Cooperating Transactions and Workspaces in EPOS: Design and Preliminary Implementation

Reidar Conradi*, Carl Chr. Malm
Norwegian Institute of Technology (NTH), Trondheim, Norway

1 Introduction

EPOS[1] offers *Change-oriented* versioning (COV) for software configuration management (CM). The EPOSDB has long and nested transactions. EPOS also supports software process management (PM) *within* a transaction and its workspace through task networks and their project infrastructure.

The paper deals with EPOS extensions for *inter-transaction coordination*. This relies on intentional configuration descriptions and *ambitions* to describe change propagation into other versions. Raw (textual) merging comes for free in the COV model. Semantic merging is facilitated by pre-commit propagation and negotiation among overlapping transactions, according to agreed-upon protocols.

In the following, the problem area is first characterized with references to existing CM systems and DBMSes. Then EPOS and specially COV are summarized. Lastly comes a presentation of our transaction and cooperation model: background, design, technical solutions, problems, and ideas on future work.

2 Background and Previous Work

CM assumes a shared and versioned repository to store *software products*, consisting of general software components and their dependencies. Few tools operate directly on the versioned repository. A software producer (a user or a team) must therefore work against a file-based *workspace*. This contains a *checked-out* configuration, i.e. a specific version of a chosen subproduct (Sec. 3.2).

Evaluation, check-out/in and commit of configurations must be planned, described, monitored, and coordinated – both on a conceptual and technical level.

Traditional DBMSes have a strict consistency concept, coupled to serializible (short and system-executed) transactions. For distributed and network-connected DBMSes, there is a two-phase commit protocol.

CM involves many concurrent actors and long update times. Since updates may involve

*Address: Div. of Computer Systems and Telematics, NTH, N-7034 Trondheim, Norway. Phone: +47 7 593444, Fax: +47 7 594466, Email: conradi@idt.unit.no.

[1]EPOS, Expert System for Program and ("Og" in Norwegian) System Development, is supported by the Royal Norwegian Council for Scientific and Industrial Research (NTNF) through grant ED0224.18457. Do *not* confuse with the German real-time environment EPOS [Lem86].

hard-to-predict and partly overlapping versions or subsystems, traditional locking procedures will cause intolerable delays. CM therefore needs non-serializible (long and user-executed) transactions. This may lead to update conflicts, which later must be reconciled by version *merging*. But sometimes *no* merge is needed, because of independent development paths.

Digression: When *all* transactions in an unversioned DBMS commit, there is *one* canonical and consistent version of the database (DB). In contrast, CM maintains and control *permanently* and *mutually inconsistent* (sub-)DBs!

Algorithms to ensure consistency in multi-layer storage systems (cache coherence, synchrony between local and global databases) resemble those used for data exchange between long transactions. Work on crowd control and groupware is also relevant.

There has been much interest in database *triggers* [Sto86]. That is, to have an *active* DBMS, which automatically performs consistency checks and side-effect propagation according to explicit event-condition-action rules. For CM we must also consider version propagation incl. negotiation about propagation rules, and that side-effect propagation can be very time-consuming and presume unobtainable access rights. Classic DBMS triggers inside short transactions are therefore insufficient. A possible but not satisfactory solution is to use *notifiers* to handle free-standing or delayed actions.

Simple CM systems, like SCCS [Roc75] and Make [Fel79], offer no help for cooperating transactions. Adele [BE87] has high-level configuration descriptions and some workspace control, but only triggers to start rebuilds. PCMS [HM88] has document-flow templates, and Mercury [MK88] uses attribute grammars to guide simple change propagation. ISTAR [Dow86] has subcontracts, but little formal cooperation. NSE [Sun88] is strong on workspace control, and DSEE [LM85] has some support for handling change requests. Few combined CM and PM system can adequately treat cooperating transactions, or can handle configurations as conceptual entities both inside a database and in an external workspace.

3 EPOS Background

We shall present EPOS and its versioning model, COV. COV is fundamental to understand our transaction concept, and is therefore treated in some detail.

3.1 EPOSDB, Data and Task Model

The core of EPOS is a client-server based *EPOSDB* [OrMrGB90]. Its *EPOS-OOER* data model can express application-specific *product and task models*, as sets of types in a DB schema. Long attributes are represented as external files.

EPOSDB implements COV, and offers a version-transparent interface. Each transaction is connected to a change job, a project task. Its configuration is evaluated (Sec. 3.2) inside EPOSDB, and checked-out (i.e. converted) to a workspace. The workspace contains:

1. *Files* and their dependencies: the *Product Structure*, PS.
2. A *task network* to support PM.
3. Some *control information*: the *Project Knowledge Base*, KB.

The Project KB can in principle be changed on-the-fly, and is *customized* using normal COV. Many tasks communicate with developers through a screen window, mouse, and keyboard.

