*
The *LyeeALL* database on the client machine stores all data that is to be put under version control. Contents from this database can be read from and written to the MetaRepresentation, and contents of the MetaRepresentation can be written to the database. Hence, this database is the interface to the programmer, since its contents are originally changed through the LyeeALL tool. Changes in the Lyee program made using LyeeALL are reflected in this database, thus finding their way into LyeeSCM, and vice versa.

d) *SCMInfo*
The *SCMInfo* structure is created by LyeeSCM and stored on the client machine. This structure contains additional data that is needed for keeping track of the different aspects of the stored configurations. In the SCMInfo structure, all version information about the last synchronized state, i.e. the version tag of the last synchronization for every configuration item is stored. Additionally, the latest version of the configuration item itself is also stored in the SCMInfo structure. This data about the last version is needed for comparing and merging configuration items in case of version conflicts.

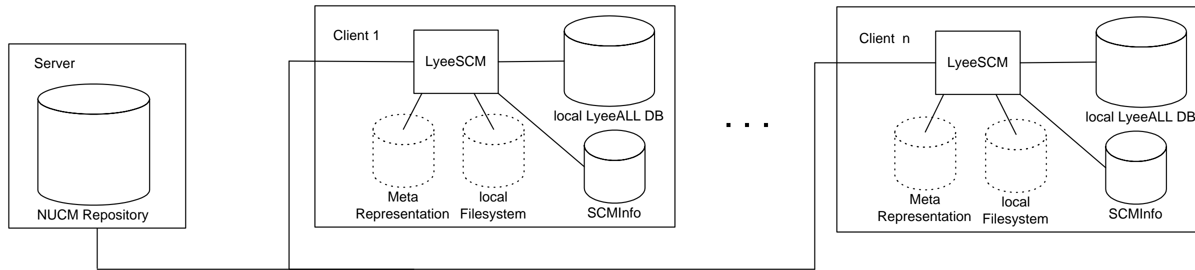

Figure 4: Structure of Lyee SCM System

e) *Local File System*
The *local file system* on the client machine is needed for operations of LyeeSCM and used to store a working copy of contents from the NUCM repository. Is is mentioned here for completeness only.

Figure 4 shows the structure of the LyeeSCM system: clients and servers are drawn as surrounding boxes, persistent data sinks are indicated by barrels. If the data sinks are only present during program runtime, they are drawn using dotted lines.

5.2 Different Steps of Configuration Management

After identifying the elements of LyeeSCM, we have to find out how these parts have to work together in order to establish the configuration management. For this purpose we list the major operations that are necessary to provide the full functionality of the system.

a) *Importing a Project into the SCM System*
This operation has to be carried out first if a Lyee software project has not been put under LyeeSCM control so far. The Lyee project has to be read from the local machine and stored in the NUCM repository on the server.

b) *Exporting a Project into Lyee*
This operation creates an instance of the project in the local LyeeALL database in the case that a project under SCM control is not yet present in the LyeeALL database. This way, a developer can join the development of the software.

c) *Checking the Status*
When there is a project in the local LyeeALL database, a valid SCMInfo structure is available on the local machine, and the project is stored in the NUCM repository on the server, we occasionally may want to check the status of the system. This operation then provides information on whether parts of the system have been changed, added or removed remotely or locally.

d) *Synchronizing*
This operation performs all changes that were identified when checking the status in order to synchronize the local LyeeALL database with the NUCM repository. Consequently, after the completion of this operation, the global version of the software and the local version in the LyeeALL database will be the same.

Now, we will take a more detailed look at some of the above procedures. Since these considerations will present all relevant aspects of the concept, we can concentrate on the subset of the above procedures listed below.

5.3 *Importing a Project into the SCM System*

This operation has to be carried out when a program exists in the LyeeALL database and shall be put under version control. The following steps have to be performed in this case: Initially, the local file system is cleared, and a new and empty SCMInfo is created. Then, the local LyeeALL database is scanned for all entries belonging to the project. From these entries, the MetaRepresentation is built. All items from the MetaRepresentation are written to the local file system according to the transformation rules to make them compatible with NUCM. For all items written to the file system, their information is added to the SCMInfo structure. Finally, all items written to the local file system are added to the NUCM repository using the import functionality offered by NUCM. At last, the local file system can be cleared.

