CAGISTrans: A Transactional Framework for Cooperative Work

Heri Ramampiaro and Mads Nygård
Dept. of Computer and Information Science
Norwegian University of Science and Technology (NTNU)
N-7491 Trondheim, Norway
Email:{heri,mads}@idi.ntnu.no

Abstract

The problem addressed in this paper deals with data sharing as a means to cooperate. Our main focus is on transactional mechanisms to manage concurrent accesses to (partially) shared data and resource bases. Several transaction models have been developed for this purpose. However, they may still be unsatisfactory with respect to support for runtime changes and heterogeneity.

The paper outlines our contributions to solve these problems, based on a transactional framework – called CAGISTrans – for customising transaction models to application needs. As part of the solution, we propose a new workspace approach to support data exchange. Further, the paper investigates necessary elements of an application specific transaction model, and discusses how they can be organised to cope with dynamic environments. Heterogeneity, on the other hand, is addressed through the use of a middleware-based architecture, focusing on portability and interoperability. Finally, the paper discusses the use of XML as a transaction specification language. This includes the advantages that can be gained and the challenges that must be considered.

Keywords: Transactions and Cooperative work

1. Introduction

Advances in computing and network technology are changing the way people carry out their work. Increasingly, work is performed in teams distributed over the network, where people get together to have the work done without strict organisational and power structures. As a result, the induced work environments are dynamic; the members and the structure of the teams may vary from time to time, and the imposed requirements may be in continuous change. This makes it important and challenging to provide suitable tools to facilitate this type of cooperative activities.

The field of CSCW – Computer Supported Cooperative Work – has emerged due to the need to provide appropriate computer-based tools to support group work [1]. The support needed spans from informal interaction among co-workers – e.g., meeting and conferencing systems – to sharing and exchange of information. Our main focus is on data sharing and information exchange, common within product design – e.g., Software Engineering, CAD/CAM, among others – and product manufacturing.

Support for sharing through computers may be achieved by allowing concurrent access to shared resources, such as databases, web-servers, etc. But, though sharing is important, considering product design and manufacturing, the avoidance of inconsistency is crucial. This calls for the provision of mechanisms to synchronise, coordinate and manage the aforementioned resource access.

Transactions and transaction models, originating from the database community, have been widely used in managing concurrent accesses to shared data. However, the main problem with traditional transactions is their strictness, especially with respect to support for long-running activities and cooperation due to their atomicity and isolation requirements [2]. To overcome these limitations, numerous advanced transaction models have been suggested [2,3,4]. The emphasis has generally been on the relaxation of atomicity and isolation. Still, in term of cooperative work, it is widely agreed that the goal has not yet been attained. For instance, many of the models were suggested with specific applications in mind, having fixed semantics and fixed correctness criteria. Thus, they may fail in providing sufficient support for wide areas of applications.

A possible solution is to provide a framework for specifying and tailoring the transaction models to the application needs. This idea is itself not new. Most prominent are ASSET [5], TSME [6], and RTF [7], among others. However, although these provide the ability to specify and implement extended transaction models suitable for specific applications, we believe there are problems that still remain unsolved. First, the support for dynamic environments – e.g., process shift – is needed but still not fully supported. One of the main reasons is that the aforementioned specification must be done before the execution of the actual transactions. Hence, they may offer little or no support for adjustment during runtime. Second, cooperative work is diverse in nature [1], making the support for heterogeneous environments highly relevant. However, existing solutions are mainly built on DBMSs – database management systems. This means that other resource management systems than databases, such as web-servers, etc., may not be well supported. For this reason, the heterogeneity aspect may not be adequately addressed.

The work presented in this paper was motivated by a need to cope with these limitations. Towards this end, the rest of the paper focuses on the investigation of ways to support cooperation that is dynamic and heterogeneous in nature. Our main contribution is the development of a transactional framework called CAGISTrans, that not only allows the definition of application tailored transaction models, but is also aimed at supporting specification refinement at runtime. The name CAGISTrans denotes transactional support for cooperating agents in a global

1 This work has been supported by the Norwegian Research Council through grant 112567/43 via the CAGIS-project.
information space – i.e., CAGIS for short.

