Planning Support for Cooperating Transactions in EPOS

Reidar Conradi*, Marianne Hagaseth
Norwegian Institute of Technology (NTH), Trondheim, Norway

Chunnian Liu
Beijing Polytechnic University, Beijing, P.R. China

Abstract

This paper describes a way to reduce the number of conflicts that may arise when several users cooperate to solve a task using a common database. The manual interaction between the users are made easier by supporting the project managers in planning of activities. Based on some interaction and an initial project division, we can analyse the connections between the activities, given as the result of impact analysis. Based on this, the project manager can choose to adjust the initial partitioning to reduce the dependencies between activities.

The impact analysis can also be used as a help for the project manager to schedule the activities. By doing this, more of the conflicts can be avoided.

However, conflicts cannot be completely avoided if some degree of concurrency should be achieved. The cooperation between users must be handled by allowing close interaction during the activities.

1 Introduction

The paper reports work on automatic planning of cooperating transactions in the EPOS Process Support Environment. Such assistance can be used by project managers to help organise software production (development, maintenance), according to incoming change-requests and according to the product/version space[1] of the software and related resources (tools, humans). It is a formidable task to manually "navigate" in the space of possible actions here.

The goal is to have powerful, low-level database mechanisms, e.g. propagation rules and locking. These can be driven by more high-level policies, e.g. communication protocols. These again are instrumented and/or generated from domain knowledge (Section 5). All this will be incorporated by an enlarged EPOS transaction model. The current paper reports work on a Transaction Planning Assistant to help organizing breakdown, scheduling and cooperation of subtransactions based on domain-specific information. An analogy to our transaction planner is AI-based program generators for (concurrent) programs. More classic project-planning to manage budgets, timing, and human resources are not considered.

2 Related work

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

Software engineering involves many concurrent actors and long update times. Since updates may involve *hard-to-predict and partly overlapping versions or subsystems*, traditional locking procedures will cause intolerable delays. Software Engineering (like CAD/CAM and related fields) therefore needs non-serializable (long and user-executed) transactions (often called design 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.

*Dept. of Computer Systems and Telematics, Norwegian Institute of Technology (NTH), N-7034 Trondheim, Norway. Phone: +47 73 593444, Fax: +47 73 594466, Email: conradi@idt.unit.no.

[1]The product space of the software is the actual software objects, and the relations between such objects. Version space of the software is the actual versions selected of the software product.

Many newer transaction models (e.g. in Gandalf, Marvel, COO from Nancy), use nested transactions, and these must handle both inter- and intra-transaction coordination. Description of other transaction models for cooperation is found in [1, 2, 3, 4].

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

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 (CSCW) is also related, but with focus on team organization and communication, and often not considering product/version structures (except graphic group-editors). There is often a globally persistent and shared blackboard, although there may be local and temporary workspaces. We can mention NSE from Sun [5], SPICE at CMU, and Lotus Notes, all with a central server and local database copies, but where the policies for broadcasting updates are rather strict. Experience from NSE indicates that there are few Write-Write conflicts (less that 1 per mill).

There has been much interest in database *triggers* [6]. That is, to have an *active* DBMS, which automatically performs consistency checks and side-effect propagation according to explicit event-condition-action rules. We must also consider "inter-version" propagation included, 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 versioning systems, like SCCS [7] and Make [8], offer no help for cooperating transactions. Adele [9] has high-level configuration descriptions and some workspace control, but only triggers to start rebuilds. PCMS [10] has document-flow templates, and Mercury [11] uses attribute grammars to guide simple change propagation. ISTAR [12] has subcontracts, but little formal cooperation. NSE is strong on workspace control, and DSEE [13] has some support for handling change requests. Few DBMS systems for software engineering can adequately treat cooperating transactions, or can handle configurations as conceptual entities both inside a database and in an external workspace.

Typical domain-independent, non-linear planning algorithms can be found in IPEM [14] and TWEAK [15]. TWEAK gives a formal treatment to the subject of non-linear planning. IPEM tries to integrate planning, execution and monitoring in fine granularity, mainly for exception handling. Both TWEAK and IPEM address the non-linear planning problem in a domain-independent way, with examples mainly from the Block World domain (robot applications).

