Information Technology and Management 5, 23–64, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

CAGISTrans: Providing Adaptable Transactional Support for Cooperative Work – An Extended Treatment

HERI RAMAMPIARO * and MADS NYGÅRD heri@computer.org
Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU)

Abstract. The theme of this paper is on transactional support for cooperative work environments, focusing on data sharing – i.e., providing suitable mechanisms to manage concurrent access to shared resources. The subject is not new per se. In fact, in terms of transaction models and frameworks, several solutions already exist. Still, there are some problems that are not solved. Among these are the problems that result from the dynamic and heterogeneous nature of cooperative work. Our solution is to provide transactional support that not only can be tailored to suit different situations, but can also be modified following changes in the actual environment while the work is being performed – i.e., adaptable transactional support. As part of this, we have identified and extracted the beneficial features from existing models and attempted to extend these to form a transactional framework, called CAGISTrans. This is a framework for the specification of transaction models suiting specific applications. To handle dynamic environments we propose a new way of organizing the elements of a transaction model to allow runtime refinement. In addition, we have developed a transaction management system, built on the middleware principle, to allow interoperability and database independence. Thus this addresses the problems induced by the heterogeneous nature of cooperative environments.

Keywords: database transactions and CSCW

1. Introduction

The proliferation of computers and advanced network technology, such as the World Wide Web and the Internet, has undoubtedly changed the way people carry out their work. Work is increasingly performed in teams distributed over networks, where groups of people get together to have the work done without strict organization. As a result, the work environments are dynamic, in continuous change and heterogeneous, thus creating the important challenge of how to provide efficient support for the induced cooperative activities.

The required support spans activities from informal interaction, which is mainly based on direct communication among involved parts, meeting and conferencing, to the sharing and exchanging of data, which is mainly based on concurrent access to a shared resource base such as databases, web-servers [32]. The main focus of this work

* Address for correspondence: IDI, S. Saelands vei 7-9, NO-7491 Trondheim, Norway.

is on the latter, more specifically on the support of concurrent access to shared resources that are common in product design like Software Engineering, CAD/CAM and product manufacturing.

In this respect, the main concern is to ensure the consistency of the data shared among cooperating individuals. This has been a subject of intensive research within the database community through the use of transactions over the past two decades. Unfortunately, traditional transactions founded on ACID (atomicity, consistency, isolation and durability) properties [14] are too rigid for advanced applications. In particular, the support for long-running activities and sharing is strongly restricted due to the atomicity and isolation requirements [9].

In order to overcome this limitation, numerous transaction models have been suggested [9,22,28]. Still, it is a widely accepted fact that the goal has not yet been attained. Most of the existing models were suggested and developed for specific applications, with fixed, tailor-made semantics and correctness criteria. Thus, they may be unsatisfactory with respect to cooperative work support.

An accepted solution is to provide the possibility to customize transaction models for specific applications. In fact, the lack of consensus on which transaction models are appropriate for what situations advocates the development of a unified transactional framework allowing such a customization. This idea is not new. Examples of existing frameworks are ASSET [3], TSME [11], and RTF [1], among others. However, despite their ability to specify and tailor transaction models for specific applications, there are problems that are still unsolved. First, adequate support for dynamic environments is lacking. One of the main reasons is that the frameworks just referred to do not provide sufficient support for runtime changes. This is because the specification of a transaction model must be accomplished before the involved transactions can be executed.

Second, cooperative work is diverse [32], making openness crucial. However, since the existing frameworks are mainly built on database management systems (DBMSs), only few can adequately support other types of resource bases such as web-servers and file systems.

The work on the development of a transactional framework reported in this paper is motivated by the need to overcome these limitations. In an earlier paper [29], we provided an overview of what kind of support our CAGISTrans framework should provide. The present paper elaborates on how this is achieved in practice. The remainder of this paper is organized as follows. To put our research in perspective, section 2 describes relevant previous work. Section 3 proposes a new way to organize the transaction model elements which aim at meeting the requirements of dynamic cooperative work environments. To derive these elements we use adaptations of the ACID requirements as our baseline. One of our main findings is a way to separate the model specification into design time and runtime specifications. Section 4 shows, in particular, how transaction specification and execution can be managed at runtime. The emphasis includes the use and management of user defined correctness criteria that are suitable for concurrent and dynamic environments. We also address the second main problem – heterogeneity. Section 5 presents the transaction management system supporting the specification of transaction models and the execution of the involved transactions. The section also discusses the elements of the CAGISTrans framework that have been implemented. One of the new aspects of our CAGISTrans framework that most existing transactional frameworks have de-emphasized is integrated support for workspace management. Section 6 describes the way we provide such a support in our framework. Section 7 provides explicit comparison of our CAGISTrans framework with those described in section 2. Finally, in section 8 we discuss our approach and results so far.

2. Previous work

In the last couple of decades, several transaction models have been developed for non-traditional database applications. Some of these are presented and discussed in review works and papers [9,22,28]. As we initially pointed out, however, many of the existing models have been suggested with a specific application in mind. Thus, they may be unsatisfactory with respect to supporting wide ranges of application scenarios. Nevertheless, many of these models provide useful and important foundations for unified transactional support for cooperative work. Most notable are Moss' *Nested transaction model* [23], providing a basis for modular modelling of transactions, *Sagas* [10], providing a basis for transaction compensations, which were further exploited in *Open nested transactions* [37], a generalization of Moss' nested transaction model, *Split and join transaction models* [15], providing the idea of dynamic restructuring of transactions, and *Cooperative transaction hierarchy* [24] that presents the idea of cooperative transactions using user-defined correctness criteria.

Due to the fact that existing models have restricted application areas, the trend is towards the development of frameworks for transaction models that are tailorable to different situations. One of the earliest frameworks in this category was ACTA [5]; a framework providing a foundation for synthesizing and reasoning about transactions. However, although ACTA is a very useful formal tool to specify and verify new transaction models, it is merely a formal and theoretical framework, that does not provide any means of making transactions operational during runtime.

Motivated by this, ASSET – A System for Supporting Extended Transactions [3] – was suggested to implement the ideas of ACTA with O++ language primitives. The idea is to allow the coding of extended transaction models (ETMs) using the O++ programming language, and to make these models operative on top of a DBMS. The ASSET primitives are `begin`, `abort`, `commit`, `delegate`, and `permits`, allowing a transaction models designer to create transactions, delegate resources among transactions, and permit dirty reads etc. among them. In [3], the authors have shown the use of these primitives to specify ETMs, demonstrating the usefulness of the framework. However, requiring a transaction models designer to step down and program transactions may be risky with respect to bugs. Moreover, since a transaction model has to be coded and compiled before the involved transactions are executed, it is necessary to have complete a priori knowledge about the tasks to be carried out as well as the induced sharing patterns. This makes it rather difficult to provide adequate support for dynamically evolving collaborative activities.

Another framework that is similar to ASSET, that allows specification and implementation of ETMs, is TSME – the Transaction Specification and Management Environment [11]. Unlike ASSET, TSME was developed as a complete transaction management system with a programmable transaction manager that enforces the specified transaction models during runtime. The main building blocks of the transaction specification in TSME are dependencies. These are classified into *state dependencies*, specifying dependencies related to transaction states; begin, abort and commit, and *correctness dependencies*, specifying dependencies related to correctness criteria; determining which concurrent executions of complex transactions preserve consistency and produce correct results. TSME is a promising framework that has further demonstrated the advantages of allowing a model designer to specify several application specific transaction models within a single environment, and supporting them during runtime. One of the main strengths of TSME is its extensive support for execution control, allowing sophisticated coordination of transaction execution. However, TSME does not provide support for dynamic restructuring, making it less appropriate for dynamically evolving environments. Moreover, the transaction manager component in TSME is apparently built from scratch, and integration with existing DBMSs seems to be difficult.

RTF – the Reflective Transaction Framework [1] – is another framework similar to those described above which aims at specifying and implementing application specific ETMs. Unlike ASSET and TSME, the main focus of RTF was to develop modules that implement existing ETMs on top of commercial TP-monitors. The basis components are *transaction adapters* – i.e., add-on software modules providing extensible transactional services for advanced applications. Using these adapters, RTF extends the facilities of a TP-monitor, allowing it to execute transactions beyond the ACID models. Initially, RTF was implemented on top of Encina,[1] demonstrating the applicability of the framework. It is worth noting, however, that RTF does not provide support for user-defined correctness criteria. This was beyond the scope of RTF, and was left for future studies [1]. Rather, the focus was on provision of extensible lock protocols handling concurrency. Moreover, RTF is mainly a database-centred framework, and does not explicitly address the support for other resource bases that are not supported by the actual underlying TP-monitors.

The CAGISTrans framework is aimed at providing the possibility to specify and implement transaction models fitting different situations. In this sense, it shares the basic objectives with the frameworks described above. However, we attempt to go further by developing a framework that extracts the beneficial features from existing models and frameworks, and try to extend these to cope with the remaining problems that are not fully addressed in the previously developed models and frameworks – i.e., means of supporting fully dynamic and fully heterogeneous environments. An explicit comparison of our CAGISTrans framework with existing frameworks is provided in section 7.

[1] See http://www.transarc.com/Product/Txseries/Encina/Brochure2.0/encma.html.

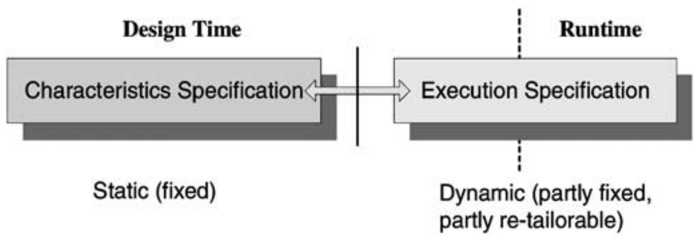
Design Time
Runtime
Characteristics Specification
Execution Specification
Static (fixed)
Dynamic (partly fixed, partly re-tailorable)

Figure 1. Illustration of the distinction between characteristics and execution specifications.

3. Distinction between characteristics and execution specifications

Dynamic environments are generally characterized by unpredictability and interactiveness. Both properties imply that it is not always possible or practical to have a complete transaction schedule in advance. To cope with this, we have argued the necessity of allowing both design time and runtime specifications of transaction models [29]. This implies the necessity of organizing the elements of transaction models in such a way that parts of the specification can be done before the actual transactions are executed, while the remaining parts can be accomplished during runtime. To address this need, the CAGISTrans framework provides transaction modelling building blocks that are organized in two separate, but connected parts [29]: (a) the *characteristics specification*, consisting of blocks that must be defined before runtime, and (b) an *execution specification*, consisting of blocks that are modifiable while transactions are being executed (see figure 1).

Blocks belonging to the characteristics specification are fixed and possible to define in advance. Those falling into this category are (1) the ACID requirements to be applied – i.e., customized non-ACID properties, (2) the relationship among transactions – i.e., structures and dependencies, (3) the designated correctness criteria – i.e., user-defined or database dependent criteria, and (4) the policies to be used based on the applied criteria – i.e., the mechanisms to be used and the rules for how and when to use them.

Blocks belonging to the execution specification are those that are only partly predictable and possible to modify during runtime. Blocks in this category are those being used to control and manage the behaviour of transactions (see section 4).

3.1. Customizing the ACID properties

We have chosen to use the ACID properties as the basis for the transaction characteristics specification. The rest of the elements are derived as a result of the specified properties.

The ACID properties have traditionally been recognized as being the ultimate concept for achieving correctness. Unfortunately, their undue strictness has made them inappropriate for use in advanced, possibly cooperative applications. For this reason, many, if not all of the existing advanced models or frameworks have advocated the need to compromise on the ACID requirements. On the other hand, given the diversity of application requirements, we stress the ability to seamlessly tailor the ACID requirements so as to meet different needs. As some situations within a single advanced application still demand parts of the ACID requirements to be preserved, most situations may see them as just a burden. Thus, the proposed solution is to allow users to customize the ACID properties according to the needs of their applications, rather than providing them with fixed properties. Basically, the idea is to allow users to first identify the main requirements, and thereafter to designate the appropriate properties. But what requirements are relevant to customize? As we pointed out in [29], only atomicity and isolation requirements should be adaptable. Both consistency and durability should be preserved by definition. This is despite the fact that some transactions may have to tolerate some temporary inconsistencies, and that parts of a transaction execution may not be required to be permanent.

3.1.1. Preserving consistency and durability

Fundamental requirements underlying transaction processing are that (1) executing transactions always transform a database from one consistent state to another consistent state, and (2) once the execution is terminated the final result is permanent [13]. These requirements are necessary both to ensure the correctness of the data shared among users and make sure that all committed results survive possible crashes. These are the main motivation for the need to preserve both the consistency and durability properties.

Full consistency implies that a set of user defined or database dependent criteria must be provided. In addition, policies determining relevant mechanisms and specifying the rules for their applicability are necessary [26,29].

Further, full durability implies the use of logging facilities that maintain all necessary information about executing transactions on a persistent store. In addition, there is a need for mechanisms that allow transactions to explicitly discard the effect of committed results such as using transaction compensation [17].

3.1.2. Switching between full and relaxed atomicity

Existing transaction models for advanced applications have emphasized the necessity of relaxing the atomicity property to meet the requirements of long, interactive transactions. The suggested solution is to provide abort management with finer granularity than that provided by full atomic transactions – i.e., allowing partial aborts. However, there are still situations where full atomicity is necessary to achieve acceptable processing. Ideally, users should be provided with the ability to choose the way to manage transaction aborts. This is why we have stressed the necessity for seamlessly switching between full and relaxed atomicity. The strength of such an approach is the ability to manage transaction termination exactly according to the application needs, thus increasing the flexibility.

We argue the necessity of specifying the transaction structure, distinguishing between *flat* and *nested structures*. In this way, transaction abortion can be managed in accordance with the actual structure. We assume that transactions that are short-lived and do not involve interaction, are flat. They are seen as atomic meaning they are according to the all-or-nothing law.