The task network resembles a Petri net. It is managed by an *Activity Manager* written in Prolog, doing planning, execution, and propagation control [COWL90]. Rules for change propagation (busy, periodic, opportunistic, lazy etc.) are expressed by task types, which contain PRE- and POST-conditions, a CODE program, and various constraints. Special inheritance rules applies. A production rule like PRE–CODE can be compared to IF–THEN clauses in conventional triggers.

The EPOS Activity Manager (AM) runs as an *application* on top of EPOSDB. The AM will set up the basic transaction/project structure. Low-level synchronization and notification is handled by the DB, and the AM application is expected to handle possible conflicting situations on a sufficiently high-level.

PM in EPOS was initially limited to tasks (e.g. rebuilds) *within* a single-version[2] workspace. We are now generalizing to multiple workspaces to control propagation/communication (e.g. merges) *between* workspaces. A message passing facility was added for this.

Figure 1 shows Product Structures in two cooperating transactions, with internal and external change propagation.

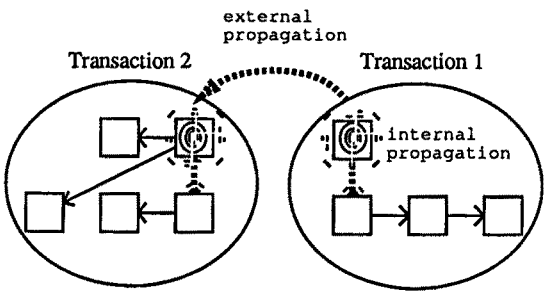

Figure 1: **Internal vs. external change propagation after an update**

3.2 Change-Oriented Versioning, COV

Change-oriented versioning [Hol88] [LCD*89] resembles and generalizes conditional compilation. COV is *orthogonal* to data model, schema, instances (objects and relationships),

[2]Most published PM work share the same fate.

and surrounding infrastructure. In COV, a completed *functional change* is described by a boolean[3] *global option*[4], e.g. `MachineSun` or `BugFixCommandA`. There are *version-rules* (constraints, preferences, defaults) to express legal or preferred option combinations.

A DB *fragment* or delta (a text line or relational tuple) is tagged by a *visibility*, a logical expression. A visibility is evaluated to `False` or `True` under a given *version-choice*, being a complete and legal set of option bindings. A version-choice corresponds to a "configuration thread" in other CM systems. A version-choice with `Unset` option values is called an ambition (Sec. 4).

A specific, *bound version* of the entire DB – a non-versioned *sub-DB* – consists of DB fragments with `True` visibilities. If some visibilities still are `Unset`, the version is partially bound.

A new option creates a "mirror" DB. This is initially equal to the existing DB, and is invisible to versions that have not included the new option.

3.3 Configuration Description and Evaluation in EPOSDB

A configuration is described by a DB query or intentional *config-description*, CD = [VD, PD]. The VD or *version-description* is a partially bound version-choice. The PD or *product-description* is a tuple of [Root objects, ER types].

A *bound configuration*, a Config, can be expressed as Config = Evaluate-config(CD, DB). The evaluation goes in three mapping steps, see Figure 2:

- C1) VC = Bind-version(VD, version-rules)[5].

  The more high-level and partially bound VD is evaluated to a low-level and fully bound version-choice, VC, through stored version-rules [GKY90]. The mapping typically incorporates more detailed "revision options" in the version-choice, and may change as the version-rules evolve.

- C2) Sub-DB = Version-select(VC, DB).

  This mapping facilitates "COV-propagation" or raw data merging *between* configurations, if the changes have wide-enough visibilities. This is the main feature of COV, since it eliminates the need for separate, parallel variants.

- C3) Config = Product-select(PD, Sub-DB).

  This mapping selects a closed Product Structure, a configuration, by constraining the bound sub-DB. The product closure is defined by a hierarchical product model, with separate `Family` interfaces/bodies and general dependencies. The mapping may change, due to structural changes in the product (really a C2 change).

A forth step will then generate a workspace, WS:
C4) WS = Check-out(Config, DB-WS map).

Note, that version-selection (C2) is done *before* product-selection (C3), being the inverse of

[3]The domain is really *ternary*: `False`, `True`, and `Unset` – or F, T and U.

[4]The options are separate from normal DB attributes. Indeed, an option is in itself an entity and does in particular have attributes like CreateTime, IntendedImpact, Name, ResponsiblePerson etc.

[5]An ambition A is also evaluated. The VC is a point within the multi-version space indicated by A.

common practice in most CM systems. Here, a back-bone *system model* must be generated *before* the individual component versions can be selected via a configuration thread. The latter ordering may create problems, if the system model can be re-structured, i.e. it is versioned. Adele offers intermixed version/product selection to handle this.

