Defining and Supporting Concurrent Engineering policies in SCM

Jacky Estublier, Sergio García, Germán Vega
LSR-IMAG, 220 rue de la Chimie
BP53
38041 Grenoble Cedex 9, France
{jacky.estublier, sergio.garcia, german.vega}@imag.fr

Abstract. Software Configuration Management addresses roughly two areas, the first and older one concerns the storage of the entities produced during the software project; the second one concerns the control of the activities performed for the production / change of these entities. Work space support can be seen as subsuming most of the later dimension. Indeed, work space is the place where activities take place; controlling the activities, to a large extent, is work space control.This short paper presents our concurrent engineering experience in Dassault Systèmes, as well as our new approach in the modeling and support of concurrent engineering for large teams.

1. The problem.

The Dassault Systèmes Company develops a family of large systems, notably Catia, the world leader in computer aided design CAM. This system alone contains more than 4 millions LOCs, is developed simultaneously by 1000 engineers which produce a new release every 4 months. The last two numbers show that Dassault Systèmes experiences very high concurrent engineering constraints. Indeed, the time where a software engineer was "owning" the set of files needed for his job, for the duration of his job is far away. Take the Dassault Systèmes example:

In average, at any point in time, there exist about 800 workspaces each containing in average 2000 files. Obviously each file may be present in more than one workspace at a given point in time; our numbers show that a given file is contained in 50 to 100 different workspaces simultaneously. The consistency of data submitted to concurrent access is known for long by the data base community, which have established the transaction consistency criteria. In our case, consistency would mean that a file could be changed in a single workspace at a time. Unfortunately, the duration of a activity can be days and weeks: a file would be locked for too long and severe dead lock would occur.

In the Dassault Systèmes context, this hypothesis has been measured. At any point in time, in average, a file is changed simultaneously in 3 workspaces, with maximum around 30; but most files are not changed at all. In average, a few hundred files are changed in a workspace before it is committed. Thus, still in average, a workspace conflicts permanently with about 100 other workspaces. Not allowing simultaneous

changes for a file would reduce concurrent engineering almost to null, most of the 1000 engineers being forced to wait.

The direct consequence of this is that merges occur frequently, depending on the policy and the kind of software. The average for an application file is 2 merges a year, for a kernel file 0.4 a year. These numbers may seem low, but averages are meaningless because most files are not changed or merged at all. Conversely, files under work are subject to many changes and merges. We have records of more than 200 merges a year for the same file, which means about 1 merge per workday. Globally, about one thousand merges occur each day at Dassault Systèmes.

We think these numbers clearly show that concurrent change does exist on a very serious scale. Controlling this situation is a real critical issue. For us concurrent engineering control means:

- Merge control (what is to be merged, when...)
- Activity structuring and control .
- High level policy modeling and enforcement.

We will see shortly theses topics. For more info on activity structuring and control see [11][12][15][18].

1.1. Merge control

The above values apply to files whereas, in this work and at Dassault Systèmes, we deal with objects (files are atomic attributes in our object model). Our experience shows that concurrent changes to the same attribute of different objects (typically the source code) as well as changes to different attributes of the same object (like responsible, state, name, file name, protection etc.) are very common. Merging must address both cases. For example, restructuring, renaming and changing files are common but may be independent activities undertaken by different persons. Raising the granularity from file to object makes appear new kinds of concurrent changes, which may produce new kinds of merges (typically composition changes). It is our claim that objects concurrent change control subsumes traditional file control and provides homogeneous and elegant solutions to many difficulties that currently hamper concurrent software engineering.

Suppose we denote by A0, A1,... Ai the different values of an attribute A, and Ai = Ci(A) the value of A after change Ci is performed. The classic (ACID) transaction concept says that Ci and Cj are performed in transaction if the result A3 = Cj (Ci (A)) or A3 = Ci (Cj (A)). We extend that definition saying that, given an object A, changes Ci and Cj on the that object can be performed in a concurrent way if there exists a function M (merge) such that M (C1(A0), C2(A0)) = Ci(Cj(A0)) = Cj(Ci(A0)) = M (C2(A0), C1(A0)) .

This means that the result of a merge is the same as if changes Ci and Cj were performed in sequence on A0, irrespective of the order. If an exact merge function existed for each attribute, concurrent engineering would always lead to consistent results! Unfortunately, for a given attribute, such a merge function either (1) exists, (2) is an approximation, or (3) does not exist at all. Typically, the composition attribute has an exact merge, sourceFile merge is an approximate function (the usual merges), and most attributes like fileName have no merge at all.

