Reconciling Software Configuration Management and Product Data Management

Jacky Estublier
LIG Grenoble University
220, rue de la Chimie BP53
38041 Grenoble Cedex 9
France
jacky.estublier@imag.fr

German Vega
LIG Grenoble University
220, rue de la Chimie BP53
38041 Grenoble Cedex 9
France
german.vega@imag.fr

ABSTRACT

Product Data Management (PDM) and Software Configuration Management (SCM) are the disciplines of building and controlling the evolution of a complex artifacts; either physical or software. Surprisingly, these two fields have evolved independently; their respective solutions to the same problems are incompatible and their properties are different. PDM is good at modeling while SCM is good at building and supporting concurrent engineering. From a software engineering perspective, the challenge is to take the full potential of strong modeling capabilities, while preserving good concurrent engineering support. The paper shows that rich modeling, flexible evolution, and concurrent engineering supports have conflicting requirements and that a solution requires rethinking the concepts of evolution, versioning and modeling. We have developed a system, called CADSE (Computer Aided Domain Specific Environment), in which a product (software, physical or both) is modeled in a way similar to PDM and in which concurrent engineering and evolution is supported in the SCM way. To that end, the system is driven by models; evolution alone being defined through different models. The paper describes our system and discusses the early lessons of its first years of practical use.

Categories and Subject Descriptors

D2.2 Design tools and Techniques, D2.3 Coding tools and techniques, D2.11 Software Architectures, D2.12 Interoperability.

General Terms

Algorithms, Management, Design, Experimentation.

Keywords

Software engineering, product engineering, SCM, PDM, CAE, CAD, CAM, System models, Models.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC-FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia. Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

1. INTRODUCTION

The symbiosis between the material and software parts of a product is continuously increasing. The vast majority of industrial goods, in the near future, will include software, from tiny devices (a few lines of code), to cars and planes (millions of lines of code). In a given product, the boundaries between its software and physical parts are pretty loose and may move during design, implementation and even commercial life. This is well known, but the surprise is that the tools and methods used in these different areas (software engineering and product engineering) are not only different but incompatible. Provided the deep relationships between the parts, it becomes more critical every day to manage the complete product, all along its life, whatever the nature of its parts.

The situation is long understood and many efforts to reconcile product engineering and software engineering have been undertaken without success. Reconciling these fields is much more challenging than it seems and requires rethinking fundamental underlying assumptions and concepts.

Tools, techniques and environments in both SCM and PDM aim at supporting and controlling the design, development and evolution of a product, be it material or software. But the fact of whether the targeted product is material or software is not a mundane difference, because software has a unique property: its source code is a model of the targeted software product and its executable code is the product. For that reason SCM and Software engineering tools are managing models, source code and executables and emphasize building, testing, fast changes through concurrent engineering; while PDM emphasizes "only" modeling.

This paper is written by software engineers for software engineers. Our goal is not a detailed comparison between SCM and PDM but (1) to identify the major reasons why these two fields are incompatible, (2) to find out the salient features of each one; and (3) to find a way to reconcile these fields, taking the best of each and improving the state of the art in both fields.

2. SCM / PDM MAIN DIFFERENCES.

SCM and PDM have many similarities and differences. For an extensive study, refer to [1][4][13]. This paper focuses only on the two most salient differences: model support and evolution support.

2.1 Model support

In PDM three classes of entities can be identified:

- business model, made of **business items**,
- detailed model, and other files called **data items,**
- targeted real physical objects.

For example, the business model of a bicycle contains a *frame* business item two *wheel* business items and their relationships. The detailed model of a wheel is a data item, i.e. a file produced by a CAD system. A real bicycle is the real object.

Clearly, PDM emphasizes product modeling. Models and metamodels are well defined and normalized. Although each PDM system takes some liberties with the standards, the following can be identified:

- EXPRESS formalism. It is the Meta-Metamodel for business modeling [19].
- Application Protocol (APxyz Metamodels). They specialize EXPRESS for a given business domain (AP209: Composite & Metallic, AP214: automobile etc) [3][20].
- The model (e.g. a Peugeot 307 model (sic), conforms to AP214, itself EXPRESS conforming).

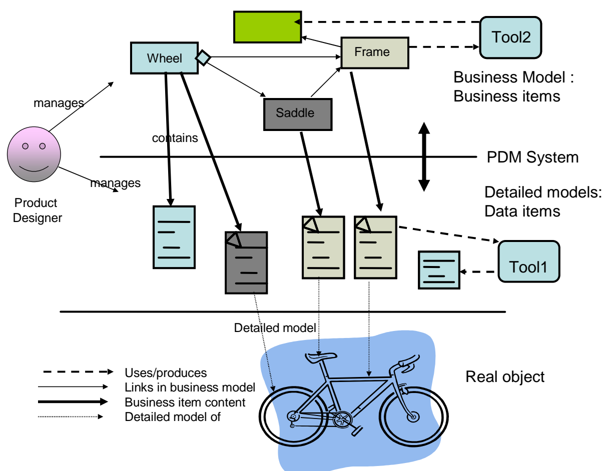

Figure 1. PDM basic architecture

A major goal in PDM is data exchange. For that reason it emphasizes (1) the metamodel normalization (the APxxx) and (2) the data structure vision. This explains why EXPRESS is mostly a data description language, with good provision for (static) consistency constraints, but without behavior (no methods). Most often the semantics are found in the associated data items (detailed models).

The engineering entities, including the detailed models, are called *Data Items.* They are the files produced and used by the different tools, like CAD tools. The content of these files is unknown from the PDM perspective. A business item can "contain" one or more data items.

The real object is in no way managed by the PDM system, even if the evolution of CAM systems is to provide better support for manufacturing (production line design, robot program generation and so on).

The business model, with respect to the detailed models and engineering artifacts, plays the role of a superstructure which controls the global product structure and its links with the data items, all along its life cycle, including its commercial life.

In software engineering, the data item layer contains the detailed models (e.g. UML), source code, files and directories. But since the target software application is also an electronic artifact, produced almost for free from the source code, the real object also pertains to the data item layer. For a software engineer, the source code and the target application are almost the same thing. The data item and real object layers are collapsed. The business model layer also exists in SCM, in the form of a system model. Unfortunately, despite different prototypes [21] and even trials from industrial systems (e.g. Clear Case), the versioning of such a system model proved to be too problematic, both from technical and usability points of view. Today no SCM systems propose a system model.