By contrast, assuming that long-lived transactions are nested, consisting of several constituent transactions, aborts of transactions can be managed in a more controlled manner – cf. Moss' nested transactions [23].

We propose specifying transaction atomicity by means of transaction dependencies. This is based on the dependency theory from, e.g. [5,11]. Note that existing solutions involving dependencies assume the existence of parent-to-children abort dependency schemes, specifying by default that if a parent transaction aborts then all its children automatically abort as well. The abortion of a child, on the other hand, does not have any direct effect on the parent. The source of dependencies in such a case is thus the transaction structure.

The main disadvantage of such an approach is that abortion of a parent always discards the effects of its children, making it impossible to define alternative, more relaxed, abortion schemes. For example, consider a software project consisting of several activities. Although the project is cancelled, it might be desirable to "save" the results of some already completed activities that could be useful in future projects. Moreover, there are situations where it is necessary to define the effects of a child's abortion on its parent as well as on possible siblings. This stresses the necessity for defining a more generalized dependency scheme than that based on structure. We call this an *atomicity abort dependency scheme*.

The idea is to allow an explicit designation of a desired abort specification depending on the application needs. In this sense, we may define prevailing abort dependencies regardless of any existing access dependencies or their structural dependencies.

Consider a transaction T with constituent transactions $t_0, t_1, t_2, \dots, t_n$ – i.e., $T = \{t_0, t_1, t_2, \dots, t_n\}$. To allow the transaction scheduler to deduce which transactions have to abort in case transaction t_i fails, it manages sets, containing lists of affected transactions, called $\mathit{AbortSet}(t_i)$:

$\mathit{AbortSet}(t_i)$ is the set of transactions that have to abort if t_i aborts:

$$\mathit{AbortSet}(t_i) = \{t_j \in T \mid \mathit{abort}(t_i) \rightarrow \mathit{abort}(t_j),\ i \neq j\},$$

where $T = \{t_0, t_1, t_2, t_3, \dots, t_n\}$.

This means that $\mathit{AbortSet}(t_i)$ is a set of transactions t_j such that if t_i aborts, t_j must also abort.

In the CAGISTrans framework, there is a data structure maintaining a set of $\mathit{AbortSet}(t_i)$, that the transaction manager uses to determine which other transactions must abort in case a specific transaction aborts. This means that when transaction t_i aborts, the scheduler will execute `DoAbort(t_i)` as follows:

```
(1) mark $t_i$ "aborted";
(2) if $AbortSet(t_i) \neq \emptyset$ then for each $t_j \in AbortSet(t_i)$
    if $t_j$ is not already aborted then DoAbort(t_j).
```

Figure 2 illustrates this by a nested transaction T_1 with sub-transactions $T_{1.1}$, $T_{1.2}$ and $T_{1.2.1}$. From this, we get the constituent transactions $T = \{T_1, T_{1.1}, T_{1.2}\}$ and $T' = \{T_{1.2}, T_{1.2.1}\}$. The arrows denote the specified abort dependencies among the involved

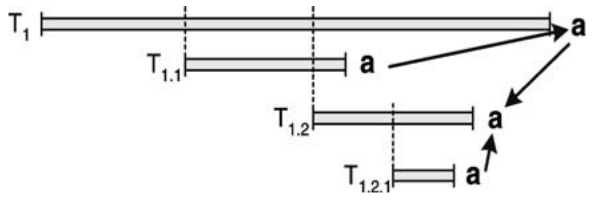

Figure 2. Illustration of abort dependency.

transactions – i.e., $T_{1.1}$ depends on T_1, and T_1 and $T_{1.2.1}$ depend on $T_{1.2}$. Hence, following our definition, $AbortSet(T_1) = \{T_{1.1}\}$, $AbortSet(T_{1.1}) = \{\ \}$, $AbortSet(T_{1.2}) = \{T_1, T_{1.2.1}\}$ and $AbortSet(T_{1.2.1}) = \{\ \}$.

As this indicates, if T_1 aborts then $T_{1.1}$ must also abort. The abortion of $T_{1.2}$ will, cause T_1 and $T_{1.2.1}$ to abort. In other words, the abortion of $T_{1.2}$ will affect its parent transaction T_1 as well as its child $T_{1.2.1}$. But the abortion of $T_{1.1}$ or $T_{1.2.1}$ will not affect other (sub) transactions.

How does all this affect the execution of transactions? We have assumed that all transactions with relaxed atomicity are nested. The effect of the specifications of the atomicity properties are relevant in the way transactions realize their specified structures during runtime. This means that some transactions may have to restructure – i.e., spawn new transactions and delegate some responsibilities, making it necessary to provide operations for dynamic restructuring (see section 4 for elaboration).

3.1.3. Switching between full and relaxed isolation

Traditional multi-user database systems often assume that transactions do not need to cooperate on data, only compete, giving them the impression of being solely in charge of some specific resources. Clearly, such an assumption contradicts the cooperative work philosophy. In fact, the sharing of tentative data is needed to make cooperation possible. Therefore, transactions must be able to reveal their intermediate results. However, this may again cause undue cascading abortion, which was originally one of the main reasons for the isolation requirement. Hence, it is important to have an acceptable abortion scheme to ensure that only those that are directly affected by a failure have to abort. The other cooperating transactions should be able to proceed as normal. To achieve this, we again use the *AbortSet* and its corresponding algorithm, above, but now we are also interested in dependencies among transactions that are not "family" related. Thus, instead of only traversing the *AbortSet* containing children of a specific transactions, the abortion algorithm will also go through an "abort set" containing cooperating transactions that should be aborted with a specific transaction.

Formally, we now define a constituent transaction set T_i as $T_i = \{t_{i.0}, t_{i.1}, t_{i.2}, \ldots, t_{i.n}\}$, $i = 1, \ldots, m$. Thus, the new *AbortSet* is defined as follows:

$$AbortSet(t_{i.j}) = \left\{ t \in \bigcup_{i=1}^{m} T_i \;\middle|\; abort(t_{i.j}) \rightarrow abort(t) \right\},$$

where $T_i = \{t_{i.0}, t_{i.1}, t_{i.2}, \ldots, t_{i.n}\}$, $i = 1, \ldots, m$.

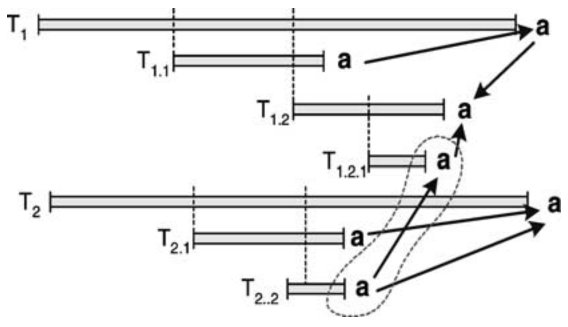

Figure 3. Illustration of abort dependency across two nested transactions.

Table 1
Isolation with corresponding correctness criteria and applied policies.

	Full isolation	Relaxed isolation
Correctness criteria	Serializability	User defined, see section 4.3
Applied policies		
Mechanisms	Strict locking and awareness	Flexible locking, workspaces and awareness
Rules	2PL protocol	User-controlled lock protocol and collaborative lock protocol

This means that the abort set of a transaction $t_{i.j} \in T_i$ may contain transactions that are children of other transactions.

To illustrate this, consider the two nested transactions T_1 and T_2 depicted in figure 3. Using our definition, $AbortSet(T_1) = \{T_{1.1}\}$, $AbortSet(T_{1.1}) = \{\ \}$, $AbortSet(T_{1.2}) = \{T_1, T_{1.2.1}\}$, $AbortSet(T_{1.2.1}) = \{T_{2.2}\}$, $AbortSet(T_2) = \{T_{2.1}, T_{2.2}\}$ and $AbortSet(T_{2.1}) = AbortSet(T_{2.2}) = \{\ \}$. This implies that an abortion of $T_{1.2.1}$ will cause $T_{2.2}$ to abort, while $T_{2.1}$ may proceed as normal.

In addition to this relaxed abortion scheme, it is necessary to have relaxed correctness criteria allowing the above-mentioned sharing of intermediate results. Of course, such criteria will be user-defined since not all applications have the same needs. How such a criterion is provided in the CAGISTrans framework is discussed in section 4.3.

Further, to achieve a specified criterion, we have to define which policy should be applied. Such policies are necessary to determine the mechanisms that are relevant and the rules for when and how to use the mechanisms. Table 1 provides an overview of the correctness criteria and policies associated with the isolation property.

For full isolation, we apply the serializability correctness criterion and use the standard locking policies described in the literature [2]. For relaxed isolation, on the other hand, we apply user-defined correctness criteria, which are elaborated on in section 4. The relevant mechanisms are flexible locking – *user-controlled locks* and *collaborative locks* (see section 3.1.3.1), workspace usage (see section 6), and awareness mechanisms (see section 3.1.3.2).

3.1.3.1. Flexible locks

The *user-controlled lock* protocol is defined and implemented as follows:

(1) Locks are acquired to lock specific objects – i.e., each object has an attribute telling which transaction is holding a lock on it.

(2) No lock mode – e.g., read-lock vs. write-lock – is required, as the lock is associated directly with an object rather than with an operation.

(3) Once an object is locked, no other transactions may acquire a lock on that object until the existing lock is released.

The main characteristic of this locking protocol is that locking is basically done interactively, without following any strict automatic locking rules. Each lock is accompanied by a notification tag providing information about the owner of the lock and the object being locked. In this way, when a lock request is issued the system will be able to provide the necessary feedback about the current lock held.

In cooperative activities, two or more transactions may have to cooperate on the same object. The locking protocol above does not permit such cooperation. To allow this, we apply *the collaborative lock* protocol. This is defined and implemented as follows:

(1) As with user-controlled locks, these locks are acquired to lock specific objects.

(2) And again, no lock modes are required.

(3) But, although an object is locked, other transactions may also acquire a lock on that object, regardless of the operation semantics.

(4) When a transaction requests a lock on an object already locked by another transaction, it will be informed about the locking situation, and asked to specify the intension: *browse*, *incorporate*, or *modify.*

    (a) If the intension is to *browse*, the lock is granted right away.

    (b) If the intension is to *incorporate*, the lock is granted, but the sequence of future actions must be specified using *demands* (see section 4.3).

    (c) If the intension is to *modify*, the lock owner and the requester will first be warned. Thereafter, they are required to solve the conflicts through negotiation.

As can be inferred, this locking protocol allows collaborating transactions to hold conflicting locks. These are similar to those found in the Solaris[2] file system. However, the main difference is that before a lock request is granted, the intension must be specified, and concurrent updates are allowed only after explicit agreement among the collaborative partners.

Note, user-controlled locks may be degraded to collaborative locks upon request. In other words, if an object is locked using a user-controlled lock, this lock can be degraded to a collaborative lock when another transaction requests to access the object.

[2] Solaris is a trade-mark of Sun Microsystem. See http://www.sun.com.

However, before such access can be accomplished, *a permit* relationship must be established (again see section 4.3).

3.1.3.2. Awareness mechanisms

Awareness in CAGISTrans is realized with event notification mechanisms. They are aimed at providing users engaged in an activity with knowledge about events that are of possible interest to them.

There are several types of events that are relevant. For those that are related to concurrent execution of transactions, significant events may be associated with transaction start, transaction termination, acquisition of locks and object change. These are realized as follows:

(1) *Notify on begin*. When a transaction starts executing, a notification message is broadcast to all involved parties.

(2) *Notify on terminate*. When a transaction commits or aborts, a notification message is sent to all involved parties.

(3) *Notify on lock*. When a lock is acquired or released by a transaction, a notification message is broadcast to all involved parties, specifying which type of lock was acquired or released and on which data the lock is/was applied.

(4) *Notify on change*. When a data object is being altered by a specific transaction operation, a notification message is sent to all involved parties.

With relaxed isolation, all four notification events are useful due to potential cooperation. With full isolation, on the other hand, the knowledge about the existence of other transactions is normally less crucial. Therefore, none of the notification events are primarily required. Still, events (1) and (2) may be useful. For example, to ensure correct results and avoid unnecessary work, execution of project plan updates must be atomic and isolated. Hence, all affected engineers would appreciate notification about the beginning and the termination of this update, so that they can accommodate their activities accordingly.

Summarizing this, notification types (1), (2), (3) and (4) are relevant when the isolation property is relaxed, whereas types (1) and (2) might be useful when the isolation property is full (see table 2).

Table 2
Relevance for notification types.

	Full isolation	Relaxed isolation
Notify on begin	Useful	Relevant
Notify on terminate	Useful	Relevant
Notify on lock	N/A	Relevant
Notify on change	N/A	Relevant

Table 3
Possible combinations of atomicity and isolation properties.

	Full atomicity	Relaxed atomicity
Full isolation	OK	OK
Relaxed isolation	Not always convenient	OK

3.2. Analysing the combination of isolation and atomicity properties

Table 3 indicates possible combinations of atomicity and isolation properties of transactions. Combining full atomicity and full isolation is readily OK as for the ACID models – cf. the literature [14].

We may also combine full isolation with relaxed atomicity, allowing transactions to provide partial rollback or controlled abort management, but at the same time prohibiting cooperation. An example of a transaction model having this combination is the nested transaction model [23].

However, combining full atomicity and relaxed isolation may not always be convenient due to the cost of rollback. An example is open nested transactions [37] allowing sub-transactions to cooperate. If a transaction aborts then all associated transactions must abort too, whether they have performed conflicting operations or not. Clearly, such a requirement would not be suitable within cooperative environments involving several associated interactive activities of long duration. Anyhow, this combination could be useful for small cooperative tasks such as those that do not involve too much invested effort but must still meet the all-or-nothing objective for the results to be reliable.

