Consistency Management for Multiple Perspective Software Development

Wai Leung Poon
Dept. of Computing,
Imperial College,
180 Queen's Gate,
London SW7 2BZ, UK
wlp@doc.ic.ac.uk

Anthony Finkelstein
Dept. of Computer Science,
City University,
Northampton Square,
London EC1V 0HB, UK
acwf@cs.city.ac.uk

Abstract
This paper discusses consistency management for realising viewpoint-based software development in an aggressively decentralised environment. Relations between design artefacts are specified in consistency checks. Possible interference between concurrent actions of users is handled by modelling the interaction between these actions terms of transactions. We briefly describe how software process model knowledge can be exploited to enhance concurrency subject to the correctness criteria of serialisability. Open issues and the research agenda underlying the development of such an environment are sketched.

KEYWORDS: concurrency control, transaction management, software process, software development environment

1. Viewpoint-based Decentralised Environments

Software development commonly involves geographically distributed participants with different objectives and expertise. A wide range of data representations, tools and development methodologies may be adopted. This is referred to as multiple perspective software development [1]. Such situations can be modelled in terms of the collaboration of autonomous development environments, each of which encapsulates one or more dimensions of the heterogeneity. The resulting system, which is composed of a collection of autonomous environments facilitates reuse and evolution as loosely coupled environments may be added, removed or replaced. In addition, because the system does not rely on centralised control, it avoids a single point of failure and permits higher levels of concurrent activities thus enhancing productivity.

An environment may be characterised in terms of five components namely *domain*, *style*, *work plan*, *specification* and *work record*.

The *domain* component describes the area of concern of the environment and sets its context in the overall system under construction.

The *specification* component describes the domain in the representation scheme supported by the style component. The specification component contains design artefacts. Design artefacts created in the software development are stored in design databases. A simplistic view of design database is adopted. A database is a collection of design objects. Each design object is a pair of domain and value. The value determines the *state* of an object. We focus on only two basic operations on objects namely read and write.

The *style* component describes the representation scheme supported by the environment. The domain component and the style component prescribe the desirable relations [2] between artefacts within an environment and between different environments. Ideally, the *consistency* of a design is defined in terms of a collection of *consistency constraints* on the design objects. Each consistency constraint governs a collection of design objects. It specifies acceptable combinations of states of design objects. A design will be in a *consistent state* if all the consistency constraints are satisfied.

The *work plan* component describes development actions together with a strategy for producing desirable products. The software process model being examined is described by a collection of rules in terms of "situation-action" pair discussed in [3]. An action will be carried out if the current state of the process model meets the situation described. To complete the specified action, a sequence of operations have to be carried out within a software development environment or across different ones. Operations are carried out automatically or manually.

The *work record* component records the history and current state of the development.

It is important to ensure that all consistency constraints are satisfied eventually. Ideally, when changes have been made, tests could be performed to check if the consistency constraints have been satisfied. However, it is not always possible to state all consistency constraints in advance. Consistency constraints may be elicited during the software development. Some may not be stated explicitly. For instance, a user may create several objects in particular states and assume that they are consistent with each other without explicitly defining any test to check the consistency. Because the consistency of such changes are beyond the control of the supporting environment, we assume that a user, who performs any sequence of operations on objects, will bring the design from one consistent state to another one if the initial state of the design is a consistent one and the update satisfies all explicitly stated consistency constraints. Using these assumptions, common causes of inconsistency are:

Incorrect actions. Users perform actions to change design objects in such a way that one or more consistency constraints are violated.

Non-atomic actions. A consistency constraint may govern a collection of design objects. If only one operation can be performed on one of the objects at a time, this inevitably means violating the consistency constraints temporarily until all changes on related objects has been made.

Bad dependencies. Inconsistency may be caused by the uncoordinated actions of different users on shared or related objects. A description [4] of the three types of problems are: lost update, uncommitted dependency and inconsistent analysis .

Different perspectives. Contradicting requirements and unresolved conflicting design decisions may be advanced by users in different environments.

We model the interaction between different environments in terms of transactions and shows how the problems described above may be addressed in a unified framework.