In practice, in SCM, and software engineering in general, a single layer exists containing detailed models, source code, different files (documents, scripts, etc.) and the executables.

2.2 Evolution control

In SCM and PDM, a core issue is the ability to control complex artifacts evolution; however, the very concept of version is understood differently.

We define that B is a version of A if, for some concern, B looks identical to A. It means that A and B share some characteristics such that, if only these shared characteristics are of concern, A and B are equivalent. For example, if you require the functions defined in an interface, two classes implementing that interface can be seen as equivalent, and therefore versions of each other.

This definition, to be of practical use, requires knowing: (1) what are these shared characteristics and (2) what are the characteristics of concern. These two conditions are usually not satisfied.

In practice, the bottom line is to recognize a difference between logical and historical versioning [18].

Two artifacts are **logical versions** (also called variants) if they are intended roughly for the same purpose (they fill the same functionality), but in different context (e.g. Windows and Unix versions). Logical versions exist and are maintained in parallel, none is intrinsically "better" than the other; using one or the other depends only on the context.

Two artifacts are **historical versions** (also called revisions) if one is a modification of the other, the new one is assumed to fill the same purpose and to be "better" than the previous one, in all contexts. Historical versioning leads to the definition of a "predecessor" relationship, therefore a line of revisions, sharing the same purpose, which altogether can be seen as a single *logical artifact;* i.e. the same artifact along the time dimension.

Historical versioning is supported in very much the same way in both camps. It is a mechanism that fits the "improvement" intentions like "I fixed, I rewrote, I completed"; therefore creating a revision does not require recording additional information: the "improvement" intention is implicit and the purpose is the same for all revisions.

Versioning is only a mechanism (a means) for the support of the goal: evolution control. But evolution has different dimensions that should be defined, modeled and supported independently. These models include:

- **Cooperation**. Define how concurrent engineering should be handled.
- **Composition**.

- **Variability**. Indicate which combinations of logical artifacts constitute a valid *assembly* (configuration).
- **Compatibility**. Provided an *assembly*, computes which revisions can fit together.

- **Evolution**. Defines how changes are related to versioning.
- **Technology.** Defines the versioning mechanism: concept and storage.

Composition models emphasize the fact that a complex product (a mountain bike) is a composite item defined by structure constraints (2 wheels, a frame…) and constraints on the elements of that structure (26” wheels, disk brake …). With respect to our definition of logical versioning, the constraints are the “characteristics of concern”; all elements sharing these characteristics are equivalent from the point of view of the composite item and constitute an equivalence class (all 26” wheels are valid candidates for a mountain bike). Therefore, creating a complex product requires selecting a candidate in each equivalence class.

Variability and compatibility are traditionally distinguished because the former defines the intention and purpose; while the later is related to technical compatibility between revisions (i.e. version 2002 of wheel XY is not compatible with disk brakes ZT revision …).

Evolution control is supported very differently in SCM and PDM; let us summarize their main characteristics.

2.2.1 *PDM*

Cooperation model. In PDM, business items (the elements of the business model) are stored in a database and data items in a file system. Changing a business item means changing a record in a database, using the usual transaction strategy. It means that only one person can change a business item at a time.

Data items are managed in a conservative way. To be changed, a data item is copied into a specific location and locked during the change. In both cases, there is no real concurrent engineering, and therefore no explicit cooperation model [16][17].

Variability, in PDM, is based on the semantics attached to the specific relationships *substitute*, *alternate*, and *optional*. In STEP, *alternate* parts are interchangeable in all occurrences whereas *substitutes* are interchangeable only in particular contexts. The *option* concept allows describing generic structures using quantifications (a bike has 2 wheels). The product is defined with all its possible options, then, depending on the option selection, the number of instances can be changed (a child bike has 4 wheels), components can be added or removed.

Compatibility is handled using the *effectivity* concept. An effective relationship is established from a revision (origin) to a set of destination revisions, expressing the fact that the relationship holds between the origin revision and any of the destination revisions. Effectivity expresses a range of compatibility; this range can be expressed by serial number range, i.e., a list of revision sequences like (1-3; 6-12) meaning revision 1, 2, 3, and 6 to 12 are compatible. It can be a time period (01/01/2001-15/08/2003) meaning any revision created between 1st January 2001 and August 15, 2003. EXPRESS, the basic data model, includes effectivity as a primitive concept, but does not include any logical variability control. In practice each PDM system implements a sub-set of the logical variability concepts.

***Evolution model* is undefined.** It is up to the developer to decide how his/her change will be mapped to the versioning concepts: creating *alternate* or *substitute* relationship, creating revisions, extending the effectivity range ….

The versioning ***technology*** used in product engineering is rather simple: business objects follow a single line of “revisions”; and each revision can be associated with a single line of “version” of the associated data item, but only the most recent version is available.

2.2.2 *SCM*

Cooperation. SCM relies on the concept of workspace. A workspace is an isolated part of a file system in which all the files needed for a job are copied. Many workspaces can work concurrently on copies of the “same” files. Multiple changes on the same file are reconciled using “mergers”. Large scale concurrent engineering is common in SE. Nevertheless, these systems do not provide any formal cooperation model. In the optimistic strategy, it is simply expected that merges will be possible; in the pessimistic one, files need to be locked before to be changed.

Variability. This model does not exist explicitly in SCM. The properties and intentions of logical entity (variant) is not made explicit. SCM systems rely on the concept of workspace and baseline to define which combinations of variants make sense, in a pragmatic way. Many Architecture Description Languages (ADLs) and system models define a concept of equivalence, but ADLs and system models are not used in commercial SCM system. Most component models manage a single concern (implement an interface). Only research prototypes have addressed the issues of equivalence and semantic based selection [5][21] and these prototypes did not transition to successful industrial practice[15]. Product line environment are based on the concept of feature, which is a selection mechanism [14].

Compatibility. This model does not exist either. A workspace contains at most one revision of a given logical entity. A relationship, in a workspace, is defined between two revisions, and is assumed to hold between these two revisions.

