Improving Cooperation Support in the EPOS CM System

Jens-Otto Larsen, Bjørn P. Munch, Reidar Conradi
Dept. of Computer Systems and Telematics,
Norwegian Institute of Technology, N-7034 Trondheim, Norway.
Email: {jensotto,bjornmu,conradi}@idt.unit.no

Patricia Lago
Dip. di Automatica e Informatica,
Politecnico di Torino, I-10129 Torino, Italy.
Email: patricia@polito.it

Abstract

Support for cooperation is important for the users of a Configuration Management (CM) system, whenever two or more people are simultaneously working on the same product. However, many CM systems today are still lacking in this respect. We have added some cooperation support tools to our own CM system, and we show how these assist the CM users in coordinating their work and resolving conflicts. While our implemented tools are tailored for our home-grown environment, the techniques and general approach are applicable also to other CM systems.

1 Introduction

One of the problems in Software Configuration Management (SCM) regards cooperating work on large systems. A substantial part of the total software costs is associated with maintenance and further development. On large software systems, there are usually several people doing development simultaneously, and their work is not always independent.

A programmer may want to work in isolation, to avoid surprises when e.g. a library or a common header file is suddenly changed by someone else. On the other hand, (s)he may get in trouble when the work is finished, and turns out to be "wrong" because it is based on an outdated version of the same library/header file. Other kinds of problems arise when two developers want to change the same part of a product concurrently. We may want to let them do that, so that both can go on with their work.

Situations like these inevitably show up in large development projects. Of course, it is always a good idea to try to organize the work in such a way that this does not happen too often, but it is not possible to avoid it totally without putting severe restrictions on the work. Instead, an SCM system should give support for *planning* for, *detecting* and *dealing* with such conflicts, rather than trying to *avoid* or *work around* them at all costs.

To aid cooperation between users of a CM system, one should not only consider access control to objects, but also provide users with information that can be used to e.g. plan activities.

The EPOS CM system (ECM) is built on top of a general, versioned database (EPOSDB) and offers a set of commands to access software components stored in the database. We have added functionality in a number of areas to support cooperation among ECM users: The underlying database has been enhanced to allow more flexible access to data and we have implemented a set of tools that is used to enter and retrieve information.

The rest of this paper is organized as follows: Section 2 explains the problem of cooperation in CM, mentions some related work, and introduces our approach. Section 3 provides a general overview of our existing CM system. Section 4 explains how we have added different types of cooperation support. Section 5 sums up the main points of our contribution, and suggests some further work.

2 Cooperation in CM

The main cooperation aspects are identified in [BJP94]:

Collaboration demands that involved agents share common data/ information, and is often based on a shared database to handle access to, and storage of software artifacts.

Communication refers to the exchange of information and messages among agents. It usually follows a (set of) specified protocol(s).

Coordination means the adjustment of agents' work towards a common goal. This happens through an explicit process model to manage organization, evolution and consistency of the whole software project.

The most obvious starting point when trying to add cooperation support to an SCM system is to focus on software objects and access to these. Most SCM systems have a shared database of software objects, and use some mechanism to synchronize object access. Some systems provide transactions to support concurrency control, recovery, and a way to encapsulate larger engineering activities. Few systems provide facilities for communication between users or for user coordination at a higher level than object access. With this focus in mind, we would expect that further cooperation support will be concerned with communication about the objects managed by the system and that declaration of activities/transactions that need coordination are centered around the objects they will access.

2.1 Related Work

Traditional DBMSes have a strict consistency (or integrity) concept, related to the data model and provide concurrency control and recovery for programmed transactions [Gra81]. The most commonly used correctness criterion is *serializability* which can be guaranteed in a number of ways, e.g. by following the two-phase locking protocol.

