Cooperating Transactions against the EPOS Database

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

1 Introduction

Traditional, unversioned DBMSes have a strict consistency model, coupled to serializable (short and system-executed) transactions. Versioned DBMSes contain partly overlapping, but mutually inconsistent sub-databases. Non-serializable (long and user-executed) and loosely coordinated transactions are needed, specially since update work can last for weeks.

The EPOSDB\(^1\) offers Change-oriented versioning (COV) [LCD+89] for software configuration management (CM). A transaction is connected to a change job or project task. This contains an intentional configuration description (2.1), which can be evaluated to a configuration, which again can be checked-out (converted) to a file-based workspace. Transactions can be long and nested (2.3).

EPOS supports software process management (PM) within a transaction through task networks and their project infrastructure. Tasks are managed by an Activity Manager, which includes an integrated Execution Manager and Planner [Liu90]. Rules for change propagation are expressed by task types with PRE-CODE-POST properties and various constraints. Propagation can be busy (triggered), periodic, opportunistic, or lazy (goal-oriented like Unix make). It can be automatic (derivative), or manual (human actor) with arbitrary review or negotiation procedures. Project customization is expressed by subtyping, type versioning using COV, and dynamic inheritance mechanisms.

The paper deals with the EPOS extensions for inter-transaction coordination. Raw (textual) merging comes for free with COV. Semantic merging is facilitated by pre-commit propagation and negotiation among overlapping transactions, according to agreed-upon protocols.

2 Cooperating Transactions: Background

2.1 Change-Oriented Versioning, COV

In COV [LCD+89], a functional change is described by a boolean global option. Version-rules (constraints, preferences, defaults) express legal or preferred option combinations. The database (DB) consists of fragments (deltas) tagged by a visibility, which is evaluated to False or True under a given version-choice, VC. The VC is a legal and complete option binding. A version is simply a sub-DB, consisting of DB fragments with True visibilities. An option binding with possibly Unset values is called an ambition, A. It specifies a multi-version DB, where changes from the local transaction will be visible. The VC is used for read and A for write (i.e. commit), and VC is a point within the multi-version space indicated by A.

A version-description, VD, is an incomplete version-choice. A product-description, PD, is a tuple of [Root objects, ER types]. A configuration is evaluated from a config-description, CD = [VD, PD], in three steps (see Figure 1):

- **C1** (VC,A) := Bind-version(VD, version-rules); see [GK91].
- **C2** Sub-DB := Version-select(VC, DB); facilitating COV-propagation between versions. One modified DB fragment, propagated from another transaction, is enough to make a configuration “dirty”.
- **C3** Config := Product-select(PD, Sub-DB); generating a Product Structure (PS) closure. Note that version-selection (C2) is done before product-selection (C3)!

A forth step will generate a workspace, WS:

- **C4** WS := Check-out(Config, DB/WS map).

\(^1\)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 ED0221.8457.

\(^2\)The domain is really ternary: False, True, and Unset.
new transaction. A consequence can be that the on-going transaction must be constrained, i.e. its ambition narrowed. Alternatively can the new transaction be constrained, delayed (i.e. serialized!), or delegated to somebody with proper access rights.

Policies for handling conflicting updates in overlapping transactions can be:

- **Priority**: let the first, last (default in EPOSDB), or some other 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*: One writer and several readers. The latter must nevertheless reconcile changes, due to transitive dependencies of a modified component.
  - *Only read locks* or no interference: No problems.
  - *No locks or optimistic synchronization* (default in EPOSDB): This may require subsequent merging.
- **Merging/integration**: the most general solution, and used in EPOS (3.1).

2.3 Nested Transactions

A nested, child transaction overlaps and constrains that of its parent, and possibly overlaps that of its siblings. 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. But with whom should inter-transaction cooperation be established? And how (manual or automatic) and when (lazy or busy) should it be carried out?

3 Cooperating Transactions: The Details

We shall describe subtasks for cooperating transactions, workspace organization, and communication protocols.
3.1 Subtask Infrastructure for a Transaction

A project task for a transaction will receive a change request, CR, and produce an updated configuration, NewC'. The PROJECT task describes the current transaction. Figure 2 shows the necessary subtasks and simplified data flows to implement cooperating transactions:

Figure 2: Tasks to support cooperating transactions.