Evolution model is undefined. It is up to the developer to decide to create variants, revisions, or new items, when convenient.

Technology. Software engineering uses a tree of variants (a branch) and revisions (a point on a branch). Branches are traditionally used for expressing logical (alternate or substitute) and cooperative versioning (a branch per workspace). Revisions support historical versioning. The basic versioning systems (CVS, Subversion …) provide a storage facility, but the intention and purpose of versions, and their relationships are not explicit. In high-end SCM systems (ClearCase, CM/Synergy, …) it is sometimes possible to associate attributes to item types and to define hyperlinks. They do not, however, provide facilities for versioning those hyperlinks and attributes.

3. CONCURRENT ENGINEERING AND EVOLUTION MODELS

This short analysis of SCM and PDM shows the following:

- The strong point of PDM is the successful use of a business model which relates the product structure with data objects.
- The strong point of SCM is its ability to support evolution and concurrent engineering in isolated workspaces.

- In both camps, the versioning and evolution capabilities cover only a tiny fraction of the possible evolution spectrum; and each camp has selected different strategies.

The challenge addressed in our work is to find a way to support both strong points: combining the use of a business model related to engineering items, but still providing support for concurrent engineering.

This issue has been addressed in SCM when attempting to support a system model (which is equivalent to the PDM business model). These attempts have produced prototypes and even industrial trials but turned out to be practically infeasible [15]. This failure can be explained by two factors: (1) the system model has to be maintained manually, and is therefore is expensive and unreliable; (2) the versioning of a system model proved to be problematic, as explained later. The cost of maintaining and versioning the system model largely outweighed the expected benefit.

Our claim is that the difficulty comes mostly from the fact that each domain has made different underlying assumptions, and for that reason uses different evolution models. The implicit hypotheses common to both fields are the following:

1. A "work" can last for a long time, and may require the availability of many different items.
2. A work usually changes only a (tiny) subset of the required items.
3. An item can be a significant undertaking, with many relationships with other items.

The issue is to satisfy these common hypothesis and the two specific characteristics: items are both business and data items, and the system provides strong evolution and concurrent engineering support.

Evolution and concurrent engineering means that the same item can be changed "simultaneously" by different persons. Provided the long duration of tasks and the many items required (hypothesis 1), a "work" has to be performed in an isolated place containing copies of all the needed items: a workspace. Provided that a work only changes a small number of items (hypothesis 2) that are highly interrelated (hypothesis 3), concurrent engineering has as consequence independent evolution and versioning of individual items.

Combining the common hypotheses and the two specific properties, logically leads to the following requirements:

Requirement 1: **Work is performed in workspaces**.
Requirement 2: **Items are versioned individually**.

Unfortunately, requirement 2 is difficult to satisfy if items are parts of a rich model. Indeed, requirement 2, in product engineering, would require:

- Versioning and merging logical and engineering items individually,
- Managing the evolution of complex models made of independently evolving items.

Although merging engineering items has been a topic of intense literature [6][8][10], there is no generic merging algorithm. Nevertheless, most tool vendors propose features for comparing / merging specific engineering items. The first point can be solved.

It is the second point which is challenging. Many propositions for model merging exist in the literature based on UML [11] [12] or specific metamodels [2][7][9], but in all cases, the whole model is versioned, not the individual items. It is "easy" to retrieve old models, to store modified models, but in general, individual items and shared only in read-only libraries. It is not possible to build a new model assembling pre-existing items that can be tailored / changed by the different users of these items.

In this paper, we claim that it is because evolution models are missing (historical compatibility, logical variability, evolution and concurrent engineering) that it has not been possible, so far, to support both the requirement set by product engineering and software engineering systems. Before discussing these evolution models, we present shortly our data model and our basic workspace strategy.

3.1 CADSE architecture

In our work, we borrow from PDM the idea that a product can be seen at three levels: two models (business and data items) which coexist and complement each other, and the real object, which is not managed by the system (see Figure 1). In our case, since the real object can also be managed, the bottom layer contains the detailed models, the engineering artifact and the real object; the top layer contains the system model, which corresponds to the PDM business model.

In CADSE, the system model, as in "traditional" SCM systems, is made of items and relationship meant to represent the structure and architecture of the software system in terms of files, systems and subsystems. Unlike "traditional" SCM systems, in CADSE, the system model describes the software in a language (metamodel) specific to the application domain. The system model language is a *domain model* which describes either business domain concepts (e.g. an account in a bank), technical domain concepts (e.g. a component, a sub-system), or something in between (e.g. a driver).

The bottom layer is the development platform, it contains a wide range of engineering artifacts (from detailed models to executables), of various granularity and size (from a fraction of file to gigabytes), a large number of tools (compilers, editors), IDEs and the operating system.

In our system, a workspace contains not only the engineering artifacts, as usual, but also the system model. The software engineer can act upon the artifacts or the system model, and an automatic and dynamic mapping translates actions on items into actions performed on engineering artifacts, and vice versa. The solution we propose (CADSE) is realized following the architecture summarized in Figure 2.

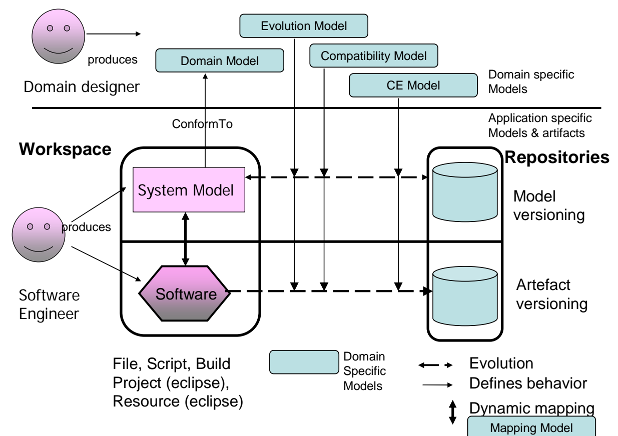

Figure 2. A CADSE architecture

In this paper we concentrate on the issues arising from evolution control and concurrent engineering. First, we have to give some information on how we defined the system model, and on the different evolution models we have identified.

