Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT

Knowledge-Based Systems 16 (2003) 441–447

Knowledge-Based SYSTEMS

www.elsevier.com/locate/knosys

Configuration management for Lyee software

V. Gruhn*, R. Ijioui, D. Peters, C. Schäfer

Faculty of Mathematics and Computer Science, University of Leipzig, Chair for Applied Telematics/e-Business,[1]
Klostergasse 3, Leipzig 04109, Germany

Abstract

This article presents a configuration management concept for software projects using Lyee methodology. To illustrate this concept, an introduction in configuration management is given. Then, the structure of Lyee programs is defined by sets and their dependencies. From this structure, the actual configuration management concept is deduced and discussed by rendering the structure for an existing configuration management testbed and describing the involved key players as well as the necessary procedures.
© 2003 Elsevier B.V. All rights reserved.

Keywords: Software configuration management; Lyee software; Network-unified configuration management

1. Introduction

Software configuration management (SCM) has been defined as the discipline of controlling the evolution of complex software systems [1]. As a vital task of professional software development, it also has to be performed in software projects developed using Lyee methodology. In this article, we present a SCM concept for Lyee. We first give a short introduction into SCM. Then, we define the structure of Lyee programs using mathematical definitions in order to provide a sound basis for further discussions. These definitions allow mapping the entities of Lyee programs onto configuration items introduced before. Finally, we depict the architecture of a system using an existing SCM tested.

2. Software configuration management

SCM can be regarded as an extension of general configuration management (CM). A standard definition of CM can be found at Refs. [2,3]. It describes the following CM procedures.

Identification reflects the structure of the product, identifies components and their types, making them unique and accessible in some form. *Control* addresses the release of a product and the changes to it throughout its life cycle. This is done by having controls in place that ensure consistent software through the creation of a baseline product. *Status Accounting* records and reports the status of components and change requests, and gathers vital statistics about components in the product. *Audit and Review* validate the completeness of a product and maintain consistency among the components, ensuring that the product is a well-defined collection of components.

Further surveys on SCM [3] extend this definition to include procedures like *construction management* that addresses the construction and building of products, *process management* that ensures the implementation of the organization's procedures, policies and life cycle model, and *team work control* that deals with the work and interactions between multiple developers.

According to Ref. [4], specific terminology is used to describe SCM in more detail: A *(software) object* (item) is any kind of identifiable entity put under SCM control. An object may be either *atomic*, i.e. it cannot be decomposed any further (internals are irrelevant to SCM), or it may be *composite*. A composite object is related to its components through *composition relationships*. Furthermore, there may be *dependency relationships* between dependent and master objects.

As these software objects are the key factors in approaching a SCM concept for Lyee software, we have

* Corresponding author.
E-mail addresses: gruhn@ebus.informatik.uni-leipzig.de (V. Gruhn); ijioui@ebus.informatik.uni-leipzig.de (R. Ijioui), peters@ebus.informatik.uni-leipzig.de (D. Peters), schaefer@ebus.informatik.uni-leipzig.de (C. Schäfer).

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

0950-7051/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.knosys.2003.08.004

to identify software objects with the mentioned characteristics within Lyee programs.

3. Static program structure

Identifying software objects for SCM implies that a structural representation of the considered software has to be found. This representation has to be formed in such a way that it regards software as a set of software objects that may be decomposed until they are atomic. Furthermore, these decomposition possibilities have to be modeled in the structure. This can be accomplished by using software objects that are sets of other software objects and that have *is-part-of* relations to other software objects.

Another Lyee-specific aspect has to be taken into account: in Lyee software, there are parts of software that are provided by the person writing the program. We can consider them 'programmer's input' to the Lyee system. On the other hand, there is software that is provided by the Lyee framework. For our SCM purposes, we have to make a clear distinction between these two: only software objects which are programmer's input have to be regarded for SCM, as the remainder is irrelevant in this context.

If we regard an initial description of Lyee program structure [5,6], we face the problem that identification of clear dependencies is difficult. Furthermore, the required decomposition is not supported and the distinction between programmer's input and the rest is not really clear. Another approach to describing Lyee programs is presented in Ref. [7]. In this article, Souveyet and Salinesi describe a meta model of Lyee. However, Souveyet and Salinesi do not take the distinction between framework-provided elements and programmer's input into account.

As there are no other approaches (due to the fact that Lyee is still evolving), we decided to remodel the structure of Lyee programs, aiming for two goals: to provide elements that show their compositions and dependencies, and to clearly separate between programmer's input and system-generated or framework-related parts.