2. Bridging the gap between flexibility and strictness using workspaces

A widely accepted solution to support cooperation is to provide workspaces – i.e., virtual spaces or places where groups of distributed people can cooperate in solving a problem or performing a task. Usually, one distinguishes between private and shared workspaces. This makes it possible to alternately perform work in cooperation, and carry out individual activities in private. Such a separation has been considered important since it is neither desirable nor practical to share all data all the time, making control and management of workspace accesses necessary.

However, in many of existing systems, the main problem is that either the control and management are left to the users themselves to figure out – cf., groupware-based approaches, or they are unduly strict – cf., database-centred approaches. To bridge this gap, there is a need to organise the workspaces in such a way that different levels of sharing can be enabled. This can be achieved by providing workspaces with nested structure. Moreover, operations allowing accesses to workspaces at different nesting levels are needed. To cope with this, we extend the traditional check-in/check-out models – e.g. [8] – with advanced check-in and check-out operations. Finally, there is a need for mechanisms to synchronise and coordinate the involved accesses. The main benefit with these extensions is the ability to control the accesses to workspaces, while at the same time increasing the possibility of sharing, compared to traditional check-in/check-out models.

2.1. Workspace organisation

Consider a software development process involving several engineers. Because of the size and complexity of the software artifact to be developed, the engineers are organised in small groups developing different but connected modules. To allow flexible cooperation, engineers belonging to the same group have to be provided with a shared workspace where they can exchange their document – i.e., program codes, etc. – specific to their module. However, until reaching some stage of their coding process, each engineer needs to work in private. This is true – e.g., when the code being written is still so incomplete that sharing would not be reasonable. To cope with this, each engineer must have a private workspace. Further, when a module reaches some maturation phase, members of other related groups may want to have accesses to what has been achieved so far. Therefore, the module and related data can be made available by copying them to a public workspace. This may, in addition, be necessary when the modules are to be integrated into a single software artifact.

As can be inferred from this simple illustration, at least two levels of data sharing may be needed; sharing among engineers within the same group, and sharing among and across different groups. As a result, distinguishing between private and public workspaces alone would not be sufficient. In addition, we may need group workspaces. This means that workspaces must be organised, as illustrated in Figure 1, to form a nested structure, with generic nesting levels. The idea is to compromise on isolation among workspaces, but at the same time limit or avoid unintended and unauthorised manipulation of data at a specific workspace.

Another aspect is management of objects. As a supplement to versions, we may apply different levels of object consistency through object states. The idea is that each object is associated with a state depending on the nesting level of the workspace where it resides. We distinguish between intra-workspace consistency, intra-group consistency and inter-group consistency. In other words, objects in private workspaces are only intra-workspace consistent. When they are released or copied to a parent – i.e., group – workspace, their consistency level is upgraded to intra-group consistency. Since such objects may still be inconsistent regarding the final results – e.g., the final software artifact, sharing is restricted to members of the group that owns that workspace, only. And, when the objects are checked in to the public workspace, they are inter-group consistent.

2.2. Extended workspace access operations

Extension of the traditional check-in/check-out model is required as a main consequence of the workspace and data organisation. This extension serves as a means to manage the relevant object states. Moreover, additional operations may be used to distinguish between pre-release and final release of objects. CAGISTrans provides the following operations to extend the check-in and check-out operations:

- **write-check-out** and **read-check-out**: distinguish the intention of the check out operation – i.e., read and write intentions – to facilitate control of sharing and appropriate object states.
- **upward-check-in**: pre-release of data to a parent group workspace – i.e., check-in up one level only.
- **check-in**: final release of data to the public workspace.
- **refresh**: update a local copy of data with the one residing in the parent workspace – e.g., check-out from a group workspace to a private workspace.
- **Workspace data operations** such as read, write, update – i.e., read and then write, and some advanced operations (see Section 4), which are needed to manipulate data at a
specific workspace, including those needed for consistency level upgrade, after an upward-check-in.

