Concurrent Engineering support in Software Engineering

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

Sergio Garcia  
LSR-IMAG  
220, rue de la Chimie BP53  
38041 Grenoble Cedex 9  
France  
sergio.garcia@imag.fr

Abstract

The evolution of Software Engineering methodology, from waterfall to spiral, from spiral to agile, indicates that high concurrency, iterative development and short cycles are key factors for effective Software Engineering. It is widely accepted that supporting (i.e., formalizing controlling, automating and optimizing) concurrent engineering processes is needed to increase predictability of cost, quality and development time.

Unfortunately, current systems (e.g., workflows, Software Configuration Management ...) are too simple and deterministic; they do not include real support for concurrent engineering. We claim this shortcoming is one of the major reasons why current workflow and process support do not significantly help in the support of software engineering.

In this paper we present the Celine system, which extends workflows with the definition of high-level executable description of concurrent engineering and therefore contributes to provide effective control over cost, quality and development time.

1. Introduction

Software engineering lacks the same level of predictability of cost, quality and development time as most engineering fields. This is partially due to an insufficient understanding of the development process and especially of how to deal with a large number of participants working simultaneously.

Mastering the process requires making it explicit using a formalism with clear semantics. A process description (a process model) is a vehicle for discussion and analysis that brings deeper understanding of the different process issues [14]. A process model is also needed for execution in order to enforce, automate or assist the process [3].

Workflows and software process support are technologies aiming at making processes explicit. The workflow engine “executes” an explicit description of the process by guiding the user to perform the different activities in an expected order [19].

Software engineering is characterized, with respect to other engineering disciplines, by an unprecedentedly fast and deep evolution of techniques, tools and needs. Many tools, techniques and methods are obsolete before reaching their maturity and practitioners routinely face novelty (“I never did that before”), uncertainty (“I do not know how it works”), instability (“it is buggy”), and requirements evolution (“I want this new feature”). This context explains why the traditional sequential and deterministic waterfall model, used in virtually all engineering disciplines, did not satisfy the software engineering needs. It was gradually replaced by methods that better fit the needs of software engineering: spiral, XP and now agile, which emphasize the fact that simple improvements have to be done and tested immediately, before planning the next improvement toward an ever moving target, using an ever moving technology.

Therefore modern software processes have the following fundamental characteristics:

• Many short cycles,
• High concurrency, and
• Non determinism.

On the other hand, workflow technology emphasizes processes with strict ordering, predictability and repeatability. A typical workflow model is the following [5][9][10]:

Fig 1 A typical workflow model

Usual workflow systems assume that:

• A process is a partial order of steps to be executed in order; completion of the last step means the process goal is reached.
• Processes are a fully repeatable sequence of steps.
• The product, goal of the process, is considered finished at the end of the process and, therefore, keeping its versions is usually not a concern. [1]
Most often, workflow systems use a unique copy of the data, stored in a global store, and either forbid concurrency or leave concurrency unmanaged [7].

In contrast in software engineering:

- The goal is never reached and actions are undertaken in apparently a non-deterministic way.
- Concurrent processes are hardly deterministic; each execution is significantly different.
- The product, goal of the process, is never finished; keeping intermediate versions of the product is a critical issue.
- Many copies of the product under way are continuously and concurrently modified.

In modern software engineering, the different phases are repeated continuously and often overlap so that the software can evolve incrementally, until the end of the product life (if successful, 10 to 30 years). No step definitely finishes the process and keeping the different versions of the product is a major concern [4].

Fig 2. A typical concurrent engineering process.

In figure 2, a typical concurrent engineering process is shown, where each developer contributes by sending his work to a central person and by taking from the central person the current state of the product under way. This is a classic CVS (Concurrent Version Systems) of SCM view, where the central node is simply a database.

Clearly, workflows do not satisfy modern software engineering needs. The only common aspect in both camps is that concurrency, if allowed, is left undefined and, therefore, not explicitly supported.