As a starting point for our modeling, we chose to examine the structure of the LyeeALL database. Fortunately, the designers of this tool made similar assumptions and only modeled these parts of the system in their database, which are programmer's input and not framework-related. By investigating the LyeeALL database structure, we were able to identify and define a set of software objects that can be used to model the structure of Lyee programs.

3.1. Structural definitions

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. Since we tried to emphasize the parts of Lyee structures that are of structural relevance for our SCM purpose, the definitions are incomplete to a certain extent: in all cases, there are additional attributes which we left out to keep the definitions simple. This is possible as those attributes are of technical nature and thus irrelevant for the program structure. All omitted attributes have been subsumed in sets named X that can be found in all definitions. We begin with the topmost entity of a Lyee program, the project.

Definition 1 (Project). A Lyee project is a tuple $p = (\nu, S, X)$ consisting of the name of the project ν and a set of systems $S = \{s_1, \ldots, s_n\}$ with $n \in \mathbb{N}$.

This definition of the project is quite obvious and motivated by the inspection of the database structure used by LyeeALL, as the systems do not occur in the documentation of Lyee.

Definition 2 (System). A Lyee system is a tuple $s = (\nu, D, F, X)$ with ν as the name of the system. $D = \{d_1, \ldots, d_n\}$ is a set of process route diagrams (PRDs), $F = \{f_1, \ldots, f_m\}$ is a set of defineds with $n, m \in \mathbb{N}$.

This definition reflects the expectations from the Lyee theory. The topmost structural unit within projects are PRDs. Thus, composing a system out of PRDs is correct. As the defineds are associated to parts of the PRDs, they should not be separated from the definition of the system. However, defineds can be used in different PRDs. Hence, modeling defineds and PRDs as two different sets within one system is appropriate.

Definition 3 (Defined). A defined in terms of Lyee theory can be specified as a tuple $f = (\nu, t, X)$, where ν is the name of the defined and t its type. Typically, the type t of the defined will be one of $\{screen, database, file, \ldots\}$.

A defined refers to an entity such as a database table or a dialog window. The name of this entity will be reflected by ν. The type t of the defined has been isolated in the definition for SCM purposes, since this type indicates which additional data (boundary software) not covered by the LyeeALL database has to be processed. For example, in case of $t = screen$ the definition of the related dialog window has to be put under SCM control, too.

Definition 4 (Item in Defined, also: Defined Item). An item in a defined can be specified as a tuple $i = (\nu, f, X)$ where ν is the name of the item and f is the defined that it belongs to.

The items of a defined are, for example, columns within a database table or dialog fields within a dialog window. Thus, for this definition, the database structure matches the definition from the theory.

Definition 5 (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. $B = \{b_1, \ldots, b_n\}, n \in \mathbb{N}$, is a set of base structures. $b_0 \in B$ is an initial base structure, p_0 an initial pallet.

According to the expectations from the theory, defining a PRD mainly as a set of base structures is reasonable. In a PRD, the first base structure has to be tagged. This is done by specifying the initial base structure b_0 in the definition. At this point, adding the initial pallet p_0 might seem superfluous. However, as the routing will be defined between pallets in later definitions, it is helpful to have the initial pallet within the definition. From Lyee theory, we can derive the invariant that the initial pallet p_0 has to be the $W04$ pallet of the initial base structure b_0.

Definition 6 (Base Structure). A base structure is a tuple $b = (\nu, b_p, p_4, p_2, p_3, X)$ where ν is the name of the base structure, b_p is the parent base structure, and p_4, p_2 and p_3 are pallets. If a base structure has no parent, we write $b_\mathrm{p} = \emptyset$.

From this definition, it is obvious that a base structure consists of three pallets p_4, p_2 and p_3. The parent base structure b_p was inspired by the database structure of the LyeeALL tool. This is also corresponding to Lyee theory, as this element provides the structural dependencies between base structures that are necessary for the framework to automatically establish routing vectors.

Until now, our definitions are rather obvious and have been created straightforwardly. Next, we abstract and regroup some information to formulate the following definitions precisely. Interestingly, this abstraction has also been performed by the creators of the database structure of the LyeeALL tool. The main idea behind the abstraction is to associate all kinds of Lyee vectors with words.

