Process Model and Awareness in SCM

Jacky Estublier and Sergio Garcia

LSR-IMAG, 220 rue de la Chimie
BP53
38041 Grenoble Cedex 9, France

{Jacky.Estublier, Sergio.Garcia}@imag.fr

Abstract. The development of large and complex systems, under hard time constraints, requires the participation of many developers working concurrently. SCM systems allow concurrent access to software artifacts, but provide poor support to maintain data consistency when concurrent changes are performed on the same artifacts. This problem can be reduced if developers are aware of the others work and warned about the conflicts that may arise, allowing the users to manage the risks more effectively.
Awareness, without any knowledge about the cooperative process and system models cannot help much, and indeed is not very much used today. We claim that awareness takes its potential only when it takes into account the cooperative process, and the system model in use. This paper, based on the experience gained with our tool Celine, explores the relationships between awareness, process and system models, and shows how the knowledge of these models can be used to improve the relevance of an awareness system.

1. Introduction

One of the key factors for the success of software in a competitive environment is time to market. This fact, added to the increasing size of software systems, leave organizations little choice other than to increase the number of engineers that work on a same product at the same time [5]. Unfortunately, rising concurrency increases the possibility of inconsistent modifications and puts the software stability at risk. SCM systems allow concurrency, but provide simplistic and inadequate support to handle this risk.

The databases community has in depth studied the problem of data concurrent modifications. There are, however, fundamental differences that make concurrent software engineering a field of research on its own. Those differences can be summarized in the following two points [12]:

1. Consistency definition. In software engineering there exists no adequate global criteria, such as serializability, to enforce the correctness of concurrent modifications. Serializability is not adequate because software is composed of complex and tightly interrelated data, such that most modifications have potential side effect on a large and a priori unknown sub set of the data. This means that two concurrent

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

modifications are almost never serializable under the classic databases definition [12][13].

2. Long duration. In databases, it is common to abort a failed transaction. Engineering tasks take days of human work, which makes it unacceptable to drop them. [12]

Usual SCM tools allow concurrent work by providing private workspaces and mergers to try to reconcile concurrent work; but they do not provide effective mechanisms to coordinate developers working concurrently. Without such mechanisms, project managers must either impose simplistic restrictions on concurrency (file locking) or they must leave all the burden of avoiding data corruption to the developers. When the number of developers increases, the risk of modifications being combined inconsistently increases.

It has been argued that this problem can be palliated by raising the awareness of developers about the work of their partners [1][3]. The main hypothesis is that if developers are given a continuous insight over the group activities, they can detect and handle potential conflicts more effectively during the development process. The challenge of an awareness system is to provide the user with the relevant, and only the relevant information. This paper explains how process and data models are related to awareness and how their knowledge allows building a smart awareness system.

Section 2 presents some background and section 3 explains what is the purpose of awareness. Section 4 relates cooperative process with awareness, 5 with cooperative policies and business processes. Section 6 introduces shortly the relationship between system model and awareness. Section 7 presents Celine an section 8 concludes.

2. Background

2.1. Cooperative engineering

We can characterize cooperation as a group of persons pursuing the same goal. In software engineering, the goal is to transform a software application from its actual state say V0 into a new state say V1.

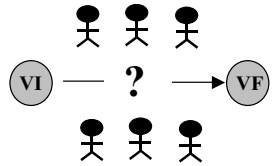

Fig. 1 Cooperation

The role of cooperative software engineering is to control and support how this transformation happens [8]. There exist several basic strategies. For instance, work can be serialized, allowing only one developer to modify the software at a time.

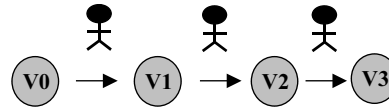

Fig. 2 Serialization

To increase the development speed, the work load can be broken down in several tasks and subtasks that are assigned to different developers and executed in parallel. Those tasks should be designed such that they involve different parts of the system; unfortunately, provided the strong coupling between the software artifacts, it is likely that more than one developer will need to change the same software artifact. Concurrent change of the same artifact must be possible. But since most development activities require exclusive access to the data, concurrent software engineering needs a system that supports multiple copies, called cooperative copies, of the developed software. Concurrent engineering is about the control of how these cooperative copies are created and reconciled into a common result.