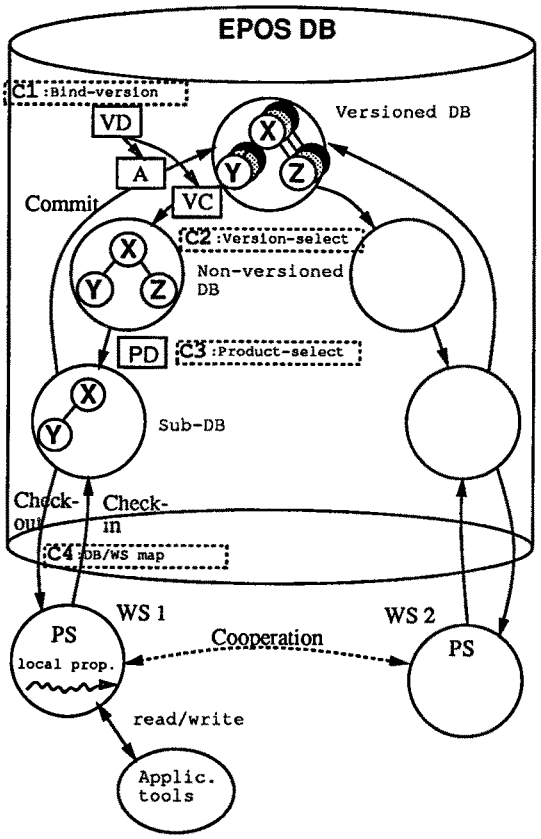

Figure 2: EPOSDB mappings

4 EPOS Transaction Model

EPOS supports long and possibly nested transactions. Short transactions may be needed to atomize internal EPOSDB operations, and are simple to add.

4.1 Transaction, Ambition and Version-status

A transaction works on a configuration, described by and evaluated from a config-description. There is also an *ambition* to specify the *scope* of updates on this configuration. The ambition is an option binding with possible `Unset` option values. It identifies a potentially large *set* of versions, or sub-DB slices of a versioned DB universe, where the local changes from the *current* transaction will be visible after commit.

Ex. Updating under ambition A, with old visibilities V and new visibilities V':
A new fragment gets $V' = A$, an unchanged one $V' = V$, and a deleted one $V' = V \wedge \neg A$.

Ex. Assume that a VC has options `Ox` and `Oy` set to `True`. This implies that its corresponding version (= visible fragments) may change, if another transaction introduces fragments with visibilities `Ox`, `Oy`, or `Ox` $\wedge$ `Oy` (the latter represents merging). See next subsection on overlapping transactions.

Lastly, a transaction is characterized by a *version-status* with values such as `Stable`, `Released`, `Reviewed`, `Tested`, `Compiled`, `Designed`, `Raw`, `New` etc. The version-status indicates the proposed reliability of the committed version. And *one* new or changed DB fragment, propagated from another transaction, is enough to make a configuration "dirty" or `Raw` (Sec. 5.4 for merging). To stop such propagation we can declare a version(-choice) `Stable`, i.e. *immutable*.

Ambition and version-status can be adjusted before commit, as a result of negotiation and carried-out merging and testing. However, the version-choice must always constrain the current ambition, and it can only be extended with *new* options after transaction start.

4.2 Cooperating Transactions with Partial Overlap

After commit of a transaction, the affected or *overlapping* configurations are candidates for reevaluation. These configurations may either be *ongoing*, i.e. we need cooperating transactions, or they may be *released*, i.e. we should notify their responsibles for possible reevaluation. E.g. Adele will prepare a log of proposed revisions to an old configuration, so that these can be included in a future, reevaluated configuration.

There are two kinds of *overlap*, corresponding to the previous C1- and C3-mappings:

- **Ambition-overlap**: mutual version visibility.
  Two ambitions A1 and A2 are either *disjoint*: there exists at least one option with a `False` value in one ambition and a `True` value in the other; or *overlapping*: no option can be `False` in one ambition and `True` in another (but `Unset` is OK). Special cases of overlaps are *constrainment* (useful in nested transactions) or *identity* (useful for parallel work on disjoint subsystems).

  No ambition-overlap corresponds to classical variants.

  Example 1. Overlapping ambitions.

  Assume that there are three old and two new options: `O1-O3`, `Ox`, and `Oy`. Also assume that transaction T1 introduces Ox with ambition A1=aaaTU and with version-choice VC1=vvvTT. The three option values vvv have bound option values, possibly

constraining the aaa values. Assume further that transaction T2 introduces Oy with A2=aaaUT and with VC2=vvvTT. This implies that T1's updates *at least* (depending on aaa) will become visible to T2, and inversely.

- **Product-overlap**: shared subsystems.
  A Product Structure constrains the sub-DB selected by a version-choice. Product-overlap simply means non-empty intersection of actual PS components. A difference tool will similarly identify the non-intersecting components, or possibly more fine-grained fragments.