Definition 7 (Pallet). A pallet is a tuple $p = (\nu, L, R, X)$ consisting of a set of logical units $L = \{l_1, \ldots, l_n\}$ and a set of routing vectors $R = \{r_1, \ldots, r_m\}$ with $n, m \in \mathbb{N}$. ν is the name of the pallet.

According to Lyee theory, different logical units are present on a pallet. The abstraction above is the key to modeling the routing vectors as a set of vectors that is part of the pallet. This is obvious as routing can only occur between pallets.

Definition 8 (Logical Unit). A logical unit is a tuple $l = (\nu, f, t, W, X)$ where ν is the name of the logical unit and $\{t \in \mathit{input}, \mathit{output}\}$ is its type. f is the defined that the logical unit is assigned to. $W = \{w_1, \ldots, w_n\}$ is a set of domain words with $n \in \mathbb{N}$.

Modeling a logical unit in this way is reasonable. According to Lyee theory, words are linked to defineds in logical units. Furthermore, we have to take into account that for words, which should be input or output by the logical units, signification vectors have to exist. For this reason, it is appropriate to group the signification vectors with the words in the logical units. As logical units deal with the input and output of words, the input/output attribute t is well-founded.

Remark (Action Vectors, Signification Vectors). Here, one might expect the definition of action vectors. From a structural perspective, this might seem natural. However, action vectors in Lyee theory are always implicitly given by the use of logical units. This is the reason why they can be created automatically by the Lyee environment.
For every input logical unit, an input vector has to be created. It is also obvious that this input vector has to contain the input statements for all words in the logical unit. The same situation occurs for the output vectors and the structural vectors.
A special situation occurs for the routing vectors: several routing vectors do not need to be captured by the definitions here, as they are implicitly given by the theory (and thus by the framework). Only those routing vectors which are explicitly given by the programmers have to be captured by definitions (see below).
The signification vectors are not modeled explicitly, as their information will be combined with the domain words later.

Definition 10 (Routing Vector, Routing Word). A routing vector in Lyee is a tuple $r = (\nu, S, t, p, X)$ with the name of the vector ν, the type $t \in \{\mathit{duplex}, \mathit{continuous}, \mathit{multiplex}, \mathit{recursive}\}$, and the pallet p. $S = \{s_1, \ldots, s_7\}$ is the signification information that represents the seven boxes of a predicate vector.

The separate modeling of the type of the routing vector t is necessary since the type of routing has implications on the structure of the software as a whole and must be regarded when a component-oriented view of Lyee software is established.

Definition 11 (Domain Word or Word). A word in terms of Lyee theory is a tuple $w = (\nu, i, S, X)$ with ν as the name of the word. i is a defined item the word is associated to, and

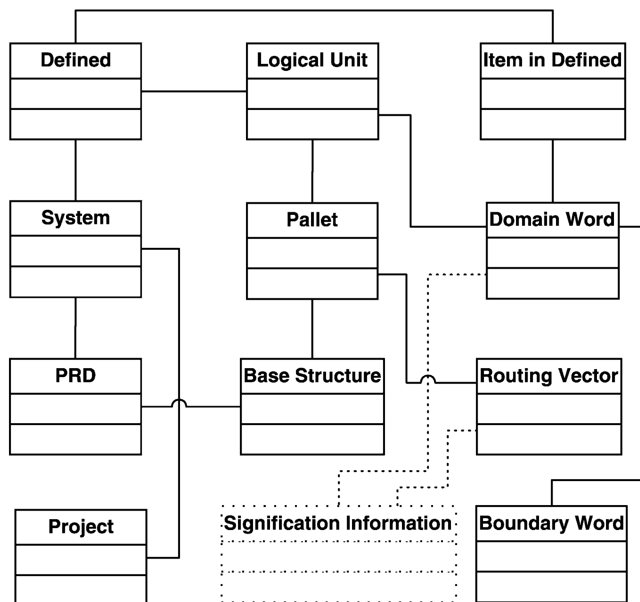

Fig. 1. UML class diagram of static program structure.

$S = \{s_1, \dots, s_7\}$ is the signification information from the signification vector of the word that is used to calculate the word's value.

Definition 12 (Boundary Word). A boundary word in terms of Lyee theory is a tuple $w_b = (v, w, X)$ with v as the name of the boundary word and the word w. The boundary word w_b can be regarded as a reference to the word w.

The boundary word is a word used by different logical units to access words from other base structures. This construct can also be found in the database.