As opposed to this, a combination of relaxed isolation and relaxed atomicity will always be OK, assuming the consistency and durability requirements can be met. This last combination is one of the main issues that we will elaborate on in the rest of this paper.

4. Management of transactional behaviour during runtime

Adaptability is a prerequisite for the support of dynamic cooperative environments. Adaptability means the ability to tailor the provided support to different needs, including those that may change while involved activities are in progress. In the context of transactional support, meeting such a requirement is not a trivial matter. The main challenges are generally how to apply changes, while the actual transactions are being processed.

To cope with these challenges, we have argued for the separation of design time and runtime specification of transaction models. Further, there is a need to separate the execution specification into two parts – *fixed* and *modifiable* parts to enable runtime modifications [29]. Three main building blocks are necessary to manage the transactional behaviour during runtime:

- Operations used to manage the execution of transactions.
- Advanced operations specifying the actions a transaction can execute.

- Rules defining constraints to manage and control the effect of transaction executions.

These three will be covered in sections 4.1, 4.2 and 4.3, respectively.

4.1. Management operations

The management of transaction execution is achieved through operations that manage initiation, termination and restructuring, respectively.

4.1.1. Managing initiation and termination

We can assume that every transaction starts executing with a *begin* and eventually terminates with either a *commit* – if it succeeds, or an *abort* – if it has to discard any changes. To allow the widest possible application areas, CAGISTrans is designed to enable both automatic management – i.e., transparent initiation and termination applicable to ACID transactions, and interactive management – i.e., user-controlled initiation and termination relevant for cooperative interactive environments. The former is realized by allowing the system to perform the initiation/termination without any intervention by the user. The latter, on the other hand, is realized in such a way that a user can freely choose the way transactions execute and terminate. However, to avoid anarchy there is a need to define dependencies among the involved transactions. For example, initiation of a transaction T_i may not be accomplished if a *begin dependency* [5] specification says that another T_j has to start first. Similarly, a specific transaction T_i may not be terminated if there is a *commit dependency* [5] specification that enforces a T_i to wait for another transaction T_j to finish. Also, aborting a specific transaction T_i may cause another transaction to abort depending on the prevailing abort scheme (see section 3.1).

4.1.2. Managing dynamic restructuring

The ability of a transaction to delegate responsibilities and spawn new transactions during runtime is referred to as dynamic restructuring. It has been identified as a useful cooperation primitive in addition to being a means to realize a specified structure during runtime. Dynamic restructuring was originally proposed with the Split and Join transaction model [15], using a split operation to divide an ongoing transaction into two or more transactions, and a join operation to merge two or more ongoing transactions into a single transaction.

The main reason for adopting the basic ideas of this notion is the ability to support open-ended activities. By allowing a transaction to delegate responsibilities, it will be able to release resources that are no longer needed before it terminates. In this way, resources can be made available to other transactions at an earlier stage. For example, although a transaction aborts, some of its resources may still be used by others still executing transactions.

In CAGISTrans we speak of two types of responsibilities: object locks and operations. A transaction may thus transfer locks on specific objects to another transaction, giving it the responsibility to do certain tasks, such as accomplish updates on these objects. It may also transfer operations to another transaction, giving it the responsibility to commit these operations. From this perspective, the difference between our approach and that of the Split and Join transaction model is our use of both operation and object lock delegation. Restructuring in the Split and Join transaction model is based on object lock transfers only. Moreover, while the transactions in that model stick to serializability, our framework allows user-defined correctness criteria.

Our dynamic restructuring will depend on the specified structure. Flat transactions do not spawn new transactions, but may delegate responsibilities to other transactions. This requires that a transaction maintains two sets respectively containing object locks and the operations for which it needs to delegate responsibilities. We call these `ObjSet` and `OpSet`, respectively. An operation `delegate(`$\mathtt{t_i}$`,`$\mathtt{t_j}$`,ObjSet,OpSet)` performs a resource transfer from a transaction t_i to another transaction t_j. This means that when a transaction t_i executes `delegate`, this transaction will release its locks on all objects in `ObjSet`, and t_j will immediately take over these locks. In addition, all operations in `OpSet` will be removed from t_i's history, before they are transferred to t_j's history. Consequently, when t_i terminates, the operations that have been delegated to other transactions will not be affected.

Unlike flat transactions, nested transactions may spawn new transactions. Thus dynamic restructuring may involve the generation of constituent transactions. In that case, `delegate(`$\mathtt{t_i}$`,`$\mathtt{t_j}$`,ObjSet,OpSet)` transfers responsibility from t_i to one of its sub-transactions t_j. At the time this operation is issued, t_j may be active or not. If t_j is active, then the restructuring is achieved as above. If it is not active, it will be initiated as a child of t_i, before the resource transfer is accomplished as before.

4.2. Advanced operations

Advanced operations mean commands specified at an abstraction level higher than, but based on, *read* and *write* operations. As we pointed out in [29], there are several reasons for using advanced operations in CAGISTrans. First, there are types of activities that cannot be easily modelled with read and write operations due to the induced complexity. By allowing this type of abstraction, however, we increase the degree of modularity, thus simplifying the modelling task. Further, advanced operations allow us to exploit the semantic knowledge about an operation. An example of such semantics is the return information from executed operations – e.g., the changes operations have made. Another example is the intention of an operation. Here, for instance, we can speak of two types of read intentions: *browsing* and *incorporation*. Browsing means that even if an object being read by a user had a different value, this would not cause the user to perform different actions. In other words, with browsing, changes will not affect users' future actions. On the other hand, if the intention is incorporation, changes on an object read are likely to affect the way users act.

How do we realize advanced operations with intentions? We define a general advanced operation as a tuple $Op = \langle Optype, InSet, OutSet, Intent \rangle$, where *Optype* denotes the type of operation, *InSet* is the set of input objects – i.e., objects read, *OutSet* is the set of output objects – i.e., objects written, and *Intent* indicates the intentions of each input argument.

Using this, we can specify a general conflicting rule based on operation commutativity [36]; two operations are in conflict if the order in which they are executed matters. Disregarding intentions, this means that two operations $Op_1 = \langle Optype_1, InSet_1, OutSet_1 \rangle$ and $Op_2 = \langle Optype_2, InSet_2, OutSet_2 \rangle$ commute if the objects read by Op_1 are different from the objects written by Op_2, the objects read by Op_2 are different from the objects written by Op_1, and Op_1 and Op_2 do not update any common object:

$$(InSet_1 \cap OutSet_2 = \emptyset) \wedge (InSet_2 \cap OutSet_1 = \emptyset) \wedge (OutSet_1 \cap OutSet_2 = \emptyset).$$

With intention knowledge, we can relax this commutativity rule such that even if the intersection between an *InSet* and *OutSet* pair is not empty, the two operations will still commute when the objects in the intersection are for browse with the inputting transaction.

In practice, we substitute the *Intent* in the operation tuple with a *BrowseSet*, assuming that all objects found in *InSet* but not in *BrowseSet* are by default incorporated. Based on this, the new commutativity rule can be formally defined as follows. Considering $Op_i = \langle OpType_i, InSet_i, OutSet_i, BroweSet_i \rangle$:

$$\begin{aligned} Compatible(Op_i, Op_j) \Longleftrightarrow\ & (OutSet_i \cap OutSet_j = \emptyset) \\ & \wedge (\forall Ob \in (InSet_i \cap OutSet_j), Ob \in BrowseSet_i) \\ & \wedge (\forall Ob \in (InSet_j \cap OutSet_i), Ob \in BrowseSet_j). \end{aligned}$$

This means that two operations Op_i and Op_j are compatible iff (1) Op_i and Op_j do not perform updates on any common objects, and (2) objects read by one of the operations and updated by the other operation only are for browse with the inputting transaction. Note, this assumes that all elements of $BrowseSet_i$ are elements of $InSet_i$; $\forall Ob, (Ob \in BrowseSet_i) \rightarrow (Ob \in InSet_i)$.

To illustrate this, we can consider a software development environment that provides the commands *Edit_interface*, *Edit_class*, and *Compile_class* for a coding process. Assume that all these are abstractions of read and write operations. Suppose that two modules are to be created as part of the coding process as illustrated in figure 4: a graphical user interface module (*GUI* for short) and a process module (P) processing inputs from the *GUI* and producing data to be displayed by the *GUI*. Further, *Edit_class*(*GUI*) implements the interface part of the process module, called process interface (*P_i* for short). As indicated in figure 4, this means that changes on *P_i* will affect the execution of *Edit_class*(*GUI*). But, as depicted, although *Edit_class*(*GUI*) reads *GUI_i*, changes to this object will not directly affect that operation. It is similar for *Edit_class*(P) and *Compile_class*(P). The workspace operations will be elaborated on in section 6.

Using our advanced operation type, we get the specifications given in table 4.

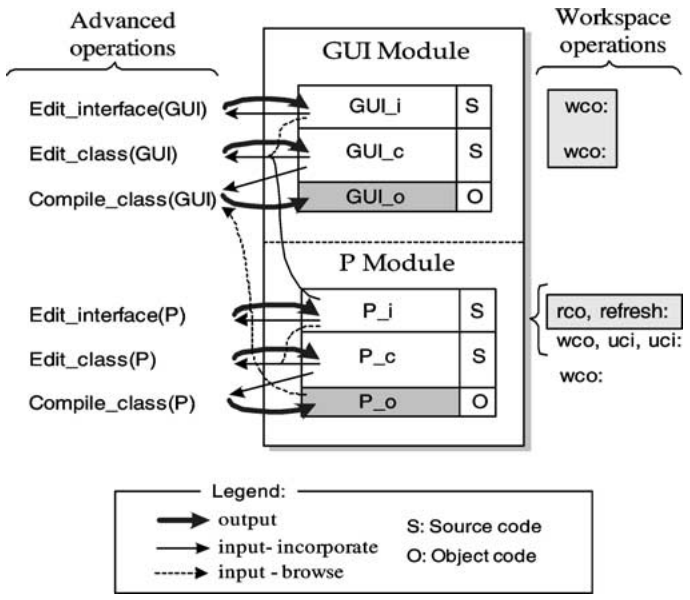

Figure 4. Advanced operations illustration.

Table 4
Illustrations of advanced operation specification.

Operations applied on *GUI*
$Edit\_interface(GUI) = \langle Edit\_interface, \{GUI\_i\}, \{GUI\_i\}, \{\ \}\rangle$
$Edit\_class(GUI) = \langle Edit\_class, \{GUI\_c, P\_i, GUI\_i\}, \{GUI\_c\}, \{GUI\_i\}\rangle$
$Compile\_class(GUI) = \langle Compile\_class, \{GUI\_c, P\_o\}, \{GUI\_o\}, \{P\_o\}\rangle$
Operations applied on *P*
$Edit\_interface(P) = \langle Edit\_interface, \{P\_i\}, \{P\_i\}, \{\ \}\rangle$
$Edit\_class(P) = \langle Edit\_class, \{P\_c, P\_i\}, \{P\_c\}, \{P\_i\}\rangle$
$Compile\_class(P) = \langle Compile\_class, \{P\_c\}, \{P\_o\}, \{\ \}\rangle$

Applying our advanced compatibility rule, we get for example:

$$\neg Compatible(Compile\_class(GUI), Edit\_class(GUI))$$
$$\neg Compatible(Compile\_class(P), Edit\_class(P))$$
$$\neg Compatible(Edit\_class(GUI), Edit\_interface(P))$$
$$Compatible(Edit\_class(GUI), Edit\_interface(GUI))$$
$$Compatible(Edit\_class(P), Edit\_interface(P))$$
$$Compatible(Compile\_class(GUI), Compile\_class(P))$$

We will use these results in the illustrative examples in the rest of this paper.

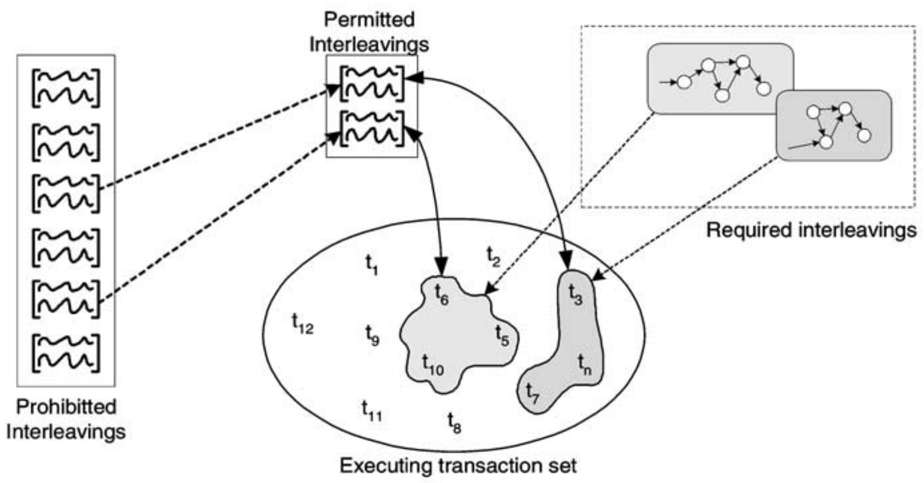

Figure 5. Illustration of the use of the three constraint tools.

4.3. Managing and controlling transactional behaviour

Transaction executions must be managed in a controlled manner to achieve acceptable data consistency. Traditionally, serializability has been the ultimate and widely accepted criterion to achieve data consistency [2]. Serializability ensures that any transaction execution is equivalent to some serial execution, and thus always produces correct results. However, a serializability criterion requires transactions to be isolated, thus prohibiting cooperation. To overcome such a restriction we have included a set of user-controlled correctness constraints to allow a controlled sharing of data, and ensure that transactions terminate with consistent final results. Such constraints may be used when the serializability criterion is considered inappropriate.