  Note: Product-overlap assumes ambition-overlap. It does not "help" that two PS versions are identical at transaction start, if there is no ambition-overlap to announce intended merging.

  Example 2a. Product-overlap in Figure 3.

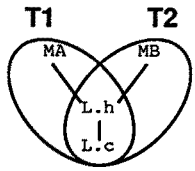

Figure 3: Product Structures with shared subsystems

The degree of overlap can be used to assess the *potential impact* (side-effects) of the intended changes of a new transaction. A consequence of such impact analysis can be that the new transaction is either constrained, delayed (i.e. serialized!), or have to be delegated to somebody else (see points 3 and 4 in Example 2b below).

Policies for handling conflicting updates in overlapping transactions can be:

- *Not_do_anything*: arbitration or anarchy!
- *Priority*: let the first, last (default in EPOSDB), or some arbitrary committing transaction win.
- *Rollback*: prevent any overlapping transaction from committing. This may waste weeks of work!
- *Locking*: prevent multiple updates on shared components by access locks:
  - *Read/write locks*:
    Only one actor can write, while several others may read. RCS uses this mechanism. Note that exclusive write locks may introduce deadlocks.
  - *Only read locks* or no interference:
    If none of the actors can or will write, there will be no problems.
  - *No locks* or *optimistic synchronization* (default in EPOSDB):
    A project administrator should assign well-defined jobs to ensure maximum parallelization and minimum interference. However, inconsistent updates may occur, and require subsequent merging (see below).
- *Merging/integration*: the most general solution, and being presented in this paper.

It may happen that several versions may have to be merged – but when, in what sequence, and by whom?

Note, that non-overlapping write access does *not* remove the need for general merging[6], due to (transitive) dependencies of a modified component, cf. trickle-down recompilation. See Sec. 5.4 for merging details.

Example 2b. Two cooperating transactions T1 and T2, with the same ambition and version-choice.

The T1 configuration contains the three modules from Figure 3: MA which uses L.h, which again is implemented by L.c. Similarly, the T2 configuration has the three modules MB, L.h and L.c. We can think of the following update situations:

1. Non-overlaps:
   MB or MA can be updated without interference, except for local recompilations.

2. Overlaps, with local and later global propagation:
   Local updating of L.h by T1 (if allowed) implies possible changes and recompilations of the two *clients* MA and L.c inside T1. A change in L.c only implies a local recompilation.

   However, any change to shared L components must sooner or later be propagated to transaction T2. This may be followed by subsequent textual merges, if T2 had already made updates on these (and propagation of the merged version back again!). But even if T2 has not updated these, we must perform normal change propagation *within* T2's workspace, to account for the "T1-imported" changes on these components.

3. Constraining overlap:
   An alternative is for T2 to delay, or even resist T1's announced changes to L. The latter case could lead to T1 being prevented from doing the announced changes. Another solution is for T1 to constrain its ambition, or for T2 to add a new sub-option to isolate itself.

4. Delegation of updates to somebody else:
   It may also happen that neither T1 nor T2 has the access rights to change the L components. Then, what if T1 still insists on having certain changes to these? A possible solution is for T1 to send a change-request message to the responsible maintainer of L, with information about requested functionality, ambition, and urgency(!). But will T2 accept to be forced to comply with T1's proposed change? Again, T1 or T2 may have to constrain their ambition-overlap.

We can conclude, that there is a strong demand for flexible communication and negotiation policies, even for very simple systems.

[6] We can consider the *entire* software product as *one* text for simplicity.

4.3 Nested Transactions

A *nested, child* transaction overlaps that of its *parent*, and possibly that of its siblings. In addition, the child is *constrained* relative to its parent in the ambition and/or product part. This implies *write locks* of version subspaces (sub-DBs) and access rights only to product subspaces within these versions. However, we are not constraining access to whole instances, only to versions of these instances.

After child commit, changes are propagated to the parent, which must handle possible update conflicts. We should therefore prepare later merging by pre-commit negotiation and propagation between parent and children. I.e. with *whom* should inter-transaction *cooperation* be established, and *how* (manual or automatic) and *when* (lazy or busy) should it be carried out?

A simple transaction tree is shown in Figure 4.

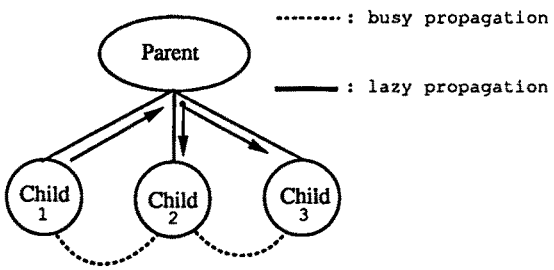

Figure 4: Nested transactions, with alternative propagation patterns

Example 3. Two different changes X and Y on module M.