2 Scenario

The behaviour of the environment may be best illustrated by an example. We illustrate how users collaborate with each other in terms of transactions while skipping details of enabling techniques until the next section. First of all, we assume that at a certain point of the development, the specification, which is composed of miniature data flow diagrams, is as shown in figure 1.

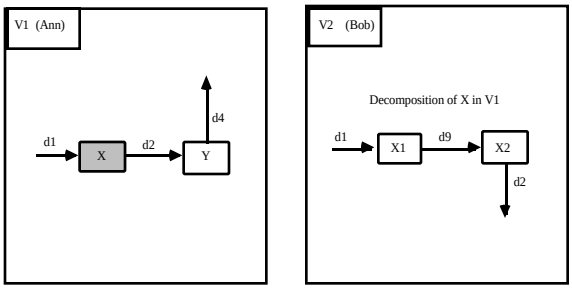

Figure 1. Initial states of two consistent specifications.

The diagram V2 is a decomposition of the node X in the diagram V1. Ann and Bob are responsible for the development of V1 and V2 respectively. We assume that the diagram is represented by two kinds of objects namely node and link. A node has a name as its sole attribute. A link has three attributes: a name, the node which it connected from and the node which it connected to. The name of node and link are assumed to be unique. A data flow diagram is self-consistent if every link has a source and a destination unless it flows in or flows out of a diagram. We consider that two perspectives are consistent with each other as long as the data flows which flow in or out of a diagram V2 match those of node X in diagram V1. Based on the definition, the initial contents of the specifications stored in two environments are as follows:

V1	V2
node(x)	node(x1)
node(y)	node(x1)
link(d1, #, x)	link(d1, #, x1)
link(d2, x, y)	link(d1, x1, x2)
link(d4, y, #)	link(d2, x2, #)

represent an external connection

Now, Bob decides that it is necessary to replace the process x2 with x4 as shown in figure 2. Without committing ourselves to the details of implementation, operations in the transaction, which we will referred to as T1, can be represented as:

```
insert[node(x4)]
read[flow(d9, x1, x2)]
write[flow(d9, x1, x4)]
read[flow(d2, x2, #)]
write[flow(d2, x4, #)]
delete[node(x3)]
commit
```

For expository simplicity, we use the short notation, $i_1[x4]$ $r_1[d9]$ $w_1[d9]$ $r_1[d2]$ $w_1[d2]$ $d_1[x3]$ c, t o represent the sequence of operations. The symbols **i**, **d**, **r** and **w** represent the operations of insert, delete, read and write respectively. In addition, either **a** or **c** is used as the last operation of a transaction to denote that the transaction is aborted or committed respectively. The objects are referred to by their name while the exact values of their attributes are ignored. The number in suffix, '1' in this case, may be used to denote that operations belong to the same transaction in case of possible ambiguity. Because T1 does not violate any known consistency constraints, a commit operation, c, can be used to end the transaction.

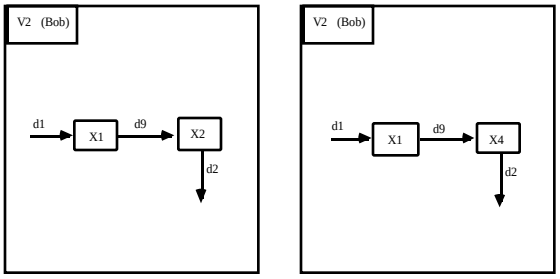

Figure 2. Bob decides to replace the node x2 with x4.

In a multi-user environment, it is possible that concurrent tasks are performed by users who access related objects at the same time. One cause of concurrent access is that operations in one perspective may need concerted actions from other perspectives in order to maintain the design in a consistent state. For instance, Ann thinks that it is desirable to produce a data flow d3 from the node x as shown in figure 3a. Because the node x is further decomposed in perspective V2, it is beyond her knowledge or authority to decide whether d3 can be made available. The decision depends on the confirmation of Bob. This dependency can be represented by checks between two environments. Hence, a transaction in Ann's perspective triggers another transaction which is to be performed in Bob's perspective. As a result, a dependency is established between two transactions. On the one hand, Ann's transaction should not be committed until the Bob's one has succeeded. On the other hand, if Ann retracts her changes by aborting the transaction, it would be reasonable to abort the transaction in Bob's perspective as well.