2.2. Asynchronous Cooperation with Workspaces

SCM is not the only field supporting cooperative changes; indeed all engineering domains face the cooperative engineering challenge. It is the nature and relationship between data, the nature and duration of activities and the development process that makes that some solution are relevant in some engineering domains but not in others. For example, Computer Supported Cooperative Work (CSCW) systems aim at controlling how a document is co-authored, which looks pretty close to software engineering, but CSCW often follows a blackboard metaphor where multiple participants work simultaneously on the same document. In those systems, even the smallest modification is visible to anyone "as soon as" it occurs. Cooperative editors implement that strategy propagating “instantaneously” all changes over the network, for all the document views to be permanently synchronized. [16] [17]

The CSCW approach is not adequate for software engineering because most software changes are performed as a set of partial modifications that make sense only within the context of a single task. Interleaving these partial modifications would therefore produce a document in an inconsistent state, prohibiting compilation and test until the end of the whole work. In software engineering, instead, each task makes sense by itself (e.g. fixing a bug), and should be compiled and tested independently. Such tasks require time and, usually, the availability of a significant fraction, if not all, the code.

This is precisely the role of workspaces: to host complete, isolated copies of the software during arbitrarily long periods of time, allowing several independent development activities to be performed simultaneously and independently. Developers can modify their private copies, leave them in inconsistent states for arbitrarily long periods of time, perform operations on them without being subjected to the others changes, and vice versa [1][4]. This separation between private local copies and public copies is known as workspace isolation.

Unlike CSCW systems, in software engineering, the synchronization among the different workspaces is done explicitly and usually involves a complete and consistent unit of work. What means “complete and consistent” is highly related to the workspace purpose and development process. A large part of cooperative engineering *policies* consists in defining “complete and consistent”, and to define between which workspaces synchronization should occur.

2.3. Divergence and reconciliation

Our hypothesis of a final common result implies that eventually all the cooperative copies are merged into the final result. Therefore cooperative engineering is fundamentally about merge control. The problem is twofold:

- **Data consistency** : Does merge provides a consistent result?
- **Process** : Are changes performed by the right person, on the right artifact at the right time?

The two points seems unrelated but experience shows that the process purpose is to make in such a way merges have the best chance to produce consistent result, and to be performed by the persons capable to solve conflicts if any. [2]

Serialization being the only well known consistency criteria, the merge function is a substitute for serialization : it takes two changes performed independently and concurrently on two copies of the same data and is supposed to produce the same result as if the changes have been produced in sequence i.e. the “second” change being performed on the data produced by “first” change. Unfortunately, the merge function depends on the nature of the data. For example for sets, the merge function exists and always produces the right result; for lists, only approximate merge functions exist; for complex data structures, only specialized algorithm taking into account the data semantics can perform merges. Programs are often assimilated as lists (of lines code) for which only an approximate algorithm exists; language specific mergers also exist but do not provide much better result and therefore are not used in practice [14].

Line merging is therefore a difficult and error prone task that needs human expertise and understanding of both the software and the modifications. When two modifications are found incompatible, large parts of them might have to be discarded, with the loss of hours or days of work. The risk of incompatible changes, and the complexity of merge, increases with the amount of changes to be merged. It is well known that, if waiting too much before to merge, the merge becomes very complex and risky, if not impossible. The only solution is keep divergences “small enough” by merging the different copies as soon as possible. These early merges create intermediate "agreement" points that serve as the base for the next reconciliation, reducing the gap that needs to be closed each time.

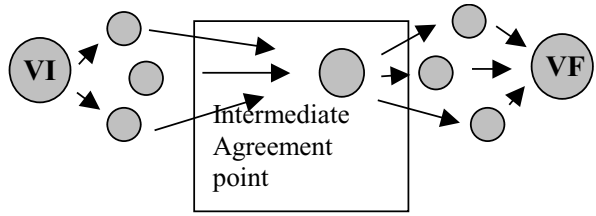

Fig. 3 Cyclic reconciliation.