Assume that X and Y are independent, so that the corresponding update jobs, Tx and Ty, will produce two *variants*, Mx and My. Tx and Ty can be carried out in parallel or in arbitrary sequence. If there later is a need for merging, this can be done by a third update job, Txy, producing Mxy. The start version for Mxy is either Mx or My.

Given our model of nested transactions, we will rather make a parent transaction Txy, with two child transactions Tx and Ty. The parent may have to delegate the merges to a third Tmerge child, because of its initial and irreversible version-choices – see Figure 5. However, the parent can instruct Tx and Ty from the start to cooperate on producing a *common* version before commit.

5 Supporting Cooperating Transactions: the Details

To define and execute cooperating transactions we need project-level information, such as change requests, work procedures (= task types), config-descriptions, workspace layout descriptions, and available resources (tools, users). Change propagation is supported by

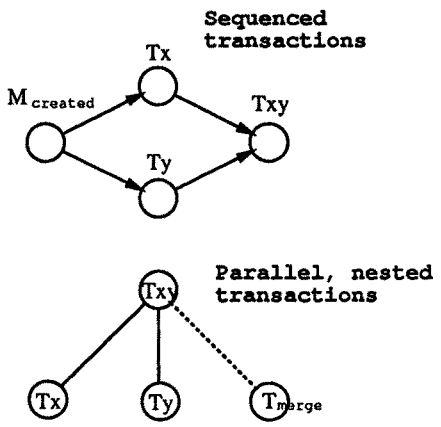

Figure 5: Sequential vs. parallel merging

message passing, communication and negotiation protocols, and local reconciliation with merging and client propagation.

In the following we shall describe subtasks for cooperating transactions, workspace organization, communication protocols, and change reconciliation.

5.1 Subtask Infrastructure for a Transaction

Each transaction is controlled by a change job of type `Project`, a *transaction task*. It will receive a change request, CR, and produce an updated configuration, NewC'.

A ROOT-PROJECT task describes the never-ending, most global transaction, while PROJECT describes the current transaction. A transaction task is connected to a parent and possible children and sibling transaction tasks. There is also a list of committed transactions, FINISHED, whose config-descriptions are candidates for future reevaluation.

Figure 6 shows the PROJECT subtasks and simplified data flows to implement cooperating transactions:

1. START-CHECKOUT: Initiate transaction, evaluate a NewC configuration from a chosen CD, and transfer and convert NewC from EPOSDB to the workspace.

2. OVERLAP-NEGOTIATE[7] for control-level negotiation: Analyze ongoing sibling transactions for possible ambition/product-overlap with NewC, and send out messages to negotiate a *contract* or protocol for future communication and propagation.

3. WORKING:

[7]The overlap analysis should technically be done *before* check-out.

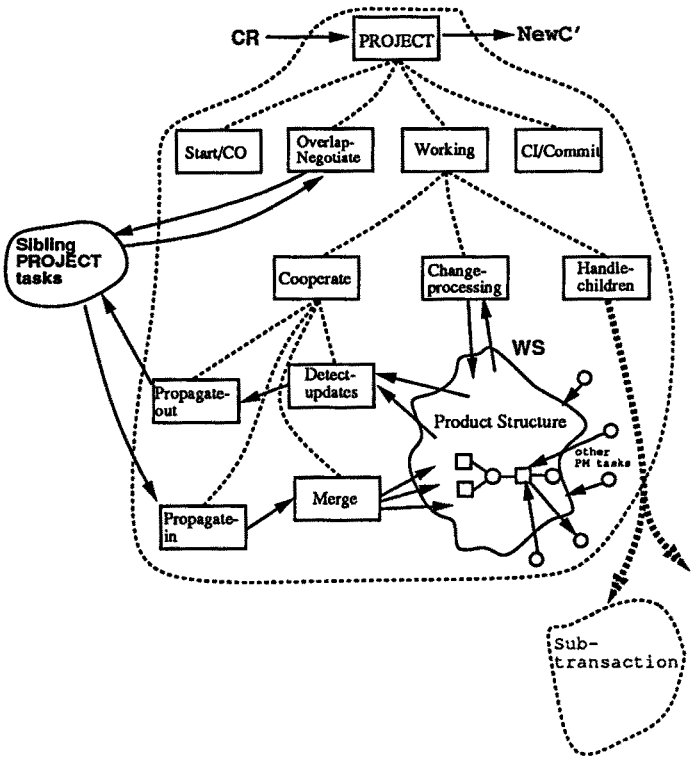

Figure 6: Tasks to support cooperating transactions

- COOPERATE: data-level exchange and negotiation.
  - PROPAGATE-IN + possible MERGEs: Analyze and negotiate incoming change-notifications. For instance, it could fork off a new MERGE subtask to reconcile an incoming and a local module.
  - DETECT-UPDATES + PROPAGATE-OUT: Monitor the local workspace, filter and analyze the changes, and possibly forward change-notifications to overlapping transactions.