In this work, we will present our approach that makes concurrent engineering processes explicit and controllable. This approach includes a language for the definition of concurrent engineering policies. In section 2, we describe the limitations of current process technology, then in section 3 we introduce our language, illustrated with different examples, and finally conclusions are presented.

2. Concurrent Engineering and collaborative processes

The common vision of a collaborative process in disciplines such as software engineering is that of a group of humans having a collective duty (for example, to provide the next release of a software) and working concurrently, but in a coordinated way, to produce a common unique result (the next release).

Concurrency is hard to avoid in collaborative processes because not allowing people to work simultaneously on the same data would imply either that the participants have to wait to get access to that data or that they have to work on different and independent pieces of the same software. Unfortunately this is unrealistic because, in software engineering, strong and often unknown dependencies exist among most parts of the software [18]. In practice, it is not possible to split software in a large enough number of independent parts.

No concurrency results in great loss of time and increased delays; something that cannot be accepted under ever increasing time-to-market constraints. Conversely, concurrency increases the amount of merges, which increases the risks of getting an inconsistent result. The concurrent engineering dilemma is to find the optimal compromise between high speed (and high concurrency) and low risk (no merges, but low concurrency). Concurrent Engineering (CE) support aims at increasing concurrency while limiting the risks related to merges.

2.1 Merging issues

Fig 3. Merging

Concurrent modification of data A will produce data B and C; the production of the common end result will require reconciling B and C into a single data D: it is the merge operation. A merge is expected to take two copies, B and C, that have a common origin (or ancestor) A, and to produce a new copy D that ideally is what would have been produced if C changes had been performed on B (it is the usual database serialization criteria).

Usually software merge is assisted by a “merger”. Currently, the popular “mergers” assimilate software source code to a list of text lines and simply add and remove lines based on the fact these lines are or not present in the ancestor [1]. This proves to be effective as long as modifications are small and independent, but is clearly insufficient in the general case. Mergers that take into account the semantics of a particular programming language exist, but they do not eliminate the risk of an inconsistent result (they only ensure that the program will compile). These “semantic mergers” are not widely used, which suggests that their improvement over simple text-based mergers is not
significant enough to sacrifice a more general tool. In general, a merge requires the participation of a human with the necessary expertise and understanding of the involved transformations. Even when performed by the right person, a merge remains an error prone operation. Merging is necessary to produce a unique common result from concurrent changes but, in the general case, there is no way to prove that a merge is correct. Merging introduces a risk of inconsistency and is the main limitation to concurrent engineering. Because merges represent a major inconsistency risk in CE, a CE process must include features to closely control which merges are allowed and which are not.

We call merge control that part of a CE process that allows designers and team leaders to define explicitly, beforehand, which parts of the software can be changed concurrently, by whom; who and when these changes will be merged.

For any engineering based on multiple copies and isolated workspace, a large part of CE control is merge control. This is clearly different from other CE approaches, like Group Support Systems [21] focusing on interaction patterns but not on its direct consequences on product evolution.

It is easy to realize that, currently, no environment provides support for merge control and, therefore, no real control over concurrent engineering. For example, SCM systems provide isolated workspaces containing copies of the shared data and mergers, but do not propose any concurrency control or merge control. Process control in SCM is limited to change control based on a workflow. Concurrent engineering is possible, but since there is no merge control, concurrent engineering is left to the responsibility of the practitioners.

It is interesting to mention that we do not know of any engineering domain supporting concurrent engineering based on copies and merges. For example, in Computer Supported Concurrent Work (CSCW)[13] technology, either the smallest modifications are propagated as soon as they occur making all the users work on virtually the same data or the shared document is split in independent parts, where each user works on a different part. This approach eliminates the need for merges, but does not provide the isolation property required in software engineering. In Product Data Management (PDM), probably due to the complexity of the data model, the concepts of working copies and merges is not supported, which makes PDM not suitable for software engineering [6].