As a consequence, cooperative engineering is a compromise between (1) maintaining isolation until the complete work is done and (2) merging as often as possible. It is the development process that defines and controls that compromise.

2.4. The development processes

How data is modified, when, by who, and how work is combined towards a single result is what we call the development process. From a data flow perspective, the development process can be represented by a directed graph, where nodes represent workspaces, and arcs represent the data flowing from origin node to destination node.

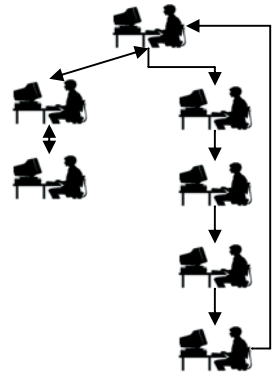

Fig. 4 Example of a static graph of workspaces

The development process can be controlled by different processes defined at different levels :

1. *Concurrent Engineering process*, which defines the graph topology i.e. its nodes and arcs. This graph defines how team work is structured and which paths the data can follow.

2. *Policy process,* which manages the data flow along the cooperative graph.

3. *Business process* which includes the change control process, engineering methodology, business activities, i.e. which activities are to be performed on which entities, for which reason, by whom and so on.

In this paper we will focus in the first level only, but our tool Celine covers point 1 and 2 [6], and is integrated with Apel [7] and Melusine [9], which cover point 3.

3. Awareness : purpose and issues

Given that the merge algorithm for software source code is not perfect, merges can be made safer enforcing the following practices:

- Merge often, so conflicts are easier to solve.
- Allow only concurrency on unrelated changes, so to reduce the probability of conflict.
- Give the task to merge to the person most likely to perform the right job.

A mixture of these strategies can be provided either by implementing a well defined process (the concurrent engineering policy) or by trusting the developers to apply the safe practices by themselves. Unfortunately, without any information about the work of their partners; developers do not know what are the current changes, who did them, for what purpose and so on. Therefore developers cannot apply the above practice. This is why many companies simply prohibit concurrent engineering for being too risky. The purpose of awareness is to give the developers the necessary context and information allowing them to apply the above good practice, making concurrent engineering "safe enough" for the advantages of a faster development significantly outweigh the risk of inconsistent or impossible merges [3].

Since the problem of safe concurrent engineering is clearly the problem of controlling the merge, the main criteria to evaluate the effectiveness of an awareness system is to measure to what extend the system helps the developer in deciding what to modify and when to merge.

We call *absence of concurrent engineering process* the fact modifications can flow without control between any pair of workspaces. Under this chaotic model, data can follow any possible path, making it impossible to foresee how and where a merge will occur. Without process, an awareness system can only inform all the users about the differences among all the existing work copies.

For a large numbers of workspaces, not only the system cannot give any hint on what is likely to occur (anything can occur), but also displaying this raw information leads to the problem of cognitive overload [1]: when too much information is presented, the cost of understanding it is too high for awareness to be of any value. For example, suppose 50 concurrent workspaces containing 100 files each, and for an average of 1 changes per file[1], a classic awareness system will display 5000 awareness messages which is much too much to consider; further, if you do not know

[1] These values are those of the application on which Celine is used today and are typical of many software development projects; but our Dassault Systems customer has 1000 concurrent workspaces, containing 2000 files in average, with 3 changes in average for each file, i.e. 6 millions awareness messages to display to each developer!.

why these changes are performed, and in which state they are, the information is of little help.

So far, all the awareness systems we know are ignoring the process, and therefore are of little value. We believe it is the reason why these systems have not been adopted by practitioners. An awareness system, to succeed, must provide simultaneously much less information and much more relevant information taking into account the process and the product models, in order to forecast the upcoming merges. The relationship between awareness and process models, and between awareness and system models will be presented in the following sections.

4. Concurrent Engineering Process and Awareness

4.1. Concurrent Engineering Process

The *concurrent engineering process* defines how workspaces are organized to communicate their work. This process defines which workspace can send its artifacts to which other workspace.

Concurrent engineering is usually not chaotic, but relies on a graph that guarantees some properties. For example, CVS, RCS, Oracle 8 workspaces, as well as most SCM systems are based on a star topology: a central database serves as a hub for multiple "satellite" workspaces that are not allowed to communicate directly. This approach is attractive because it ensures that the central database will always contain a copy that is not too old compared to the developer's copies, making it optimal to represent the collective work of the team at any time.