3.2 System model and Domain model

In CADSE, the system model has to conform to a domain model (the system metamodel). The domain model describes the business domain concepts, architecture style, or any technical or conceptual structure and conventions that a system model must be compliant with in that domain.

The language in which a domain model is expressed is rather simple; it describes only structural concerns. A domain model is made of Item types and Link types that have attributes.

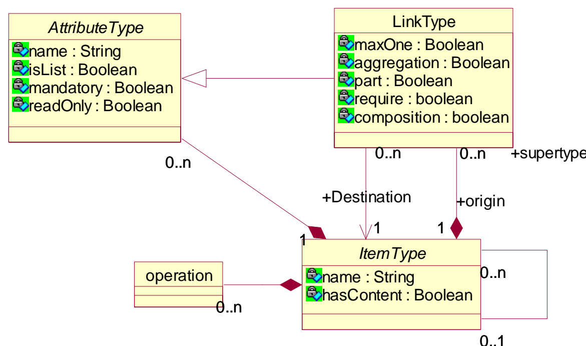

Figure 3. The Domain Metamodel.

Attributes can be Strings, Date, Integer, Boolean and Enumerate; they can be monovalued or multivalued. The special attribute *content* can be mapped to engineering items like files, directories, resources, projects, database records, entities managed by specific tools, like CAD or SCM systems, and so on. For illustration purposes, let us suppose that the content is a File. A link is an attribute whose value is a reference toward another (logical) item.

A system model describes a particular application in the domain, and therefore conforms to the domain model; it is made of items and links which are instances of the domain model item types and link types. Figure 4 is an example of a simple technical domain model which expresses that a software application must be made of Components related to bugReports, and handled by an Engineer.

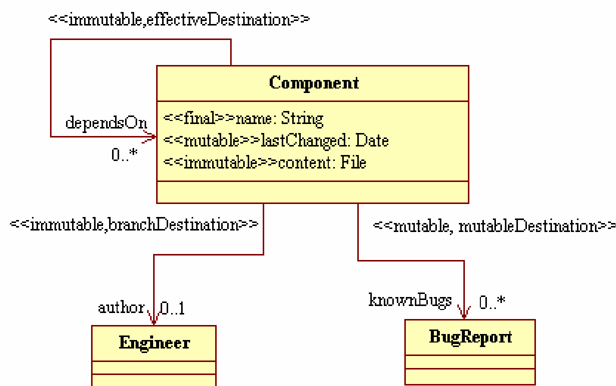

Figure 4. A domain model example.

The versioning technology used is similar to product engineering: an item has an immutable *identifier*, say X, and is versioned as a numbered sequence of revisions. An item *reference* is its identifier followed by the revision number: X.1, X.2 …

Following the traditional software engineering strategy, we restrict a workspace to contain at most one revision of a given item. We define the operation *import (ref)* that brings from repository into a workspace a given item in a given revision; and *commit (id)* that sends back the item from workspace to the repository.

3.3 Evolution model

Traditional SCM systems only manage files with a unique strategy: any change will produce a new revision. A revision is a snapshot of a file or a set of files at a given point in time and is therefore immutable. But if items are made of different attributes, links and *content*, an item revision can be seen as something created at a given point in time, but that still can evolve. The creation and evolution strategy for an item is defined by the evolution model.

Suppose for example that a given *component*, X.n, is imported (checked-out) in a workspace, then changed and committed back (check-in) to the repository. What should be the versioning strategy?

3.3.1 Attribute changes

We would like, for example, to be able to express that *name* must be set at item creation, and never changed; *lastChange* should instead be set to the current date, and *content*, containing the actual source code, should be immutable.

The evolution model consists of associating to each attribute and link the following policies:

- ***Mutable***. The attribute can be changed in the database on any existing revision (e.g. lastChange)
- ***Immutable***. If the attribute changes, a new item revision must be created (e.g. content)
- ***Final***. The attribute cannot be changed (e.g. name)

Note that usual SCM systems only handle contents (files) with unique and mandatory *Immutable* policy.

In our system, the behavior depends on which attributes of X is changed and on its evolution policy. For instance, in our example, the resulting behavior is ignoring the change of *name*, updating *lastChange*, and creating revision X.n+1 if *content* has changed. Each attribute may have a different evolution strategy. See Figure 5 for the state transition diagram.

3.3.2 Link changes

A link is both an attribute (in its origin item) and an entity by itself whose value is a destination item plus some management information like effectivity and state. Changing a link means either changing the attribute (AC in Figure 5), or changing the destination item (DC in Figure 5). Changing the link attribute means adding/removing a destination if the attribute is a list, or linking to another item (AC for Attribute Changed).

The link destination depends on the type of link. Links can be *Normal*, *Effective* or *Branch*. *Normal* means that the destination is a single revision; *Effective* expresses a range of revisions like (1-3; 6-12) meaning revision, 1, 2, 3, and between 6 and 12; *Branch* means all revisions of the item. For example, if a link currently has destination Y.12, the link attribute is Y and the ***destination*** is either:

- Normal : the destination revision number (e.g. 12),
- Effective : a range of revision (e.g. (2-7; 12)),
- Branch : nothing.

Therefore, if the destination changes its revision number (e.g. from Y.12 to Y.13), the link attribute is unchanged (still Y), but the link destination will be changed (from 12 to 13), its effectivity extended (from (2-7; 12) to (2-7; 12-13)), or nothing if the link is *branch*.

Link attribute change. The attribute characteristics apply to link attributes: *knownBugs* should be *mutable* (we can add and remove a *bugReport*) and *dependsOn* should be *immutable* (if adding/removing a dependency, a new revision of the origin item should be created).

Link destination change. Take the example of the *dependsOn* link. If X.n *dependsOn* Y.m and Y is changed, what should be the behavior of the system? It depends on the kind of change performed on Y and on the intended semantics of link *dependsOn*. If the change on Y is such that it requires changing an *immutable* attribute in X (a method signature change in Y for example), new revisions Y.m+1 and X.n+1 should be created. In other cases, Y may have changed one of its *immutable* attributes, therefore producing Y.m+1, but X.n still works fine with Y.m+1 (changing the implementation of a method for example). We would like to say that X.n is "compatible" with Y.m ***and*** Y.m+1.