We believe that predictability of cost, delay and quality in software engineering can only be approached if the software engineering process is mastered in both its deterministic (workflow), non-deterministic and concurrent aspects (merge control). Our goal is to propose process support that extends usual workflow like formalisms (for the deterministic part of processes) with merge control (for the non-deterministic nature of concurrent work).

3. The basic Celine concepts

Celine is our environment for the support of concurrent engineering, specifically tailored for software engineering. Celine defines CE as follows.

- **A common result.** Concurrent engineering refers to the work of a group of persons working together to deliver eventually a common unique resulting artifact called the reference. Virtually all systems agree that software CE requires:
  - **Asynchronous collaboration.** Software developers need an isolated copy of the software, called a workspace, to perform their usual activities (changes, compilations, tests, etc.) without being disturbed by external changes.

In software, CE takes the form of data transfers among workspaces. Without additional concepts and mechanisms, any transfer is possible, which results in what can be qualified as a chaotic process [Fig 4].

The issue is to transform this chaotic situation into controlled CE. Based on our experience, the first issue we have identified is scalability. Indeed, the number of potential data transfers increases very fast with the number of persons involved. For groups containing more than a handful of persons, there is clearly a scalability issue.

3.1 Scalability: the group concept

To address the scalability issue, Celine adds the following requirements:

- The reference, at any point in time, is found in a specific unique workspace called the reference workspace.
- A group is the set of workspaces sharing the same reference workspace.

With these two new requirements, a workspace pertains to a single group and communication only occurs among members of the group. Therefore, seen from outside, a group can be abstracted by its reference workspace, as if it were the result of a single person. Conversely, any workspace in a group can be the reference workspace of a lower level group. In this way, the complete organization can be organized in a
tree-like structure where nodes are reference workspaces. This brings scalability to the number of participants that can work on the same project.

This concept of group and reference workspace proved to be the key to CE control in large organization, without really bringing limitations. Indeed, whatever the organization size, the issue turns to solve CE inside a group. Interestingly, each group can support a different CE policy; it is the duty of the CE system to connect groups, still satisfying their CE policies.

For example, in our Dassault Systems customer, 1200 engineers are working simultaneously on the same software (CATIA). The 1500 simultaneous workspace are structured into a tree of groups having a depth of 8; each group containing only a handful of participants. The group at the top of the hierarchy supports very strict CE policies for the validation and management of the customer release (at the real top), in the middle of the tree, the usual SE work is performed, with “good” CE control, at the bottom of the hierarchy is found groups developing prototypes or performing trials, with very relaxed CE policies and constraints.

In the remaining of this paper, we will focus on how CE can be defined and supported inside a single group.

### 3.2 Concurrent engineering policy

Even with a handful of participants, without rules, transfer of data can occur at any time between any pair of participants and therefore, merges can occur anywhere at any time, which means that there is no CE control at all. This is why we call it the chaotic process, or absence of CE policy. To eliminate this chaos, a CE policy can constrain transfers in two ways:

- **Statically**: which transfers are valid?
- **Dynamically**: when are transfers valid?

The static part defines the communication graph, which is the deterministic aspect of a CE process. The dynamic aspect builds over the communication graph, controlling when people can communicate and merge their work, controlling in that way concurrency, non-determinism and merges. The static and dynamic constraints over the process correspond respectively to:

- Communication graph: It defines the legal flow of data among workspaces, according to the kind of activities undertaken in these workspaces, i.e., their role.
- Merge rules: Merge rules define the required conditions for merges to be legal; indirectly controlling when and which work can be performed and transferred.

Merge rules, described in more details later, define “only” what are the valid merges in the form of a triple \{A, B, C\} meaning that work performed in workspace A and work performed in workspace B can be merged in workspace C.

We believe that these definitions and concepts are very general and cover the whole spectrum of CE needs. The next section gives more information about how our Celine system works.

Fig 5. Concurrent Engineering=graph+merge rules

### 4. The Celine system

Celine is based on a few basic concepts: logical entities transferred between workspaces following a communication graph and satisfying merge rules. We define below these concepts.