The constraints in CAGISTrans explicitly specify transaction interleavings that are (1) *prohibited*, (2) *allowed*, and (3) *mandatory*. Constraints (1) are realized using *conflicts*, which define operations that cannot execute concurrently to hinder incorrect effects. Constraints (2) are realized using *permits*, which identify operations that in general are considered conflicting by (1), but that still can execute concurrently. Finally, constraints (3) are realized with *demands*, which specify operation sequences that must appear to achieve correct executions. Different combinations of conflicts, permits, and demands thus constitute the user defined correctness criteria in CAGISTrans.

Figure 5 is a general illustration of how combinations of the three constraint types can be exploited as user defined correctness criteria. A table containing a set of prohibited interleavings specifies the *conflicts*. This consists of a list of operation pairs that cannot be executed concurrently, for the currently executing transaction set. Consider now that $T_1 = \{t_5, t_6, t_{10}\}$ and $T_2 = \{t_3, t_7, t_4\}$, are two sets of cooperating transactions. An analysis shows that the prevailing *conflict* constraints are, for T_1 and T_2, too strict. There are, in particular, two specific pairs of operations that we would like to allow to occur concurrently. To cope with this, we must specify a *permit* relationship

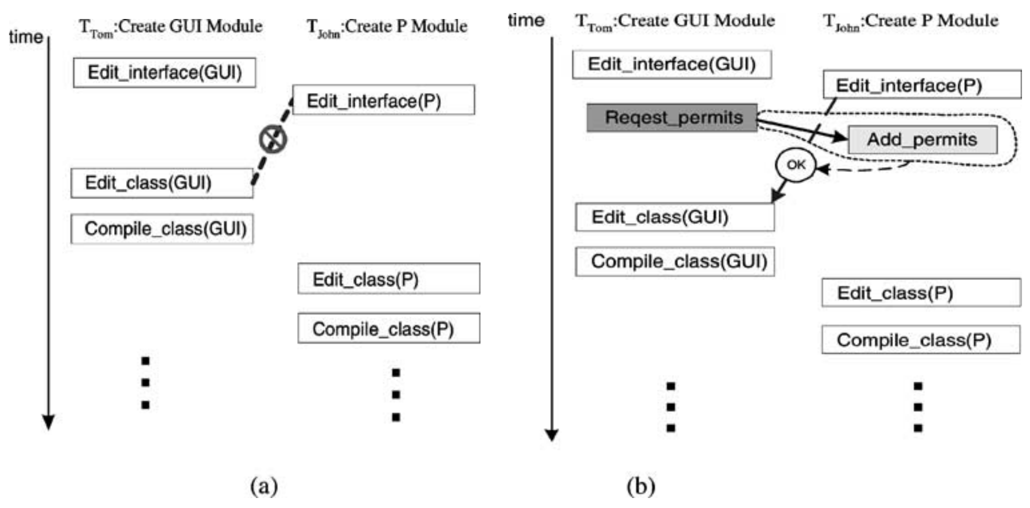

Figure 6. (a) Incorrect scenario, and (b) correct scenario using conflicts and permits.

for each transaction set. However, the two permitted interleavings may now introduce some concurrency anomalies. Therefore, we might also have to specify *demand* rules, determining sequences of operations that the transactions in T_1 and T_2 have to execute. Note that such *demanded* interleavings must still obey the conflicts defined for the two transaction sets.

4.3.1. The necessity of conflicts, permits and demands

This section discusses the necessity of diverse combinations of the three constraint types.

Case 1: Conflicts and permits only

This case assumes that we only have conflicts and permits, where the permits allow some specific transactions to violate some of the conflict rules. An analogous example is allowing a transaction t_j to write an object read by another uncommitted transaction t_i. As long as t_i's read does not affect its future computation, any changes to this object will not be critical. But if this is not the case, problems might arise. This means that there are situations where conflicts and permits alone would not be sufficient to guarantee correctness. Rules specifying what sequences of actions must be executed for the final results to be acceptable from the viewpoint of both t_i and t_j are also needed.

To illustrate this, let us say that two engineers, Tom and John, are assigned the responsibility to create the two modules *GUI* and *P* from section 4.2. Figure 6(a) illustrates a possible interaction scenario.

According to our commutativity (conflict) rule, this scenario is illegal because of the concurrent execution of *Edit_interface*(*P*) and *Edit_class*(*GUI*). As a result, Tom must wait until John finishes. Assume, however, that Tom and John find such a wait unacceptable. Hence, they need a permit relationship to enable their interaction. We may define this as *permit*($T_{\mathrm{Tom}}T_{\mathrm{John}}$, [*Edit_class*, *Edit_interface*], [*GUI*, *P*]). This allows Tom and John to execute *Edit_class* and *Edit_interface* upon *GUI* and P, even if this violates a commutativity conflict. Taking this permit relationship into account, figure 6(b) shows the scenario that may be considered correct.

Note, to achieve recoverability [2], we must ensure that T_{Tom} does not commit before T_{John}, specified as a commit dependency, and if T_{John} aborts then T_{Tom} must abort too.

Case 2: Permits and demands only
This case only combines demands and permits. Permits identify allowable interleavings while demands specify steps that must appear. Since this assumes that conflict rules do not exist, permits are used to allow accesses to data that are controlled via locking. This implies that we will not be able to fully exploit the benefit of advanced operations, and so be unable to customize conflicts. The reason is that we would not be able to reason about conflicts appearing beyond read/write.

Nevertheless, we may regard locks as a specialization of conflicts. Hence, this case is a special case of case 4.

Case 3: Conflicts and demands only
This case is equivalent to that of [24]. This combination may be applied to achieve correctness. However, since we may not directly refine a conflict definition (also true for the original conflicts from [24]), there is no way to relax any pre-specified conflict rules if the requirements change. Nevertheless, we often prefer to allow some limited number of transactions to disregard some specific conflicts rather than change these conflicts for all executing transactions collectively. Although cooperation could be useful for Tom and John, allowing a third party to observe the changes made may not necessarily be convenient, since the results may still be incomplete. In other words, removing some specific conflicts, thus opening up for all the involved transactions, may sometimes be inconvenient with respect to consistency of the final results. Therefore, we may also need permits in addition to conflicts and demands.

To illustrate, consider the coding example involving Tom and John in figure 7(a). Due to our conflict rules and the lack of permits, Tom and John may not execute *Edit_interface*(P) and *Edit_class*(*GUI*) concurrently. This means that only one of the two engineers can update both *P_i* and *GUI_c* at a time. For this reason, Tom and John agree that Tom should do the necessary updates on *P_i* as well as *GUI_c*, while John modifies *P_c*.

This far, the interaction between Tom and John's transactions is legal. However, to make sure that all interface information included in *GUI_c* are complete, before producing the object code of *GUI*, Tom has to browse the object code of P. Therefore, the scenario in figure 7(a) must be considered invalid. Tom has to wait until the object code of P is available.

Hence, Tom and John need a demand specifying that before *Compile_class*(*GUI*) can be executed – after *GUI* and P are modified, *Compile_class*(P) must be performed. As a result, the interactions depicted in figure 7(b) represent a valid sequence of actions,

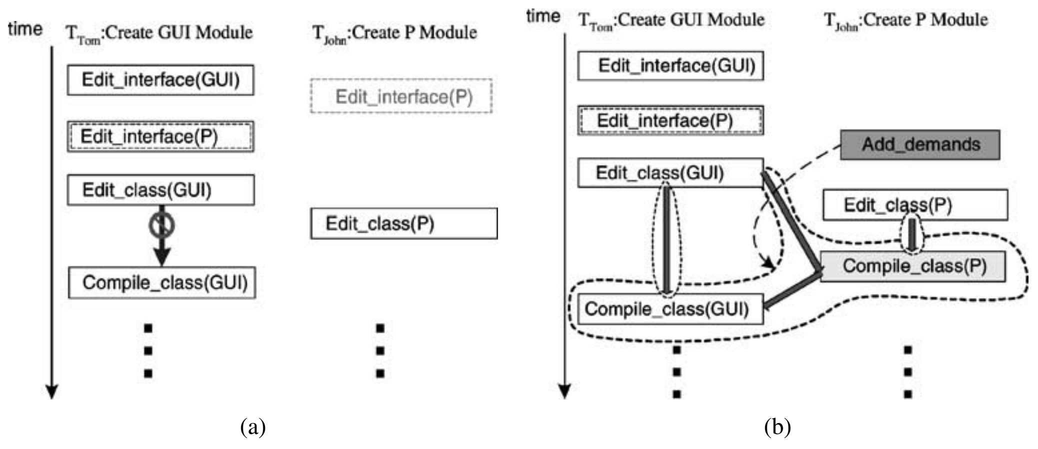

Figure 7. (a) Invalid scenario according to conflicts and demands constraints, and (b) valid scenario following conflicts and demands constraints.

using both conflict and demand constraints. A discussion of demand additions (the thick lines) is provided in section 4.3.2.

Case 4: Conflicts, permits and demands
The main conclusion of the discussions in the three cases above is that we may need all the three constraints to ensure correctness and enable some degrees of flexibility – i.e., controlled sharing of data. To illustrate this case, recall our coding example with Tom and John from case 1, and consider the scenario in figure 8(a).

Suppose that John has to make some modifications to *P_i* after Tom has created the *GUI*-class. Because of the prevailing permits (see case 1), this is OK. However, due to these modifications, the *P_i* incorporated by Tom is no longer up to date. This implies that Tom ought to re-execute *Edit_class*(*GUI*) to reflect the changes made to *P_i*. A natural way to ensure this is to add a demand enforcing the execution of *Edit_class*(*GUI*) after *Edit_interface*(*P*). Figure 8(b) shows a scenario following this rule.

Here, the commit dependency between T_{Tom} and T_{John} still applies, meaning that after the first *Compile_class*(*GUI*), Tom must wait for John's commit before he can terminate too.

4.3.2. Handling dynamic re-specifications
We have stressed several times the ability to modify the three constraint types – i.e., conflicts, permits and demands – at runtime. Table 5 summarizes our conclusions from an analysis of the possible adjustments.

4.3.2.1. Conflicts
Conflicts should not be modifiable during runtime to avoid unmanageable complexity. However, indirect refinements are still possible. For instance, we can apply user-controlled locks (see section 3.1) to gain more restriction, and we can add permits to gain more flexibility.

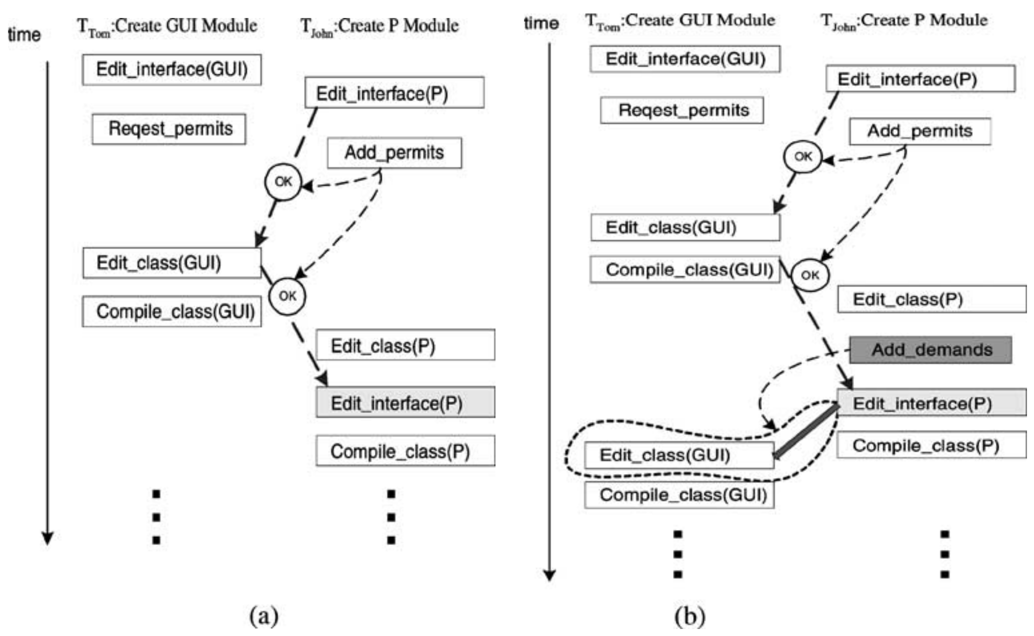

Figure 8. (a) Illegal scenario due to incompleteness of conflicts and permits, and (b) legal scenario using conflicts, permits and demands.

Table 5
Properties of the execution constraint tools.

	Addition	Removal
Conflicts	N/A[a]	N/A[a]
Permits	Always OK	May cause difficulties
Demands	May cause difficulties	Always OK

[a] Conflicts are not modifiable.

4.3.2.2 Permits

Addition of permits. Apart from the concurrency anomalies that may occur, which are managed separately [29], the introduction of permits at runtime does not cause any problems since this will only affect operations that will be performed in the future.

To handle a new permit relationship – i.e., *permit*$(t_j, t_i, Op\_set, Ob\_set)$, allowing t_i and t_j to perform a set of operations *Op_set* on a set of objects *Ob_set*, a specification manager (see section 5.3) executes the following steps:

Add_permit$(t_j, t_i, Op\_set, Ob\_set)$

(1) Check whether conflicts for *Op_set* and *Ob_set* exist. If this is true, define the permits.

(2) If no conflicts exist, check whether the data in *Ob_set* are locked:

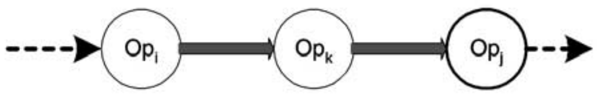

Figure 9. Illustration of an In-Between constraint.

(a) if the lock type is user-controlled, request degradation. If this can be granted, degrade the lock to a collaborative lock and define the permits;

(b) if the lock type is collaborative, define the permits.

(3) Otherwise, no permits are necessary.