Software engineering involves large data sets at client sites and long update times. Since the scope and sequence of updates are hard to predict and may involve overlapping/versioned subsystems, traditional locking procedures may cause intolerable delays. Thus, software engineering – like concurrent engineering in CAD/CAM and VLSI – also requires support for long-lasting and user-controlled transactions, often called *design transactions* [KKB85].

In addition to the integrity constraints implied by the data model, a software database will often pose additional, user-defined consistency constraints on software objects, e.g. on object state. A last item worth mentioning is that software objects are interdependent and that a change to one object may affect other objects, e.g. due to file inclusion.

A standard work procedure when serializability is not applicable, is to check-out copies (i.e. temporary versions) of shared data at project start, perform the updates locally, then check-in, and perform final merging/reconciliation/integration. A similar situation will arise if versioning mechanisms are actively used to increase concurrency in a system: a number of versions is created and these must be integrated at some later stage. The problem with these approaches is that without a formal criterion for correctness of a concurrent execution of transactions, we are not able to reason about the inconsistencies that may arise.

Some newer database transaction models are treated in [BK91, Elm92]. These models try to increase concurrency in the systems by using less restrictive correctness criteria. Most of the new transaction models propose *nested* transactions [Mos85], or grouping of cooperating transactions [Ska89]. These new models have not been put into use, partly due to unavailability of implementations, and partly due to the recovery requirements underlying these models. Automatic enforcement of the ACID principles by undoing transactions is not always desirable in CM systems where applications and users can reestablish consistency manually.

Simple versioning and rebuild systems, like SCCS [Roc75] and Make [Fel79], offer marginal help for cooperation. Even more advanced CM systems like ClearCase [Leb94], and Adele [BEM93], all with high-level CM features, have very crude transaction and cooperation facilities. Some dedicated Software Engineering DBMSes, like DAMOKLES [DGL86], provide open and flexible long transaction models, but lack a user framework that enables cooperation support. Software engineering environments like Marvel [Kai90], COO [God93], and EPOS [MCL+95] all provide some cooperation functionality based on their own databases.

2.2 A Small Cooperation Example

To illustrate the problem of cooperation, consider the following simple and typical scenario from development of an imaginary software system, using RCS [Tic85] for source code management:

The imaginary `dbaccess` module (subsystem) contains, among others, these files:

`dbacc.h`:	Public interface to entire module.
`update.c`:	Manages update calls to the database; parts of module implementation.
`cache.h`:	Interface to cache management sub-module.
`cache.c`:	Implementation of the above.

As part of an overall system upgrade, developer Bob is given the task of adding some more update functionality to this module. This entails adding a new function declaration to `dbacc.h`, and implementing the new function in `update.c`. This new function also needs some extensions to the caching, so he makes changes to `cache.h` and `cache.c` as well. He checks out all those files with a lock, and since the changes are closely related to each other, he will not check them in until he is satisfied with all, and he will in practice enforce a serial access pattern.

Alice is also working towards the same system upgrade, but in another module, when her test application crashes. She is able to track the problem to a bug in the internal cache management, in other words she needs some fix to be made in `cache.c`. Since Bob is already in the process of doing (different) updates, we have a conflict situation.

RCS will refuse to let Alice check out `cache.c` for updates if Bob has already done so, thus preventing parallel work to take place at all.

There are several possible ways for Alice and Bob to resolve their conflict, but any of these solutions will either hamper their work (e.g. if Alice decides to wait until Bob is finished; i.e. sequential work), or there is no real support from RCS (e.g. Bob bypasses the system by taking Alice's changes into his own working copy).

2.3 Adding Cooperation Support

A simple versioning system like RCS can provide only limited information related to cooperation problems, e.g. we can find out who has locked an object, but we cannot find information about the context in which the object is being updated.

When building our CM system we have assumed a transaction-like work model, i.e. developers will often change several objects as part of one logical unit of work. CM users do their changes in workspaces which are connected to transactions in the underlying versioned DBMS. Both workspaces and transactions may be nested to reflect work breakdown. Objects are retrieved from the database, placed in workspaces as files and later returned to the database.