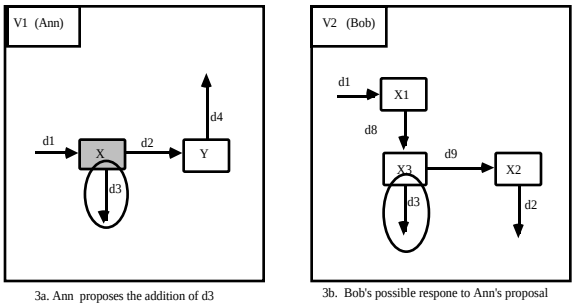

Figure 3. Dependency between transactions across different perspectives

Bob, who is responsible for constructing the diagram V2, decides that neither the process x1 nor the process x2 can produce the data flow d3. Hence, he adds an extra step named x3, which takes an intermediate result, denoted by d8, from x1 and produces the original d9 for x2 while producing d3 as required by Ann. The changes, which will be collectively referred as transaction T2, are

shown in the figure 3b. (Of course, it is also plausible that Bob rejects Ann's proposal, then a new proposal may have to be made.) The short notation, $i_2[x3]$ $r_2[d9]$ $w_2[d9]$ $i_2[d8]$ $i_2[d3]$ c, is used to denote the following operations:

```
insert[node(x3)]
read[flow(d9, x1, x2)]
write[flow(d9, x3, x2)]
insert[flow(d8, x1, x3)]
insert[flow(d3, x3, #)]
commit
```

Nevertheless, there is interference between two transactions on the data flow d9. In fact, there are conflicts between insert and delete operations as well but we ignore these for the moment to simplify the discussion. An undesirable portion of interleaving of operations is "$r_1[d9]$ $r_2[d9]$ $w_2[d9]$ $w_1[d9]$", which allows the update written by the T2 to be overwritten by T1. The difference between the expected combined effect and the undesirable one is shown in figure 4. Hence, it is essential to co-ordinate the actions taken to put the proposed changes in the two alternatives into the design.

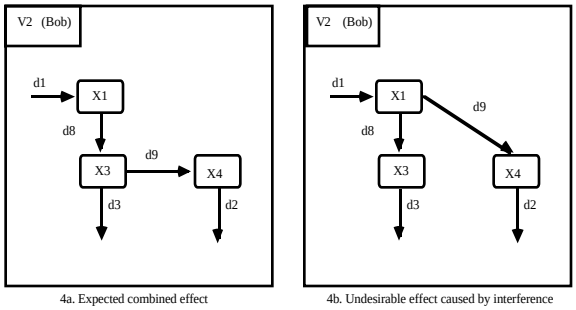

Figure 4. Comparison between an expected and undesirable combined effect.

A possible way to avoid the interference is to delay execution of the conflicting actions of T1 until T2 has committed. Hence, a possible execution of the transaction which leave the database in a consistent state is (with irrelevant sequence of operations represented as "*"):

```
T1: *                    r1[d9] w1[d9] * c
T2: * r2[d9] w2[d9] * c
```

As a result, the conflicting operations in T1 should not be performed until all operations in T2 have been carried out and committed. To avoid the delay, we may opt for accessing the information immediately once the conflicting actions have finished. The corresponding sequence of execution is:

```
T1: *                r1[d9] w1[d9] * c
T2: * r2[d9] w2[d9] * c
```

This scheme does enhance the concurrency of the execution but it is done at the expense of possible cascading abort. The reason is that T1 depends on the tentative update made by T2 since changes made by T2 has not been committed yet. If T2 is in fact aborted, new operations will have to be performed in T1 to compensate for the changes or T1 will have to be aborted as well. Hence, the first option is preferable as it reduces the dependency between different activities.

3 Concurrency Control

Managing complex relations between objects can be tedious and error-prone. As illustrated above, two possible measures can be taken. Firstly, to detect incorrect actions, the consistency constraints can specified between different objects. If a user performs operations on some objects and violates some consistency constraints as a result, the environment will be able to advise user to carry out further automated or manual operations to update dependent objects. Operations may require co-operation between users in different environments. Secondly, the sequence in which objects are accessed can be controlled so that effect of one's non-atomic action is not visible to others and undesirable dependency can be avoided.