  All these activities are guided by the agreed-upon protocol, and interfere with the CHANGE-PROCESSING subtask below.
- CHANGE-PROCESSING for normal, single-user PM: contains subtasks for work decomposition, sequencing, and change propagation within the workspace. Thereby the work according to CR will be carried out, and a NewC' will be created and validated to some degree. Subtransactions can be started via the HANDLE-CHILDREN subtask.
- HANDLE-CHILDREN: starts and terminates children transactions.

4. CHECKIN-COMMIT: Transfer the updated workspace containing NewC' to

EPOSDB, and propagate/negotiate changes to siblings. Then commit transaction to parent via EPOSDB, and negotiate with the *parent's* HANDLE-CHILDREN task. Overlapping, released configurations will be notified through their responsible tasks, having some suitable activation rules.

5.2 Workspace control

Check-out and *check-in* converts between the sub-DB and an external workspace[8] (files, Prolog facts). Check-out is batch-wise, not demand-driven, and we should reuse workspace data between transactions. The sub-DB objects must be kept structurally in synch with the workspace files (C4 mapping). *Locks* and access rights can be appended on the checked-out files, but this is p.t. not offered by EPOS.

Workspace *layout* is important, since tools must be properly fed and common subproducts effectively shared. For source programs we have chosen a close mapping between a `Family` breakdown in the DB and a file directory in the workspace. Shared families will be represented by symbolic links, and are usually set up by the parent transaction.

Only leaf transactions are supposed to do any real update work. Non-leaf transactions possess workspaces which serve as a shared pool for their children. These workspaces will not be regenerated upon child commits. We may also consider language-specific check-out tools to control e.g. `#include` directives in source programs.

The organization and connectivity of workspaces strongly influences the mechanisms for change propagation. E.g. use of global directories will make updates on these immediately visible, cf. NSE's symbolic links and Example 4 below.

Example 4. Partly shared workspaces.

A typical situation is a *parent* transaction T, with N *children* Ti. These may work on disjoint parts of a product to achieve a common goal. T's workspace is often a *global* pool of shared program components. Each Ti will temporarily keep components undergoing updates in local workspaces. When such components have been sufficiently tested, they will gradually be "pre"-checked-in or promoted to the global pool. This may cause change propagation into other local workspaces.

However, suppose that one of these Ti's wants to *delay* the local effects of a changed and promoted module M'. This is normally done by temporarily retaining a local copy of the previous version of M, or by equivalent manipulation with symbolic links or search paths. Such temporary changes in the local workspace should be properly recorded, for later integration of the new M'. This illustrates the diffuse borderline between pre-check-in propagation, check-in, and commit – see Figure 7. If commits were cheap and convenient, they might offer a more uniform solution.

[8] Remember, that the project infrastructure (Project KB) around a transaction/workspace can be customized using normal COV.

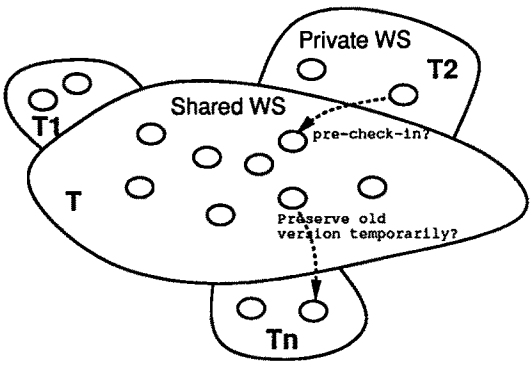

Figure 7: Common vs. private workspaces

5.3 Inter-Transaction Protocols: What, When, How?

The OVERLAP-NEGOTIATE subtask will establish a `ProtocolPolicy(Ti,Tj)`[9] for each directed (Ti,Tj) pair of overlapping transactions. Based on this information, a task network is instantiated and instrumented to implement the protocol *between* the actual workspaces. This is very close to the task network being automatically planned *within* a workspaces.

Within certain limits, the protocol can be re-negotiated, partly or in full. Changes in ambition- or product-overlap may change the network of cooperating transactions. We will initially assume stability here.

The expected work pattern is that such transactions have "shared" the Product Structure reasonably between themselves wrt. updates. The main communication pattern should be to incorporate the others' updated components into one's own environment in a controlled way.

The protocol contains the following information:

- **Granularity**: *what* shall be propagated (before check-in), e.g.:
  - Selected types: e.g. entire subproducts vs. single components.
  - Selected attributes, specially files being long attributes.
- **Timing**[10], or *when* to receive and implicitly when to send:
  - HARD coupling (*Busy*): All changes done by others are propagated immediately to me.
  - SOFT coupling (*Semi-Busy* – recommended): Propagate or promote such changes after manual *confirmation* by the *other* transaction.
  - TIGHT coupling (*Other-lazy* – default): Propagate after the *other's* check-in/ commit.