Unfortunately, for highly collaborative projects the number of workspaces under the central database can quickly become too big to keep awareness information tractable.

A concurrent engineering process well suited for awareness must satisfy the following properties:

1) Scalability: the process should be able to host a large number of workspaces while keeping the amount awareness information tractable.
2) Merge control relevant information: the process should provide the means to forecast when, where and whom will perform merges.

4.2. Scalable processes : Groups.

We have already mentioned that a cooperative work aims at producing a single result called the project's *reference copy*. Conversely, we call *work copy,* the copy that each developer can directly modify and that contains his/her work.

In our work, we made the hypothesis that the reference copy is hosted by a well identified workspace, called the reference workspace. The reference workspace

contains at any time, the actual visible result of the collective work. This hypothesis is often satisfied in practice, because software development is a continuous process where successive reference copies need to be produced, each developer feels more comfortable if is known where is the actual reference copy. In most usual systems, the data base plays the role of reference workspace.

We call a *group* a set of workspaces that have an explicit reference workspace, and where only the reference workspace communicates with the other workspaces.

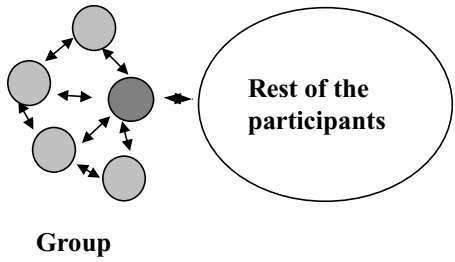

Fig. 5 Group

Groups are useful in practice because [12]:

- Groups easily match work division (it can be in charge to realize a task).
- Groups may use different concurrent engineering policies.
- Groups facilitate overall management, by hiding their operational details from the rest of the organization.

Groups easily match administrative and geographical divisions within an organization.

Interestingly from the awareness perspective, a group can also be organized hierarchically by allowing a workspace in the group to become the reference workspace of a sub-group. Such a hierarchy permits partitioning a task into sub-tasks. At Dassault Systems, the development of the Catia system uses a 7 level group hierarchy, allowing different types of concurrent engineering policies ranging from very strict at the top levels where are located the “Best So Far” and “Public” releases, to light constraints “in the trenches” : the development workspace groups at the bottom of the hierarchy [6][18].

From awareness perspective, a groups hierarchy based on tasks and subtasks is interesting because it makes groups a good candidates as the context for users to be aware of. Limiting awareness to a group activities raises the scalability of the system: the amount of information depends only on the number of workspaces within a group not on the total number of workspaces.

Modifications in the different workspaces of a group are usually semantically related because groups are created to perform particular tasks cooperatively, and therefore the likelihood to work on overlapping sets of files is pretty high. A group

member can examine his partner's modifications under the light of the group's mission, which provides an help in understanding its logic and intention.

4.3. Distance and States

Groups give scalability to awareness, but without a mechanism to make group awareness more precise about the forthcoming merges, the awareness information remains too broad and the developer has difficulties to measure to what extend others changes have consequences in his/her personal work.

We need a mechanism that indicates to the developer where and how divergences between his and somebody else work will be merged. Group awareness can be made richer by using the properties of the internal process.

We call *distance* between a source copy and a destination copy, the number of nodes (workspaces), along the cooperative graph, the first copy has to cross before to meet the destination copy.

Each time a copy moves from a node to the next one along the graph, the changes performed in the source node are potentially discarded and/or merged with the work performed in the new node. Therefore, for a long distance, the changes performed in the source copy are likely to reach the destination node much later and significantly transformed. The analysis of "far away" changes is likely to be at least partially irrelevant.

The shorter the distance, the more urgent and relevant the divergence analysis.

The distance is a good measure of the awareness information relevance; it is the main way an awareness system takes vantage of the cooperative process information.

As mentioned above, the cooperative policy defines when an through which node a change will move from its actual location to the next one. It is the combination cooperative graph, cooperative policy that defines the distance. The distance being determined by the cooperative graph and cooperative policy, the selection of a cooperative graph is of critical importance for concurrent engineering and awareness.