Most AI planning rules have no formal Input/Output specifications, which are essential for software development tasks. Project customisation can be done by simple rule grouping and substitution, even if the rule space is rather flat. Process evolution can generally be supported by replanning and re-execution.

In the rest of this paper, we will first summarize the EPOS background, and then present the planning extensions for cooperating transactions.

3 The EPOS context

EPOS is a Software Process Environment [16]. Internal process models are represented as object-oriented and typed networks, being automatically (re)built. The task networks and all associated model information reside in a sub-database under the versioned EPOS–DB [17] [18]. Database accesses executed by above tasks are regulated by the embedding sub-transaction/sub-project [19]. The process model is expressed by an object-oriented and reflective process modeling language, called SPELL [20].

3.1 Consistency Model

The underlying consistency model for impact analysis is that consistency must be related to the whole product structure, not only to single objects. In the EPOS–DB, consistency of single objects is easily achieved since each EPOS–transaction maintains its own copy of the object. At the commit time, either the whole object

or none of it, is reflected in the database, ensuring that each single object always is consistent as viewed by the database. However, when the relations between objects are taken into account, the definition and maintenance of consistency becomes much more complicated. The problem is caused by the following:

- Overlaps in the set of objects that are accessed:

  There may be overlaps between the set of objects accessed by two different, concurrent[2] EPOS–transactions. This means that two objects accessed by two concurrent EPOS–transactions may be consistent when viewed separate, but inconsistent, when the relationship between the objects is taken into account.

- Different semantics on the relations:

  Maintenance of consistency in the database is difficult because of the semantics of the relations between objects. This problem is more difficult than just having referential integrity in a relational database. By referential integrity, the database can ensure that all tuples referenced in a foreign key field exists in some other tables. However, relations in an object–oriented model can have different semantics, thus, such simple rules cannot be used to maintain consistency. The best we can do is to follow the relations, and notify the users about possible inconsistencies that may have arisen due to changes to some of the objects. Then, it is up to the application to decide what actions must be taken to maintain the consistency.

The result of this is the following definition of consistency, viewed from the database:

1. A single, isolated object is always consistent viewed from the database, since every transaction maintains its own copy of the object.
2. Several, related objects are consistent if the database ensures that every changes are notified to the affected transactions, and if each owner of a transaction has had the the possibility to compensate for the updates.

4 Cooperating transactions in EPOS

EPOS–DB offers nested and long (non-serializable) transactions in a client-server architecture. Each internal database transaction is associated to a `Project` task and to an external file-based workspace. A transaction operates on a given database *version* "slice" (the visible sub-database), selected by a *version-choice* serving as a read filter. The transaction also specifies the scope of local changes, selected by a *version-ambition*, i.e. which other database versions might be impacted. The version-choice and the version-ambition describe a part of the *version space*, and are intentionally expressed as sets of "option" bindings. An ambition implies a *write lock* on the associated version subspaces (sub-DBs), and with access only to product subspaces (sub-products) within these versions. However, we are not constraining access to whole instances, only to versions of these instances.

The relevant part of the *product space* is described through an intentional *readset* (a set of root components and a set of directed relation types for transitive closures) and an extensional *writeset* (enumeration). Components may have normal read/write locks.

The above version- and product space descriptions are part of the *workorder* for the associated project task. In addition comes process-related information (tools, humans, time-constraints) and given cooperation protocols.

A child transaction overlaps and constrains that of its parent, and possibly overlaps that of its siblings. Transaction *overlap* is primarily defined in the version space by overlapping ambitions. In case of version overlap (no version overlap means classic variants), it is interesting also to consider product overlap.

After child commit, changes are propagated to the parent, which must handle possible update conflicts between the children, using e.g. policies like Rollback (intolerable), Priority (the last one wins?), Access locks (classic access locks), Optimistic (soft locks with notifiers), followed by Integration/reconciliation

[2] Concurrent means that the transactions are overlapping in time.

(negotiation, merging) etc. Clearly, such update conflicts will disappear, if strict serialization and locking are globally enforced. However, this will cause excessive waiting among developers. Thus, pre-commit cooperation is a pragmatic way to prevent, regulate and clarify update conflicts in case of overlaps.