2. High Level Concurrent engineering policies

Experience shows that customers find extremely difficult to design their own concurrent engineering strategies. Our claim is that planning and implementing different strategies is a hard task in existent SCM tools because of the lack of an adequate level of abstraction. Any decision made using classic SCM concepts (branches and revisions) is error prone, due the unexpected effects it might have.

We propose a basic work model (the group) and a set of operations on which concurrent engineering policies can be easily designed. This vision allows us to take CE policies conception to a more adequate level, using abstractions that are more natural to concurrent work (group's workspaces and operations among them) instead of the traditional low level ones (branches, revisions and merges). We have defined a language that lets us express the most common forms of group based C.E. policies in a simple straightforward way.

2.1. Product data model

A product is a typed and named object composed of attributes (name-value pairs) and a complex object representing the document in a domain or tool specific format. Attributes are either *common* or *versionable*; document formats can be data based schema, XML, or any other, but most often it is a simple file system representation i.e. a hierarchy of files and directories, where each file is seen as operating system attributes (name, author, rights …) and a content.

2.2. Workspaces

A workspace contains one or more documents. A workspace is made of two parts: the *working area*, and the workspace *local history*.

The *working area* is a repository in which documents are represented following their original format, and under the tool required for their management; and attributes can be seen and changed through our specific attribute management tool. In Dassault Systèmes the working workspace is simply a FS containing the documents. [4]

The workspace *local history* is a repository in which is stored some *states* of the associated workspace (i.e. its value at a given point in time). The local history is meant to be local to the participant's machine and therefore can be used while working off-line. A workspace manager is responsible of providing the mapping operations between the working workspace and the local history repository.

In a large development environment it is possible to have different policies regarding the frequency and the motivation of creating new versions; it is also possible to have different version managers according to the different versioning requirements and different emplacements. Our system doesn't address the problem of versioning; it is capable to work with heterogeneous version managers, s long as a few functions are

available (i.e. the CI and CO functions) [5]. In our simple CE tool, the local history can be either CVS or Zip, and the working workspace is a FS containing the documents, plus the attribute management tool.

From the "outside world" only a single workspace state is visible, it is always one of the states stored in the local history; not necessarily the latest one. The working workspace is not directly visible.

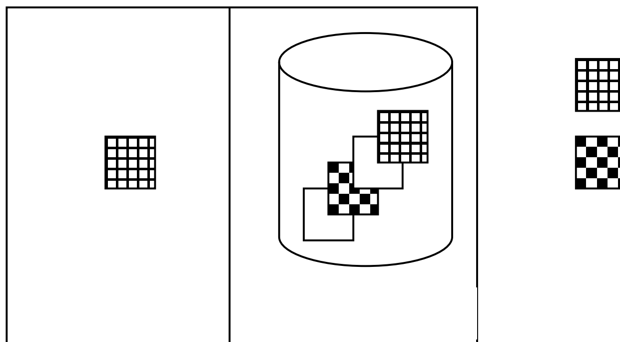

Fig. 1. Workspace

2.3. Groups

A group is a set of workspaces working on the same products to achieve a common goal. One of them, the *parent* workspace, is in charge of synthesize and integrate the work made by the child workspaces. The *child* workspaces can only communicate with the parent.

Comparing with databases, the parent workspace plays the role of the central DB and the Childs the role of the cache for the concurrent transactions.

The complete group behaves as its Parent alone; from outside, the Parent is a "normal" workspace in which is performed the whole job of the group. It is thus possible to build groups where Childs can be parent of a lower level group. The work structure is therefore a tree where nodes are either parent or child workspaces, and arrows the containment relationships. Containment between two groups may mean either work decomposition into concurrent activities (each group performing a different task), or different level of validation (top level nodes being more validated that leaves).

In practice, companies use both approaches simultaneously; Dassault Systèmes uses currently at least a 6 level work space hierarchy; the 3 top level being called GA (general availability), BSF (Best So Far), and Integration which represent different levels of validation, while the 3 (or more) lowest levels represent a decomposition in task and sub-task for the current work to do.

It is also interesting to point out that a group defines de facto a cluster of highly related work spaces; the Parent work space playing the role of global repository for the

group. It allows for a real distribution of repositories as shown in [6]; there is no longer any need for a common central repository; distributed and remote work is handled that way. It is our belief that concurrent engineering of very large teams cannot be handled without such a distribution of repositories, not found in classic SCM tools [9].