In the absence of cooperative policy (which is the usual case), with only the cooperative graph as information, the minimal distance is easy to compute, but the maximum distance in the general case is infinite, and the real distance unpredictable.

For example, on one extreme, for a graph being a line or a circle, the awareness system can computes distances, i.e. it can exactly forecast when a given change will reach a given developer. In that case, the information is only for information purpose since developers have no way to influence the distance. On the other extreme, without graph (each node is liked to all the others), and without policy, the minimum distance is always 1, but the real distance is totally unpredictable. This context makes the developed somehow nervous, since each change can be of maximum relevance (the distance can be 1) but since the real distance in unknown, the relevance is unknown; they are too many information to consider and no hint to know which one to consider first. The awareness information is useless.

4.3.1. The simple star topology

For the star topology, where the reference workspace is in the center, the minimum distance is one with the reference workspace, and two for the other workspaces. This simple topology is very much used, since it clearly identifies a unique workspace (the reference workspace) as being more relevant than any other one. If each group uses a star topology, the whole workspace structure is a tree of groups.

We define the concept of state of an artifact copy as

- The distance between that copy change and the reference copy,
- The distance between somebody else change and that copy.

In a star topology, from the point of view of a workspace different from the reference workspace, a change can only be in one of the following state :

- Distance to the reference : 0 (Unchanged) or 1 (Modified)
- Distance from a foreign change : 1 from the reference workspace (Obsolete), 2 from any other workspace (Changed).

In Celine, we call these states respectively modified, obsolete and changed, for user convenience, the combination (Modified / Obsolete) is called *Conflict* which means next move of that copy to or from the reference workspace will require to perform a merge.

For example, a user that is aware of a “conflict” state might decide to perform an early merge. A “changed” file might require no particular attention from the user, but a dangerous combination of “changed” and “modified” states might trigger the decision of one of the developers to stop further modifications of a file, or to both developers to exchange information to solve the emerging conflict early.

4.4. The extended star topology (Public and private workspaces)

The star topology is one of the most popular because it separate changes in two groups : those promoted to the reference workspace are implicitly made “official” and available to the rest of the group members, and those still in the developer workspace are supposed to be underway changes, with unknown level of stability and permanence. This difference is well expressed by the distance 1 with the reference, and 2 with the underway changes.

It would be interesting to have more complex types of processes within a group allowing more variability in distance to classify divergences as more or less urgent/relevant. The extended star topology is a good example of such a process.

In an extended star, each workspace is made of two parts : the first one (private) contains the current state of the developer’s work in progress, and the second one (public) holds the latest snapshot that the developer wants to share with the rest of the group. With respect to the simple star, this topology has the following advantages :

- The developer can make public a “stable” version of his/her work and continue working.

- Making public a version can be performed even when working off line.
- The promotion of the public versions into the reference workspace can be done at a later time, in an order defined by the policy or by an "integrator" working in the reference workspace.

In the reference workspace, the same occurs, the promotion occurs in the private area, and can be made public only after some work (e.g. validation) have been performed. In this topology the distances and states, are as follows :

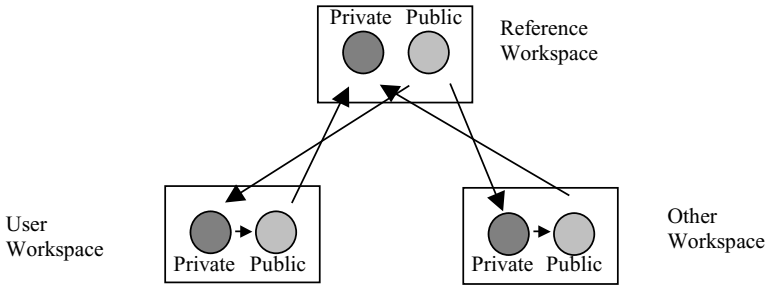

Fig. 6 Extended star graph.

- To reference workspace : 1 from public (Modified), 2 from private (Locally Modified)

From others workspace : 1 from public reference workspace (Obsolete), 2 from private reference workspace (Locally Obsolete), 3 from developer public workspace (Changed), 4 from developer private workspace (Loc*ally Changed*).