Change propagation occurs in two steps:

1. First, we have to decide mutual version visibility or overlap, and set up (low-level) and pairwise **inter-transaction** protocols for pre-commit negotiation and propagation. That is, to *whom* should inter-transaction cooperation be established, *what* components are involved (granularity, type), *how* (automatic or manual) and *when* (eager or lazy) should it be carried out etc.? The presented planning work aims at giving more high-level support for such coordination.

2. Then, there is conventional **intra-transaction** change propagation regulated by normal task networks, regardless of the source and nature of the change. The existing Planner is used to (re)generate such networks, using domain knowledge in form of task types, product structures etc. [21].

5 High-level Transaction Planning in EPOS

`Project` is the EPOS term for execution environment, from a full scale project to a simple task performed by a single user in his own work environment.

Good planning can reduce the need for manual cooperation between the developers. A Project Manager *meta-tool* uses its local **Transaction Planning Assistant (TRAPLAS)** to advice the human project manager on this. TRAPLAS tries to minimize dependencies between proposed subprojects, or to minimize the cost of such dependencies, by attempting to:
1) partition an update job into "natural" subprojects,
2) schedule such subprojects, and
3) suggest communication patterns, that can be expanded into cooperation protocols.

The planning depends on the readsets and writesets associated with each project. An *impactset* is computed for each pair of transactions to describe their inter-connection. That is, if a component is in the writeset of one transaction and directly or transitively in the readset of another, it belongs to the impactset of the former. Typical relations used to generate the readset and impactset are `FamilyOf` (subsystem hierarchy), `ImplementedBy` (between interface and body), and `DependsOn` with subtypes (general dependencies). Each component and relationship has an associated *weight* and each relationship also a *direction*.

The *domain* knowledge used by TRAPLAS consists of global and local consistency constraints stored in types and meta-types, intentional project goals (e.g. ambitions, writesets), added semantics specially on relations, existing product/task structures, product ownerships, and possibly personnel allocation. This knowledge can be manipulated and versioned on a project basis.

5.1 Project Partitioning

This section describes how impact analysis can be used to support the project manager in the partitioning of a project into subprojects. This project partitioning is done before the actual work starts.

The `Project` type in EPOS–PM has two subtypes, `CompositeProject` and `AtomicProject`. `CompositeProjects` have subprojects, while `AtomicProjects` have not. The projects in EPOS–PM are organised in a hierarchy which follows the transaction hierarchy in EPOS–DB. This is because one EPOS–PM–project is connected to one EPOS–DB–transaction. We assume that the project partitioning continues until the leaf node projects are all atomic. The relation between project and transaction is that each project has one transaction associated with it. Both projects and transactions are organised in a hierarchy so that a project can contain several subprojects and a transaction several subtransactions.

5.1.1 Initial Project Partitioning

In the initial project partitioning, a composite project is divided into n subprojects. These subprojects may be either composite projects or atomic projects.

1. **How the initial partitioning is determined:**

   An important question is *how* the initial partitioning of the composite project is determined. This question includes:

   (a) Which criteria are used to do the initial partitioning?

   If impact analysis is used to reduce the interconnections between the subprojects before the result is presented to the project manager, the initial clustering is not important. However, the project manager may want to use some special strategies for the initial partitioning, even if this may be changed later.

   i. One possibility is to describe the partition in terms of data that is to be accessed by each subproject.
   If each object already has an ownership associated with it, this can be used to define the partitioning. This means that every object must have an owner associated with it, and that each object can have only one owner.
   Another way to partition the data between the subprojects is to follow the subsystem structure. In this case, the ownerships information is not used in the initial project partitioning. However, it can still be used in the impact analysis, for instance included as weights on objects or relationships. This can then be used to compute the interconnection between two projects, and in the next step be used to adjust the initial project division.

   ii. The initial project partitioning could be done based on the tasks that the project consists of. In this case, we would need a way to specify the tasks, and also a way to relate these definitions to the actual data, since the impact analysis needs predefined access sets as input.

   (b) How many subsets are created?