CE policies are defined for a group, a whole hierarchy of groups can have different polices reflecting the different methodologies and consistency constraints of each level. Typically, the groups closer to the top level, which can be seen as holding the official copies of the documents, have more restrictive policies then the "bottom" groups, who need higher concurrency levels.

2.4. CE Global Policies

A product is a composite entity that has a number of internal consistency constraints. In a similar way as transactions, a number of changes must be done before a product recovers a new consistent state; this means that only complete new consistent states can be communicated to other workspaces, not only a sub sets of changes. Therefore operations between workspaces involve only complete product states.

Provided a group with Parent workspace P and a child workspace W, these operations are the following:

Synchronize (W,D): If Document D does not exist in workspace W, it is copied from P. Otherwise, the currently visible D version of P, is integrated in W working workspace. In other words, changes made on D in P since last synchronize operation, are integrated in W.

Integrate (W,D): All modifications made on the visible version of D in workspace W are integrated into the D in the Parent's working workspace. In practice, it means that changes performed in W, since last Integrate, are propagated in P.

These operations are critical for any CE policy. It is important to know who is entitled to decide to synchronize or integrate two workspaces (the Child or the Parent one ("w" or "p"), and if the synchronization or integration are performed interactively ("i"), or "batch" ("b"), as well as to decide the level of consistency of such critical operations.

By default, interactive merging allows unsynchronised integrations (s), merges (m) and conflicts (c), and batch (b) do not. For example "I" alone means that integration is initiated by the Parent workspace, in interactive mode, merges and conflicts allowed. Unsynchronised (s) means that the receiving workspace is the current common ancestor, which means the operation is simply a copy of the current workspace to the receiving one.

"Ibsm" means that the integrate operation (I) is asked by the Parent workspace in batch mode(b); it can involve merges (m) but not conflicts. For example, CVS enforces "Iws" and "Sbmc" assuming the Parent WS is the CVS repository; i.e. Integration is at the child initiative (w), in batch mode (b), and allows no merge nor conflict; while Synchronize are batch (b), but accept merges (m) and conflicts (c).

Of course "m" implies "s", and "c" implies "m", thus "smc" and "c" are equivalent. In order to make policies more readable, by convention "Ip" is equivalent to "I" and "Sw" to "S"; and "S" and "P" alone is interpreted as "Swismc" and "Ipismc".

The other possible operations are:
Publish (D): The integrator stores its working state of document D in the local history and made it visible. All child workspaces are notified that there is a new visible version to synchronize from.
Propose (W,D): Workspace W stores its working state of document D in the local history; that version becomes the visible version, and the integrator is notified that there is a new version of D, on the workspace W ready to be integrated.
Change (C, D, W): Pseudo operation Change means workspace W *is allowed* to perform a modification on document D, this is, it can modify its attributes.

2.5. Policy definition

Defining a policy means defining the valid history of operations a workspace can perform on a given product. This can be made by defining a state machine representing the different valid operations (the inputs of the state machine) for a given state of the workspace (a particular step in the work cycle of the workspace). A global policy defines the rules for a particular child workspace and on a particular product. We can then define a global policy using the following conventions for the operations:

- I : Integrate, as discussed above (with possibly suffixes p, w, i, s, b, m, c)
- S : Synchronize, as discussed above (with possibly suffixes p, w, i, s, b, m, c)
- Pu : Publish
- Pr : Propose
- C : Change

The parameters of the operations are the workspace and the product for which the policy is being defined. We use regular expressions syntax as the base for our language. For examples:

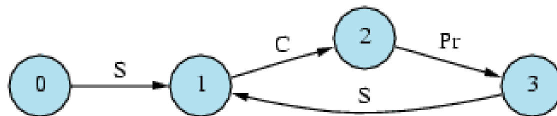

Fig. 2. (S C Pr)*

In this simple global policy, a workspace must first to Synchronize (S) to have the current visible version of the product in Parent, and only then it can start its work, modifying what it needs to change(C), and finally, propose (Pr) its working version to be integrated. The cycle then starts again. Integrate is not indicated, it means that the Parent workspace decides, based on the Childs ready for integration, which one to integrate. But, in this policy, integration may involve merges and conflicts that the user working in the Parent workspace will have to solve.