Clearly, the relevance of someone else change is well represented by the distance (four levels), but the number of states is now larger, especially if combined with the state of its own changes (three levels including the unchanged one) (4*3 = 12) . Even with this simple topology, the number of states increases pretty fast, making complex graph without cooperative policies intractable, falling back again in the cognitive overload mentioned above.

5. Policies, business processes and awareness

5.1. Cooperative Engineering Policies

A concurrent engineering policy is a process that enforces a particular strategy for concurrent engineering, by the definition of the path that can be followed in the cooperative graph, between two nodes. From a practical point of view, cooperative policies are implemented using a "Lock" primitive, which reserve the right to change a copy to a single workspace in a group, and controlling which operation (like promote, synchronize, publish, etc) are allowed given the circumstances. [6][11]

Whatever the implementation, a policy constrains the distance to a narrow range of values, if not a single one, even in complex graph. A policy is a way to use flexible and complex graph, still maintaining the distance into predictable are relatively small values.

If the awareness system knows the distance, as constrained by the cooperative policy, the number of states to manage can be small enough for proposing a good awareness system simple, relevant and useful. Indeed, In the absence of policy, only very simple graphs, like those presented above can be used, and awareness is a substitute to explicit policies : the visibility provided by awareness can be used by developers to decide themselves what is the policy to use at any point in time.

Except Celine, we do not know any system that propose explicit and high level cooperative policies, and that couples policy and awareness. This topic of defining and implementing policy in Celine was the topic of a previous paper [6], but our way to define policies will be modified, to improve and generalize the way policy and awareness can work in symbiosis.

5.2. Awareness and business models

Policy models are process models intended to control the data flow along the cooperative graph. Their main purpose is to make cooperative engineering safer, through a better control of merges, and incidentally to improve awareness functionalities.

Change control is the process that defines which change to implement, which groups to create to support these changes, with which policy and so on. It is our claim that change control would highly benefit from the existence of explicit cooperative policies, cooperative processes, and awareness, as presented in this paper. Our system Apel, since long, have addressed these issues [7].

Business process cover more high level processes addressing other topics, not necessarily related to changes. Our Apel system have been designed with these issues in mind, but is not discussed here.

It is our claim that these different levels of processes cannot be addressed in a single formalism, but benefit from each other, and should be connected in some way. Our work on Mélusine address this issues of heterogeneous processes interoperability [9].

6. System models

A system model is the definition of the relationships between the entities forming a system. In software engineering it includes the definition of the relationships between the different files present in a workspace. Some relationships may indicate a strong semantic dependency between files, meaning that a change in one of them, semantically, is also a change in the other one, called indirect change.

This information is of importance for awareness for two fundamental reasons :

- Changes are not only direct but also indirect changes,
- Granularity of operations like promote, synchronize or lock is not the file, not the whole workspace, but the transitive closure of the dependency operation.

System model information is very important for cooperative engineering control, since it increases the relevance of the information provided. Without system models, systems are taking conservative policies : the granularity of operations (promote, ..) is the whole workspace and only direct changes are considered by the awareness system.

We have defined an extension of the Eclipse framework for the complete support of system models and we are in the process of integrating Celine with this workspace system. We hope this experience will be reported in a future paper.

7. Celine

The Celine system has been built initially for the support of cooperative engineering policies [6][10]; once deployed we have realized that most companies are not yet ripe for policies, and prefer to rely on the developers expertise to handle cooperation, provided that the right information is available to them. This is why we have added awareness to Celine. We have first hard wired the star topology, then the extended star one, and integrated with cooperative policies.

Celine is daily used in production at STMicroelectronics by a group of engineers that with 70 workspaces (in average) collaborate on the development of large number (a few hundred) of microelectronic design files that for all purposes can be considered as software source code. Within those files, a kernel group of around 100 files is under daily modification. The number of concurrent modifications is high for the kernel files (50 concurrent changes in average), and the number of developers in the team make impossible to rely on informal communication channels to find out who is modifying what. Indeed, before to use Celine, that group was forced to lock file (using the synchronicity version control system), and therefore to avoid concurrent engineering, with high penalty in overall productivity.