   If ownership is used, initial project partitioning leads to the same number of subprojects as the number of owners. If the ownerships information is not used, but projects are defined based on substructures, the project manager may define as many subprojects as he wants. If subprojects

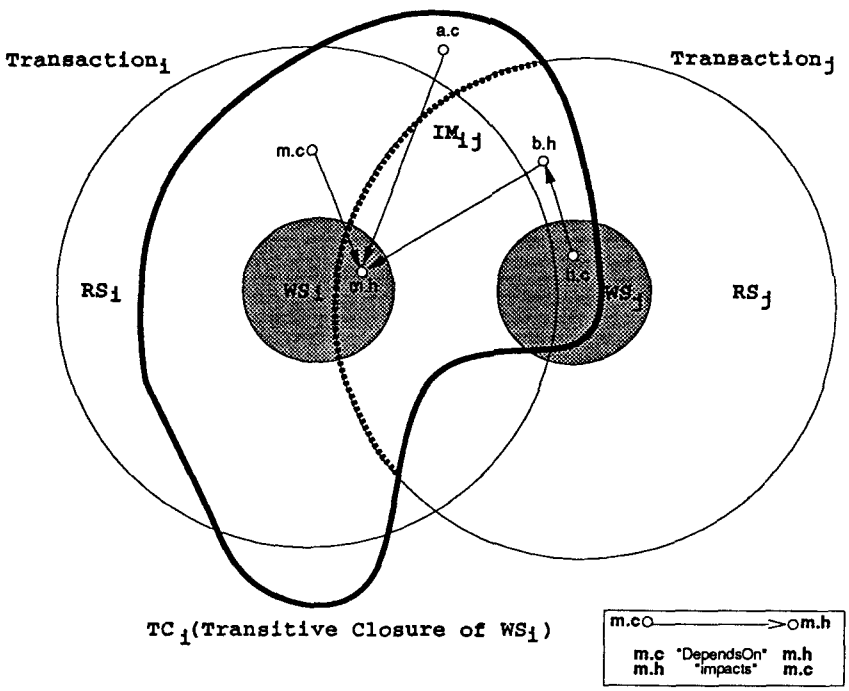

Figure 1: The Result of Impact Analysis.

follows the specified tasks, it is natural to have as many subprojects as tasks, or possibly less if some tasks are executed in the same project.

2. **How the partitioning is described:**

   The projects are described by its readset (*RS*) and writeset (*WS*). This means that the project partitioning is described by dividing the readset and writeset into *n* readsets and writesets. Thus, if a composite project is described by $[RS, WS]$, its subprojects are described by $[RS_i, WS_i]$, for all *n* subprojects. The readsets of the subprojects may possibly be overlapping. The same is true for the writesets of the subprojects. Also, the writeset of one project may overlap with the readset of another project. Further, we always have that $WS_i \subseteq RS_i$.

   The readset is specified by first listing the objects that the subproject can read. Then, relations from these objects are followed to include all objects that relates to these objects. This step is repeated until no more objects can be added to the readset. The dependency relations we are following are `DependsOn`, and its subtypes `TextIncludes` and `SourceImports`, and the relation `ImplementedBy`. For instance, if a module *x* is in RS and *x* `DependsOn` module *y*, or *y* is `ImplementedBy` *x*, then *y* is added to RS. In addition, relations in the product structure are followed, for instance, `FamilyOf` and `ComponentOf`.

   The writeset is specified by just listing the actual objects that are accessible for writing.

3. **How the interconnection between subprojects is described:**

   The following kinds of overlaps can arise due to the partitioning of the readset and writeset of the composite project:

   (a) $RS_i \cap RS_j \neq \emptyset$:
   Overlaps in the two readsets causes no problems as long as none of the common objects are updated.

   (b) $RS_i \cap WS_j \neq \emptyset$:
   The transactions can still execute in isolation, even if their readsets and writesets overlap. However, certain sequences of reads and writes to the common objects will lead to inconsistencies. By inconsistency, we mean that the transactions are interleaved in a non–isolated way, and that a transaction are allowed to read data changed by another transaction before it is committed. One sequence of accesses that leads to inconsistency is a `read` by $Transaction_i$, followed by a `write` by $Transaction_j$, followed by a `read` by $Transaction_i$. Another sequence is a `write` done by $Transaction_i$, followed by a `read` by $Transaction_j$, followed by a `write` by $Transaction_i$[3]. Since we allow these sequences to happen, every overlap of readsets and writesets are handled as possible conflicts when impact analysis is done in the project partitioning.