To support cooperation we wanted our CM system to be able to provide users with more information about other users' current and planned work, spanning from identification of which objects are being changed to user-oriented descriptions of work intentions. To allow for more active cooperation, the system should be able to give its users notifications about events caused by other users. In addition, the system must be extended with operations for more flexible sharing and exchange of objects between users working in different workspaces. Although the versioned database can be used to permit concurrent work, and to allow coexistence of several versions under development, we believe that cooperation is best supported by separate concepts and mechanisms.

We did not want to embed a particular work process in our CM system. We wanted the system to support a wide range of situations, ranging from active collaboration with frequent communication and coordination, to situations where users want to work in isolation, or are unaware of other users activities, and postpone necessary interactions to integrate own changes with those done by others. We believe that the most common work patterns will be somewhere between these extremes: the CM users will need to work in isolation for periods, then coordinate with others before they continue with other activities.

Since our system is centered around workspaces and transactions, it was natural to connect the cooperation support to these concepts. The cooperation support that we have added can be summarized by:

1. Declaration of work intentions: Planning information (e.g., resources, calendar and dates, and expected work sets), are specified in a formalism comprehensible to both users and the system. This permits us to reason on the plans against the actual state of a project: potential cooperation problems can be identified and solved in advance. These specifications are primarily meant for a priori coordination of work.

2. Transaction information is stored in the database and maintained when users are performing operations related to transactions. Some examples of the information are: A general description of the work intention, the transaction structure (hierarchy), and the set of accessed objects. This information is accessible through a set of tools/commands, and can be used to detect and/or avoid access conflicts at any time.

3. Whenever a user invokes a command that will update the database contents, a notification message is sent to all users that may be affected by the event. The notification messages are displayed and managed through a separate user interface, with features for filtering and user subscription to message classes. For instance, users may exchange textual messages in an e-mail like way; they can subscribe to a certain class of messages connected to events they might decide to have interest in. In this way, users may customize the amount and type of messages they are going to receive.

4. The transaction model has been extended with operations allowing flexible exchange of objects between transactions. These operations are available to the CM users through additional commands.

3 ECM - The EPOS Configuration Management System

This section discusses ECM, the EPOS configuration management system, as well as various aspects of the underlying database, EPOSDB [Mun93]. The main task of the product support is to manage storage of, and access to, an evolving, complex product. The product may consist of files and/or database objects.

In figure 1, we give an overview of how the user interacts with ECM, and its components to be described below. Note that an actual user might also be using the process modeling tools developed for EPOS, but these are not being discussed here.

3.1 The ECM Product Tool

ECM is a command-line or Prolog interface to the EPOSDB, which manages the user's files and checks them into and out of the database. Checked-out configurations are stored in *workspaces*, which are directory hierarchies matching the product structure (see below on product model). Files can be checked in and out incrementally, or as block check-out or check-in.

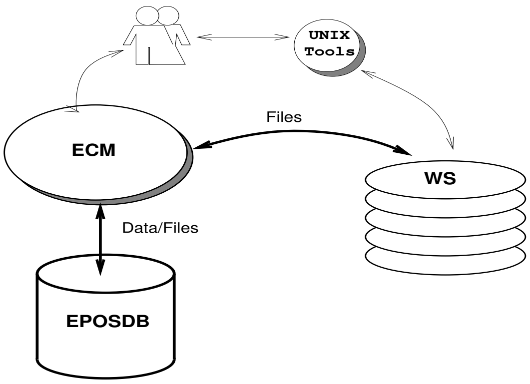

Figure 1: How the things fit together.

We have also developed a graphical user interface to ECM, CHAT which also provides access to the cooperation support facilities described in the next section.

ECM is also extended with commands to move or copy files between transactions within the database, both between ancestor and sibling transactions. These commands are used to exchange files between users/workspaces in a controlled manner and at a fine granularity.