In that team, the topology of the development process is a simple star. Celine provides each user with a view of the data that exposes the different state of each file, as explained in section 4.3.

In average, an engineer has 5 to 10 active simultaneous workspaces. To reduce the amount of awareness information, we encapsulate the workspaces of a particular user as a single source of divergences, displaying a single icon for a file even when present in more than one workspace. The Celine interface provides a way to find precisely who are the participants that have performed modifications and also to visualize the differences between any of them and the user's local copy.

Celine natively supports the star and extended star topology, on top of a CVS or synchronicity repositories. Celine handles and supports the basic operations (promote, synchronize, lock and so on), as part of its cooperative engineering policy (shown in the right menu bar in the above picture).

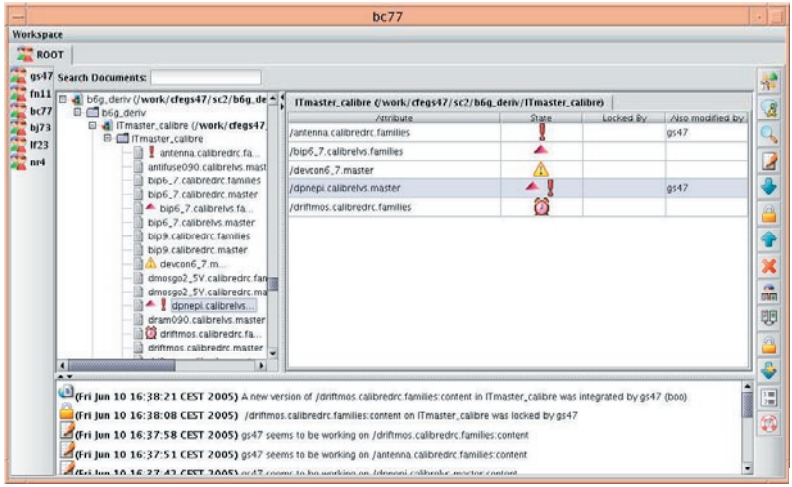

Fig. 7 Celine

Since arbitrary cooperative graphs are special cases of cooperative policies, we envision to extend Celine by a deeper integration with the cooperative policy support layer. The goal is to deduce automatically from the cooperative policy what are the possible states, and to generate a specific awareness support optimized for that policy.

Currently, Celine is in the process of being integrated with workspace supporting system model (as an extension of Eclipse). The current evolution of Celine is to support uniformly all the levels of process, individually or in combination, and to take vantage of the system model to generate an optimal and highly relevant awareness information.

8. Conclusion

Large software development projects need some concurrent engineering control to reduce the risks inherent to simultaneous modification of sensitive artifacts (e.g. source code). It is clear than a pessimistic (locking) strategy is not only too restrictive but also inadequate as it does not take into account semantic relationships among the different software elements.

In the absence of adequate means to control concurrent engineering, all the burden of dealing with copy reconciliation is on the shoulders of developers. Worse, conflicts are detected only when copies are merged, not when conflicts are created. Awareness has been proposed as a mechanism supporting concurrent engineering, providing developers with continuous insight of the team's activity. The hypothesis is that developers, with that information, will anticipate conflicts, and will reduce the probability of tricky or impossible merge, making cooperative engineering “safe enough”.

The above hypothesis does not hold in the general case, because it generates too much and not relevant enough information. As a matter of facts, these systems exist but are not used. We have shown that cooperative engineering relies on different models:

- Cooperative graph model (workspace topology)
- Cooperative policy model (allowed path along the graph)
- Business process model (partial order of activities)
- System model (data dependency)

Our claim is that, to be relevant and useful, an awareness system must take into account information coming from these models.

The cooperative graph introduce the concepts of group and distance. The group concept allows scalable system and solves the awareness cognitive overload issue. We have shown that an advanced awareness system can automatically transform the distance into a state displayed to the developer. We have shown why distance is the relevant information from cooperative engineering point of view, but distance can be computed only for simple graphs.