Removal of permits. In contrast, the removal of a permit relationship may not always be straightforward. Difficulties may arise due to existing conflict specifications. For example, some operations in the *Op_set* may be conflicting, thus causing an execution to be invalid. However, this only applies if the involved operations have already been executed. To manage the removal, the specification manager executes the following steps:

$Rem\_permit(t_j, t_i, Op\_set, Ob\_set)$

(1) For all operations in *Op_set*, check for conflicts. For each conflict found, check in the log whether the operation has already been executed. If the operation is found in the log, report the conflict to the user, providing options for further actions: (i) invalidate (through abort), (ii) ignore (not recommended), or (iii) cancel (no removal).

(2) Remove the permits.

4.3.2.3. Demands

Addition of demands. Although there are no problems adding permits, introduction of demands is not always so straightforward. The following discussion considers different cases and situations that the specification manager must handle when a user adds demand constraints.

Case 1: Force an occurring operation Op_k in-between operations Op_i and Op_j

An *In-Between*(Op_i, Op_k, Op_j) constraint requires that if all three operations Op_i, Op_j and Op_k occur, then Op_k must succeed Op_i and precede Op_j in sequence. This is illustrated with a linking graph in figure 9.

See case 3 in section 4.3.1 for an application of this constraint type. The worst case here is when such an addition is requested just before Op_k is to be executed. It is OK if Op_j has not already been executed (irrespective of whether Op_i has been executed) – it is just a matter of adding the linking requirements and scheduling the involved operations appropriately (if all three occur). But if Op_j has already been executed, it has to be re-executed – with possible recursive effects – when Op_k is executed (if Op_i also occurs).

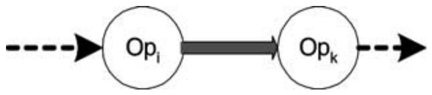

Figure 10. Illustration of an Occur–Follow constraint.

Case 2: Force a non-occurring operation Op_k to occur and follow operation Op_i
An *Occur–Follow*(Op_i, Op_k) constraint requires that if a specific operation Op_i occurs, then the given operation Op_k must also occur and follow Op_i in sequence. This is illustrated with a linking graph in figure 10.

See cases 3 and 4 in section 4.3.1 for three applications of this constraint type. Such an addition typically involves forced scheduling of operations. The only recursive effects here would be other forced executions but no re-executions.

Note that operations may be involved in several demand constraints – of both types even – at the same time.

Removal of demands. As seen with the adding of permits, the removal of a demand does not pose any difficulties. This is because occurring patterns of operations are not prohibited by nonexistent demands.

5. Support for heterogeneous cooperative environments

Cooperative work environments are heterogeneous in addition to being dynamic. Therefore, openness is crucial. Previously, we have argued the necessity of supporting changes in the environment, while users are executing their transactions. In the same line, we stress the necessity of providing the support for a wide range of resource management systems. Motivated by this, we have developed the CAGISTrans transaction management system as a middleware [29]. Our objective is to provide a system built on top of available DBMSs, which at the same time can offer features that extend these DBMSs in terms of distribution support, increased resource availability and database independence, as well as providing flexible cooperative transaction management support.

Figure 11 depicts the architecture of the CAGISTrans transaction management system. A natural implication of the distinction in figure 1 is to provide an environment that allows both design time specification of transaction models and runtime management of transaction execution. Therefore, the transaction management system is divided into two separate environments, consisting of a *specification environment* and a *runtime management system*.

5.1. The specification environment

An important aspect of our system is that all specifications are encoded in XML. Therefore, the most dominant components of this environment are those used to define, parse and validate transaction model specifications.

In this environment, a model designer specifies the desired transaction characteristics and the initial execution constraints and operation sets, before he/she executes

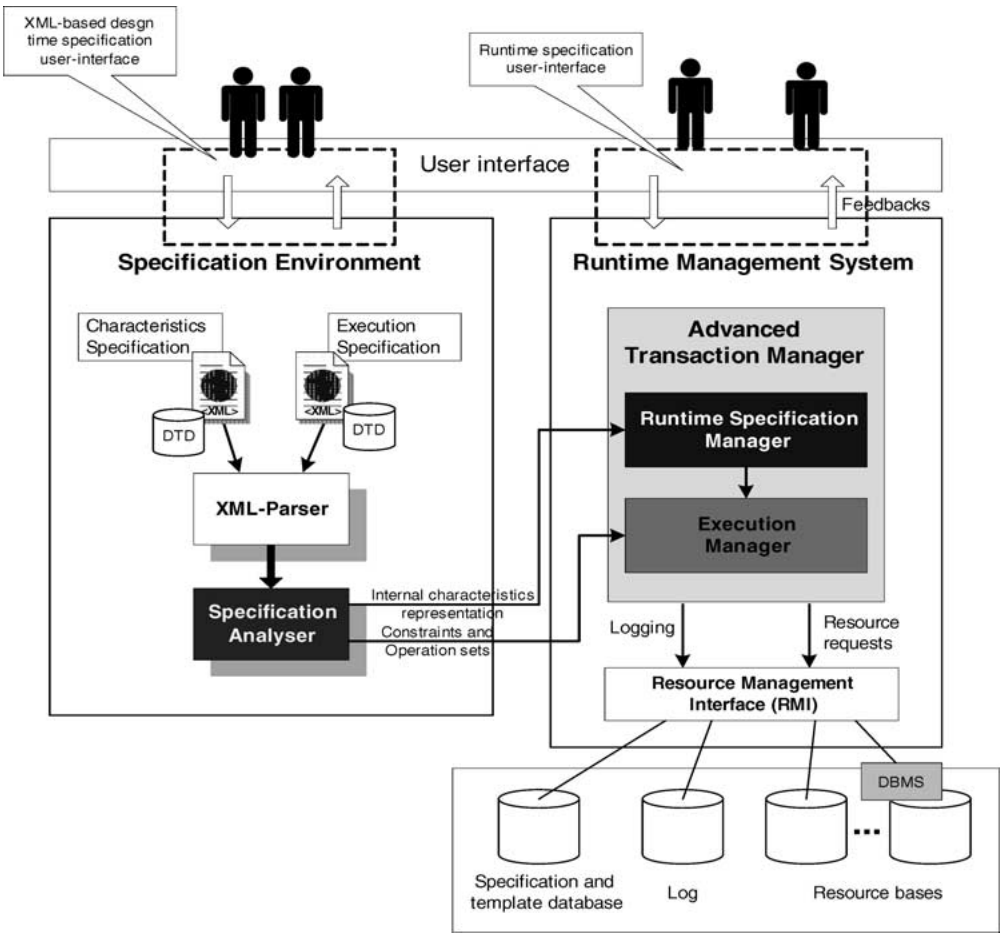
XML-based desgn time specification user-interface
Runtime specification user-interface
User interface
Feedbacks
Specification Environment
Runtime Management System
Characteristics Specification
Execution Specification
<XML>
DTD
XML-Parser
Specification Analyser
Internal characteristics representation
Constraints and Operation sets
Advanced Transaction Manager
Runtime Specification Manager
Execution Manager
Logging
Resource requests
Resource Management Interface (RMI)
DBMS
Specification and template database
Log
Resource bases

Figure 11. Architecture of the transaction management system.

his/her transactions. These are then passed through an XML parser, which checks the specification against prespecified DTDs – document type definitions. Next, a specification analyser goes through the specification and translates the generated elements from the XML parser into (a) internal representations of transaction characteristics – i.e., dependency formation rules, enabled correctness criteria and applied policies, and (b) operations sets and execution constraints – i.e., conflict table entries, initial permit sets, and initial demand rules.

Referring to figure 11, we have implemented an *XML*-based design time specification of transaction characteristics based on definitions of the desired "non-ACID" requirements. In addition, a complete *XML/DTD*-based language has been implemented. Here, DTD plays the role of a language dictionary. To parse specifications, we have integrated an *XML-parser* from IBM – i.e., the XML4J-parser.[3] An integral part of

[3] See http://www.alphaworks.ibm.com/aw.nsf/techmain/xml4j.

Table 6
Overview of implemented components.

Blocks	Implemented	Comments on not fully implemented components
User interface		
Design time specification	60%	Fully functioned graphical user interface is not finished yet Using standard text editor to hard-code the specification
Runtime specification	100%	
Characteristics and execution – XML and DTD	100%	
XML parser	100%	
Specification analyser	60%	Automatic check for supported model is not available
Runtime specification manager	80%	Handling of advanced conflict rules is only designed
Execution manager	75%	Realization of dynamic restructuring is only designed Enforcing advanced conflict rules is not supported
Resource management interface	90%	Interfaces to standard ACID transaction using JTA/JTS is implemented but not integrated and tested yet

the CAGISTrans prototype is also analysing and translating the outputs from the XML parser – i.e., the *specification analyser (SA)*. An overview of the implementation status of the components is provided in table 6.

5.2. The runtime management system

The representation of the internal characteristics from the specification environment is used by the runtime management system to control and manage the execution of transactions. Its main component is the *advanced transaction manager*, responsible for making sure that the execution of transactions conforms with the transaction characteristics and execution specifications from the specification environment. This manager again consists of two sub-components; the *runtime specification manager (RSM)* and the *execution manager (EM)*. Before transactions are executed, the RSM checks that the specifications can be supported. The CAGISTrans system is designed to operate on top of DBMSs. Thus, if a user chooses to rely on an underlying DBMS to handle correctness control, the RSM checks – through the *resource management interface* (RMI) – that such support is present. Unfortunately, many existing DBMSs have their own ways of implementing concurrency control. For example, it is often impossible to access the lock tables that are managed by these systems. Instead, some systems allow locks to be explicitly acquired and released through SQL-statements.[4] All in all, "cooperative" management of con-

[4] See for example the GET_LOCK and RELEASE_LOCK statements in MySQL.

currency control between a CAGISTrans system and DBMSs can only be achieved to a certain degree. A way to overcome this limitation is allowing the underlying DBMSs to do the required control of sharing. As such, correctness of data managed by the DBMSs can be assured in each system, but achieving global consistency (across the systems) is still an important challenge that must be further considered. This issue has been a subject for intensive research in the last couple of decades. Several solutions have been proposed in the literature [4,20,21]. However, it is widely agreed that managing consistency across multiple heterogeneous and autonomous DBMSs is an issue that still deserves further attention.

From a transaction commitment point of view, CAGISTrans can implicitly affect the way underlying DBMSs handle commitment by enabling or disabling automatic commitment – i.e., “auto-commit”. There are database systems that allow this specification through JDBC drivers.

From these perspectives, checking the underlying DBMSs for the support provided means checking (1) which type of isolation level is supported for each DBMS and (2) whether auto-commit can be enabled or disabled as needed.

As an alternative to relying on DBMSs, the user may want to rely on the advanced support provided by the CAGISTrans system. In that case, the RSM will manage the execution of user transactions in cooperation with the EM.

Further, recall that new specifications may be introduced during runtime. The RSM provides the necessary support for the required modifications. For example, when new constraints are to be introduced, the RSM first checks the actual specification. Then it gives the user all the necessary information on the actions that must be taken. To illustrate, consider the addition of new constraints to the current set of demands as discussed in section 4.3. The RSM checks the log – i.e., an execution descriptor – for all executed operations. If any new constraint may cause invalidation of some operations because a specific operation is to appear between two already performed operations, the RSM will inform the user about this, requesting him/her to choose the way to proceed – either to cancel the addition or to allow an invalidation.

Validation of transaction executions is managed by EM. The EM uses the specified correctness criterion to control the execution of a specific transaction. The EM can be seen as a transaction scheduler in the sense that it is responsible for executing the transactions on the basis of requests. In addition, it is responsible for managing the execution of relevant management operations. This includes applying the prevailing dependencies, which are relevant when issuing *begin*, *abort* or *commit* (see section 4.1). Further, the EM allows transactions that have a nested structure to *spawn* new transactions upon request. And, if necessary, the EM also accomplishes *delegation* of operations among transactions.

As shown in table 6, we have implemented a major part of the *runtime management system*. More specifically, we have designed and implemented the *user interface* which supports interactive transaction initiation and termination. The RSM component is implemented to handle read/ write-based conflicts only. Handling more advanced conflict rules has been designed, but this is still not integrated in the CAGISTrans prototype.

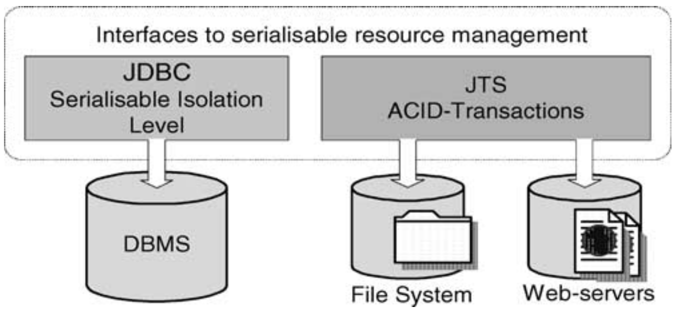
Interfaces to serialisable resource management
JDBC
Serialisable Isolation Level
JTS
ACID-Transactions
DBMS
File System
Web-servers

Figure 12. Interfaces to serializable resource management systems.

Permits and demands are, on the other hand, part of the prototype implementation. The current implementation of RSM also supports specification of the two latter constraints during runtime. Further, the EM component allows validation of transaction executions using permits and demands. However, even though dynamic restructuring is designed, it is still missing from the EM prototype implementation. Finally, some advanced operations are integrated and supported by the EM. These include the operations used to handle access to workspaces that we will elaborate upon in section 6.

5.3. Ensuring correctness

In CAGISTrans, there are two possible ways to achieve desirable final results: enforcing serializable execution and adopting user defined correctness criteria.