#### 4.1 Workspace

In this section we describe how Celine implements the principles shortly introduced above. In particular, we will define the operations through which the developers perform and communicate their work and then we will present our concurrent engineering language.

In Celine, a **workspace** is defined as the union of a **working copy** and a **repository**.

- The **working copy** is a place in a file system that contains the software copy that the workspace owner can change freely, and on which the different development tools (editors, compilers, etc) can operate.
- The **repository** contains a sub-set of the successive states of the working copy, stored as an immutable sequence of revisions.

Developers interact with their workspace through three basic operations: EDIT (editing, creating, renaming … files of the working copy), LOAD, which replaces the value of the current working copy by the value of one of the revisions stored in the associated repository; SAVE, which creates a new revision in the repository with the actual value of the working copy.

Communication among participants is enabled by
the operation TRANSFER, which copies a logical entity from a SOURCE workspace repository to a TARGET workspace working copy. A TRANSFER can result in:

- a copy of the SOURCE last revision in the TARGET workspace, if SOURCE is a modified copy of the current TARGET working copy,
- a merge, if both SOURCE and TARGET are different modifications of a common origin (called the common ancestor), or
- nothing, if SOURCE and TARGET are identical, or if the TARGET supersedes the SOURCE.
- A transfer is called promote (or push) from the source workspace point of view, and synchronize (or pull) from the target workspace point of view.

![Diagram of transfer between workspaces](image)

**Fig 6. Transfer between workspaces**

Celine proposes as default a standard 3-way text merger, but user defined mergers can be provided for specific kinds of data. For certain types of data (such as sets with add and delete operations) a merge function is known to perform always correct merges; in that case, concurrent engineering is less critical. For software source code, text mergers usually perform a good job, but human intervention is generally needed, if only to check the result of the merger. In some other cases, merges are always manual, such as solving the simultaneous rename of a file. Celine assumes the worse case scenario, where human participation is required.

Celine provides a user interface to perform SAVE, LOAD and promote / synchronize operations. Enforcing a policy takes the form of allowing or disallowing changes in a working copy (EDIT not allowed) and TRANSFER operations, based on the knowledge of the operations performed in all workspaces, the communication graph, the roles and the policy definition.

In the next three subsections we will present our language based on three fundamental concepts: logical entity, communication graph and merge rules.

### 4.2 Logical entities

In most SE environments, software is simply a set of files while software relies on logical objects rather than physical files. This mismatch explains many of the limitations found in actual SE environments. For example, in most version control systems the system can avoid concurrent modification of files (using locks), but there is no way to protect a consistent set of files (a logical object) from concurrent changes. There are many examples of such logical objects: a “package”, a “module”, a sub-system, etc [18]. In some advanced SCM systems, logical entities are defined, but no CE policy [20].

Celine defines a logical entity as a named set of files. Logical entities are not necessarily disjoint; they can overlap or contain other logical entities. In Celine, the basic operations (TRANSFER, SAVE, LOAD, EDIT) and policies are defined on logical entities not on files. Since development tools operate on files (editors, compilers, mergers, etc.), we need a function that maps logical entities to their corresponding files and vice versa.

Our language assumes a user-defined system model and its corresponding mapping function between logical entities and files. For user convenience we provide, as default, a simple language based on regular expressions that allows the definition of logical entities. This is clearly a weak replacement for a real system model and its corresponding mapping function, but used with good file naming conventions, it proves to be a viable intermediate solution.

Examples of such expressions include:

- `sources : src/*.java`
- `package : src/*/.*.java`
- `plugin : src/interface/*/.*.h, src/implementation/*/impl.cpp`

The first example defines `sources` to be the logical entity made of all the Java files directly below the `src` directory; the second example defines `package` to be each set of Java files found in the directories below the `src` directory. `Plugin` designates those pairs of files for which “*” matches the same string in both expressions. Note that `package` is a label for a set of (similar) logical entities and that `sources` is a `package`.