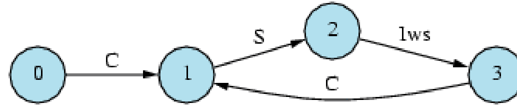

Fig. 3. (C S Iwbs)*

In this policy, the workspace starts working on the product (C) followed by a synchronize (S) and an integrate (I), both executed at the current workspace initiative. Since S is performed right before I, merges, if any, are performed in the child workspace during the S operation. This is consistent since S alone stands for Swismc i.e. interactive with merges and conflict allowed. In this policy, Integrate will not involve any merges; this is why it is often used.

The Change Operation

The pseudo operation C indicates when, in the execution of a policy, modifications can be made. This does not means that actual changes on the product attributes are mandatory, but only the possibility to perform them. We then always interpret C as (C)? meaning a change is always optional.

Integrator initiated operations and behavior of the integrator workspace

The user working in the Parent workspace (when existing) plays the role of group manager. He/she manages the reference version of the group, he/she decides what product to integrate (if "Ii"), notifies when new versions are available ("Pu"). It will be seen later that he/she has also special role in managing locks, solving conflicts and handling failures, using for that purpose, special operations.

The global policy defines the behaviour of a single child workspace. For a given state, only some operations are allowed, and its execution fires the transition to the next state, where new operations are allowed. Some of the group operations, however, are not under the control of child workspaces, but are initiated by the integrator. It is desirable to clarify some aspects of the interaction between integrator and child workspaces, such as the fact a newly integrated version has to be immediately published, or that a workspace must wait for a Publish to continue working.

Our strategy is to gather all these details in a single global policy, in order to have a single place to look to know the policy.

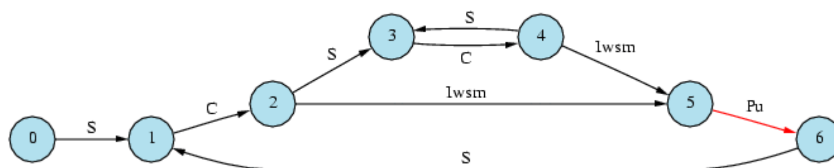

Fig. 4. ((S C)+ Iwsm Pu)*

In this policy, a workspace can synchronize and modify a product several times, but it needs to do at least one synchronization/modification cycle("(S C)+") . When the workspace integrates its work, at its initiative (Iwsm), a publish is automatically

executed, to notify all workspaces that a new version is available in the integrated workspace.

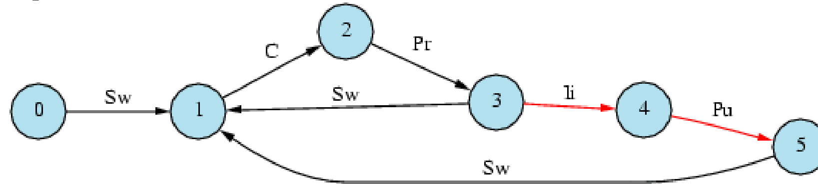

Fig. 5. (S C Pr (Ipic Pu) ?)*

In this policy, the workspace does not need to wait for "publish"; after proposing a modification (pr), it can synchronize again and continue to work. A new "propose" will however replace the visible version with the new visible state of the product in the workspace. When the user working in the Parent workspace integrates one of the proposed versions, it is published immediately.

2.6. Local Policies

The global policy assumes all attributes can be changed and that conflicts, if any, are solved during the "synchronize" or "integrate" operations. According to the nature of the attribute, merge functions may not exist, in which case the work will be lost during synchronization. If the merge function is approximate, the merge operation is risky and may require manual intervention. Even if perfect, merges may be not desirable without control.

Sensible data can be protected using the consistency constraints for S and I operations defined in the global policy. An operation that violates the imposed constraints is banned, and a special recovery mechanism must be applied. This is an optimistic strategy, while recovery might be an expensive operation; violations of the consistency constraints are expected not to occur often. In local policies a pessimistic approach using locks can be used to protect consistency of the data held by the most sensible attributes.

Local policies have a name and are defined for families of attributes, for examples:

AttributeName => PolicyName

Standard expansion rules for files and directories apply, with the product directory as the root directory. Operating system file attributes are precised using ":" after the file expansion. :

FileSuffix:FileAttributeName => PolicyName

Examples:

Date => Policy1

src/*.c:name => Policy2

Which means that the "Date" attribute is managed following the Policy1 policy, and the name of C files "*.c:name" found under the "src" directory are managed following the Policy2 policy.