In some cases, like for *author*, which is an item, we would like to say that we do not care about the *author* current revision, as long as it is the same author.

The evolution model allows us to express all of these cases; each link may have one of the following characteristics:

- ***MutableDestination***. Changing the revision number of the destination item does not have an effect on the origin item, but the link now targets the new revision.
- ***ImmutableDestination***. If the destination item changes its revision number, a new revision of the origin item must be created with the link targeting the new revision.
- ***EffectiveDestination***. The origin item will be compatible with both the previous and new destination revision (the link targets both revisions).
- ***FinalDestination***. Destination is not allowed to change.
- ***BranchDestination***. The target of the link is the item Id, not a specific revision. Creating a new revision of the destination item does not even change the link.

If the destination changes of its revision number (i.e. it becomes *new-revision*), the link destination state becomes *changed* if its characteristic is *mutableDestination*; *new_revision* if its characteristic is *immutableDestination*. *Branch* links are always *unchanged*.

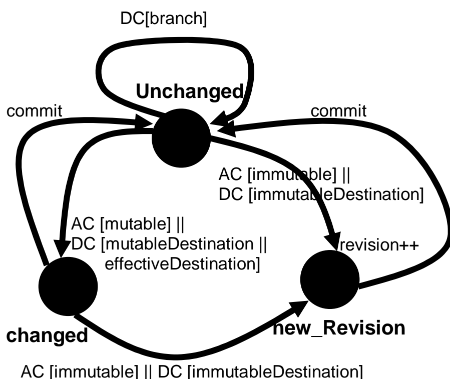

Figure 5. Attributes and links State transition diagram

The state of a link attribute is the combination of the link attribute and the link destination states, *new_revision* having priority over *changed*, and *changed* having priority over *unchanged*.

The state of an item is *new*_revision if at least one of its attributes or link is *new_revision*, *changed* if at least one attribute or link is in state *changed*; *unchanged* otherwise.

3.3.3 *Item state propagation*

If the state of item Y becomes *new_revision*, its revision number is immediately incremented, without waiting for the next commit. But changing Y revision number may have immediate side effects on items X that have links toward item Y; changing X state to *changed* or *new_revision*, depending on the link attribute and value characteristics; incrementing X revision number if becoming *new_revision* and recursively.

Taking the Figure 4 domain model, at a given point in time the following situation may occur (Figure 6).

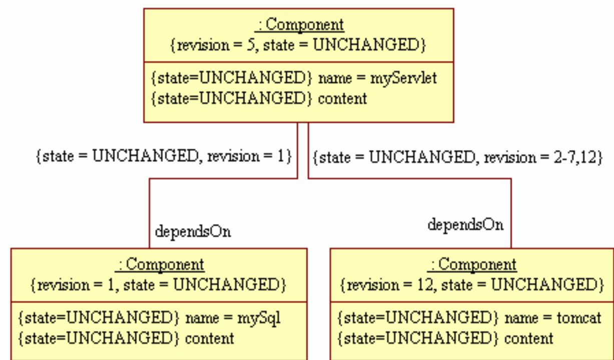

Figure 6. A system model at a given point in time

If a file pertaining to the Tomcat *content* attribute is edited or if a new version of Tomcat is imported, changing that file changes immediately the Tomcat *content* attributes which state becomes, *new_revision* since the *content* attribute has characteristic *immutable* for *components.* The Tomcat attribute *content* being now *new_revision*, Tomcat itself becomes *new_revision* and its revision becomes 13. This state change propagates to the *dependsOn* link, which updates the effectivity range and changes its state to *changed* since *dependsOn* is declared *effective*. In turn this change propagates to the *myServlet dependsOn* attribute and finally to *myServlet* itself which becomes *changed*. See Figure 7.

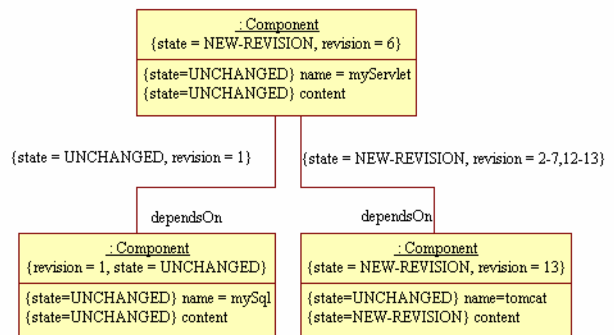

Figure 7. System model after a file change in Tomcat

This computation is performed completely locally only based on the state and characteristics of attributes and links pertaining to the current workspace, and the result is displayed in real time. In

CADSE, the complete state of the system is always available and the consequences, sometimes far reaching, of any action are immediately visible.

Therefore, depending on the evolution model, complex state changes may occur in a workspace. Side effects are computed without any need for being connected to the repository, and the system permanently provides the current state of the local workspace items. An evolution model is defined as characteristics attached to the attributes and links ***definitions***, therefore defined on the product metamodel. It means that it is the duty of the domain expert to define both the product metamodel and the evolution model; in this way defining the behavior of the system when the product evolves. The important consequence is that the designers and developers of a product in the domain automatically apply the (right) evolution policy, and do not have to know the detail and the intention of the domain designer.

3.4 Concurrent engineering model

Now suppose that X is changed in two concurrent workspaces A and B. A commits first, what occurs when B tries to commit? The usual SCM strategy is to prohibit B to commit X and to ask B to merge X first. But this is only one of the many possible strategies; it is the usual one because, implicitly, X has single attribute *content* with characteristic *immutable*. We could envision that some kinds of merges are performed even in the data base. For example, the *knownBugs* attribute, being a list, could be merged between A and B; the system can make a perfect merge and add bugs found in A and B. The ***reconcile*** characteristics, can be set on list attributes to express this property.

For *mutable* attributes, changes performed on X in B will overwrite the changes made on X in A. For *immutable* attributes, a new X revision will be created, superseding the value of X established in A. The changes made by A on X will either be overwritten or forgotten.

In most cases overwriting or forgetting an attribute value is unwanted; characteristic ***conflict*** detect this case and the system behaves as usual SCM systems: the second commit is rejected if two conflicting changes are performed on the same attribute; a merge has to be performed in B first.