5.4 *Calculation of Status*

The necessary steps for the calculation of status are obvious. Let us consider first that the item we are investigating exists in the MetaRepresentation, the SCMInfo structure and the local file system (i.e. the NUCM Repository). Furthermore, for all considerations about status determination, we assume that all versions of all configuration items can be temporally ordered. This means that the version tag of a more recent version of a software object is greater than the tag of an older version of the same software object.

Table 1 shows all situations that can occur for a software item. The three columns N, S and M contain the version tags of a configuration item in the NUCM repository (N), the SCMInfo structure (S) or the MetaRepresentation (M). x indicates that a version tag has a "don't care" value. For other purposes, there are three possible version tags, a, b and c. These are ordered according to the relation $a < b < c$, meaning that a is older than b, b is older than c, and - transitively - a is older than c. Version tags of a, b or c indicate that the temporal order of the different items' version tags is significant. If the x tag is used, the pure existence of a configuration item is sufficient. If a configuration item is not existing in a structure, we write a dash (-).

For the temporal order of the version tags, there are two invariants that have to be fulfilled in order to have a valid set of version tags:

$$\text{Invariant 1: } S \leq M$$
$$\text{Invariant 2: } N \geq S$$

Invariant 1 (referenced as (1) in the table) indicates that the version tag stored in the SCMInfo cannot be older than the version tag stored in the MetaRepresentation. This is obvious as the data from the SCMInfo is written when the data is retrieved from the NUCM repository. At the same time, the version tag of the MetaRepresentation is set. The new tag in the MetaRepresentation can only result from a subsequent change of the configuration item. Thus, the version tag of the MetaRepresentation cannot be older than the tag stored in the SCMInfo.

Invariant 2 (referenced as (2) in the table) indicates that the version tag stored in the SCMInfo cannot be older than the version tag stored in the NUCM repository. Between two synchronizations of the SCMInfo and the NUCM repository, the SCMInfo structure is not changed. As the version tags can only grow with the time, later version tags in the NUCM repository cannot be older than the corresponding tag in the SCMInfo. Column C indicates a possible conflict that arises from the situation: A dash (-) indicates that there is no conflict. A question mark (?) indicates that there might be a conflict. The actual existence of the conflict

can be determined by performing a merge operation in that case. An exclamation mark (!) indicates a definite conflict that only can be resolved manually.

N	S	M	Item Situation	C
a	*a*	*a*	no changes	-
a	*a*	*b*	local change	-
a	*b*	*a*	cannot occur - (1), (2)	-
a	*b*	*b*	cannot occur - (2)	-
b	*a*	*a*	remote change	-
b	*a*	*b*	remote & local changes	?
b	*b*	*a*	cannot occur - (1)	-
a	*b*	*c*	cannot occur - (2)	-
a	*c*	*b*	cannot occur - (1), (2)	-
b	*c*	*a*	cannot occur - (1), (2)	-
b	*a*	*c*	local & remote changes	?
c	*a*	*b*	remote & local changes	?

N	S	M	Item Situation	C
c	*b*	*a*	cannot occur - (1)	-
a	*a*	-	local deletion	-
a	*b*	-	cannot occur - (2)	-
b	*a*	-	local deletion & remote change	!
x	-	*x*	local & remote addition	?
x	-	-	remote addition	-
-	*a*	*a*	remote deletion	-
-	*a*	*b*	local change & remote deletion	!
-	*b*	*a*	cannot occur - (1)	-
-	*x*	-	local & remote deletion	-
-	-	*x*	local addition	-
-	-	-	not sensible	-

Table 1: Version Correspondence

This completes our consideration of the calculation of the status. Of course, this status can be displayed for informational purposes, but it also is the first step towards synchronizing the local and remote versions of the software.

5.5 Synchronizing

Synchronizing means performing the changes that are deemed necessary as a result of checking the status. The following steps have to be performed in order to synchronize the LyeeALL database with the NUCM repository: Initially, the current status has to be ascertained. This is done as described in *Calculation of Status*.