Local Operations
Reserve (W,P,A) : A lock is set on attribute A of product P by workspace W, if there is no lock already. When a lock is set, only the workspace that owns it can modify the attribute.
Free(W,P,A) : A lock is removed on attribute A of product P by workspace W, if W owns that lock.

Correctness

The complex nature of the data manipulated, the semantic dependences among the attributes, and the long duration of the tasks in concurrent engineering make global notions of correctness such as serializability and atomicity inconvenient. Instead, user defined correctness criteria based on semantic knowledge of the attributes; their relationships and the tasks performed must be allowed. Studies on cooperative transactions have found in patterns and conflicts a natural way to specify different correctness criteria. Patterns define the operations that must happen, and the order in which they must happen for a transaction to be correct. On the other side, conflicts specify the forbidden interleaving between different transactions [1][2][3][10][16].

We have not defined a general language for defining any possible correctness criteria for any cooperative transaction, as proposed in [1], where the defined grammar can take into account all possible interleaving among transactions. This would make policy definition difficult, and threaten the dynamic nature of our system where new groups and policies can be easily created as the development advances. Instead we propose a tiny set of interpretations of the local policies corresponding to different levels of consistency, inspired by typical consistency constraints found in Software Engineering.

Our global policy defines the pattern of behaviour for all attributes, for example in the policy "(S C I)*" states that a synchronization, modifications and integration, in that order, are necessary for a workspace to correctly finish a cycle of work. Introducing Reserve and Free operations in the local policies we define the conflicts, reducing the possible interleaving to a sub-set that enforces our correctness criteria:

Very strict policies
Very strict policy execution might be desirable for groups of attributes that are related by strong consistency constraints.

An example is the name of the files composing a software module. Restructuring that module involves changing file names, and clearly this task should be undertaken in a single workspace at a time.

A group of attributes under a very strict policy can only be modified by a single workspace, thus locking the whole group of attributes before starting the transaction.

*Module_names = very_strict: (R S C I F)**

We protect the source file names of certain module with the local policy:

/project/libxyz/.java:name : Module_names*

If a workspace tries to modify the name of a java file in the module libxyz, it needs to reserve all the name attributes, synchronize, modify and integrate, and finally all attributes are freed.

Strict policies

A very-strict policy guarantees consistency of relationships among attributes within a group, but prohibits any concurrent work. In some cases concurrent modifications on different workspaces is still considered consistent as long as the set of attributes modified are not overlapping.

For example, a given word files should not be edited concurrently in different workspaces, but it is no problem to edit different word files concurrently. It corresponds to a strict interpretation of policy like:

*EditDoc = strict: (R Swismc C Ipismc F)**

/project/stat/.xls : EditDoc*

EditDoc policy means that all Excel documents that are to be changed in a session are to be first reserved, then a synchronize is performed (thus getting in the current version of the documents), then editing of the document is allowed; the session ends by integrating all the changed documents.

Flexible policies

The strict interpretation requires an a-priori knowledge of all the documents that are involved in a session; it is not possible to try to reserve a new document as soon as the editing session is started. For most purposes, this is too strict.

For a given attribute A, "R ... S ... C" is equivalent to "S ... R ... C" if between synchronization and reserve the attribute value did not change in the Parent workspace.

This property is at the base of flexible policies. In a flexible policy, the reserve is automatically attempted when the attribute is to be changed; this reserve succeeds only if the value of that attribute did not changed in the Parent WS since last operation S. Therefore the policy

*EditDocFlex = flexible: (R S C I F)**

/project/doc/.doc : EditDocFlex*

looks similar to EditDoc presented above, but here, as soon as a new word document is about to be changed, a reserve operation is attempted. There is no need to perform explicit reserve operations. This is much more flexible indeed, but there is two drawbacks, with respect to a strict policy :

There is a risk of inconsistency, if some documents have strong semantic consistency constraints, and one of them is not reserved,

There is a risk of dead locks if two concurrent sessions require an overlapping set of documents.

Reservation Propagation

Reserving an attribute in a group means reserving that attribute for all the group workspaces, including the Parent. But reserving the attribute in the parent may reserve it in the group to which the parent pertains, and successively. Thus, reserving an attribute may propagate to a large number of workspaces, especially because higher level groups have stricter policies, and are more likely to define policies with locks.