### 4.3 Roles and Communication graph

We introduce the concept of Role, to type workspaces according to the kind of activity(ies) for which they are intended. For example, integrator workspaces could be in charge of integrating the contributions coming from other workspaces, documentation workspaces could be used for documenting the code, test workspaces...

Constraining the exchange of data among

---

1 For convenience, a file is by default a logical entity. Therefore, if no logical entity is defined, Celine works, as all current SE environments, on files.
workspaces to well-defined paths is represented by a graph where the nodes are workspaces of a given role and the directed arcs are the “legal” communication channels. This looks like usual workflow models, except that:

- Nodes are workspaces, not activities.
- Before and after a TRANSFER, the SOURCE and TARGET workspaces work in parallel.

In Celine, only the data flow (transfers) is controlled; the control flow is not explicitly defined, but is the consequence of the merged rules (see later). The definition of a concurrent engineering policy begins with roles declaration.

\[ \text{[Read-only]} \text{role-name *:{logical_entity}} \]

Roles have an associated cardinality that indicates either 1 (simple role, the default) or * (a multiple role), which indicates that any number of workspaces can play that role at a given point in time. Every policy must include a simple role named reference for the reference workspace. Finally, each role declaration is followed by the logical entities that the workspace can contain (by default all). The reference role must always contain all the logical entities. By declaring a role as read only, the system can optimize the execution of the policy by anticipating that there will be no modifications on workspaces of that type (and prohibiting changes as well).

Once the roles are defined, the communication graph is defined in the following way:

\[ \text{role}_1 \rightarrow|<->\text{role}_2 : \{\text{PUSH}\} \text{logical_entity} \]

This means that TRANSFERS are legal from a workspace playing role_1 to a workspace playing role_2 (and vice versa, if the <-> sign is used), for the logical entities in the list.

The most basic example of a communication graph is the star topology (see figure 2), where participants can only communicate directly with the reference workspace. This topology, used in many SE systems, is interesting in that the reference workspace reflects well the real state of the collective work, since all exchanges among developers have to be in the reference. A CVS strategy is described as follows:

\[ \text{Roles} \{
  \text{reference ;}
  \text{developer * ;}
  \text{general-test ;}
  \text{test-platform ;}
\}

\[ \text{Graph} \{
  \text{reference <-> developer ;}
  \text{reference -> general-test ;}
  \text{general-test -> test-platform ;}
  \text{test-platform -> reference ;}
\} \]

\[ \text{Fig 8. A communication graph.} \]

Finally, it can also be indicated who can take the initiative of a transfer (the sender (promote) or the receiver (synchronize). In our system, a promote only creates a new revision in the local workspace repository and sends a notification to the TARGET workspace owner, who is free to perform the corresponding synchronize when convenient. The PUSH declaration explicitly asks the system to perform the transfer at the SOURCE initiative (promote does perform the transfer, as it is the case with CVS).

4.4 Merge rules

The communication graph defines the valid paths or the transmission of work, but when (seems like this should be, “what work can be done”) can that work be done and when can it be transferred is left unclear at that level. Without explicit handling of that part of the process, concurrency remains uncontrolled and unpredictable. Merge rules indicate what merges are valid and, indirectly, when transfers are legal.

Merges and the order in which work is done and transferred are tightly related, the former depending strictly on the later. We support explicit declaration of merges instead of declaring concurrency (i.e., which workspaces can work simultaneously on what objects) based on the following considerations:

- The merge is the main concern.
- Declaring concurrency leaves certain aspects of the merge, notably where and which work can be merged, uncontrolled.
- It is virtually impossible, by hand, to make the relationship between merges and concurrency.
- It would overly constrain concurrency.
In contrast, by controlling the merge, the environment can deduce how to allow maximum concurrence for
the specific merge constraints, which is usually what is
wanted. Merges can be allowed based on the following
parameters: In which workspace merges are possible
and which work is being merged. The syntax of merge
rules is as follows:

role_1, role_2 : role_3 {logical_entity}

The rule means that the modifications on a
logical_entity performed in workspaces playing role_1
and role_2 can be merged in workspaces playing
role_3.