   (c) $WS_i \cap WS_j \neq \emptyset$:
   Overlaps between the two writesets may cause inconsistencies for certain sequences of access. This sequence is basically a `write` by $Transaction_i$, followed by a `write` by $Transaction_j$, followed by a `write` by $Transaction_i$[4]. When doing the project partitioning, we do not consider the sequences of operations that can be issued by the transactions. We only consider the overlaps, that is, if there is an overlap between the writesets, we assume that the projects will be interconnected in some way.

5.1.2 Impact Analysis to Improve the Project Partitioning

After the initial project partitioning is done, an improvement in the partitioning is presented to the project manager by doing *Impact Analysis*. The impact analysis computes the interconnection between every pair of subprojects, based on the initial partitioning, and based on *weights* that are put on the relations. The result of Impact Analysis is shown in Figure 1. It shows how $Transation_i$ is interconnected to $Transaction_j$, that is, how the conflicts between the two transactions impacts $Transaction_j$.

The *interconnection* between two transactions is a way to describe how severe the conflicts are that may arise if the two transactions are executed in parallel, that is, if they are overlapping in time. By adding weights

[3] Several reads and writes can be added to these sequences. However, it is still the same two problems.

[4] Several reads and writes can be added to this sequence.

on objects and relations, the possible conflicts can be described as a number. This means that the conflict that may arise by executing these transactions in parallel can be compared with other conflicts. Then, we can present to the project manager which two transactions may lead to most conflicts. *Weights* on relations and object types are used to compute the weight of an impactset. We have that WT_{ij} is the weight of the impactset IM_{ij}.

- The big circles represent the readsets (RS) of the two transactions, with the enclosed writesets (WS) being shaded circles. This is the input to the impact analysis, together with the relations that are to be traversed.
- TC_i is the transitive closure of WS_i. It denotes the objects which may be influenced by updates made to objects in the writeset WS_i of $Transaction_i$. We call this the full impact of WS_i to the rest of the database, since it includes every object that is related to some object in the writeset. Both the product structure and other relations are traversed to obtain every object that in some way relates to objects in the writeset of $Transaction_i$.

  In Figure 1, TC_i is the area inside the fat lines, which here extends outside the two transactions displayed.
- $IM_{ij} = TC_i \cap RS_j$ is the impact of WS_i on RS_j. It is indicated in Figure 1 as the area between the dotted line and the fat line. We take the intersection between the transitive closure and the readset RS_j to be able to describe the impact of the writeset of $Transaction_i$ on the readset of $Transaction_j$.

  The weight of IM_{ij} is WT_{ij}. This can be presented to the project manager as a measurement of the impact of WS_i on RS_j. Based on this information, he can choose to move one or more of the objects in WS_i to another writeset. Then, impact analysis must be performed once more to compute the new impactsets.
- $IM_i = \cup_{j=1}^{n} IM_{ij}$, where $i \neq j$ is the total impact of the writeset of $Transaction_i$ on the readset of all other transactions. The impact of WS_i on its own readset is deleted because it is irrelevant for describing the interconnection between $Transaction_i$ and the other transactions. The weight of IM_i is WT_i.

The project manager can use the result of Impact Analysis to manually adjust the initial partitioning by reducing inter-project dependency since one transaction is associated with one project.

The project manager prepares N sub-workorders (sub-WOs) based on the final partitioning of [RS,WS]. A WorkOrder includes a change-request and a configuration description. For each sub-WO, a subtask of $PROJ$ is automatically generated. As the result, a network of subprojects (plus other subtasks such as `SchemaManager` task etc.) of the project is generated.

5.1.3 Example

An example of the usage of impact analysis is shown in Figures 2 and 3. The notation is the same as in Figure 1. The impactsets are indicated by dotted lines. Figure 2 shows the initial transaction division. $Transaction_1$ has the writeset WS_1 containing `a.c`, `a.h` and `d.h`. The readset RS_1 contains `b.h` in addition to the objects in the writeset. $Transaction_2$ has the writeset WS_2 containing `p.c` and `b.c`, and the readset RS_2 containing `b.h` and `p.h` in addition to the objects in the writeset. Further, the figure shows the impactsets IM_{12} and IM_{21}. The impactsets contain the following objects:

- IM_{12}: `p.c`, `p.h`, `b.c`, and `b.h`.
- IM_{21}: `a.c`.

The impactset IM_{12} is found by taking the intersection between TC_1 and RS_2. When the transitive closure TC_1 is computed, the edges are follwed in the reverse direction.