A way to coordinate concurrent actions of users is that operations are allowed to be interleaved as long as it is logically equivalent to a serial execution. The requirement can be analysed in terms of the well established concept of serialisability [5]. Here, we focus on the mechanisms on which such an environment can be based and techniques to enhance concurrency by exploiting knowledge of the software process.

A software process is composed of a collection of process steps. Between each step, checks are performed to decide the next appropriate actions. We simply treat each process step or check as an individual subtransaction [6]. By definition the consistency of the distributed design artefacts is maintained if and only if there is a serialisable global schedule S such that S is computationally equivalent to the result of the execution of transactions in different sites. Hence, we have to synchronise the operations between global transactions executed across different environments and between local and global transactions executed in the same site. Centralising the scheduling of all transactions is a simple solution. Unfortunately, it leads to a single point of failure and possibly incurs expensive communication between the centralised scheduler and distributed development environments, even if most transactions are executed locally. Furthermore, because the actual operations performed in a transaction usually depend on the users ad hoc decisions, it is difficult to distinguish between local and global transactions in advance. As a result, we would have to treat any transaction with uncertain status as a global transaction thus creating a possible bottleneck at the centralised scheduler. Hence, we prefer techniques which achieve global serialisability through the collaboration of decentralised schedulers.

The requirement above could be achieved by enforcing a *rigorous schedule* together with transaction management which guarantees that if a transaction has any operations conflicting with another transaction, the execution of the operations of one transaction must be delayed until the commit or abort of another transaction has been received [7].

Two phase locking can be used to enforce the rigorous schedule. We choose a locking protocol because: firstly the technique is well investigated; secondly, we seek to minimise aborting transactions, the other well known technique, timestamp [5] tends to rely on aborting to enforce the serialisability; thirdly, the phantom problem may appear if objects are created or destroyed. Granularity locking [8] provides an efficient solution to the phantom problem however, it is unclear what the performance of the its counterpart in timestamp based approaches, e.g. [9], might be. Finally, the locking protocol avoids the problem of cascading abort discussed above.

To ensure commit-deferred, the global transaction manager must ensure that the commit operations are sent to different environments only after all operations have been performed. It is enforced by requiring all local transaction mangers to acknowledge the commitment of transactions to the originator of the global transaction. Assuming that there is no communication

error and abortion of subtransaction, we may regard a transaction as typically composed of two phases. In the first phase, the transaction manager creates subtransactions which are to be executed in one or more environments. From each subtransaction, the transaction manager expects an acknowledgement of the result of execution. No commit should be sent to any subtransactions in different sites until all acknowledgements have been received. In the second phase, if all acknowledgements have been received a commit will be sent to all participating sites. Hence, locks which are held by any subtransaction should not be released to others subtransactions until receiving the acknowledgement from the root of the nested transaction. Finally, since there is no need to co-ordinate the execution order between global transactions, we can safely assimilate the role of global transaction manager into the local transaction manager.

4 Ways to Enhance Concurrency

The software process has been described in terms of a sequence of actions and checks. Each sequence is represented as a subtransaction. Apparently, each subtransaction need to acquire the appropriate lock in order to access the order. However, there are two useful features of the relationships between subtransactions that we may exploit to enhance concurrency. Firstly, users' decision on any operations do not depend on objects read by the checks. Secondly, the amount of repair operations depends on the result of the checks. It is highly unlikely that all repairs, which are designed to cater for different problems, have to be performed in any particular instance. Hence, it is conservative to acquire all the locks before the execution. Instead of requiring a transaction to acquire locks all at once, we treat each sequence of operations in a check or a repair as a separate unit in acquiring locks. This permits a higher degree of concurrency if only a few repair actions actually take place.