A merge is valid only if a rule explicitly mentions it.
By default, merges are not allowed (no concurrency).
As with the communication graph rules, if no logical
entities are mentioned, it defaults to all the logical
entities.

4.5 CE policies examples

Combining even a simple communication graph like
the star with merge rules provide a large set of
meaningful policies, as we shall illustrate.

CVS is a popular version control system, which
keeps in a central repository the different revisions of
the products to be controlled. CVS forbids any merge
in the repository; therefore, the developers are allowed
to promote only if it does not imply merges in the
repository, otherwise he/she must first synchronize
performing a local merge, and then promote. This
policy is defined as follows:

Roles {
    read-only: reference ;
    developer * ;
}
Graph { reference <-> developer }
Merge{developer,developer: developer;}

If no merges at all were allowed (removing the Merge
line), the policy would be the equivalent of a
standard “locking” strategy, where two copies of a
logical object (files) can not be modified
simultaneously.

It must be noted that the CVS system model is the
file system; consequently, the software logical entities
are not protected from merges, only the files. This is
equivalent, in our language, to not declaring any
logical entities at all. The rule

Entity {software : * }

Declaring that the whole software is a single logical
entity (called software) makes the system prohibit any
promote to the repository as long as there are files
changed in both the repository and the promoting
workspace, even if they are not the same files. It avoids
a common source of inconsistency found in CVS-like
systems.

Another popular star-based policy is when a person
is in charge of the reference copy and, therefore,
merges and modifications can (and must) be performed
in the reference workspace. This policy puts the owner
of the reference workspace (an integrator) in charge of
all merges:

Graph { reference <-> developer }
Merge {
    developer, developer : reference ;
    developer, reference : reference ;
}

In this policy, a promote performed by a developer
“only” stores his working value in his own workspace
repository and sends a notification to the integrator,
who then is free to decide in which order to perform
the integration (calling synchronize).

Both policies (all developers promoting directly to a
central repository or a single integrator with access to
the repository) are used frequently in many
organization, without any formal declaration. In the
first case each developer assumes the responsibility of
checking the consistency of his results after a merge. In
the second case, a special developer (the integrator)
receives changes from the other users and is the only
one allowed to change directly the reference copy
(usually by checking in a new version in a version
control system). Our language allows for the
formalization of both strategies in a few lines.

The star topology can also combine different roles,
for finer grained control. For example, it might be
useful to separate the software system into the kernel
and regular logical objects and to define the Kernel
and Developer roles, accordingly.

Roles {
    Reference ;
    Developer * ;
    Kernel * ;
}
Graph {
    Reference -> Developer ;
    Developer -> Reference : regular ;
    Reference <-> Kernel ;
    Developer -> Kernel : kernel ;
}

This policy says that developers, if they change a
kernel logical entity, must promote it toward a kernel
workspace, before that change can reach the reference
workspace. This policy says that only kernel
workspaces are entitled to validate changes into kernel
entities.

It is very interesting to realize that even on the graph
topology; which is the simplest we can find, a very
large number of policies can be defined. Even with a
single developer role, and no logical entities at all,
many different policies can be defined and
surprisingly, they all make sense and are used
somewhere. For example, we could want the
developers to wait for their changes to be integrated
before continuing to work, or to synchronize their
workspace before continuing to work, or to require all
developers to synchronize to the save version before to continuing to work, or to give to the integrator the decision to synchronize the developer workspace, or to allow promotes only if there is no risk of conflicts in the reference workspace etc. We do not know any system that approaches this flexibility, even on a predefined topology, without roles or entities.

If different logical entities and roles are defined, an infinite number of different policies can be defined, still on the simple star topology.