The serializability criterion is mainly supported by underlaying resource management systems. Therefore, the CAGISTrans system provides this criterion through the interfaces to serializable resource management systems, as shown in figure 12. These interfaces are realized with JDBC-Java Database Connectivity[5] – and the JTA/JTS – Java Transaction API[6]/Java Transaction Service[7] – interfaces. Using the JDBC interface, we are able to specify the isolation level of the underlying DBMS connected to the CAGISTrans system. This means that we may enforce serializable execution on that DBMS by choosing the serializable isolation level – i.e., "`TRANSACTION_SERIALIZABLE`".[8] On the other hand, if we use non-DBMS resource providers – such as standard file systems and web-servers – the CAGISTrans system will use the JTA/JTS interface to enforce serializable execution. Note, however, that this enforces a full ACID execution, making it impossible to use relaxed atomicity too.

Unlike serializability, management and enforcement of user defined criteria is achieved through the CAGISTrans system itself. Figure 13 shows how we implement this

[5] See http://java.sun.com/products/jdbc/.

[6] JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system. See http://java.sun.com/products/jta/.

[7] JTS specifies the implementation of a Transaction Manager which supports the JTA, based on the CORBA OTS [25]. See http://java.sun.com/products/jts/.

[8] See http://java.sun.com/products/jdbc/.

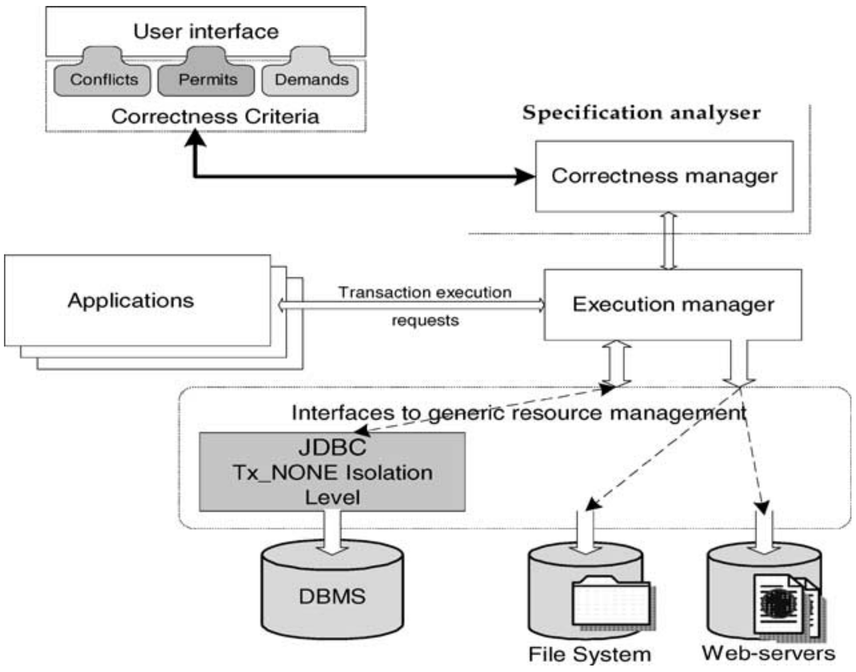

Figure 13. Managing user-defined correctness criteria.

in CAGISTrans. As depicted, we are still able to use a database as a resource provider through a JDBC interface. But, now the isolation level is "`TRANSACTION_NONE`", indicating that all transactions are handled by the CAGISTrans system. JTA/JTS, on the other hand, only allows ACID transactions. Therefore, a JTA/JTS interface for extended transactions is not applicable. Rather, the CAGISTrans system takes over the responsibility for handling advanced transactions through the specification analyser (SA) and the execution manager.

The user specifies the desired correctness criterion through a user interface, allowing the specification of conflicts, permits, and demands. These are, thereafter, handled by the *specification analyser* (SA) which checks their validity and consistency (see section 4.3.2 for a discussion). The SA cooperates with the EM to ensure that the transactions meet the specified constraints when they are executed.

Referring back to our implementation overview in table 6, the JDBC interface to the DBMSs is supported by the current CAGISTrans prototype. For example, we use a MySQL[9] server as database for storing management and control information. This database server was chosen due to its simplicity in use and implementation. Further, we have tested the resource management interfaces against a PostgresSQL[10] DBMS. We are also currently investigating use of the IBM DB2.[11]

[9] See http://www.mysql.com/.
[10] See http://postgresql.readysetnet.com/.
[11] See http://www.ibm.com/db2/.

5.4. Experiments with conflicts and patterns

The ideas underlying the correctness constraints tools of our CAGISTrans framework (see section 4.3) have originally been inspired by conflicts and patterns from the cooperative transaction hierarchy model [24].

Before implementing our constraint tools, we first tried to implement *conflicts* and *patterns*. We did some experiments with specification of grammars for conflicts and patterns. From this we learned that meeting the dynamic requirements of cooperative work was not a trivial matter. First, we had to define a complete LR(0)-grammar for the patterns and conflicts, and generate a lexical analyser using – e.g., *lex* [19], and a parser using – e.g., *yacc* [19], before we could execute the involved actions. For this to be possible, however, we also needed complete transaction (operation) sets to be carried out. Moreover, change incorporation at runtime would not be possible as we first had to modify the grammar, then generate a new lexical analyser and a new parser, before execution of the actual transactions could start again. These are the main reasons for deciding to develop our own demands and a revision of conflicts.

An additional lesson learned from these experiments was that the complexity of the grammar increased proportionally with the number of transactions and operations involved. Such a complexity would make the cost of managing the transactions unduly high. This further supports our argument for allowing runtime adjustments. In fact, by making such adjustments possible, we might allow stepwise specification, thus coping, to some extent, with the aforementioned complexity.

5.5. Comparison with other systems

The CAGISTrans system architecture distinguishes between a *specification environment*, providing a means for specification and validation of transaction models, and a *runtime management system*, offering support for execution of transactions and management of their behaviour during runtime. The main advantage of such a separation is the ability to reason about properties and behaviour of transactions before they are executed, allowing a model designer to customize his/her model as desired. Moreover, the ability to refine the model at runtime is useful as we may need to support new requirements – e.g., due to the evolutionary behaviour of the actual activity. There are other approaches that apply similar separation of specification and runtime environments. Most comparable with our work from this perspective is TSME [11]. Compared to this, CAGISTrans has attempted to improve the dynamic support. In TSME a specification is tested with respect to whether it may be supported before it is applied. Once the specification passes, and the transactions are executed, there is no way to change the provided specification before the transactions are terminated or interrupted. Moreover, support for dynamic restructuring was beyond the scope of TSME.

Further, while TSME was implemented as complete systems, our framework is built on the middleware principle, making it possible to provide support for a wide range of resource management systems.

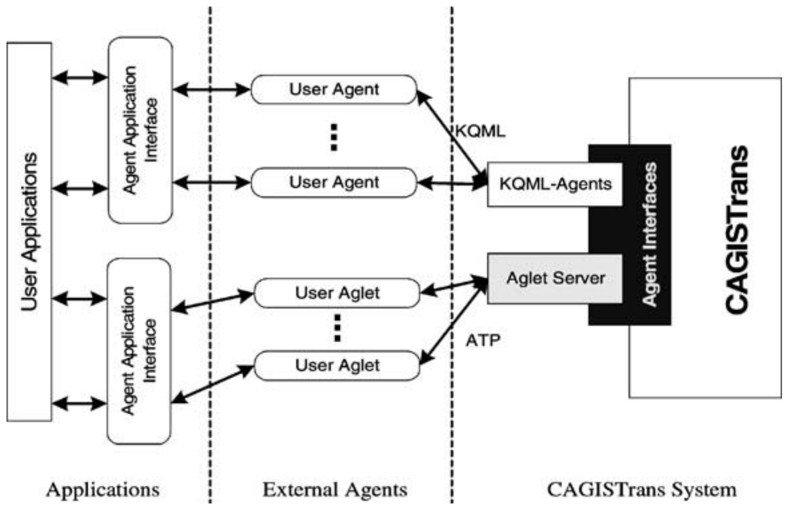

Figure 14. Interface to external agents.

5.6. Implementation using mobile agents

Proof of concept prototypes have been implemented to realise the CAGISTrans framework based on the architecture described in this section. The prototypes were mainly built using agent technology. The reason for this was primarily to facilitate the integration of distributed support – e.g., Web support. Another important aspect was mobility which can be exploited to reduce the communication cost. Also, agents can be exploited to facilitate cooperation between diverse system components and act as interfaces to external applications. To deal with such applications we have implemented an *external agent interface*.

The main purpose of the external agent interface is to allow external agents to access resources administrated through the CAGISTrans system. Implicitly, this means that if several agents cooperate via the same objects, our system provides transactional services which support the cooperation.

Currently, we have implemented a system that allows both aglets and other KQML-speaking agents to access such CAGISTrans services. This allows agents assisting users in a cooperative process to follow a cooperative policy specified for that cooperation. This means that when one or more agents access a back-end resource server they will follow the prevailing correctness criteria specified for their interaction.

Figure 14 depicts how the external agent interface in CAGISTrans is realised. As already mentioned, CAGISTrans currently supports two categories of agents: KQML-speaking agents and aglets. KQML agents communicate with the CAGISTrans system through a facilitator (see figure 15), using the KQML communication protocol. A facilitator is a service provider maintaining information about active system agents – i.e., special purpose agents forming part of the CAGISTrans system itself – that accomplish agent requests. A system agent, which is an aglet, advertises all available services to

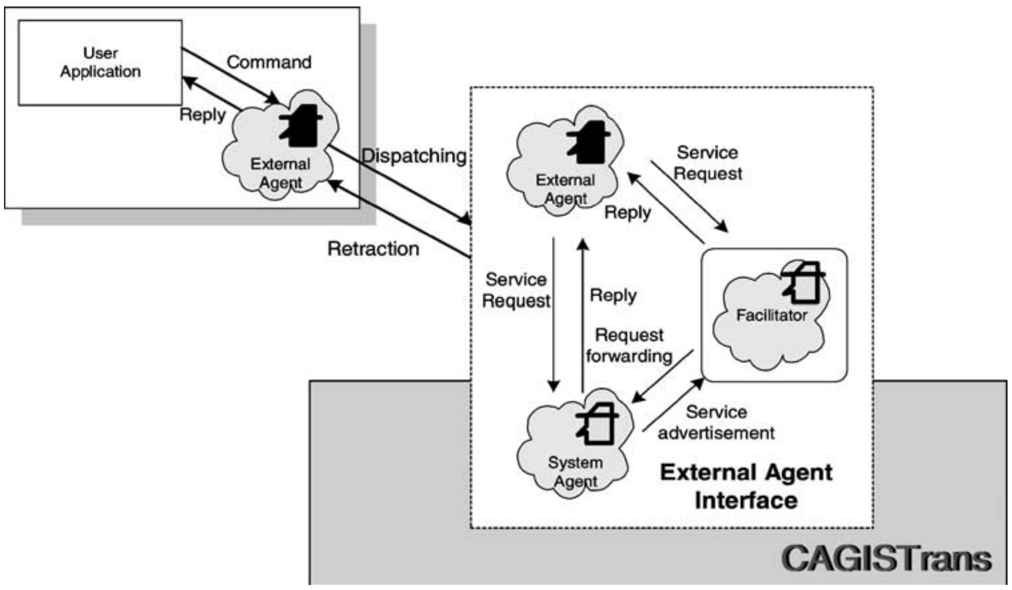

Figure 15. Interaction between an agent and the external agent interface components.

the facilitator along with its physical address and symbolic name. Such services include executing operations through CAGISTrans, mediating transparent accesses to underlying distributed resource bases, and presenting notifications for awareness purposes. This means that a client application can use agents to request executing operations such as workspace operations through the CAGISTrans external agent interface. They can also be used to access remote resource servers where documents or objects used in cooperative activities are stored. A list of such servers including their exact addresses is managed in CAGISTrans. This list can be provided by a system agent to external agents. Note that, as mentioned above, the distribution of document servers is intended to be transparent to the users. Thus, the use of agents here aims at facilitating such a transparency. Finally, external agents may "poll" notification information that is useful for awareness purposes. This means that some of the awareness services can be presented by system agents to external agents, and then again "forwarded" to users.

An external agent asks the facilitator whether there are system agents that can offer the service it needs. If the service request matches one of those being advertised, the facilitator will provide the external agent with the address and the symbolic name of an appropriate system agent. Hereafter, the two agents will continue their dialogue until the external agent has accomplished its tasks.

In addition to being KQML agents, external agents may also be aglets. Since our system agents are aglets, unlike a KQML agent, an (external) aglet may communicate directly with a system agent, exchanging messages with it until its service requests and corresponding tasks are accomplished. Thus, intervention by the facilitator is not necessary. Here, communication between two aglets follows the agent transport protocol (ATP) for message passing. See [18] for an overview of the ATP.

6. Integrating workspace management

A widely accepted approach to supporting cooperative work is to provide a shared space which can be logical or physical where groups of people can perform tasks together. Such a space is called workspace. It is common to assume the existence of both shared and private workspaces. This allows people to alternately work in a group, and carry out individual activities in private, thus enabling partial privacy. Integration of the workspace concept in CAGISTrans was motivated by the need to facilitate the management of sharing in data intensive systems as well as to further widen the application areas of the CAGISTrans framework.

However, for this to be practical we had to extend the concept. In many existing systems the control of access to the workspaces is either left to the user to figure out – e.g., many groupware-based approaches, or it is unduly strict – e.g., many database centred approaches. This makes it necessary to provide flexible support which is powerful enough to bridge this gap.

6.1. Flexible workspace support

First, it is necessary to organize the workspaces into a nested structure, enabling multiple levels of sharing (see figure 16). In addition to *individual* and *public workspaces* there are *group workspaces* that provide limited groups of people with shared space [29].

To illustrate this, let us assume that our software company consists of engineers cooperating to perform the coding task, developing a software artefact. Usually, the size of a software artefact to be developed is so large that to address the induced complexity the engineers have to be organized into small groups, coding different but related modules. To allow flexible interactions among group members, there is need for a place where these members can exchange their program codes. Note, however, that sharing is only reasonable if the code has reached a certain maturation phase. Until then each member ought to work in private – i.e., in a private workspace. Later the produced program codes can be released to a place where a larger group can access them. This process continues to ever larger group workspace until one reaches the public workspace.