The project manager can choose to change the writesets to try to reduce the impact between the transactions. In Figure 3, the object `d.h` has been moved from WS_1 to WS_2. The new impactsets contain the following objects:

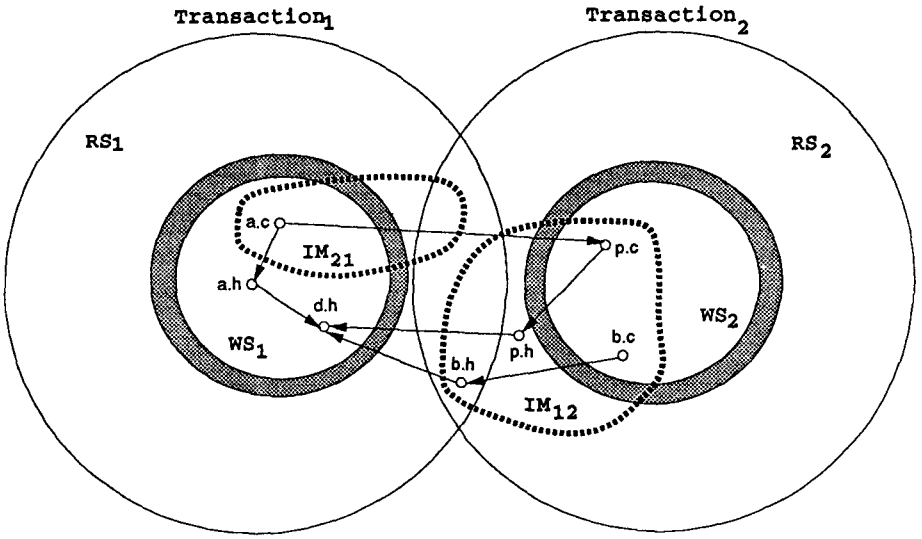

Figure 2: Example: Initial Transaction Division.

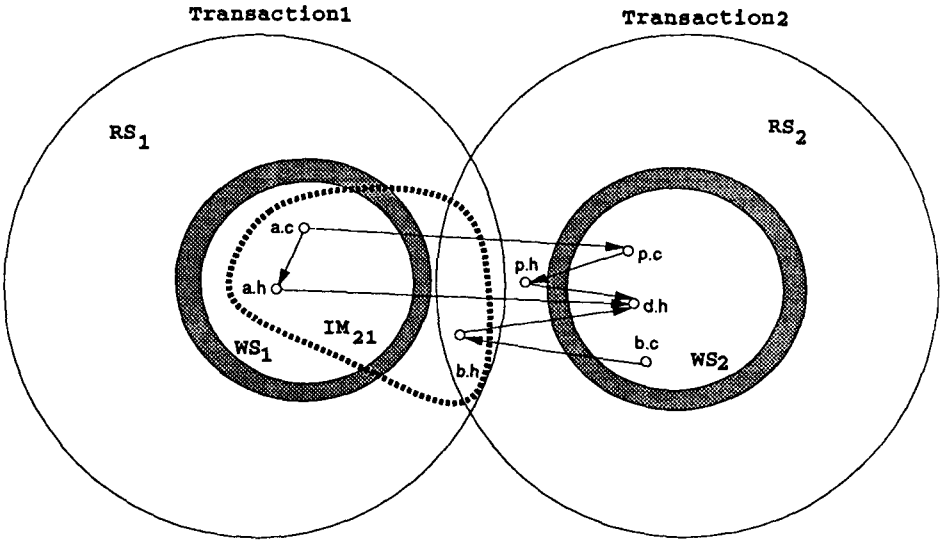

Figure 3: Example: Refined Transaction Division.

- IM_{12}: $\emptyset$
- IM_{21}: `a.c`, `a.h` and `b.h`.

This means that the reorganizing of the writesets has reduced the dependencies between the two transactions. What the impact analysis does, is to compute the actual impactsets, which are then used by the project manager to find a better transaction division.

The effect of changing the writesets of the transactions is that the definition of the associated projects is changed. This changing is done to find the projects that leads to as few conflicts between the projects as

possible. In this way, we can use the product structure to construct a project structure with fewer inter–project conflicts.