Then, all items that are marked as “locally changed” are updated in the NUCM repository by storing the changed version. All items that are marked as “remotely changed” are updated in the local LyeeALL database. All items that are marked to be added to the repository are added to NUCM, and all items that are marked to be added to the LyeeALL database are added to the database. All items that are marked to be deleted from the repository are deleted in NUCM. The same is true for the items that are marked to be deleted from the LyeeALL database.

For all items that have been added, deleted or changed, the information in the MetaRepresentation and the SCMInfo has to be updated. This is obvious since the latest version has to be present in the MetaRepresentation and in the SCMInfo structure after the synchronization.

5.6 Conflict Resolution Through Merging

As indicated above, there may be conflicts that cannot be resolved automatically. These have to be solved manually by editing the conflicting elements. How to do this depends on the type of conflict. LyeeSCM can try to solve these conflicts automatically. If and only if this automation fails, the conflicts have to be resolved manually.

Conflicts are resolved automatically by a three-way-merging operation of the item data. To perform this operation, the version from the MetaRepresentation has to be available. Also, the latest version from the NUCM repository has to be retrieved. Additionally, the version

that is stored in the SCMInfo has to be retrieved. Now, a three-way difference has to be calculated to detect the set of differences between the version in the MetaRepresentation and the SCMInfo version, and the differences between the version in SCM Info and the latest version. Then, it can be checked if the changes can be combined. If this is the case (and the changes are not conflicting), a merged version of the three files can be produced. This merged version can be added to the NUCM repository as the latest version. For these operations, we can use existing tools since we can represent the data from the MetaRepresentation and SCMInfo as XML files that can be processed like ordinary text files in terms of calculating differences and merging, if we restrict the XML file representation and enforce a static order of the XML elements within the file.

6 Configuration Management Tool

In the previous sections, we have developed concepts for configuration management using the Lyee methodology. However, configuration management is not an end in itself. Configuration management is a task that has to be performed continuously during the development of software. Thus, the concepts need to be implemented and a tool has to be created to support them. One possible implementation of the concept has already been built in the form of the LyeeSCM tool. In this section of the paper, the design of the tool will be described.

6.1 Application Design

An overview of the complete system is given in figure 5. It shows the system consisting of two major parts, the NUCM server and the client system. The NUCM server can be used as provided by NUCM and has to be run under a Cygwin [4] environment which allows Unix programs to be run in a Windows environment.

The client system that communicates with the NUCM server by TCP/IP consists of multiple parts: Firstly, there is a NUCM client library (DLL). This library is an adaption of NUCM's client application made for our purposes. Secondly, there is a set of UNIX tools (*diff*, *diff3*, and *patch*) used to provide the merge-functionality necessary for detecting and resolving conflicts. And finally, there is the actual application *LyeeSCM*. This application uses the NUCM client DLL as well as the set of UNIX tools. These two have to be embedded into an Cygwin environment, as they are originally designed for UNIX and not for Windows.

The whole application is designed as a Windows application. As Microsoft Visual C++ was used for the implementation, it was possible to use the Microsoft Foundation Classes framework (MFC). This framework provides a Document-View Architecture for the applications, as well as all necessary framework features to rapidly build windows applications. The MFC provides easy-to-use support for menus, tool bars, windows, splitting bars in windows etc. Thus, the decision to use MFC when building a windows application was quite natural.

6.2 Realization of Change Set Mode

We wanted to provide change-based versioning as well as state-based versioning for LyeeSCM. Unfortunately, NUCM does not provide the change-based mode. However, it is possible to extend NUCM to provide a change-based versioning feature in addition to the state-based one.