If we take all *is-part-of* relations between the definitions above, we can combine our software objects to a whole program structure, shown in the form of a UML class diagram in Fig. 1. Note that in the diagram, there is also a class *Signification Information* that has no direct counterpart in a definition. However, this class has been drawn in order to show that both the domain words and the routing words contain signification information.

3.2. Boundary software

In the definitions above, boundary software has not been explicitly mentioned. 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.

However, one aspect that remains unchanged is that certain boundary software is associated with certain parts of the Lyee program. For example, the GUI part of a Lyee program is related to the system items or the defineds of the definitions above. This is underscored by the fact that every dialog in the GUI has to be represented by a defined. Now, it is possible to argue about whether a dialog's boundary software has to be stored with the defined or the system. Nevertheless, one aspect that both possibilities have in common is 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 this structure than of the definitions of the static structure.

3.3. Completeness of program structure

During the presentation of the definitions, we already mentioned elements that are not covered by the static structure definitions. This is due to the fact that we only provided definitions for the elements that form the 'programmer's input' of a Lyee program. We did not set up definitions for the elements that 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 our definitions) are provided by the Lyee framework, it is plausible that we cover all elements of Lyee programs in our considerations, although we only provide definitions for those elements that are given by the programmers. This enables us to set up the CM concept for Lyee programs based on the definitions presented above.

4. Lyee configuration management concept

The process of establishing a CM concept for Lyee software can be split up into two tasks: first, a mapping has to be defined that associates the elements of the static structure of Lyee programs with software objects in the SCM system. This relates to the task of identification mentioned earlier. Then, the operations to put a Lyee program under SCM control or to retrieve a version of the program from the repository have to be specified. This corresponds to the task of control in the overview given in Section 2 of this article.

4.1. 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 structure that can be processed by a SCM system. From now on, we will use the Network-Unified Configuration Management system (NUCM) [8], a testbed for SCM systems that provides the necessary functionality to implement a SCM system, as the basis of 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 that transforms tuples and sets from our definitions into collections in NUCM, while the remaining elements of our tuples become atoms. This is a valid approach since 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 considers neither the fact that NUCM stores 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’. There, we have a boundary in abstraction below which no further decomposition of items takes place for versioning purposes. For example, in C programs, configuration items are on the file level. Source files are not decomposed for versioning purposes, and the version of a software object is equivalent to the version of the corresponding file. For a modularization concept for Lyee programs [9], 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. Hence, we can define the pallets to be the configuration items. These considerations enable us to improve the first transformation mentioned before to get to a 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 case.
- 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.

Fig. 2 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.

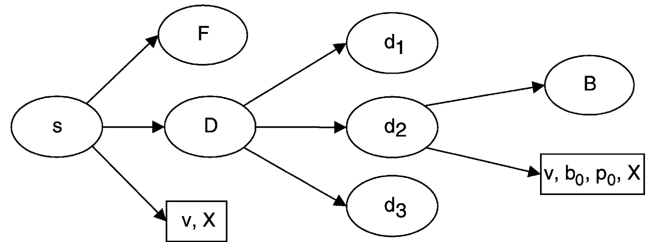

Fig. 2. One system and three process route diagrams as NUCM artifacts.

4.2. Configuration management system structure

With these rules set up, we can now transform any Lyee project in such a way that it can be put under the control of a NUCM system. We will briefly present the structure of the CM system for Lyee, *LyeeSCM*, in this section.

Fig. 3 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.

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.

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 before. In the MetaRepresentation, we also store version tag information, i.e. information whether software objects on the client machine are up-to-date or not.

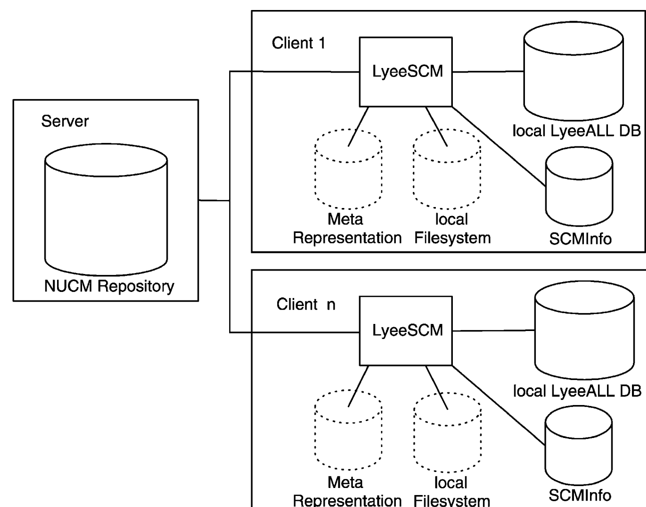

Fig. 3. LyeeSCM system structure.