ECM models software products using the data model of the EPOSDB. Software components and subsystems are represented as entities with a small set of attributes, e.g. describing the status of the component. Along with entities describing primary components, we also store the file contents of the component in the EPOSDB. System composition, and inter-component dependencies are represented through relationships.

ECM provides a small set of predefined product types: `product`, `part` which is a generalization of `subsystem` and `component`, the subsystem `composition` relationship with name attribute, and a number of dependency relations, e.g. `derivedfrom` and `srcincludes`.

To this, the user may define subtypes for various types of files, such as `csource`, `framedoc` etc. ECM has an additional library of over 100 component types, covering a wide variety of common file types.

In order to facilitate installation of already existing products into EPOS, we have built a tool for extracting the product structure from a stored product and convert it to EPOSDB storage. A product description language PDL has been designed, and the installer tool "EPIT" using it is implemented.

3.2 The EPOS Database

The EPOS Database [Mun93] is a general-purpose DBMS with features geared for SCM. The current EPOSDB runs on Sun workstations, and provides a Prolog API. All objects are versioned according to our COV versioning model [MLG+93], which we will not discuss in this paper.

The EPOSDB uses an extended ER data model with subtyping. The data model is based on the Object-Relationship model [Rum87]. There is a predefined type `longfield`; instances of this type have additional operations for checkin/checkout of the file contents to/from normal operating system files. In addition to "normal" objects, the EPOSDB also maintains a *meta-database* containg user-queriable objects representing types, active transactions, etc.

The EPOSDB supports nested, long-duration transactions which form an envelope around a unit of work. The transactions may exist through several application sessions, and are named entities. Transactions have separate storage areas for changed objects, and we can distinguish between the copies of an object residing in different transactions. Database access is usually done through the context of a long transaction, and operations may be performed on shared objects or local copies. Transactions are explicitly terminated with either a *commit* or an *abort* operation, whereupon changes are made permanent or discarded. It should also be mentioned that the EPOSDB also provides more traditional short-duration transactions with ACID properties.

We have extended the transaction model to allow fine-grained control over propagation of changes between parent and child transaction, and between sibling transactions. Objects can be individually copied out from or back into the parent transaction (or the permanent database), and local copies can be transferred into the local storage areas of sibling transactions. It is also possible to access objects residing in an ancestor transaction, but these objects may be changed by other transactions. To prevent update conflicts, the transaction model defines a set of flexible lock modes and even allows several transactions to concurrently update an object (requiring merging of the changes).

In essence, the EPOSDB provides access to a redundant set of mechanisms that can be used for enforcing a variety of correctness criteria with different implementations. The responsibility for guaranteeing correctness is partly left to the user and/or applications. However, there is a default behavior that provides failure atomicity and partial isolation.

Each ECM workspace has an accompanying database transaction which provides concurrency control and recovery for the workspaces. The transaction holds local and changed data during the lifetime of the workspace, thus guaranteeing atomicity of the changes and isolation against changes from other workspaces.

4 Cooperation Support for ECM

In this section we describe three attempts we have made at improving cooperation support within the ECM environment. This work is based on the flexibility of the underlying database and its transaction model, allowing a high degree of control over mechanisms normally used to implement automatic concurrency control.

As mentioned in section 3.2, the EPOSDB leaves to the users and applications to enforce specific correctness criteria. An alternative would be to choose one criterion and an implementation of this, which would partly prescribe the work mode, and then build cooperation support based on the situation.

Our low-support approach was chosen since we wanted the EPOSDB to be able to support a wide range of work modes which again will require different types of cooperation support:

At one extreme, users may use conservative two-phase locking to enforce a serial execution of transactions. At the other extreme, it is possible to allow full concurrency during transaction execution, and then, in case of conflicts either undo the changes or force the users to merge conflicting, concurrent changes.