When committing an item X, having a link toward Y.m+1, the system records the destinations item Id Y and the revision number m+1 (or revision range for effectivity links). Therefore, Y.m+1 must be present in the repository before X commits recursively. Committing an item may imply commits to a number of other items. The system first computes all the items to commit and then it computes the errors with respect to the repository (*conflict* characteristics); then, in a transaction, the system commits all the items.

This concurrent engineering model is very simple; it is sufficient to sufficient to support the classic policies, but not to describe advanced CE policies like in Celine [22].

3.5 Compatibility model.

Most of the issues addressed by the compatibility model are the consequences of the fact that a workspace contains at most one revision of a given item. This constraint simply matches the usual software constraints: a single version of a class is allowed in an application.

Therefore, importing (from the repository into the workspace) an item Y.m may raise a number of consistency issues.

Suppose that Y.p is already in the workspace. We have a number of different cases.

- p=m and Y.p is unchanged: the operation is null.
- Y.p has state changed or new_revision: Y.p may be overwritten (Discart) or merged (Merge).
- Y.p is unchan*ged*: Y can be reverted (Revert, m < p) or updated (Update, m > p).

If Y.p exists and Y.m is imported, all existing links L from X.n to Y.p are changed from Y.p to Y.m.

- If L is *mut*ableDestination, link L is set to Changed, and therefore X.n state is also set to Changed.
- If L is imm*utableDestination*, link L becomes, *newRevision*, and X.n becomes X.n+1 *new_revision* and recursively.

Therefore importing an item which is not the expected one may propagate to many other items. Before importing item Y.m, the system computes and displays all the sides effects {(Discart, Merge, Revert, Update), {new_revisions}, {changed}} and asks if the operation should be performed anyway (with the sides effects) or cancelled.

Now suppose that Y.m is imported and has link L toward Z.i. If Z is not currently in the workspace, we have to decide whether Z has to be imported too. If link L has characteristics ***require***, Z will be imported too, and recursively if Z has also *require* links. If L is not *require*, the system maintains a “non resolved” link, i.e., a link that references an item not present, along with its revision number, allowing the same kind of consistency check even on missing items.

But importing Z.i may have the same side effect as discussed for Y.m. and the system displays for Z.i the associated list of side effects. Therefore, depending on the side effects, the user may decide to import Z.i or not. Note that, if Y.m is imported but not Z.i, Y.m will be marked *changed* or *new_revision*, depending on link L characteristic.

Because of the different *require* link propagations, importing an item may end in importing a list of items, each one with its own associated list of side effects. The system first computes all the propagations and sides effects, as if all required items will be imported, and displays a table, in which each line corresponds to one item to be imported, and its list of side effect. The color of each line indicates its importance (Discart, Merge, Revert, Update) deselecting a line, means “do not import”. Deselecting a line “immediately” recomputes the side effects to the other items to be imported.

Once all decisions have been made, the system performs all the selected imports, and updates the workspace accordingly, in a single transaction. It is important to emphasize that all this computing is performed statically, observing the state of the repository and the state of the workspace and does not require any data transfer; this explains why it can be performed fairly quickly in interactive mode.

The involved side effect computation turned out to be rather complex, especially when adding recursive composite items not discussed here, and a mixture of selected / deselected items. The experience we already have shows that, without the help of the system, it is virtually impossible to forecast the version problems and to know the right version to use in all circumstance.

Conversely, the messages and help provided by the system makes building complex composite systems fast, safe and easy.

We believe that, fundamentally, it is this compatibility model and associated mechanism that solve the model versioning issue. This compatibility model proved to be particularly difficult to design; it requires expertise in different fields, involves complex mechanisms, and requires an advanced architecture to be implemented. These difficulties probably explains why this feature was not available until now and why, since the model versioning issue is the root of the incompatibility between model based approaches and concurrent engineering, the Software engineering and product engineering approaches have been incompatible domains.

4. IMPLEMENTATION.

We do not believe it is feasible to design and build a system supporting the features, concepts and methods spanning a large range of engineering activities, including software engineering and product engineering. It explains probably why it was not possible, so far, to reconcile SCM and PDM systems. Our approach, instead, has been to analyze the differences in the concepts and techniques used in Computer Aided Engineering environments (CAE), and to identify how these differences could be formalized. We have identified a number of dimensions on which different CAE diverge. For each one of these dimensions, we have defined a model, allowing us to express, through a model (conforming to the metamodel), the point or the area, in this dimension, used by a given CAE.

Our work led us to identify the following models:

- Domain model
- Variability Model
- Compatibility Model
- Evolution Model
- Concurrent engineering model

Other models have been identified and are supported as well, but are not the topic of this paper: correspondence, behavior, composite and build models. Providing a model for each one of these dimensions is a way to represent formally and precisely the engineering environment used in an engineering domain: a CADSE (Computer Aided Domain Specific Environment).

From a practical point of view, our work faces two challenges. The first one, which is the topic of this paper, is the capability to support engineering in a very wide range of domains, including software and product engineering. The second challenge is to reuse tools and habits. Therefore our system should:

- Smoothly integrate existing tools and systems, respecting user habits and methods.
- Be highly adaptable and extensible, to fit a wide range of uses.

4.1 Architecture and reuse

For the integration and reuse of tools and methods, we have used the Mélusine approach [14] which links abstract concepts to real tools. In CADSE, the workspace and repository domains (see Figure 8) are described in a rather abstract way and connected to current software engineering tools. In practice, a workspace is synchronized with Eclipse (NetBean support is forecasted); for model versioning the repository is synchronized with different databases (mySQL in the current version), and for artifact versioning the repository uses current versioning systems (Zip, CVS or Subversion in our implementation). The approach allows us to easily change and integrate commercial tools, including PDM ones.