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 acts as the interface to the programmer, since its contents are originally changed by using 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.

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 configurations. In the SCMInfo structure, all version information about the last synchronized state is stored, i.e. the version tag of the last synchronization for every configuration item. Additionally, the latest version of the configuration item itself is also stored in the SCMInfo structure. This data about the last version are needed for comparing and merging in case of version conflicts.

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. It is shown here for completeness only.

4.3. 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 CM. For this purpose, we list the major operations that are necessary to provide the full functionality of the system.

1. *Importing a Project into the SCM System*. This operation has to be carried out given the situation that there is a Lyee software project that 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.
2. *Exporting a Project into Lyee*. Now we assume that a project that is under SCM control is not present in the LyeeALL database. This operation creates an instance of the project in the local LyeeALL database. This way, a developer can join the development of the software.
3. *Checking the status*. Now we assume that we have a project in the local LyeeALL database that has been exported or imported. This means that there is a valid SCMInfo structure available on the local machine. Additionally, the project is stored in the NUCM repository on the server. Now we want to check the status of the system. This operation provides information on whether parts of the system have been added or removed remotely or locally, or were just changed.
4. *Synchronizing*. Synchronization comprises performing all changes that were identified as necessary by ‘checking the status’. These steps have to be performed in order to synchronize the local LyeeALL database and 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.

The elements of LyeeSCM mentioned before and the procedures listed here were combined to form a working SCM system. The system is capable of meeting all requirements that arise if CM is requested for the software development using Lyee.

5. Discussion and further work

The detailed elaboration of the CM concept has taken place in the meantime, and the LyeeSCM tool has been implemented and tested as a first step towards the validation of our concepts. Additionally, a concept for splitting Lyee programs into modules, which can be developed independently, has been derived by using the features of the definitions listed above. As it is not possible to describe all aspects of that work in this article, we focused on some aspects to give the reader a rough idea of the LyeeSCM system and its key players. One part of work that has to be performed next is the validation of the concepts shown above. In order to do this, the LyeeSCM tool should be used for CM in typical projects of Lyee software development to demonstrate its suitability. 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 and thus useful for the Lyee community.

6. Conclusion

We have defined the program structure of Lyee software using sets and their relationships. Using these definitions, we were able to formulate a configuration management concept for Lyee software. For this concept, we described the structure of the configuration management system, as well as its major procedures, which served as a basis for the implementation of the LyeeSCM configuration management tool. Using this configuration management concept, the software process using Lyee methodology is supported by another toolkit meeting the needs of large-scale system development.

References

[1] W.F. Tichy, Tools for software configuration management, in: J.F.H. Winkler (Ed.), Proceedings of the International Workshop on Software Version and Configuration Control, Teubner Verlag, Grassau, Germany, 1988, pp. 1–20.

[2] The Institute of Electrical and Electronics Engineers, New York, IEEE Guide to Software Configuration Management Plans, ANSI/IEEE Standard 828-1990, 1990.
[3] S. Dart, Concepts in configuration management systems, in: P.H. Feiler (Ed.), Proceedings Third International Workshop on Software Configuration Management, Trondheim, Norway, 1991, pp. 1–18.
[4] R. Conradi, B. Westfechtel, Configuring Versioned Software Products, in: I. Sommerville (Ed.), Software Configuration Management: ICSE'96 SCM-6 Workshop, LNCS 1167, Springer, Berlin, Germany, 1996, pp. 88–109.
[5] F. Negoro, Principle of Lyee Software, in: Proceedings of the International Conference on Information Society in the 21st Century, 2000, pp. 441–446.
[6] F. Negoro, The Predicate Structure to Represent the Intention for Software, in: Proceedings of the International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2001.
[7] C. Souveyet, C. Salinesi, Generating Lyee Programs from User Requirements with a Meta-model based Methodology, in: Proceedings of Lyee W02, 2002, pp. 196–211.
[8] A.v.d. Hoek, D. Heimbigner, A.L. Wolf, Generic, Peer-to -Peer Repository for Distributed Configuration Management, in: International Conference on Software Engineering, 1996, pp. 308–317.
[9] V. Gruhn, R. Ijioui, D. Peters, R. Queck, C. Schäfer, Aspects of Lyee Configuration Management, in: Proceedings of Lyee W03, 2003 (in press).