5. Policy and awareness

Policies can be supported by the software environment by enforcement and/or by awareness.

Enforcement is the active participation of the software environment to guarantee that the process is executed as planned. This enforcement is implemented by allowing or denying certain operations at a given time, according to the process model.

One problem with the enforcement approach is that some policies might be “ideal best practices,” but too constraining in practice, which may result in users rejecting the tool either because in some special cases they have to violate a rule (panic mode, etc) or because they know that the rule can be safely violated. To overcome this problem, we have worked on the second approach for the support of CE policies: group awareness. The awareness strategy consists of giving the user the necessary information for him/her to decide when to transfer work, what files not to edit, etc. The assumption here is that it is the developer who better knows the risks of concurrent engineering and how to deal with them in each particular case [11][12]. What he/she needs, is to have a clear view of what is happening in the developer group in order to make the right decisions about when to transfer data, whether to edit or not a certain file, etc. In this approach, the role of the software development environment is to provide the relevant information and only the relevant information that will help the user to follow the rules, … if possible. The awareness system makes use of the process information to filter the potentially thousands of events down to the information that is useful for each developer. The policy enables discovering what is the “relevant” information. Again, what is relevant, from a given user point of view, is either what merges he/she will have to perform or merges that somebody else will have to perform over his/her own changes.

For example, at STMicrosystems we have put in service a version of Celine that provides awareness. The policy definition and the system implementation permit Celine to always “know” everything about forthcoming merges and the risk of merges. The system is based on the concept of distance; the distance being the number of transfers that are needed before a merge is to be undertaken. Celine associates a different icon to each distance and displays the corresponding icon in front of each file. Celine, on demand, provides all the information on a forthcoming merge: who did the changes, when, which changes, what other workspace did other changes, how long before to have to merge and so on.


We believe that it is because merge control is at the heart of the system that we can solve the issues related to CE. It is a major departure with respect to the traditional view of workflow and processes. The “CE process” is in fact the set of all processes that satisfy the merge rules. This set of processes is not made explicit, because it is very large, often infinite, and because it is simply not relevant. As defined above, what is relevant is to allow the maximum concurrency, while controlling the risk of inconsistency that comes with merges. It is exactly what the system does, nothing more and nothing less. The process, in its traditional view, is a means, not the goal. Traditional processes not only do not describe what is relevant, but they also overly constrain concurrency, do not support non-determinism, and do not fit well with short cyclic processes. We believe that our system addresses the issues related to CE:

- Non-determinism. The system does not prescribe or impose when transfers are made, nor any sequence of actions, nor any ordering, but “simply” prohibits those transfers and changes whose (potentially far away) consequence will lead to an invalid merge. Any action that does not violate, now or later, a merge rule is valid. In practice, CE processes are highly non-deterministic.
- Cyclic process. The fact that a reference workspace exists at any point in time fits nicely with cyclic processes. A cycle is “simply” the set of activities undertaken by a group between two successive specific states of their reference workspace (like “tested”, “valid”, “official”).
- High concurrency. It is the direct consequence of the presence of merge rules. Only concurrent actions that will eventually violate a merge rule are prohibited. The system guarantees the maximum possible concurrency that satisfies the merge rules.
- Deterministic process. A “classic” workflow can be seen as a special case, in which the communication graph is isomorphic with the workflow model (role == activity), with no merge at all, and a single cycle.

A major contribution of the approach is that the user describes only his/her problem: which merges are to be avoided; not how his/her problem can be solved; they
do not deal at all with low-level features like files, versions, branches or locks.

6.1 Implementation and Technical issues

In Celine, a policy is enforced “simply” by prohibiting executing actions that will eventually produce an illegal merge. Compiling a policy consists of computing a state diagram in which states are sets of triples (A, B, C), where A, B and C are workspaces and the triple means that a change performed on A is currently in B and not yet in C. A transition is the consequence of one of the three operations edit, promote, and synchronize. On this state diagram, states corresponding to illegal merges are identified and all routes leading only to these states are illegal. The transitions that lead to one of these illegal routes become illegal and the operations that produce these transitions are illegal. To prohibit edit, locks are set on files only when a transition is to be prohibited and promote and synchronize operations are simply disallowed.