The workspace domain is driven by four (meta) models (domain, variability, evolution and compatibility). The correspondence model maintains the consistency between the item and its associated content in the workspace. For example, an item can be associated with an Eclipse project; creating a dependency between two Eclipse projects may automatically create a dependency link between the corresponding items. Similarly, creating a relationship between two items may create a project dependency, create or change files, update the classpath and so on. Workspace is itself an Eclipse plugin, which proposes new editors (for items, attributes, links, composites items etc.), strongly coupled with the usual Eclipse concepts and tools. Workspace gives Eclipse a high-level and hierarchical view of projects, resources, files and plug-ins. Workspace subsumes the actual Eclipse concept of workspace, and can be seen as the high level workspace view of Eclipse.

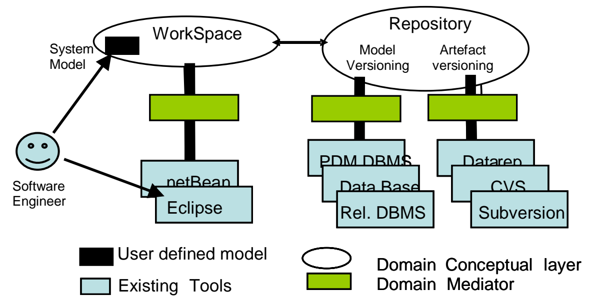

Figure 8. CADSE implementation architecture

In Mélusine, since each domain is autonomous, the workspace domain can be used alone. Indeed, workspace alone provides a number of very useful facilities, and has been used as-is inside and outside as our team for a long time.

4.2 CADSEg: a multi model approach

Even using the Mélusine tools and approach, building a CADSE manually is a significant undertaking that is too often not economically feasible. It is only in the product family context that Domain Specific Environments (i.e. CADSEs) have been developed; the hypothesis, in product families, is that a large number of products are to be developed in the same family, making the CADSE development cost effective. Despite years of existence and a huge commercial motivation, the fact that real product family environments are so few is a proof that a manual CADSE development is too expensive in general.

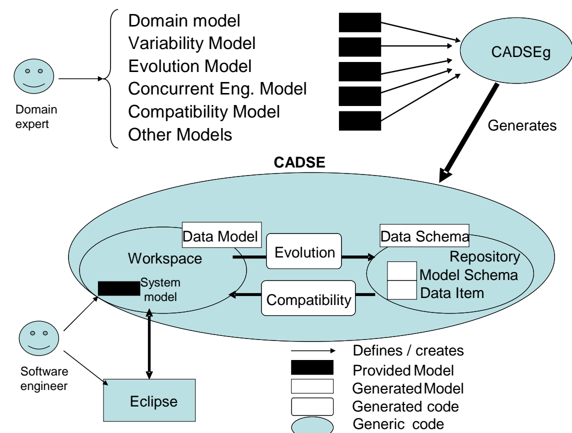

Figure 9. CADSE models and CADSE generation

Since the different models defining the targeted CADSE are available, the “simple” idea is to take these models as input and

generate the associated CADSE as output. This is the goal of the CADSE generator tool we have developed.

CADSEg being a software environment dedicated to the production of a specific class of software applications (CADSE production) can be seen as a CADSE itself. CADSEg is built using itself (bootstrapped); and is one of our most complex and well used CADSE.

5. EXPERIENCE

The authors are software engineering and SCM experts, not PDM experts. Nevertheless, our work on the topic started more than 10 years ago [13], and motivated much of our subsequent research. We first had to design and implement the Mélusine approach and infrastructure (which currently supports CADSEg and all CADSEs). Mélusine and a number of domains, including product, content and repository, have been validated through the development of industrial applications, some of them in operational use. Then, we have designed and implemented the workspace domain which supports all our software developments since February 2006. Teamwork, the domain that links workspace and repository domains, and controls the evolution models, has been designed and developed simultaneously but was ready much later.

The experience of using workspace alone is compelling, and fully supports the claims from the product engineering community. The workspace view (system model) is really a high level and conceptual view, in which the structure, properties and nesting of items can be easily seen and managed. The strong link with the engineering items allowed us to perform most of the engineering activities on the workspace view, instead of using the package explorer one. It is interesting to notice that users tend to use the workspace view more and more. This pushed us to rewrite some mechanisms like rebuilding, in a generic and high level way. Altogether, the CADSE project has developed 120 000 Physical Lines Of Code (PLOC); the largest CADSE developed (CADSEg itself) has 92 concepts in its metamodel, Melusine CADSE has 23 concepts and a few hundred logical items; efficiency has not been an issue so far.

Intrinsically, workspace is a concurrent engineering view very much in line with the SCM philosophy. We used Subversion for versioning the Eclipse data items (projects and files) managed by workspace. During the time workspace was used alone, without surprise, we experienced the fact that real concurrent engineering over a rich model cannot be managed using only the classic (file) versioning tools. Many difficulties arose that forced us to adopt the classic “strict baseline” strategy which consists of having a single branch by item, and in using only the head of all branches (the baseline).

The strict baseline approach is the SCM view of the “full model versioning” approach used by product engineering tools, and by most model driven Software engineering approaches. Clearly, the different evolution models supported by Teamwork are required if items evolve and are versioned individually, and if SCM and PDM are to be reconciled.

Teamwork development and experimentation showed us how complex and critical, historical compatibility control is. We had to rework the concepts, implementation and interfaces a number of times before reaching a satisfactory result. It is no surprise that, without such a service, PDM and SCM cannot interoperate.

A major property of the system is that most, if not all, the versioning, compatibility and consistency policies are addressed by a CADSE designer which is often done once for all in a given company. Then, product or software designers and developers do not care of versioning, concurrent engineering or compatibility issues. The system “knows” the semantic and intention attached to each attribute and link and automatically checks the conditions and automatically updates the versioning information. For example, the compatibility mechanism dynamically and automatically updates the compatibility information when developers are working. The decisions to create a new revision or not, to update the links, to extend an effectivity range, or to compute the side effect of an action are directly derived from the models, and require absolutely no action from the developers. This property proved to have major benefits:

- The policy is consistently enforced for all developers working on the same software/product
- The developers do not need to be aware of the policies and their associated controls.
- Developers do not have anything to do for the consistency, compatibility and concurrency constraints to be satisfied.

From the developer’s perspective, consistency, compatibility and concurrency are features available “for free” which proved to be key for developer acceptance. But the real fundamental consequence is that the model(s) are by construction, always in sync with the reality. This is true for all the models: the system models, which represents the software/product structure and elements thank to the correspondence model, as well as the evolution models thank to the mechanisms presented in this paper. Indeed, the models *are* the reality.