5.2 Scheduling

The $PROJECT$ owner does further scheduling of the subprojects, using advice from Impact Analysis. Note that optimal scheduling of serial or cooperative transactions, based on more detailed read/write patterns, is a NP-complete problem [22]. Also note that the partitioning decided above, is not independent of scheduling, if optimal solutions are sought, see below examples. For instance, we can commit small and more important changes first (serialization!). We can also run "well-balanced" and mutually dependent transactions in parallel with proper coordination. Alternatively, we can run "tricky" updates in a strict sequence (if both WT_{ij} and WT_{ji} are big), or apply temporary separation (as variants) followed by later merge jobs.

For instance, consider subtransactions T1 and T2, scheduled as:

T1; T2;	(serial: T2 is based on T1's work)
T2; T1;	(serial: T1 is based on T2's work)
T1 $\parallel$ T2;	(later merge: merge(T1,T2))
T1 $\Leftrightarrow$ T2;	(cooperative, thus no later merge)

The previous impact analysis is used for such planning, although we foresee a strong interaction with the human project manager.

5.3 Cooperation Protocol

Here we have to negotiate, maintain and later interpret rather low-level protocols, P_{ij}, among each pair of cooperating (overlapping) transactions (subprojects). Only atomic transactions need to have exchange protocols.

The protocol contains information on the following items:

- **Granularity**: e.g. simple instances of selected types vs. entire subproducts.
- **Coupling**[5], or when to receive. This may be: `Eager` where all changes are propagated immediately; `Lazy` (recommended) to propagate or promote changes after manual confirmation from the causing transaction; or `Delay-other` to delay propagation till after *other's* check-in/commit, or delay them to just before *own* commit (`Delay-own`).

  It is important to be able to delay the effect of other's (pre-committed) updates to a later time, cf. "copy-on-read", although we eventually have to incorporate such changes.
- **Acceptance Rule**, being either AUTO-ACK or MANUAL-ACK. AUTO-ACK requires that a notification always is sent, but no answer expected. This is followed by AUTO-COPY, if there are no conflicting textual updates (otherwise a merging must be performed). MANUAL-ACK requires explicit acknowledge after notification, followed either by:

  1. REJECT: either DELAY as we are not-yet-ready, VETO with proposed changes returned, or CONSTRAIN to limit mutual version visibility.
  2. ACCEPT, with request to AUTO- or MANUAL-COPY, see below.
- **How To Receive**, or workspace connectivity: AUTO-COPY implies a shared file, an indirect file link, or a manipulated search path. MANUAL-COPY means explicit copying and possibly merging.

The protocol may have to be adjusted when new sibling transactions start and commit. Changes in version- or product-overlap may change the network of cooperating transactions, but we will initially assume stability here. The protocol can also be dynamically supplemented, and even re-negotiated in simple cases. Some policies are not independent, e.g. MANUAL-ACK excludes AUTO-COPY.

[5] Adapted after Adele's proposed design for workspace coordination [23].

For inter-transaction transfer we partly use an internal database "tunnel" mechanism to do pre-commit propagation of general instances, partly an external mailbox mechanism for simple notifications. Neither of these are described here. In addition comes implicit communication between external workspaces with partly shared files (e.g. through symlinks), also not dealt with here. Reconciliation with merging is not dealt with either.

The idea is to have TRAPLAS in cooperation with the human manager/developer to set up general communication patterns, that can be translated into the above low-level protocols.

To recapitulate: When $Transaction_i$ makes a change, it should eventually propagate or notify the update to all overlapping transactions $Transaction_j$ (i.e. IM_{ij} indicates connections). The idea is to have TRAPLAS, in cooperation with the human manager/developer, to set up general communication patterns, mainly based on the previous impact analysis. These patterns can later be translated into the above, low-level protocols.

Thus, we will apply some heuristics on what seems like reasonable communication patterns. For instance, changes to shared libraries may be propagated rather unconditionally (`Eager`, AUTO-COPY), changes to project libraries may be propagated when the receiver is ready (`Delay-own`, MANUAL-COPY), while changes to mutually dependent modules may require much negotiation as indicated above (`Lazy`, MANUAL-ACK).

5.4 Further decomposition

The previous partition-scheduling-cooperation protocol steps can be repeated during execution of composite projects, until we reach atomic ones doing the real update work.

We have chosen to gradually decompose, not make the full decomposition in one step, even if the latter may give a more optimal plan.