[9] If there were no external workspaces, i.e. only internal DB information, many of the below policies become superfluous or even impossible to realize.

[10] Adapted after Adele's proposed design for workspace coordination.

- LOOSE coupling (*Self-Lazy*): Propagate only after my *own* check-in/commit.

• How to accept:

- MANUAL-ACK: need explicit acknowledge after notification, followed by:
  * REJECT with request to:
    R1) DELAY: OK, but not-yet-ready,
    R2) VETO: return protest message, possibly with proposed changes,
    R3) REDUCE mutual version visibility.
  * ACCEPT with request to: AUTO-COPY or MANUAL-COPY (see below).
    A difference tool can be used to assist decisions.
- AUTO-ACK: notification is always sent, but no answer expected. This is followed by AUTO-COPY, except when there is directly conflicting textual updates – see merging in Sec. 5.4.

After delivery of a new copy: Possibly first perform a textual merge and agree mutually upon this (may take several iterations); then local change propagation as usual, see next subsection.

Note: the protocol regulates multiple updates of shared subconfigurations, but it is each user's responsibility to ensure consistency and completeness of his work.

• How to receive or workspace connectivity:

- AUTO-COPY: shared file (only for NO-ACK), indirect file link, or manipulated search path.
- MANUAL-COPY: e.g. to prepare for merging.

Some of the above policies are not independent, e.g. MANUAL-ACK excludes AUTO-COPY.

5.4 Reconciliation of changes: Merging and local propagation

Once a changed component X' from transaction T1 has been announced and accepted into another transaction T2, there are two main cases:

1. Physical merging of two textual components, X' and X".

   Assume that T2 also has made its own version X". To assist the physical merge, a multi-version text editor could be used to high-light different text fragments [SBK88]. For a start, we will use `emacs` with three windows: one containing X', one containing X", and one containing the merged X'+X", called X+. The initial X+ is the COV-merged text fragments of X' and X". After testing in T2, the new X+ should be propagated back to T1, and so on.

2. Change propagation to affected, local clients.

   Assume that T2 depends on the most recent X, in order to update Y to Y'. A typical example is `#include` dependencies between an interface and its client components. All clients must in due time be notified about relevant changes and re-processed accordingly – and so on transitively. Such waves of change propagation may require manual updating and retesting, and/or automatic re-building.

Change propagation can be severely constrained by applying *fine-grained* knowledge about the Product Structure [Tic86], cf. changes in the interface vs. in the body of an Ada package.

6 Open Areas

6.1 Handling of Aborts

If all the child transactions commit gracefully, there are no problems. But what happens if a child transaction, TC, aborts *before* proper commit? If changes from this child transaction already have been prematurely propagated to sibling transactions, we face some very unpleasant choices[11]:

- Rollback of the entire parent transaction.
- Rollback of the affected sibling transactions to a consistent state, assuming one of two cases:
  - Local *check-points* are available, e.g. by intermediate copies of changed components. This implies resetting the time to a point *before* the first change received from TC, i.e. *physical* check-points.
  - We can create a subtransaction of the affected sibling transactions *every time* a major change is imported from TC. Thus, we can rollback each wave of changes, i.e. *logical* check-points.

  Our intention with cooperating transactions is pre-commit interchange in a well-structured way. With a mixture of local changes and imported changes from different sources, we can get a very complex version graph *during* the transaction. In such intertwined cases, we should perhaps resort to traditional, more independent transactions, and perform step-wise merging later.

Cf. also previous comments on the granularity of commits.

6.2 Decision making support

EPOS should act as an *intelligent assistant* to support software work and its coordination. Task types express constraints and activation rules. This information is used to plan work breakdown and sequencing within a transaction.

However, there is little explicit support for cooperating transactions and general job planning, e.g. to:

- Decide work/project structuring, based on incoming CRs and existing Product Structures.
- Assist in detailed impact analysis of announced changes.
- Structure and evolve the version space.
- Suggest suitable cooperation protocols for a given piece of work.

[11] This withdrawal situation can be found everywhere in software development.

7 Conclusion and Future Work

EPOS runs on Unix-based workstations (Sun-3), and is implemented in 13,000 lines of C and 3,500 lines of SWI-Prolog. EPOSDB is based on C-ISAM, with client-server protocols using Sun RPCs. A simple graphical user interface is available.

EPOS PM has recently been extended from single-version to multi-version workspaces, offering inter-transaction negotiation and propagation. This is based on ambitions to formalize the *intent* of proposed changes. Trial implementation started in mid-Nov. 1990, and the first results are due in April 1991. The entire PM formalism can then be used for medium-scale experiments on a multi-user EPOSDB, offering the COV paradigm for versioning.

Many of the EPOS solutions can be applied to other CM systems and to fields like document processing and crowd control.