2.3. Workspace access coordination

Correctness of each object state can be managed by synchronising and coordinating the accesses to workspaces. To facilitate this coordination, workspace operations can be executed as part of a transaction. Hence, consistency of objects that are shared among several users can be controlled by indicating necessary operations to be carried out, and enforcing the order in which they are performed. This implies that if transaction executions obey such criteria, we may assume that their results are correct. How this is done is one of the main topics of the rest of the paper.

3. Distinguishing between characteristics and execution specifications

Cooperative work is diverse and unpredictable in nature. A provision of dynamic support is, for this reason, crucial. To meet this requirement, the model needs to be modifiable during runtime. We argue that this may be achieved only by allowing both design time and runtime specifications. This means that we have to organise the specification of the actual transaction model into two separate but connected parts, consisting of one that can be designed before the transaction is executed, and another that can be modified while the involved transactions are being executed. We call the former characteristics specification and the latter execution specification (see Figure 2).

![Figure 2 Illustration of the distinction between characteristics specification and execution specification](image)

The idea is to collect elements of a transaction model that are fixed and possible to predict in advance into the characteristics part, and include the elements that are only partly predictable into the execution specification. In this sense, the transaction characteristics consist of elements that are vital and must be known prior to the execution of transactions. First, they specify the main properties of the involved transactions, i.e., the ACID properties [9]. Second, they determine suitable structures of the transactions, and define how they affect each other’s processing, i.e., the relationship among the involved transactions. Third, the elements define appropriate correctness criteria to be applied. That is, they specify whether one should rely on an underlying DBMS and/or enable user-defined criteria. Finally, the characteristics elements determine the mechanisms and policies to be utilised to satisfy the designated correctness criteria. This means; they define mechanisms to be made available, and the rules for how and when to use them to achieve the required correctness.

The execution specification is a supplement to the characteristics specification in that it consists of elements that must be present to achieve the desired characteristics during runtime. In this sense, it defines the behaviour of transactions in terms of; the operations needed to execute, manage and control transaction executions, and execution constraints. A more detailed description of these elements is given in Section 4.

3.1. Discussion and analysis

This discussion will be centred on the elements of the characteristics specification, showing how we derive the rest of the elements of the specification based on the transaction properties.

Now, why do we have to explicitly customise the ACID requirements? The ACID requirements have been used extensively within the database community as the ultimate criteria for achieving correctness within transaction processing. However, as we initially pointed out, their undue strictness has made them inappropriate for advanced applications, such as cooperative applications. Some of the ACID requirements have to be compromised to provide the required support. Which of these requirements should be relaxed depends on the actual nature of applications. Therefore, customisation is necessary.

The main advantage of customised ACID requirements is that this allows users to fit the requirements to the characteristics of the application they are involved in. For example, some situations within advanced applications may still require atomicity to guarantee the correctness of data in the presence of crashes. Other may see this as a burden since it might require people to unnecessarily repeat work that has already been considered finished. Similar arguments apply for the remaining properties.

Next, what properties would be compromisable? "Porting" of ACIDity from traditional database applications to advanced applications normally imply compromising the atomicity and isolation properties only. Preservation of both consistency and durability seems to be accepted [2,3].

3.1.1. Atomicity

Starting with atomicity, there should be two options; full atomicity and relaxed atomicity. Going for full atomicity should be possible since it may still be vital to preserve the all-or-nothing property of transactions. This may, for instance, be relevant in activities that do not involve user interaction and are short. The goal is to keep atomicity whenever it is appropriate. Relaxation of atomicity, on the other hand, may be required for several reasons. First, the presence of failures in long running activities may be expensive since lots of work could be thrown away. Therefore, it is necessary to compromise on the all-or-nothing low. And, relaxed atomicity should serve as a means to allow finer grained abort management in the form of partial abort. Further, it is common in cooperative environments that users choose the actions within their activities as they go along - i.e., interactive transactions. In such a case, requiring transactions to perform automatic abort is inconvenient, thus calling for user controlled rollback.