As mentioned in the introduction, we should also consider possible time constraints and available resources, e.g. tools, persons and their availability.

6 Conclusion and Future Work

TRAPLAS serves as a translator between more goal-oriented domain knowledge and the underlying database support. Without such a link, either more rigid update policies have to be enforced, or more flexible cooperation patterns may become unwieldy. Some issues to pursue are the following:

- **Formal transaction modeling**: a more unifying transaction model to formally express domain-knowledge, and considering consistency and user roles. We also need to formalize workspace environments and manage incremental evaluation of configurations.
- **A Transaction Description Language**, integrated or harmonized with SPELL.
- **Version space planning** to manage version-ambitions.
- **Dynamic** *writesets, ambitions, protocols etc.*
- **Validation** by realistic industrial scenarios.

Acknowledgments

Thanks to the entire EPOS team.

References

[1] Gail E. Kaiser and Calton Pu. Dynamic Restructuring of Transactions. In *[24]*, pages 265–295. Morgan Kaufmann, 1991.

[2] 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.

[3] Mary F. Fernandez and Stanley B. Zdonik. Transaction groups: A Model for Controlling Cooperative Work. In *3rd International Workshop on Persistent Object Systems, Their Design, Implementation and Use.*, pages 341–350, january 1989.

[4] Andrea Skarra. Concurrency control for cooperating transactions in an object-oriented database. *SIGPLAN Notices*, 24(4):466–473, february 1989.

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

[6] 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.

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

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

[9] Noureddine Belkhatir and Jacky Estublier. Software management constraints and action triggering in the ADELE program database. In *[25]*, pages 44–54, 1987.

[10] Tani Haque and Juan Montes. A Configuration Management System and More (on Alcatel's PCMS). In *[26]*, pages 217–227, January 1988.

[11] Josephine Micallef and Gail E. Kaiser. Version and configuration control in distributed language-based environments. In *[26]*, pages 119–143, 1988.

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

[13] 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.

[14] José A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proc. of AAAI'88*, pages 83–88, 1988.

[15] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[16] Reidar Conradi, Espen Osjord, Per H. Westby, and Chunnian Liu. Initial Software Process Management in EPOS. *Software Engineering Journal (Special Issue on Software process and its support)*, 6(5):275–284, September 1991.

[17] Anund Lie et al. Change Oriented Versioning in a Software Engineering Database. In *Walter F. Tichy (Ed.): Proc. 2nd International Workshop on Software Configuration Management, Princeton, USA, 25-27 Oct. 1989, 178 p. In ACM SIGSOFT Software Engineering Notes, 14 (7)*, pages 56–65, November 1989.

[18] Bjørn P. Munch, Jens-Otto Larsen, Bjørn Gulla, Reidar Conradi, and Even-André Karlsson. Uniform versioning: The change-oriented model. In *[28]*, pages 188–196, 1993.

[19] Reidar Conradi and Carl Chr. Malm. Cooperating Transactions against the EPOS Database. In *Peter H. Feiler (Ed.): "Proceedings of the 3rd International Workshop on Software Configuration Management" (SCM3), Trondheim, 12–14 June 1991, 166 p. ACM Press Order no. 594910.*, pages 98–101, June 1991.

[20] Reidar Conradi et al. Design, use, and implementation of SPELL, a language for software process modeling and evolution. In *[29]*, pages 167–177, 1992.

[21] Chunnian Liu and Reidar Conradi. Automatic Replanning of Task Networks for Process Model Evolution in EPOS. In *[30]*, pages 434–450, 1993.

[22] Claude Godart. COO: A transaction model to support COOperating software developers COOrdination. In *[30]*, pages 361–379, 1993.

[23] N. Belkhatir, J. Estublier, and W. L. Melo. Adele2: A Support to Large Software Development Process. In *Proc. 1st Conference on Software Process (ICSP1), Redondo Beach, CA*, pages 159–170, October 1991.

[24] Ahmed K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, 611 p., 1991.

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

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

[27] Peter B. Henderson, editor. *Proc. 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.

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

[29] Jean-Claude Derniame, editor. *Proc. Second European Workshop on Software Process Technology (EWSPT'92), Trondheim, Norway. 253 p.* Springer Verlag LNCS 635, September 1992.

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