In a CADSE, building a complex system by composition of pre existing components is highly assisted. It is driven by the domain metamodel, by the database of existing component, and the composition consistency is enforced by the compatibility model. A CADSE provides “naturally” the support expected from automatic selection mechanism [15]. The main reason why automatic selection failed to be used in the industry so far is that it relies on a software model that has to be maintained manually all along the product life cycle; and maintaining such a model is too expensive and error prone. In a CADSE, model maintenance is granted and automatic by construction.

6. CONCLUSION

The homogeneous and consistent management of the complete life cycle of products containing software, electronic, electrical and mechanical components is a clear industrial need. Unclear is the reason why, so far, despite a decade of efforts software engineering and product engineering environments, tools and methods are not integrated and cannot even interoperate [1].

The paper shows that SCM and PDM goals are very similar but, PDM emphasizes modeling and SCM emphasizes building, evolution and concurrent engineering. We believe this difference of focus explains why the solutions to the same problems are so different.

Reconciling PDM and SCM therefore requires supporting high level modeling and concurrent engineering. The contributions of this paper are:

- Identifying why high-level modeling and concurrent engineering are conflicting features.
- Finding that the root issue lies in the evolution control of models made of items that evolve individually.
- Defining the evolution models capable to solve the model evolution control issue.
- Presenting CADSEs, that implements these feature and support these models.
- Evaluating the solution we have proposed.

Provided the extremely wide spectrum of industrial domains covered by product engineering and software engineering, it is unrealistic to believe that a single tool could cover the functionalities, concepts and features required in all these domains. Our approach instead, has been to find the different dimensions that distinguish a domain from another, and to propose a language (a metamodel) in which it is possible to formally define where a domain fits in each dimension. The set of such models constitutes a formal specification of the associated domain. Our approach is to generate a Computer Aided Domain Specific engineering Environment (CADSE) from this specification. The experience with CADSE, so far, shows that, not only is the approach feasible, but that it is possible to build practical and scalable solutions.

In software engineering, most advanced features (ADL, selection, proof, validation …) require or produce high level models of the software. Unfortunately, in the software engineering field, we have been unable to (1) enforce the consistency between the model(s) and the "real" software and (2) manage the evolution and the concurrent engineering of such models. Having solved these two issues, we expect that our work will provide a way to resurrect these early attempts, providing CADSEs in which these advanced services are provided "for free".

CADSEg can be downloaded http://www-adele.imag.fr/cadse

References

[1] I. Crnkovic, U. Asklund, and A. Persson Dahlqvist. Implementing and integrating Product Data Management and Software Configuration Management. Artch House Publishers, 2003.

[2] A manifesto for model merging. Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, Mehrdad Sabetzadeh. Proceedings of the 2006 international workshop on Global integrated model management. Shanghai, China.

[3] Step. http://www.wikistep.org/index.php/Main_Page

[4] B. Westfechtel and R. Conradi. Software Configuration Management & Engineering Data Management: Differences and Similarities. In Proceedings of Software Configuration Management Workshop (SCM-8). Springer Verlag, 1998.

[5] R. Conradi et B. Westfechtel, Version Models for Software Configuration Management, ACM Computing Surveys, 1998.

[6] J.J. Hunt et W.F. Tichy. Extensible Language-Aware Merging, IEEE Int'l Conf. on Software Maintenance (ICSM), 2002.

[7] J. Rho et C. Wu, An Efficient Version Model of Software Diagrams, Asia-Pacific Software engineering Conf. (APSEC), IEEE, 1998.

[8] T. Mens, A state-of-the-art survey on software merging, IEEE Trans. On Software engineering, 28(5), 2002.

[9] D. Ohst, M Welle, et U. Kelter. Difference Tools for Analysis and Design Documents, IEEE Int'l Conf. on Software Maintenance (ICSM), 2003.

[10] H. Shen et C. Sun. Flexible Merging for Asynchronous Collaborative Systems, Confederated Int'l Conf. CoopIS/DOA/ODBASE, LNCS, 2002.

[11] P. Sriplakich, X. Blanc et M.-P. Gervais, Applying Model Fragment Copy-Restore to Build an Open and Distributed MDA Environment, MoDELS/UML, 2006.

[12] Z. Xing et S. Eleni, UMLDiff: an Algorithm for object-oriented design differencing, IEEE/ACM Conf. on Automated Software engineering (ASE), 2005.

[13] Jacky Estublier, Jean-Marie Favre and Philippe Morat. Toward an integration SCM / PDM. SCM8, Brussels, July 20th - 21st, 1998. In LNCS 439, Springer Verlag.

[14] Jacky Estublier and German Vega. Reuse and Variability in Large Software Applications. Published in Proc ESEC/FSE September 2005, Lisboa, Portugal

[15] J. Estublier, D. Leblang, A. Van Der Hoek, R. Conradi, G. Clemm, W. Tichy and D. Wiborg-Weber. Impact of Software engineering Research on the Practice of Software Configuration Management ACM Transactions on Software engineering and Methodology, Vol 14, N° 4, October 2005, Pages 1-48.

[16] A Comprehensive Configuration Management Solution, Metaphase Product Structure Manager and Advanced Product Configurator. Metaphase Technology, MW00206-A.

[17] EDL/Metaphase, Overview. Metaphase Technology, MW00200-A, 29 pages.

[18] J. Estublier and R. Casallas. Three Dimensional Versioning. In SCM-4 and SCM-5 Workshops. J. Estublier editor, September, 1995. Springer LNCS 1095.

[19] Product Data Representation and Exchange - Part 11: The EXPRESS Language Reference Manual, ISO-DIS-10303-11, ISO, Agust 1992, 138 pages.

[20] Recommended Practices for AP 203, PDES Inc., june 1995, 81 pages.

[21] J. Estublier and R. Casallas. The Adele Configuration Manager. In Configuration Management, ed. Walter Tichy. John Wiley and sons ltd, 1994.

[22] J. Estublier, S. Garcia. Concurrent Enginnering policies in Software Engineering. ASE, Tokyo, Japan. September 2006.