This computation is done once for all when the policy is compiled; at that time the whole state diagram is computed and simplified. This allows for a number of validations: inconsistent policies are identified, deadlocks, dead end (there is no route toward the reference workspace). From the state diagram, the system computes the concurrency level and detects bottlenecks (no concurrency), and produces a report showing the characteristics of the policy in term of concurrency and control flow. It allows the designer to improve the policy. The implementation showed us that it is virtually impossible to compute manually how to transform merge rules on logical entities into a file lock policy and concurrency control; conversely, to compute the consequences of a concurrency description in terms of merges. It clearly shows that automated support is required.

6.2 Current state

This language is the result of observation of the informal strategies for dealing with concurrent work that arise in the real world, as well as of the kind of communication that occurs between programmers and project managers about their processes. This observation comes from the experience accumulated by our team when collaborating with different industrial partners, starting with the collaboration with Dassault Systems on the development of the ADELE system. At Dassault System, thousands of workspaces coexist simultaneously to develop CATIA and strict policies are necessary.

The current implementation of Celine is in production at STMicroelectronics. Celine can be used with “any” version control system, but currently supports Synchronicity, CSV and Subversion. The version in exploitation emphasizes awareness, which proved to be a valuable service, especially when developers work at different geographical sites, as is the case at STMicroelectronics. Discussion with the users has also been a source of inspiration for the current state of our language. Celine supports a concurrent engineering policy engine based on our previous work [8]. We are currently developing a new engine that enforces the concurrent engineering policies as presented in this paper.

6.3 Future work

Future work includes using the state diagram to compute the state and distance for better awareness support. More precisely, we envision an interactive system that helps the designer to identify what are the relevant states and how to show them to the developer. Then, the system shows to each developer, at any point in time, what is the state of the system relevant to him and forecasts, sometimes far ahead, the merges he/she will have to perform. This is a generalization of the current awareness system supported by the Celine version currently in daily production at STMicroelectronics [4].

7. Conclusions

For workflow systems, a process is a repeatable and deterministic partial order of predefined steps, in which the satisfaction of activity ordering is key (the control flow). For concurrent engineering, a process is a non-deterministic set of concurrent activities performed on copies of the same data with the aim of producing, repeatedly, a single common result. For concurrent engineering processes, merge control is key.

We have shown that currently no system we know of provides significant concurrent engineering support. We believe that this is due to a lack of understanding of the real issues and concepts behind concurrent engineering. This lack of understanding is such that the only mechanisms provided are based on manually setting locks on files, which is much too low level to implement a meaningful CE policy.

We believe that these two classes of processes are the two extremes of a full spectrum of processes and that software engineering needs to control processes pertaining to the entire range, not only “classic” workflows. We believe that it is because the current process technology cannot define and support processes pertaining to that range that it has not been successful in software engineering.

We believe our work contributes in:
- Defining the underlying principles of concurrent engineering processes based on multiple copies.
References

[11] Sarma, A., Noroozi Z., Van Der hock, A.: "Palanitr: Raising Awareness among Configuration Management Workspaces". 25th International Conference on Software Engineering. 05 03 – 05, 2003. Portland, Oregon which merges are to be avoided, not how his/her problem can be solved (locking strategy …). From the declared desired properties, the system computes the maximum possible concurrency that satisfies these properties. Not only does the introduction of this formalism open the door for better computer aided support and enforcement of a wide range of processes, but it also provides a vehicle for discussion and reasoning about software engineering processes at a high level of abstraction. Our work reconciles workflow, software process, CSCW into a common high-level formalism in order to homogeneously support the wide range of processes found in software engineering. We believe this is a key achievement for satisfactory support of software engineering processes.