Extensions to the workspace access support are also necessary to allow flexible accesses to workspaces [29]. In brief, the operations needed are *write-check-out (wco)* and *read-check-out (rco)*, distinguishing the intentions behind the check-out operation, *upward-check-in (uci)*, checking in data from any level to the next level, *check-in (ci)*, checking in data from any workspace to the public workspace, *refresh*, updating a local copy of data with the one residing in the parent workspace, and data manipulation operations such as read, write, insert and delete, plus diverse advanced operations (see section 4.2). Note that the main difference between *uci* and *ci* is that when a user performs *uci*(*obj*), he/she only puts a copy of *obj* from his/her workspace into a parent workspace. He/she still has the ownership on *obj*. But, with *ci*(*obj*), *obj* is moved to the public workspace, and the ownership to *obj* is released.

We can now return to our coding example. Using the above workspace operations, the detailed interactions between Tom and John corresponding to figure 8(b) which

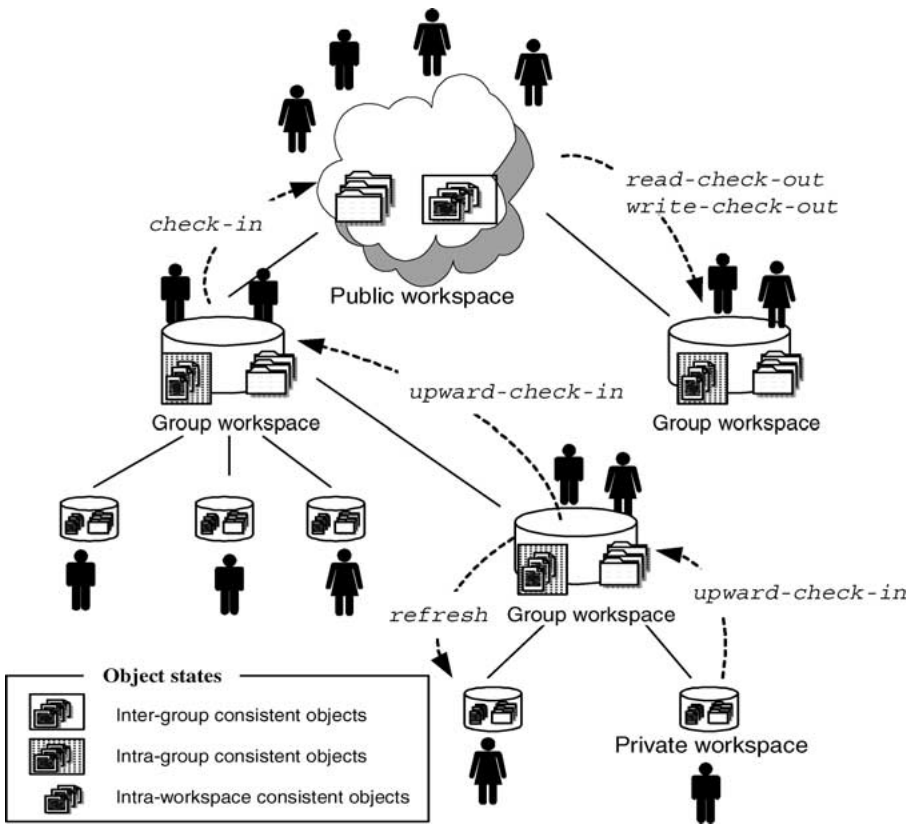

Figure 16. Nested workspaces and corresponding operations.

again is based on figure 6(b) are depicted in figure 17. Before editing the *GUI* interface (*GUI_i*), Tom must check out *GUI_i* for write – i.e., *wco*(*GUI_i*). This allows him to make all the necessary changes to *GUI_i*. John does the same with *P_i*. When John is finished, he updates the copy of *P_i* in his and Tom's group workspace by issuing *uci(P_i)*. Here, John chooses *uci* rather than *ci* since he knows that *P_i* is still incomplete. Therefore, making *P_i* publicly available is not reasonable yet. *P_i* is useful to Tom, though. Actually, Tom is now ready to edit the *GUI* class (*GUI_c*). For this, he needs a copy of *P_i* so he issues *rco*(*P_i*). Thereafter, he checks out *GUI_c* for write – i.e., *wco*(*GUI_c*). Now, while Tom is performing his updates on *GUI_c*, John finds out – after making some changes to *P_c* – that he must do some further changes to *P_i*, which he afterwards checks in to the group workspace once more. As Tom is forced to re-edit *GUI_c* (see figure 8(b)), he must update his copy of *P_i* with the latest changes. Since Tom already has a copy of *P_i*, he accomplishes this by issuing *refresh*(*P_i*), rather than another *rco*(*P_i*). Hence, Tom will now update *GUI_c* to reflect the changes to *P_i*.

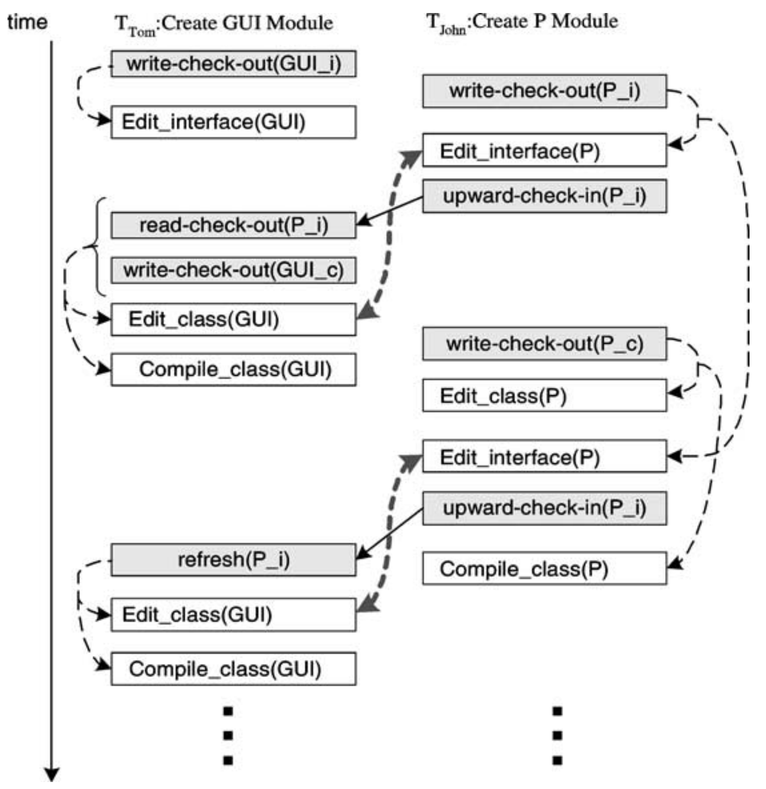

Figure 17. Illustration of the use of workspace operations.

6.2. *Mapping between workspaces and resource bases*

Our workspace concept can be regarded as an abstract concept. This means that physical workspaces do not exist by themselves. Instead, they are logically mapped to underlying resource bases, as illustrated in figure 18.

As depicted in this figure, objects reside on several types of repositories, including file systems, Web servers and databases. Then, when an object is checked-out or checked-in, it is “marked” with where it will logically belong – i.e., to a *private workspace*, *group workspace* or *public workspace*. Thus, to facilitate this mapping, we define an object as a tuple `(id, type, ws-state, address, owner)`, where

- `id` is a unique object identification;
- `type` is the object kind – i.e., `file`, `web-doc`, or `relation` (see figure 18);
- `ws-state` identifies the type of workspace that the object is checked-out from or checked-in to – i.e., `ws-state` may be `private`, `group` or `public`;
- `address` denotes the physical location of the object:
  - if the object type is `file`, the address will be a file path – i.e., “`file://<path-name>`”,

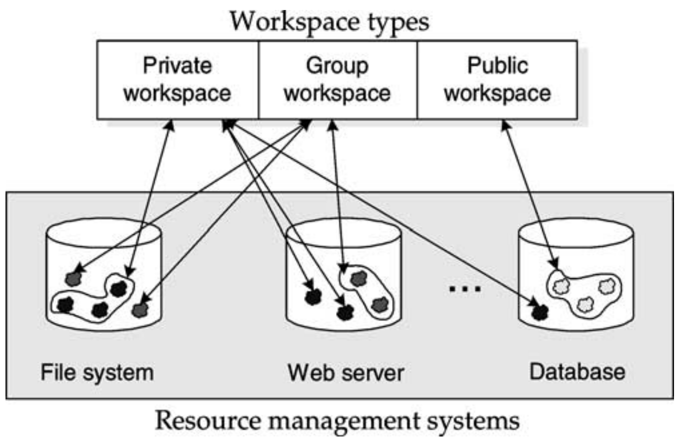

Figure 18. Illustration of mapping between several resource management systems and workspaces.

  - if the object `type` is `web-doc`, the address is either an IP-address or an HTTP URL – i.e., “`http://<ip-address>`” or “`http://<web-url>`”,
  - if the object `type` is `relation`, the address is a database URL – e.g., determining the JDBC driver – i.e., “`jdbc ://<database-protocol>:<database-host>/<database-name>`”;

- `owner` identifies the current owner of the object – i.e, *username* or *groupname.* If `ws-state` is `public`, the owner is empty.

Since workspaces are only logically defined, they can be created as needed depending on the current cooperation situation. They are simply determined by the objects' `ws-state` and `owner`. This means that objects residing in the private workspace owned by Tom are all objects such that `ws-state='private'` and `owner='tom'`. Objects in a group workspace `Gi` are all objects such that `ws-state='group'` and `owner='Gi'`. Objects in a public workspace are all objects such that `ws-state='public'` and `owner=' '`.

The above mapping facilitates the use of our extended workspace operations. This means that when an object is checked-out from or checked-in to a workspace, it suffices to update the objects `ws-state` and `owner` accordingly. The address will thus specify the link to the resource base to be used.

6.3. Correctness assurance and coordination

To ensure correctness and coordinate interactions through workspaces, we use user-defined constraints to specify prohibited interactions, permitted interleavings, and required sequences of operations, as described in section 4.3.

To illustrate, assume that the commutativity (conflict) rules from section 4.2 apply, and consider figures 6, 8 and 17. Without the permit relationship from figure 6(b), when Tom issues *rco(P_i)*, the execution manager would have to request him to wait until John has committed his updates on *P_i*. But, with the indicated permit relationship, upon request, John will be requested to make available his partial results by issuing *uci(P_i)*, allowing Tom to proceed with his task.

Further, John's repeated processing of *P_i* will affect Tom's processing of *GUI_c*. As indicated in figure 8(b), this triggers a demand relationship too. This again will force the execution of Tom's *refresh* operation, after the new copy of *P_i* is made available through John's *uci* operation.

7. Comparison with other frameworks

Our CAGISTrans framework differs from other relevant work in providing a combination of support for explicit customization of non-ACID requirements, user-defined correctness criteria, explicit support for applied policies, dynamic restructuring and workspace support. This is summarized in table 7. As illustrated, some features have been adopted from existing transaction models and frameworks, and then extended in forming the CAGISTrans framework. This will now be explained:

- *Explicit customization of non-ACID requirements*. To our knowledge, support for explicit definition of non-ACID requirements to specific applications has not been proposed before. Existing frameworks let non-ACID requirements be implicitly tailored, but do not allow users to define them in accordance with their application's needs.
- *User-defined correctness criteria*. Patterns and conflicts as user-defined correctness criteria tools were originally proposed by [34]. They were improved in the Cooperative Transaction Hierarchy [24], where they were represented as state-machines. The demands and conflicts of our CAGISTrans framework are built on these concepts. However, in contrast to patterns, demands are represented as directed graphs identifying and representing sequences of actions required to ensure correctness. In addition, while a complete set of patterns has to be defined before transaction execution – without any possibility for re-definition during runtime, demands can be modified while transactions are being executed. Moreover, our conflicts are defined as tabular relationships rather than with a state-machine. In fact, our conflict concept is more similar to that associated with semantic-based concurrency control than those proposed in [24]. For a discussion of semantic-based concurrency control issues, see [31]. The benefit here is that when transactions are to be validated, instead of checking a state-machine which is usually complex, CAGISTrans utilizes simpler tabular inquiries. Finally, permits were originally proposed with ASSET [3]. Our use of the concept also allows flexible sharing. However, permits in CAGISTrans are accompanied by conflicts and demands. For example, we utilize permits both to override specific conflicts and to enable concurrent accesses to locked objects. Hence, CAGISTrans aims at providing a more flexible, but controlled type of sharing.
- *Explicit support for applied policies*. Applied policies in CAGISTrans determine the relevant mechanisms, and specify rules for how and when to use them. Although the

Table 7
A summary of CAGISTrans features and their relation to other frameworks.

CAGISTrans feature	Inspiration or origin	Supported by other frameworks	Extension or difference
Explicit customization of non-ACID requirements	None or not known	Done implicitly in all other relevant frameworks – i.e., explicit specification not supported	Explicit support
User-defined correctness criteria	Conflicts: Cooperative transaction hierarchy and semantic-based concurrency control Permits: ASSET Demands: Patterns as in cooperative transaction hierarchy	TSME: Through correctness dependencies ASSET: Through permits	The combination of all three constraint tools Dynamic support in terms of runtime modifications
Explicit support for applied policies	None or not known	Not supported by any other relevant framework	Brand new
Dynamic restructuring	Splitting as in Split and Join transaction model Delegation as in ACTA	RTF: Dynamic restructuring ASSET: Delegation of operations (static)	Support for both lock and operation delegation Application of user-defined correctness criteria
Integrated workspace support	Classical check-in and check-out models, and Coo, EPOS and TransCoop	Not supported by any other relevant framework	Combination of unlimited nested structure, flexible workspace operations, and user-defined coordination

distinction between mechanisms and rules has long been known in both the database and the CSCW communities, to our knowledge, it is still not addressed in connection with transaction models and frameworks. Hence, our use of this concept, allowing users to explicitly fit policies in accordance with the needs of the applications is probably unique.