In practice, we will provide several interfaces to ECM and the EPOSDB , which enforce specific concurrency control measures, according to different correctness criteria.

4.1 Transaction and Workspace Meta-Data

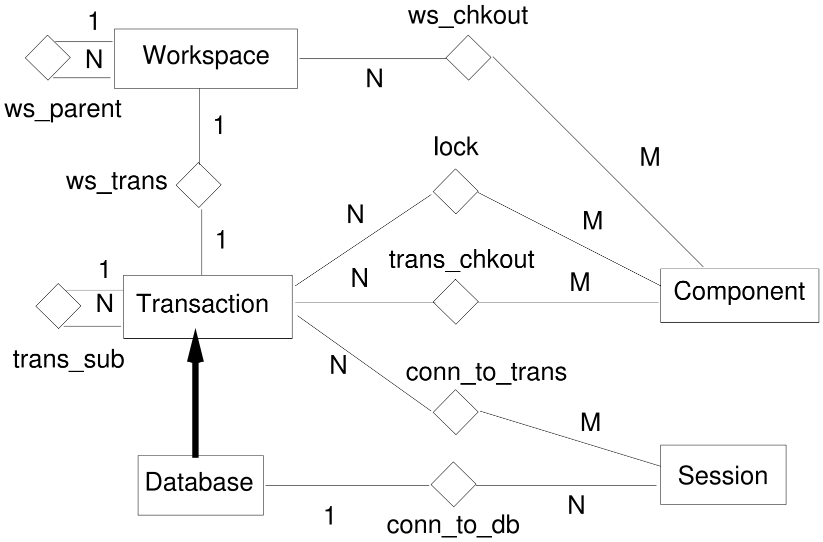

Figure 2: Schema for transaction and workspace database.

To support interactive conflict solving, we need to supply sufficient information for the users to be able to locate the source of a conflict.

Our solution is a meta-database of transaction-related data. This database can be queried from within any transaction and it is updated by various access and transaction control operations. The transaction and workspace meta-database will give a consistent view of which transactions exist and which objects are locked or checked out by which transactions.

The schema for this meta-database is shown in figure 2. The transactions and workspaces are modeled by the respective types and the hierarchies by the `trans_sub` and `ws_parent` relations. The `lock` relation type models locked components, and similarly the `trans_chkout` `ws_chkout` relations show which transactions have checked out a component.

We are building transaction and workspace browsers which are integrated with the existing component browser in CHAT. The goal is to build a system where one can easily find the source of a conflict arising from an attempted operation. The user will then have to take appropriate actions to resolve the situation: delay the operation, undo one's own work, or request the partner involved in the conflict to act.

4.2 Work-Unit Descriptions

The term work-unit, short WU, covers the notion of a "meaningful" unit of change/revision made to software objects in the EPOSDB. By storing work-units in the EPOSDB , it is possible to share descriptions of planned work with other users, so that it will be possible to coordinate and schedule larger change jobs to minimize the potential for access conflicts.

The intentions of the work to be performed in a work-unit is given by a work-unit description, WUD. This description primarily encompasses information needed to start an ECM workspace, such as ambition, choice, and read-/write-sets of named software components. WUDs are stored in the EPOSDB using a schema which is structurally similar to the one shown in figure 2.

WUDL ("Work Unit Description Language") is a textual language for defining properties of and pre-declaring work intentions for a workspace or transaction. WUDL specifications are declarative, and their purpose is to define in advance which objects will or may be accessed, not when and how.

A WU specification contains these main pieces of information about what will take place during the execution of a transaction:

- Properties of the workspace and transaction: status (planned, running, committed, aborted), owner/responsible, planned start and duration, whether it is decomposable, etc.
- Hierarchy, i.e. name of parent transaction, if any.
- Versioning context.
- Name of product to work on.
- Initial read-set and write-set of transaction.
- Access-policies: lock and access modes.
- Cooperation policies and automatic actions in case of conflicting access.