Optimistic and controlled strategies without locks.
The major reason to use locks, is to make sure that an integrate operation will not involve any merge and conflict. But our specialization of operation I and S allows such a control, a posteriori, i.e. an optimistic controlled policy, as opposed to locking strategies which correspond to pessimistic ones; the different lock interpretations ranging from very pessimistic (very-strict), to relaxed-pessimistic (flexible).

For example

*Pesimistic= strict: (R Swismc C Iwb Pu F)**

Is similar to

*Optimistic= (Swismc C Iwb<&S> Pu)**

The difference is that in the Optimistic policy "Iwb" may fail, if there are merges or conflict involved. If case of failure, the policy indicates that an operation synchronize must be performed, and Integrate attempted again "<&S>".

Implementation
We translate the policies, including their correctness criteria, into a state machine. Our implementation is efficient, because a single state machine is sufficient for a group of attribute, whatever the number of actual attributes in the group, and the state machine interpretation is straightforward. Policy interpretation is very cheap. Further, these state machines are created and executed on the workspace machine, and need little interaction with other work space to be executed. Reserve and free are those operations, as well as a delayed reserved (to check the attribute value in the parent); but in all cases, only the parent workspace is involved.

2.7. Summary

The language is based on the distinction between global and local policies. Global policies come from the fact consistency requires a product to be managed as an atomic entity; local policies come from the fact each attribute may have different consistency constraints, either technical (related to the availability of reliability of their merge function) or logical (sensitivity and criticality of the attribute, semantic relationships, and so on). The system allows for both and enforces the consistency of the whole.

The language is also fairly independent from a given data model. Indeed, it is based on attributes of entities and can therefore be adapted to almost any data model, even if the tool we have built works on a very pragmatic one: file system attributes and a number a specific attributes.

The central issue in concurrent engineering is always to find the compromise between high concurrent work (and thus very optimistic policies, or no policy at all), and reliability (and thus restrictive policies). We address this issue first, allowing different policies for different attributes, second allowing a large range of pessimistic-optimistic policies.

The optimistic / pessimistic range is split in two. In the pessimistic side, we use locks, and we provide three different interpretations of the lock instruction: very-strict, strict and flexible. But locks propagate and may seriously reduce concurrency.

In the optimist side, we provide a number of restriction in operations Synchronize and Integrate (batch/interactive, parent/child, synchronized/unsynchronised, merge/no_merge, conflict/no_conflict) which allows to avoid problems, still retaining maximum concurrency, but in this case, the policy must include failure recovery.

Altogether, the language covers a very wide spectrum of concurrency policies, going much farther than traditional database strategies, and capable to represent many database strategies (including, ACID, dirty read, non repeatable read, 2PL and others). Nevertheless, the language does not allow for any and all concurrency policy, but do correspond to actual practices in software engineering. It is our goal to propose a very simple language, intuitive enough for users to define easily the policies they need, not to define anything, thus this selection of interpretations, which corresponds to best practices.

3. The CE tool

We have built a prototype implementing the ideas presented above. Our system exposes the following features:

- It contains an interpreter of our language that runs locally to each workspace, enforcing the policies defined for the group that contains the workspace. An efficient implementation of the interpreter has been
- It implements the basic synchronization services
- It implements a distributed locking system.
- It manages dynamic creation of new groups as development evolves

Special attention is paid to the awareness capabilities of the tool. Awareness is characterized as “an understanding of the activities of others, which provide a context for your own activity” [7]. The assumption is that providing the users with the appropriate contextual information allows them to make more sophisticated local decisions. The awareness concept has been long studied and proved beneficial in the CSCW domain, but has never gained the same popularity in SCM systems.

For the vision of context to be useful it must provide only the information relevant for the user to take decisions based on it, without the unimportant information introducing noise. We use the group model to filter all the information coming from all the workspaces to the subset that is useful for each user.

The most basic contextual information needed for cooperative work is group membership, our CE Tool provides every user with a vision of the workspaces in his group, as shown in figure 6, this list is synchronized with the actual state of the group held by the integrator. Additionally, the user can find what documents each user has proposed.