- *Dynamic restructuring*. This concept was originally proposed in the Split and Join transaction model [15]. CAGISTrans applies a similar approach to restructure transactions while they are being executed. The difference lies in the way the restructuring is performed. CAGISTrans realizes dynamic restructuring by combining transaction *splitting* – from the Split and Join transaction model – with the notion of *delegation* – which originated with ACTA [5]. Finally, while Split and Join transactions apply serializability as the correctness criterion, CAGISTrans allows user-defined criteria.

- *Integrated workspace support*. In the context of transactions, the workspace concept has been extensively used in EPOS [6,7], Coo [12], and TransCoop [8,35], among others. In brief, EPOS and Coo use temporary, shared sub-databases (scratch-pads) for data exchange and integration work. TransCoop focuses on the exchange of operations instead of exchanging data between private and public workspaces, while correctness control is handled through history validation and merging mechanisms. Our workspace concept differs from these in combining several aspects. First, we use a nested workspace structure that applies *unlimited nesting levels* to regulate the degree of sharing. To our knowledge, existing approaches are restricted to *two* – i.e., private and public, and *three levels* – i.e., public, semi-public, and private. Second, our CAGISTrans framework applies several extended workspace operations enhancing workspace interaction. Finally, our approach utilizes user-defined constraint tools – cf., conflicts, permits, and demands – for coordinated workspace access.

8. Discussion and conclusion

The diversity of cooperative work means that there are extensive possibilities for customizing all offered support, including transactional support. There are many transaction models and a few transactional frameworks that have provided useful foundations for such support. Still, there are problems that must be faced due to this diversity.

The dynamic nature of cooperative work is an important challenge addressed in this paper. Our objective has been to extract beneficial features of existing models and frameworks and extend these into a new framework that is able to address this problem. The fundamental idea in our framework is distinguishing between design time and runtime specification of transactions. In this way, predictable parts of a transaction model can be specified before a transaction is executed, whereas parts which are not possible to reason about a priori are specified during runtime. The main strengths of such an approach is the ability to meet new requirements during runtime, and the ability to separate the specification of transaction models from the execution of the actual transactions. As a whole, the way we combine support for explicit customization of non-ACID requirements, user-defined correctness criteria, explicit support for applied policies, dynamic restructuring and workspace support is probably unique compared with other existing transactional frameworks.

Nevertheless, there are several issues worth further discussion. A key issue is the trade-off between system transparency and user management. From a CSCW perspective, too much system management and too little user control is unacceptable. Users might then get the feeling of losing the whole picture of what is going on, which is against the philosophy of CSCW. As a result, most CSCW applications rely on social interactions to handle concurrency. But, from a database point of view, this is unacceptable as data consistency cannot be guaranteed. Thus, this has given rise to the need for a sensible trade-off. The specific combination of user-managed specifications and system-based control in our CAGISTrans framework may give an acceptable result. Our framework has been designed to allow users to specify appropriate transaction models based on application needs, and have the system do the validation, management and control based on this. However, the cost still to be paid is that users may be required to have a level of expertise above what can be expected from an average user. This triggers the necessity of developing a graphical user interface to make our system more user friendly. This will, for instance, relieve users from coding transaction models in XML themselves.

Another important issue is performance such as transaction throughput. Traditionally, high transaction throughput has been one of the main requirements for transaction processing systems. The discussion in section 4 indicates that our framework's execution of advanced transactions may introduce some overhead. For example, several execution "parameters" have to be in place before a transaction is executed. Further, introduction of new constraints may also imply control and management tasks, which could slow down the overall speed. However, transactions supporting cooperative work exist for long periods of time. Therefore, they may be more sensitive to response time performance than system throughput. Hence, some extra time needed for validation, management and control purposes will be less critical in the global picture. For this reason, we have primarily emphasised provision of flexible support for cooperative activities rather than optimization of the transaction speed itself. Nevertheless, optimising the overall system throughput is also an important issue that deserves further development. Thus, this will be a significant subject for further research.

Another performance issue is our use of Java[12] as the implementation platform. Currently, the most significant weakness of Java is its moderate performance, at least compared with C/C++. However, Java is an interpreted object-oriented language with a capability of being executed on heterogeneous environments. In addition, Java programs are inherently capable of being transported over networks and executed at a remote system. Due to our distribution and heterogeneity requirements, the choice of Java was a simple one to make. It is nonetheless important to note that Java's moderate performance can be improved by making Java code executable instead of interpretable. So-called Java just in time compilers (JIT)[13] are already available to help improve the speed of "executing" Java applications.

Currently, there are several issues that our CAGISTrans framework has not addressed, either because they are beyond the scope of this work or because they are still part of our future investigation. Some of these may be seen as limitations of CAGISTrans. First, team work often requires full ad-hoc support, but this is not addressed. This would allow cooperating participants to perform work the way they want to. Con-

[12] See http://java.sun.com or http://javasoft.sun.com.
[13] See http://www.sun.com/solaris/jit.

currency control, for example, is often left to social protocols. However as discussed above, due to the necessity of a trade-off between user-intervention and system control this is beyond the scope of CAGISTrans. Second, no specific extended transaction model (ETM) is provided with our framework, as – e.g., with ASSET and RTF. On the one hand, this could be regarded as a limitation of CAGISTrans. On the other hand, given the diversity of existing models combined with the lack of concesus on which models that suit which situations, we have put our emphasis on the extraction of beneficial features of relevant models, and on exploitation of their combination as much as possible to provide efficient support. Third, full recovery mechanisms are not addressed. At the time we designed our system, this was beyond the scope of our framework. However, our CAGISTrans does allow the use of an underlying database management system to do things such as logging for recovery purposes. A problem first arises when the underlying resource bases are other than legacy databases, which would call for explicit CAGISTrans recovery management. Our framework provides a simple logging mechanism to manage transaction aborts.

The CAGISTrans system is aimed at being interoperable and portable, allowing people to work together across different platforms and geographical boundaries. Hence, we have designed and built a transaction management system that not only provides advanced transactional support but also operates on top of diverse resource management systems. In this sense, the development of our CAGISTrans system follows the middleware principle, making it operable in heterogeneous environments.

The system is implemented combining standard mature technology such as XML and Java[14] and advanced evolving technology such as Software Agents. We refer to [30] for a more thorough description and discussion of our general experience and specific use of these technological tools. The CAGISTrans prototypes [16,27,33] have implemented the major parts of our framework (see table 6). It is our main conclusion that the current CAGISTrans system is able to support the basic features of dynamic transaction management, allowing users to specify models and have the system execute their transactions in a flexible but controlled manner. Our future work will proceed with an implementation of the rest of the designed components. As part of this, we will further investigate the ways to make our system even more efficient and user friendly.

Acknowledgements

We would like to thank Rune Selvåg, Lars Killingdal, and Mufrid Kirlic for their contributions in the implementation of the CAGISTrans prototypes. We would also like to thank Henry F. Korth, Michel Adiba and Kjell Bratbergsengen for their valuable comments to this work.

[14] Java is a trade mark of Sun Microsystems. See http://java.sun.com.

References

[1] R. Barga, A reflective framework for implementing extended transactions, Ph.D. dissertation, Oregon Graduate Institute of Science and Technology (April 1999).

[2] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems* (Addison-Wesley, Reading, MA, 1987).

[3] A. Biliris, S. Dar, N.H. Gehani, H.V. Jagadish and K. Ramamritham, ASSET: A system for supporting extended transactions, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 94)*, eds. R.T. Snodgrass and M. Winslett (May 1994).

[4] Y. Breitbart, H. Garcia-Molina and A. Silberschatz, Transaction management in multidatabase systems, in: *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ed. W. Kim (ACM Press/Addison-Wesley, 1995) pp. 573–591.

[5] P.K. Chrysanthis and K. Ramamritham, Synthesis of extended transaction models using ACTA, ACM Transactions on Database Systems 19(3) (September 1994) 450–491.

[6] R. Conradi, M. Hagaseth and C. Liu, Planning support for cooperating transactions in EPOS, Information Systems 20(4) (June 1995) 317–326.

[7] R. Conradi et al., Transaction models for software engineering database, in: *Proceedings of the Dagstuhl Workshop on Software Engineering Databases* (1997).

[8] R.A. de By, W. Klas and J. Veijalainen, *Transaction Management Support for Cooperative Applications* (Kluwer Academic, 1998).

[9] A.K. Elmagarmid, *Database Transaction Models for Advanced Applications* (Morgan Kaufmann, San Mateo, CA, 1992).

[10] H. Garcia-Molina and K. Salem, Sagas, in: *Proceedings of the ACM International Conference on Management of Data (SIGMOD 87)* (May 1987) pp. 249–259.

[11] D. Georgakopoulos, M.F. Hornick and F. Manola, Customizing transaction models and mechanisms in a programmable environment supporting reliable workflow automation, IEEE Transactions on Knowledge and Data Engineering 8(4) (August 1996) 630–649.

[12] C. Godart et al., Designing and implementing COO: Design process, architectural style, lessons learned, in: *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)* 25–29 March 1996 (IEEE-CS Press, 1996) pp. 342–352.

[13] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques* (Morgan Kaufmann, San Mateo, CA, 1993).

[14] T. Härder and A. Reuter, Principles of transaction-oriented database recovery, Computing Surveys 15(4) (1983) 287–317.

[15] G.E. Kaiser and C. Pu, Dynamic restructuring of transactions, in: *Database Transaction Models for Advanced Applications*, ed. A.K. Elmagarmid (Morgan Kaufmann, San Mateo, CA, 1992) pp. 265–295.

[16] L. Killingdal and M. Krilic, Programmable transactions – project report (in norwegian), Technical Report, Norwegian University of Science and Technology, Student Project (April 2000).

[17] H.F. Korth, E. Levy and A. Silberschatz, A formal approach to recovery by compensating transactions, in: *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB '90)* August 1990, eds. D. McLeod, R. Sacks-Davis and H.-J. Schek (Morgan Kaufmann, San Mateo, CA, 1990) pp. 95–106.

[18] D. Lange and M. Oshima, *Programming and Deploying Java^{tm} Mobile Agents with Aglets* (Addison-Wesley, Reading, MA, 1998).

[19] J. Levine, T. Mason and D. Brown, *lex & yacc*, ed. O'Reilly, 2nd edn. (1992).

[20] S. Mehrotra et al., Ensuring consistency in multidatabases by preserving two-level serializability, TODS 23(2) (1998) 199–230.

[21] S. Mehrotra et al., Overcoming heterogeneity and autonomy in multidatabase systems, Information and Computation 167(2) (2001) 137–172.

[22] C. Mohan, Tutorial: Advanced transaction models – survey and critique, in: *Proceedings of the ACM International Conference on Management of Data (SIGMOD 94)* (May 1994) p. 521.

[23] J.E.B. Moss, Nested transactions and reliable computing, in: *Proceedings of the 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems* (1982).

[24] M.H. Nodine and S.B. Zdonik, Cooperative transaction hierarchies: Transaction support for design applications, VLDB Journal 1(1) (1992) 41–80.

[25] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, v2.2, ed. OMG (1998).

[26] H. Ramampiaro, CAGISTrans: Adaptable Transactional Support for Cooperative Work, Dr.ing thesis no. 2001:94, Norwegian University of Science and Technology (NTNU) (2001).

[27] H. Ramampiaro, M. Divitini and S.A. Petersen, Agent-based groupware: Challenges for cooperative transaction models, in: *Proceedings of the International Process Technology Workshop (IPTW 99)* (September 1999).

[28] H. Ramampiaro and M. Nygård, Cooperative database system: A constructive review of cooperative transaction models, in: *Proceedings of the 1999 International Symposium on Database Application in Non-Traditional Environment (DANTE 99)*, November 1999 (IEEE-CS Press, Kyoto, 1999) pp. 315–324.

[29] H. Ramampiaro and M. Nygård, CAGISTrans: A transactional framework for cooperative work, in: *Proceedings of the 14th International Conference on Parallel and Distributed Computing Systems (PDCS2001)* (August 2001) pp. 43–50.

[30] H. Ramampiaro and M. Nygård, Supporting customisable transactions for cooperative work: An experience paper, in: *Proceedings of the 2002 Western Multi Conference (WMC 02) – Collaborative Technologies Symposium 2002 (CTS 02)*, San Antonio, USA (January 2002).

[31] K. Ramamritham and P.K. Chrysanthis, *Advances in Concurrency Control and Transaction Processing: Executive Briefing* (IEEE-CS Press, 1997).

[32] K. Schmidt and T. Rodden, Putting it all together: Requirements for a CSCW platform, in: *The Design of Computer Supported Cooperative Work and Groupware Systems*, eds. D. Shapiro, M. Tauber and R. Traunmüller (Elsevier, Amsterdam, 1996) ch. 11, pp. 157–175.

[33] R. Selvåg, Web-transactions, Master's thesis Norwegian University of Science and Technology (2000).

[34] A. Skarra, Concurrency control for cooperating transactions in an object-oriented database, SIGPLAN Notices 24(4) (February 1989) 466–473.

[35] J. Wäsch, Transactional Support for Cooperative Applications, Ph.D. thesis, GMD/IPSI and Dramstadt University of Technology (1999).

[36] W.E. Weihl, Commutativity-based concurrency control for abstract data types, IEEE Transactions on Computers 37(12) (1988) 205–214.

[37] G. Weikum and H.-J. Schek, Concepts and applications of multilevel transactions and open nested transaction, in: *Database Transaction Models for Advanced Applications*, ed. A.K. Elmagarmid (Morgan Kaufmann, San Mateo, CA, 1992) pp. 350–397.