Two-phase locking protocols with deferred commitment as suggested above may cause deadlocks. There are three kinds of techniques to handle the deadlock namely prevention, detection and avoidance [10]. Deadlock prevention techniques, which require us to lock all objects before execution, undermines concurrent activities performed by different users. Deadlock detection techniques could incur expensive costs because resolving a deadlock is not trivial except by aborting transactions on which users may have invested substantial manual effort. The remaining option is that we monitor if there is any situation which may lead to deadlock. If so, actions are taken to avoid the deadlock. Nevertheless, merely based on the knowledge of locks being held and users who are waiting for locks, traditional deadlock avoidance, for instance [11], may abort operations unnecessarily. To reduce the number of unnecessary abortions, effort could be spent on the deadlock avoidance mechanism to improve its predictive power. The process model is one of the valuable source of information. One promising approach that we are exploring is to enhance our understanding of the likelihood of deadlock by simulating the future execution of the process model while it is being enacted.

Serialisable execution of the transactions cannot avoid the situation when contradicting proposals may be asserted by different perspectives. Transaction models manage consistency by isolating any inconsistent state from other until the transaction is committed or aborted. Hence, either different perspectives have to be treated as different objects or at most one perspective can be stored in a specification at a time. We propose to relax such limitation by treating possible conflicting changes on the same collection of objects as different versions. Objects which are engaged could be made available to other by allowing independent changes to be made on previous consistent versions of the object. With this new facility, any user may also protect tentative changes from being visible to others until he or she thinks that the updated version is stable enough. In the above example, Bob may create two versions of diagram for figure 2b and figure 3b based on the baseline as shown in figure 1a. Changes can then be made concurrently without worrying about the impact of one on the other. As a result, the delay experienced from Ann's perspective could also be shortened. However, the price to pay is the requirement to merge different versions together. Such a process may be assisted by a conflict resolution tool [12].

5 Summary and Further Work

We have outlined our proposal for consistency management through imposing consistency checks and detecting the interference between users' operations in terms of a transaction model. We have also sketched our approach to handling possible deadlock and tolerating inconsistency caused by different perspectives by allowing multiple concurrent versions of a perspective.

We have to evaluate the proposed deadlock avoidance scheme against other techniques analytically and experimentally. It is also important to investigate the impact of the process model chosen on the performance of the techniques.

Versioning brings in its train the problems of configuration management. How can a collection of version be selected so as to compose a consistency design? Which version should be selected as a baseline for a new version when several versions have been created already? We speculate that the problem could be partly addressed by analysing the work record, which stores the history of transaction, in order to provide hints on organising different versions of design objects.

Acknowledgements

We gratefully acknowledge the support of The Croucher Foundation and the EPSRC Voila project. Thanks also to our colleagues Wolfgang Emmerich, Jeff Kramer and Bashar Nuseibeh for many valuable suggestions.

References

[1] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, pp. 31-58, 1992.

[2] B. Nuseibeh, J. Kramer, and A. Finkelstein, "Expressing the Relationships Between Multiple Views in Requirement Specification," presented at 15th International Conference on Software Engineering, Baltimore, USA, 1993.

[3] U. Leonhardt, A. Finkelstein, J. Kramer, and B. Nuseibeh, "Decentralised Process Modelling in a Multi-Perspective Development Environment," presented at 17th International Conference of Software Engineering, Seattle, Washington, USA, 24-28th April 1995, 1994.

[4] C. J. Date, *an Introduction to Database Systems*, 6th ed. Wokingham, England: Addison-Wesley, 1995.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*: Addison-Wesley, 1987.

[6] J. E. B. Moss, Nested transactions : an approach to reliable distributed computing. Cambridge, Mass.: MIT Press, 1985.

[7] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz, "On Rigorous Transaction Scheduling," *IEEE Transactions on Software Engineering*, vol. 17, pp. 954-960, 1991.

[8] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, “Granularity of Locks and Degrees of Consistency in a Shared Database,” in *Modeling in Data Base Management Systems*. Amsterdam: Elsevier North-Holland, 1976.
[9] M. J. Carey, “Granularity Hierarchies in Concurrency Control,” presented at 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Colony Square Hotel, Atlanta, Georgia, 1983.
[10] M. Singhal, “Deadlock Detection in Distributed Systems,” in *IEEE Computer*, 1989, pp. 37-48.
[11] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II, “System Level Concrrency Control for Distributed Systems,” *ACM Transactions on Database System*, vol. 3, pp. 178-198, 1978.
[12] S. M. Easterbrook, “Elicitation of Requiremnts form Multiple Perspectives,” : Imperial College, London, 1991.