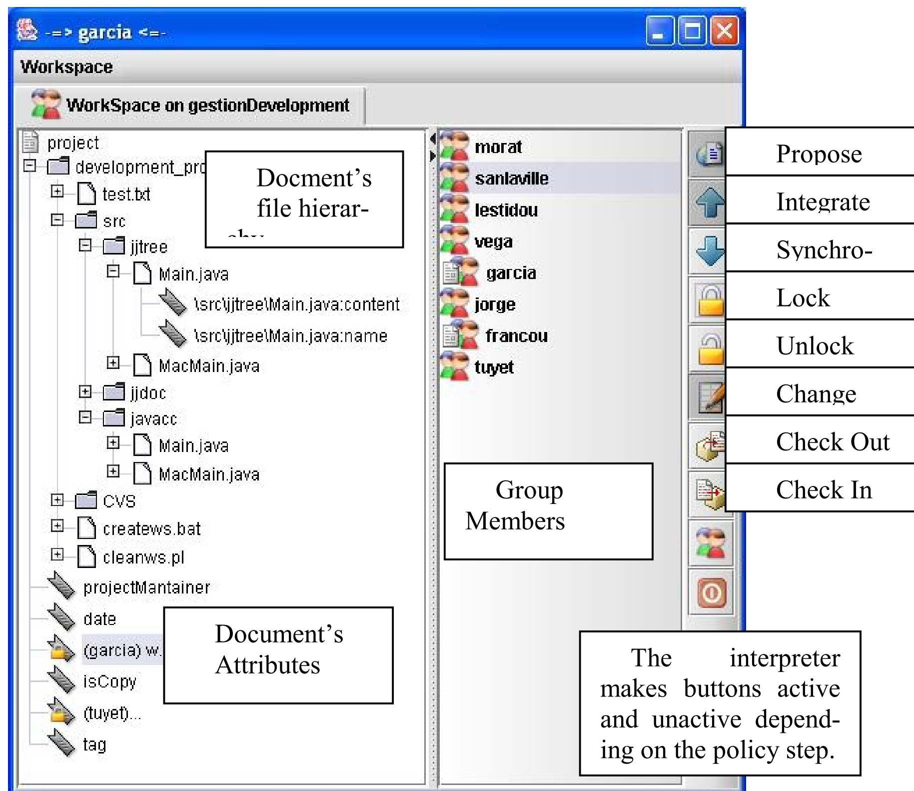

Fig. 6. The CE Tool

The tool also provides a vision of the state of the document's attributes. Based on the operations' history in a group, the attributes of a child workspace version of a document can be in one of the following states:

- Obsolete: The attribute has changed in the parent workspace and not in the child workspace.
- Updated: The attribute has the same value in the child workspace than in the parent workspace.
- Modified: The attribute has been modified in the child workspace and not in the parent workspace.
- Conflict: The attribute has been modified both in the parent workspace and in the child workspace.

Currently, the system can run using CVS or our home-grown zip version manager as the tools to store the local history, but integration with other version managers should be not difficult.

4. Conclusion

The intention of such a language is to propose, to non-expert users, a simple way to define and understand their concurrent engineering policies. Indeed the language is based on operations that user know very well, since it is what they use daily, and hides implementation concepts like branches, revisions, predecessor relationship, common ancestor and so on. Since these operations are, in a way or another, available in all SCM systems, the language is independent from any single version manager and

therefore is completely general; it can be adapted to any SCM system, and at least used by team leaders to define their policy first, before to find a way to implement it using their current SCM system.

The language; despite been simple, allows for a wide set of policies, ranging from the most restrictive (very-strict with locks), to the more relaxed (optimistic, no locks), or even no policies at all. We believe the fact workspaces are typed, as well as policies, the fact that policies can be applied independently to different attribute groups, makes the system extremely powerful and versatile, but still simple.

Indeed the experience so far proved that users find the language very convenient to define the policies; users understand fast what it means, and it allows them to reason on their policy, to discuss their respective merits, and gradually to define and select the one most convenient to them. Only this is already a major progress.

We believe that this work goes a step beyond, deriving from the policy an automatic implementation, still independent from the actual version manager in use in the company. We have built a tool that support policy definition and policy enforcement, including the special operations for recovery, not presented here. The tool is dynamic, in that groups are created and changed dynamically at execution. Of high interest, is the fact policies can be statically analysed, and properties statically derived. Among the properties, we can mention that we know if merges are possible or not, we know where and who is the common ancestor and we know which files will be involve in next Synchronize and Integrate. From these properties, optimisations can be performed including the determination of the files involved in the next Synchronize / Integrate, the computation of deltas before hand, and the avoidance of useless checks like changes or merges. Indeed, in many cases, most of the work involved in the heavy operations Synchronize and Integrate can be computed before hand. This is important since experience shows that users feel that Synchronize and Integrate are operations that slow down their work; our optimisations make these operations much faster from the user point of view and improve significantly user comfort.