Cooperative policies complement the cooperative graph, allowing to control complex and flexible graphs, still maintaining short and predictability distances. Policies allow to provide a relevant awareness information even using complex cooperative graphs. Policies and awareness are two ways to provide cooperative engineering control; policies providing and explicit and imperative process, awareness relying instead on the developer expertise. Policy and awareness seem opposite, but our experience shows they are complementary, the policy providing the way to provide relevant awareness information, even in complex situations, which in turn allows to let large parts of the process under the developer responsibility.

The Celine system is, to our knowledge, the first system that provides awareness taking into account all the above models, and which is in daily industrial production. The experience so far shows that awareness along with the other models provide a very good solution to concurrent engineering; while each one of these model, alone, falls short to address the complexity and variability of the concurrent engineering challenge.

References

[1] Sarma, A., Noroozi Z., Van Der hoek, A.: "Palantir: Raising Awareness among Configuration Management Workspaces". 25th International Conference on Software Engineering. 05 03 – 05, 2003. Portland, Oregon

[2] Cleidson R.B. De Souza, David Redmiles, Gloria Mark, John Penix, Maarten Sierhuis. “Management of interdependencies in Collaborative Software Development”. 2003 International Symposium on Empirical Software Engineering (ISESE'03)

[3] Paul Dourish, Victoria Bellotti. "Awareness and Coordination in shared workspaces" Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)

[4] Cleidson R. B. De Souza. David RedMiles, Paul Dourish. “Breaking the Code, moving between private and public work in Collaborative Software Development" ACM SIGGROUP Bulletin Volume 24 , Issue 1 (April 2003)

[5] Dewayne E. Perry, Harvey P. Siy, Lawrence G. Votta. "Parallel Changes in Large Scale Software Development: An observational Case Study" ACM Transactions on Software Engineering and Methodology (TOSEM)

[6] Jacky Estublier, Sergio Garcia, German Vega. "Defining and Supporting Concurrent Engineering policies in SCM" SCM-11 May 2003, Portland, Oregon, USA

[7] J. Estublier, S. Dami and M. Amiour "APEL, an effective Process Support Environment." First International workshop on Multi Facets Software Process Engineering. September 22-23. Tunis, Pages 123, 137. Tunisia. 1997.

[8] Bischofberger W.R. Kleinferchner C.F. Mätzel K.-U. "Evolving a Programming Environment Into a Cooperative Software Engineering Environment" Proceedings of CONSEG '95, New Dehli, February 1995, pages 95-106, Tata McGraw-Hill, 1995.

[9] J. Estublier, G. Vega "Reuse and Variability on Large Software Applications” ESEC/FSE, September 2005, Lisboa Portugal.

[10] J. Estublier “Objects control for software configuration management”. 8 June 2001. CaiSE 2001, Interlaken, Suisse.

[11] A. van der Hoek, A. Carzaniga, D. Heimbigner, A.L. Wolf. "A Testbed for Configuration Management Policy Programming," IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 79-99, January 2002.

[12] M.H. Nodine, S. Ramoswamy, and S. Zdonik. A Cooperative Transaction Model for Design Databases. In Database Transaction Models, pages 59--86. Morgan Kaufmann, 1992.

[13] A. Skarra. "Localized correctness specifications for cooperating transactions in an object-oriented database". Office Knowledge Engineering, 4(1):79-106, 1991.

[14] T Mens “A state-of-the-art survey on software merging” - IEEE Transactions on Software Engineering, 2002. 28(5): p. 449-462

[15] P. Molli, H. Skaf-Molly and G. Oster. “Divergence awareness for virtual team through the web”. Proceedings of the integrated design and Process technology, 2002.

[16] C. Sun and C. A. Ellis: "Operational transformation in real-time group editors: issues, algorithms, and achievements, " In Proc. of ACM Conference on ComputerSupported Cooperative Work, pp.59-68, Seattle, USA, Nov. 1998.

[17] M. Beaudouin-Lafon and A. Karsenty. “Transparency and awareness in a real-time groupware system”. In 5th annual ACM symposium on User Interface Software and Technology. ACM Press 1992

[18] J. Estublier “Distributed Objects for Concurrent Engineering”. International Symposium, SCM-9, Toulouse, France, September 1999. Proceedings