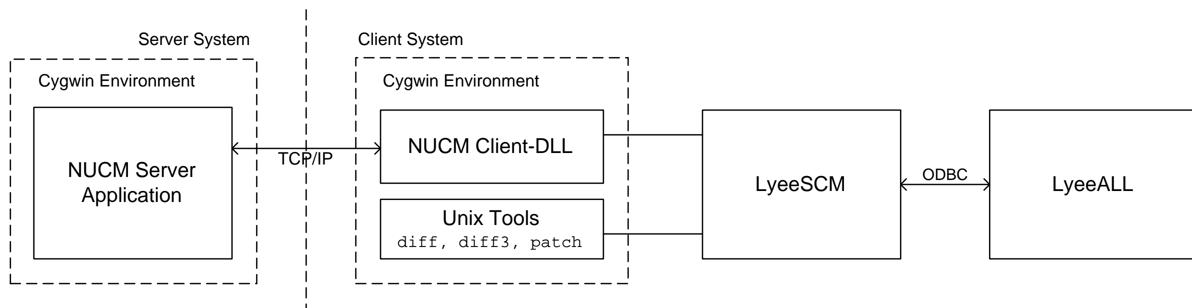

Figure 5: LyeeSCM System Structure

Internally, this is achieved by two main ideas: Firstly, we store the differences (deltas) of two versions of an atom in the repository. (For the state-based mode we simply store the complete versions.) Secondly, we add an additional atom for each collection. In this atom, we store the changes of the collection for every version, i.e. we mark the addition and the removal of atoms or collections within the collection. Physically, we do not remove items from the collections anymore.

To retrieve a special version in change set mode, the NUCM repository is opened as in state-based mode, starting from the baseline. However, every version of an atom or collection in the repository corresponds to one change set entry. Thus, for every change set entry we open the corresponding version of an atom and combine the different deltas to retrieve the final version. Similarly, we combine the deltas of the item lists if a collection is opened in order to retrieve the final list of items showing the contents of a collection in the change based versioning mode.

7 Discussion and Further Work

The redefined structure of Lyee software acts as a basis for a modularization concept that is quite necessary and will directly influence the software process of Lyee. This is so because the modularization concept clearly shows how programs can be divided into parts that can be developed independently and concurrently.

In this paper a configuration management concept for Lyee has been developed. Based on the static program structure and the modularization concept, Lyee programs are investigated and the concept is expressed. This concept is the concrete input for the realization of the configuration management tool.

The concept is bound to the NUCM system, however, as NUCM provides functionality that can be found with other systems, it would also be possible to use other systems for a configuration management tool for Lyee.

The modularization concept gives an idea how modules in Lyee programs can be identified. Based on considerations of a sensible size of these modules and general considerations about splitting software, we found a concept that allows the splitting of Lyee-based programs according to the visibility of defined items.

The implementation and first tests showed that the above concepts are working. The LyeeSCM tool presented her can (even as prototype) be used for configuration management with Lyee. However, the validation cannot provide answers to the question if development of larger systems using Lyee is superior to developing them using traditional methods. This is research work that has yet to be done.

Further research should also investigate how the development process with Lyee changes if configuration management is applied. For these investigations, the LyeeSCM tool provides a valuable contribution. It also would be interesting to see whether the modularization concept is feasible in a way that the size and composition of the detected modules meet the needs of large-scale system development with Lyee.

8 Conclusion

In this paper we presented a modularization concept for Lyee software, based on the static structure of Lyee programs identified in earlier work. This modularization concept is important for dividing a Lyee project into appropriate parts and thus enabling developers to build Lyee applications concurrently - one large enabling factor for the application of a configuration management concept. Next, the structure and the functionality of such a concept were shown. This concept has been realized as a tool. We are confident that the evaluation will prove the indications available to us at the moment, showing that the presented concepts are valid and well-defined.

All in all, through the work presented here, Lyee is extended with concepts known in software engineering as key issues for large scale system development. We therefore hope that Lyee can mature in this area and establish itself as a new and interesting methodology in the world of software engineering.

References

[1] V. Gruhn, R. Ijioui, D. Peters, and C. Schäfer. Configuration Management Concept for Lyee Software. In *Proceedings of Lyee W02*, October 2002. ISBN 1-58603-288-7.

[2] A. van der Hoek, D. Heimbigner, and A. L. Wolf. A Generic, Peer-to-Peer Repository for Distributed Configuration Management. In *International Conference on Software Engineering*, pages 308–317, 1996.

[3] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[4] Red Hat, Inc. (Ed.). *Cygwin*. http://cygwin.com, May 15, 2003.

[5] M. Siff and T. Reps. Identifying Modules via Concept Analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.

[6] W. F. Tichy. Tools for software configuration management. In J. F. H. Winkler, editor, *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 1–20, Grassau, Germany, 1988. Teubner Verlag.

[7] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.