The properties are also used to improve awareness strategies. Indeed our tool includes awareness facilities only sketched in this paper.

Altogether, we believe that our system improves the state of the art of concurrent engineering in many dimensions, both from theoretical and practical point of views. From conceptual point of view, we found a language simple, powerful, with well-defined semantics. From the practical point of view, this system improves user understanding of CE issues, improves the enforcement of these policies, improves the independence between policies and versioning tools, improves efficiency of the system, and improves user comfort and awareness.

References

[1] M. Nodine, S. Ramaswamy, S. Zdonik. "A Cooperative Transaction Model for Design Databases". Edited by A. Elmagarmid. "Database Transaction Models". Chapter 3, pages 54-85, Morgan Kauffman Publishers, Inc.

[2] S. Heiler, S. Haradhvala, S. Zdonik. B. Blaustein, A. Rosenthal "A Flexible Framework for Transaction Management in Engineering Environments". Edited by A. Elmagarmid. "Database Transaction Models". Chapter 88-121, Morgan Kauffman Publishers, Inc.

[3] A. Skarra. “Localized correctness specifications for cooperating transactions in an object-oriented database”. Office Knowledge Engineering, 4(1):79-106, 1991.
[4] J. Estublier. “Workspace Management in Software Engineering Environments”. In SCM-Workshop. Springer LNCS 1167. Berlin, Germany, March 1996.
[5] J. Estublier and R. Casallas. “Three Dimensional Versioning”. In SCM-4 and SCM-5 Workshops. J. Estublier editor, September, 1995. Springer LNCS 1005.
[6] J. Estublier. “Distributed Objects for Concurrent Engineering”. In SCM-9. Tolouse, France. September 1999.
[7] P. Dourish, V. Belloti: “Awareness and Coordination in Shared Work Spaces”. Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)
[8] C. Neuwirth, D. Kaufer, R. Chandhok, J. Morris, “Issues in the design of Computer Support for Co-authoring and Commenting”, Proc. CSCW’90 Computer Supported Cooperative Work (Los Angeles, Ca., October 1990).
[9] J. Estublier, R. Casallas. “The Adele Software Configuration Management”. Configuration Management”. Configuration Management. Edited by W. Tichy; J. Wiley and Sons. 1994. Trends in software
[10] A. Skarra. Concurrency control for cooperating transactions in an object-oriented database. SIGPLAN Notices, 24(4), April 1989.
[11] J. Estublier, S. Dami, and M. Amiour. “APEL: A graphical yet Executable Formalism for Process Modelling”. Automated Software Engineering, ASE journal. Vol. 5, Issue 1, 1998.
[12] J. Estublier, S. Dami and M. Amiour. High Level Procesing for SCM Systems. SCM 7, LNCS 1235. pages 81-98, May, Boston, USA, 1997.
[13] P. Molli, H. Skaf-Molli, C. Bouthier: “State Treemap: an Awareness Widget for Multi-Synchronous Groupware.”. 7th International Workshop on Groupware (CRIWG'01). September 2001
[14] P. Dourish and S. Blay, “Portholes: Supporting Awereness in Distributed Work Groups”, Proc. CHI’92 Human Factors in Computer Systems (Monterey, CA., May 1993).
[15] C. Godart, F. Charoy, O.Perrin and H. Skaf-Molli: “Cooperative Workflows to Coordinate Asynchronous Cooperative Applications in a Simple Way.”. 7th International Conference on Parallel and Distributed Systems (ICPADS’00). Iwate, Japan, July 2000
[16] M. Franklin, M, Carey, M. Livny: “Transactional Client-Server Cache Consistency: Alternatives and Performance”. ACM Transactions on Database Systems, Vol. 22 No. 3. September 1997
[17] C. Godart, G. Canals, F. Charoy, P. Molly. “About some relationships between configuration management, software process and cooperative work: The COO Environment”. In 5th Workshop on Software Configuration and Maintenance, Seatle, Washington D.C. (USA), LNCS 1005, April 1995.
[18] C. Godart, O. Perrin, H. Skaf. ”Coo : A Workflow operator to improve cooperation modeling in virtual enterprises”. In 9th IEEE Itnernational Workshop on Research Issues in Data Engineering Information Technology for Virtual Enterprises” (RIDE-VE’99), 1999.