A sample WU specification is shown below. This describes an activity which scope is to implement a set of identified, small bug-fixes. The components which are to be accessed are specified in detail, and these are locked and checked out at transaction start and released when the workunit finishes.

```
WORKUNIT FixMemoryLeaks_Release3.0
    RESPONSIBLE bjornmu;
    DESCRIPTION ``Fix detected memory leaks in current release'';
    PLANNED_START 010995;
    PLANNED_DURATION ``1 day'';
    PROD_ROOT 'eposdb:';
    READ_SET dm/dmcurs.c dm/dmtrans.c dm/dmqry.c tran/transrec.c
             tran/commit.c ver ll/FDcache.c ll/lowl.c;
```

```
    WRITE_SET dm/dmcurs.c dm/dmtrans.c dm/dmqry.c tran/transrec.c
              ver/options.c ver/vishash.c ll/FDcache.c;
    DEFAULT_READ_LOCK nonrestrictive_read, start, end;
    DEFAULT_WRITE_LOCK restrictive_write, start, end;
    DEFAULT_ACCESS_MODE  checkout, start;
END_WORKUNIT
```

Work-units are written in an ordinary editor and then stored in the database. ECM workspaces can be started based on stored work-unit descriptions, and we plan to provide an extended interface to ECM which follow the policies and behavior specified in a WUD. The stored WUDs can be used for longer-term planning of work.

We have implemented a planning tool [CLH95] which can analyze a set of work-unit descriptions and suggest an execution history which will minimize the set of dependencies between concurrently executing transactions.

WUDL is currently being extended to contain elements for coupling ECM closer to the EPOS software process support tools. A longer-term goal is to supply a library of concurrency control and cooperation policies which cover the most methods found in database systems. Using this library, we are able to specify work-units at a relatively high level of abstraction.

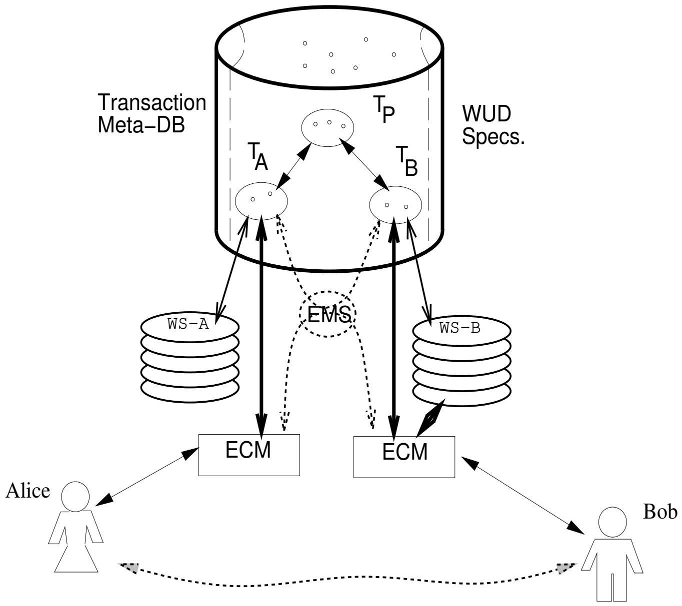

Figure 3: Cooperation between Alice and Bob.

4.3 Events and Notifications

To enable more asynchronous cooperation between software developers using ECM, we have designed and implemented a messaging system which is used for notifying users about events, e.g. database access, updates, and commit operations, originating from other transactions.

Events are specified in terms of ECM operations, so that users can easily relate to the notifications.

This message system is primarily useful for giving early notification of actual or potential conflicts in the case that transactions use non-restrictive locking or no locking at all. A workspace or transaction can subscribe to events and will receive a notification message whenever the specified event takes place.

The EMS ("Epos Message Server") is responsible for sending messages between transactions. ECM commands like check-in and check-out trigger the EMS, which will then send the notification messages to all workspaces subscribing to the event.