Still, there are many issues to be pursued:

- **Communication paradigm:**
  - Harmonization with DBMS triggers.
  - Better handling of interactive, cooperating tasks.
- **EPOS PM extensions:**
  - Type evolution and structuring.
  - Multi-actor monitoring and planning.
- **Transactions and workspaces:**
  - Introducing locks and access rights.
  - Better control of workspace layout.
  - Cheaper commits.
  - Incremental evaluation of configurations.
  - Less strict nesting of transactions.

Acknowledgements

Cordial thanks to the entire EPOS team.

References

[BE87] Noureddine Belkhatir and Jacky Estublier. Software management constraints and action triggering in the ADELE program database. In *[NS87]*, pages 44–54, 1987.

[COWL90] Reidar Conradi, Espen Osjord, Per H. Westby, and Chunnian Liu. Software Process Management in EPOS: Design and Initial Implementation. In *Proc. 3rd Int'l Workshop on Software Engineering and its Applications, Toulouse, France*, pages 365–381, 3–7 Dec. 1990. Also as DCST TR 15/90, EPOS Report 100, 12 p.

[Dow86] Mark Dowson. ISTAR — an integrated project support environment. In *[Hen86]*, pages 27–33, 1986.

[Fel79] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software — Practice and Experience*, 9(3):255–265, March 1979.

[GKY90] Bjørn Gulla, Even-André Karlsson, and Dashing Yeh. *Change-Oriented Version Descriptions in EPOS.* Technical Report 17/90, EPOS Report 102, 16 p., DCST, NTH, Trondheim, Norway, November 1990. (Accepted for Software Engineering Journal).

[Hen86] Peter B. Henderson, editor. *Proc. of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Palo Alto), *227 p.*, December 1986. In ACM SIGPLAN Notices 22(1), Jan 1987.

[HM88] Tani Haque and Juan Montes. A Configuration Management System and more (on Alcatel's PCMS). In *[Win88]*, pages 217–227, January 1988.

[Hol88] Per Holager. *Elements of the Design of a Change Oriented Configuration Management Tool.* Technical Report STF44-A88023, 95 p., ELAB, SINTEF, Trondheim, Norway, February 1988.

[LCD*89] Anund Lie, Reidar Conradi, Tor M. Didriksen, Even André Karlsson, Svein O. Hallsteinsen, and Per Holager. Change Oriented Versioning in a Software Engineering Database. In *[Tic89]*, pages 56–65, 1989. Also as DCST 27/89 — STF40-A89149 ISBN 82-595-5740-1 — EPOS Report 84, July 1989, Trondheim, Norway.

[Lem86] P. Lempp. Integrated computer support in the software engineering environment EPOS — possibilities of support in system development projects. In *Proc. 12th Symposium on Microprocessing and Microprogramming, Venice*, pages 223–232, North-Holland, Amsterdam, September 1986.

[LM85] David B. Leblang and G. McLean. DSEE: Overview and configuration management. In J. McDermid, editor, *Integrated Project Support Environments*, pages 10–31, Peter Peregrinus Ltd., London, 1985.

[MK88] Josephine Micallef and Gail E. Kaiser. Version and configuration control in distributed language-based environments. In *[Win88]*, pages 119–143, 1988.

[NS87] Howard K. Nichols and Dan Simpson, editors. *Proc. of 1st European Software Engineering Conference* (Strasbourg, Sept. 1987), *Springer Verlag* LNCS 289, *404 p.*, 1987.

[OrMrGB90] Erik Odberg, Bjørn Munch, Bjørn Gulla, and Svein Erik Bratsberg. Preliminary design of EPOSDB II. September 1990. 78 p.

[Roc75] Mark J. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, SE-1(4):364–370, 1975.

[SBK88] N. Sarnak, B. Bernstein, and V. Kruskal. Creation and maintenance of multiple versions. In *[Win88]*, pages 264–275, 1988.

[Sto86] Michael Stonebraker. Triggers and inference in database systems. In Michael Brodie and John Mylopoulus, editors, *On Knowledge Base Management Systems: Integrating Artifical intelligence and Database Technologies*, pages 297–314, Springer Verlag, 1986.

[Sun88] *Network Software Environment: Reference Manual.* Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, USA, part no: 800-2095 (draft) edition, March 1988.

[Tic86] Walter F. Tichy. Smart recompilation. *ACM Trans. on Programming Languages and Systems*, 8(3):271–291, July 1986.

[Tic89] Walter F. Tichy, editor. *Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, USA, 25-27 Oct. 1989, 178 p.*, ACM SIGSOFT Software Engineering Notes, November 1989.

[Win88] Jürgen F. H. Winkler, editor. *Proc. of the ACM Workshop on Software Version and Configuration Control, Grassau, FRG, Berichte des German Chapter of the ACM, Band 30, 466 p.*, B. G. Teubner Verlag, Stuttgart, January 1988.