- **START-CHECKOUT**: Initiate transaction, evaluate a NewC configuration from a chosen CD, and transfer and convert NewC from EPOSDB to the workspace.
- **OVERLAP-NEGOTIATE**: For control-level negotiation: Analyze ongoing sibling transactions for possible ambition/product-overlap with NewC, and send out messages to negotiate a contract protocol (3.3) for future communication and propagation.
- **WORKING**:
  - **COOPERATE**: data-level exchange and negotiation.
  - **PROPAGATE-IN**: Analyze and negotiate incoming change-notifications. For instance, it could fork off a MERGE subtask.
  - **PROPAGATE-OUT**: Monitor the local workspace, filter and analyze the changes, and possibly forward change-notifications to overlapping transactions.
  - **MERGE**: Perform textual merge using a multiwindow editor and special merge tools, and agree mutually upon this (may take several iterations); then local change propagation as usual.

All these activities are guided by the agreed-upon protocol, and interfere with the CHANGE-PROCESSING subtask below:

- **CHANGE-PROCESSING** for single-user PM: contains subtasks for work decomposition, sequencing, and change propagation within the workspace.
- **HANDLE-CHILDREN**: starts and terminates children transactions.
- **CHECKIN-COMMIT**: Transfer the updated workspace with NewC' to EPOSDB, and propagate/negotiate changes to siblings. Then commit transaction to the parent via EPOSDB, and notify overlapping, released configurations.

3.2 Workspace control

A DB-external workspace contains files and some control information. The DB-internal configuration must be kept structurally in sync with the workspace files. Only leaf transactions do any real update work, while non-leaf or parent transactions manage shared workspaces for their children. Workspace layout and connectivity is important, since tools must be properly fed, common sub-products effectively shared (by symbolic file links), and private workspaces dynamically adjusted. However, this is not yet supported. We will also need check-out tools to augment #include directives in source programs.

**Example.** Consider a parent transaction T, with N children Ti. These work on disjoint parts of a product to achieve a common goal. T’s workspace is a global pool of shared program components, and 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. However, suppose that one of these Ti’s wants to delay the local effects of some other’s 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.

This illustrates the diffuse borderline between “pre”-check-in propagation, check-in, and commit. – see Figure 3.
3.3 Propagation Protocols

The OVERLAP-NEGOTIATE subtask will establish a ProtocolPolicy(\(\text{T}_i, \text{T}_j\)) for each directed \((\text{T}_i, \text{T}_j)\) pair of overlapping transactions. Based on this information, a task network is instantiated (planned) and instrumented to implement the protocol between the actual workspaces\(^3\). 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.

A protocol contains information on:

- **Granularity:** *What* shall be propagated before check-in, e.g.:
  - Instances of selected types: entire subproducts vs. single components.
  - Selected attributes, specially files (long attributes).
- **Timing**\(^5\): *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.
  - LOOSE coupling (*Lazy*): Propagate only after check-in/commit.
- **How to accept:**
  - MANUAL-ACK: needs explicit acknowledge after notification, followed by:
    * REJECT with request to:
      R1) DELAY (OK, but not-yet-ready),
      R2) VETO (return proposed changes),
      R3) CONSTRAIN mutual version visibility.

\(^3\)This is very close to the task network being automatically planned within a workspaces.

\(^4\)Some policies are not independent, e.g. MANUAL-ACK excludes AUTO-COPY.

\(^5\)Adapted after Adele’s proposed design for workspace coordination.

4 Conclusion and Future Work

EPOS runs on Sun-3 workstations, and is implemented in 13,000 lines of C and 3,500 lines of SWI-Prolog. Trial implementation of cooperating transactions started in mid-Nov. 1990, and the first results are due in April 1991. The extensions amount to new types for transaction subtasks, mailboxes, and a generalized Planner.

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:

- Multi-actor monitoring and planning.
- Introducing locks and access rights.
- Better workspace control.
- Cheaper commits.
- Handling of pre-commit aborts.
- Less strict nesting of transactions.

References