CHAT also provides a user interface for displaying and browsing messages, and replying to them or sending user written messages. The messages are stored as entities locally to the receiving transaction.

We are working on a system which can respond to EMS messages and perform specified actions, e.g. automatically check out a component which has been changed in a sibling transaction.

4.4 Cooperation Support Example

Turning back to example from section 2.2, we will now look at some possible situations and solutions with our cooperation support. We will assume that Alice and Bob work with one workspace each and that their corresponding transactions are siblings, as shown in figure 3.

The problem was that Alice needed to fix a bug in `cache.c` which is copied into T_B and further checked out into Bob's workspace. Alice can find out who is using or updating `cache.c` through WU specifications and the transaction meta-database. She can then choose to postpone her own testing and do the bug-fix on `cache.c` in a separate transaction (which might be the preferred solution according to the development/maintenance process) or go on.

In the case that Bob has not locked the file, Alice will be permitted to check out `cache.c`, and the last one to finish work will discover that there is a conflict and merge the changes (or undo own updates). If Bob did lock the files he is working on, Alice must wait.

In both cases, Alice may request Bob to transfer `cache.c` to her transaction/workspace, so that she can perform her changes and then either keep the object or return it to Bob. Bob can also return his local copy of `cache.c` to the parent transaction, wait for Alice to do her update, and then, if needed, continue working on the object.

Bob and Alice can also use EMS to get notifications about the others actions. This is useful when any of them is waiting for the other to release locks or finish work on `cache.c`

5 Conclusion and Further Work

We have presented a CM system which has been extended with operations and mechanisms to support cooperation between users of the system. We have discussed different types of information that can be entered into the system and how it can be used for planning, conflict detection and identification of partners involved in both possible and actual conflicts. We have presented tools that can be used to retrieve information on demand and a tool that can display messages about events caused by other users.

Referring to the cooperation aspects listed in section 2, our system solution to cooperation is as follows: collaboration among human agents is supported by ECM and its data exchange facilities, EMS and the transaction meta-database assist in information exchange, and support the system in informing users of conflicts or evolving situations of general interest. Lastly, coordination may be achieved through active use of WU pre-declarations.

By integrating the cooperation support with the CM system we are able to offer more precise information related to the objects managed by the CM system, and result in more reliable and easily maintainable information.

The users also have support from the CM system for propagating changes to each other, so that they can resolve the conflicts resulting from parallel work. The system is customizable and can be used to support a range of work modes, from loose to tight cooperation.

The idea of integrating work related information into the CM system should be applicable to other systems, even much simpler ones. We would need to implement tools to enter and retrieve this information, and users must be made aware of the importance of providing information about work intentions.

Further work on the CM system will be to get more experience with using the system in realistic situations and then to identify other types of information that may be included. We will also need to build support for using WU specifications in work-planning. The notification system will generate a large number of messages and we will need to extend the possibility for filtering messages. We will also try to build tools that can respond to messages by performing prescribed actions according to cooperation protocols, e.g. to acquire the latest changes to an object and put it in the workspace. Finally, the low level support for merging changes will be extended with more advanced tools and techniques.

References

[BEM93] Noureddine Belkhatir, Jacky Estublier, and Walcelio Melo. Software Process Model and Work Space Control in the Adele System. In *[Ost93]*, pages 2–11, 1993.

[BJP94] M. Brodie, M. Jarke, and M. Papazoglou, editors. *Proc. of the 2nd International Conference of Cooperative Information Systems*, Toronto, Canada, May 1994.

[BK91] Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[CDW87] Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik, editors. *Proc. IFIP WG-2.4 International Workshop on Advanced Programming Environments, 16-18 June 1986, Trondheim, Norway.* Springer Verlag *LNCS 244*, 604 p., March 1987.

[CLH95] Reidar Conradi, Chunnian Liu, and Marianne Hagaseth. Planning Support for Cooperating Transactions in EPOS. *Information Systems*, 20(4):317–326, June 1995.

[DGL86] Klaus Dittrich, Willi Gotthard, and Peter C. Lockemann. DAMOKLES — a Database System for Software Engineering Environments. In *[CDW87]*, pages 353–371, 1986.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications.* Morgan Kaufmann, 650 p., 1992.

[Fel79] Stuart I. Feldman. Make — a Program for Maintaining Computer Programs. *Software — Practice and Experience*, 9(3):255–265, March 1979.

[Fel93] Stuart I. Feldman, editor. *Proceedings of the Fourth International Workshop on Software Configuration Management (SCM-4)*, Baltimore, Maryland, May 21–22, 1993.

[God93] Claude Godart. COO: A transaction model to support COOperating software developers COOrdination. In *[SP93]*, pages 361–379, 1993.

[Gra81] Jim N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Databases*, pages 144–154, September 1981.

[Kai90] Gail E. Kaiser. A flexible transaction model for software engineering. In *Proc. 6th International Conference on Data Engineering*, pages 560–567, Los Angeles, CA, February 1990. IEEE Computer Society. Invited paper.

[KKB85] H. Korth, W. Kim, and F. Bancilhon. A Model of CAD Transactions. In *Proceedings of the 11th International Conference on Very Large Databases*, pages 25–33, 1985.

[Leb94] David B. Leblang. The CM Challenge: Configuration Management that Works. In *[Tic94]*, chapter 1, pages 1–37. John Wiley, 1994.

[MCL+95] Bjørn P. Munch, Reidar Conradi, Jens-Otto Larsen, Nguyen Ngoc Minh, and Per H. Westby. Integrated Product and Process Management in EPOS. *Journal of Integrated CAE*, 1995. (Forthcoming in special issue on Integrated Product and Process Modeling), 30 p.

[MLG+93] Bjørn P. Munch, Jens-Otto Larsen, Bjørn Gulla, Reidar Conradi, and Even-André Karlsson. Uniform Versioning: The Change-Oriented Model. In *[Fel93]*, pages 188–196, 1993.

[Mos85] J.E. Moss. *Nested Transactions: An Approach to reliable Distributed Computing.* Information Systems Series. MIT Press, Cambridge, Massachusetts, 1985. (PhD thesis from 1981 at MIT).

[Mun93] Bjørn P. Munch. *Versioning in a Software Engineering Database — the Change Oriented Way.* PhD thesis, DCST, NTH, Trondheim, Norway, August 1993. 265 p. (PhD thesis NTH 1993:78).

[Ost93] Leon Osterweil, editor. *Proc. 2nd Int'l Conference on Software Process (ICSP'2), Berlin. 170 p.* IEEE-CS Press, March 1993.

[Roc75] Mark J. Rochkind. The Source Code Control System. *IEEE Trans. on Software Engineering*, SE-1(4):364–370, 1975.

[Rum87] James Rumbaugh. Relations as semantics constructs in an object-oriented language. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 466–481, Kissimmee, Florida, October 1987. In ACM SIGPLAN Notices 22(12), Dec. 1987.

[Ska89] Andrea Skarra. Concurrency Control for Cooperating Transactions in an Object–oriented Database. *SIGPLAN Notices*, 24(4):466–473, February 1989.

[SP93] Ian Sommerville and Manfred Paul, editors. *Proc. 4th European Software Engineering Conference* (Garmisch-Partenkirchen, FRG)*, Springer Verlag* LNCS 717, September 1993.

[Tic85] Walter F. Tichy. RCS — A System for Version Control. *Software — Practice and Experience*, 15(7):637–654, 1985.

[Tic94] Walter F. Tichy, editor. *Configuration Management.* (Trends in software). John Wiley, 1994. ISBN 0-471-94245-6.