To achieve relaxed atomicity by means of partial rollback, a user or a transaction model designer needs to specify the desired structure of the transactions to be run - i.e., choosing between flat and nested, and the intended depend-
encies among transactions, explicitly. First, relaxed atomicity is most relevant for transactions having a nested structure. Therefore, to allow partial abortion, the choice would fall on nested structure. Second, the effect of the abortion of one transaction on other transactions needs to be specified. This means that a user or a transaction model designer needs to define the abort dependencies among the involved transactions. These will explicitly determine the transactions that have to abort, regardless of their read-write dependencies. Hence, partial rollback is accomplished by first checking the abort dependency specifications, and then aborting only (sub-)transactions that are included in these specifications. Table 1 summarises the relevant elements impacted by the atomicity property.

![Table 1: Atomicity and relevant elements (Boldface means the dependencies are mandatory)](image)

In sight of the distinction between characteristics specification and execution specification, atomicity mainly affects the way transactions achieve their specified structure. Recall that transactions with relaxed atomicity are nested. This means that some of these transactions likely need to restructure – i.e., spawn and delegate some responsibilities – during runtime. Therefore, to allow this, a primitive for transaction restructuring is required as part of the transaction execution building blocks.

### 3.1.2. Consistency

Consistency needs to be fully preserved to ensure correctness of the final result of a transaction execution. Though some kinds of inconsistency may have to be tolerated by transactions executed in cooperative environments, who is interested in introducing inconsistencies in a database or other involved resource base? Further, this is one of the main reasons research on transactions is relevant at all, also within the context of cooperative work. In our opinion, the concept of transactions might be meaningless if consistency of the final results cannot be achieved.

![Table 2: Consistency and relevant elements](image)

However, there is a challenge that needs to be considered. The relaxation of the atomicity property implies that a transaction may commit parts of its total processing. Thus, how to define correctness of such a transaction? As an answer, specific correctness criteria should be prerequisites (see Table 2). A minimal set of criteria must be defined before a transaction is executed. This means that either a user defines an appropriate criterion or a possible underlying DBMS provides one – e.g., serialisability. This means that the execution specification must contain a block that allows the definition of such constrains.

### 3.1.3. Isolation

As for the atomicity properties, there should be two isolation options: full isolation and relaxed isolation. Full isolation is encouraged when sharing is not a prerequisite. As an example, in the cooperative programming environment in Section 2, one would need to work in private until a source code has reached some maturation phase – i.e., first release, and so on. However, in the context of cooperative work, full isolation would more often be a burden. Moreover, transactions within cooperative environments normally have long duration. Therefore, requiring other transactions to wait until specific transactions commit would be unacceptable.

![Table 3: Isolation and relevant elements](image)

Table 3 summarises the relevant elements impacted by the isolation property. To achieve relaxed isolation there is a need to specify and apply correctness criteria that are more relaxed than serialisability, but still constraining the transaction execution to achieve consistency. Further, to allow sharing among transactions, a sharing primitive explicitly specifying the interleaving that is allowed must be provided. And, to achieve the criterion specified, policies defining mechanisms and the corresponding rules for their usage should be provided. For instance, diverse locking types (isolation) can be used combined with awareness primitives – i.e., notification. Moreover, to cope with problems that possible cascading aborts may cause – e.g., aborting all transactions that share the same data, relaxed isolation (further) motivates the need to explicitly define the abort dependencies upon the involved transactions. In other words, if a transaction fails, not all cooperating transactions have to be affected. Rather, only those covered by such abort dependencies are required to rollback.

Regarding the distinction suggested in this paper, there is an implication similar to that for the atomicity property. As already mentioned, relaxed isolation requires that a user need to specify correctness criteria. This means that specification of how transactions view and affect other’s processing must be specified. More details on how this is achieved are discussed in Section 4.

### 3.1.4. Durability

As for the consistency requirements, the final results of
transactions must be permanent. Provision of persistence for committed data is one of the key philosophies underlying DBMSs. The main idea is that once the results are committed, it should survive crashes and other system failures.

Table 4 summarises the relevant elements concerned with ensuring durability. An important question is when to make data permanent in long running activities that may involve cooperation? It is reasonable to assume that long-running transactions are split into several sub-transactions, resulting in a nested structure. Consider that each sub-transaction wants to commit its results. These results can be made permanent by saving them on the underlying repository. But, how such commits affect the final results depends on the commit dependency specifications for the involved transactions. Further, due to relaxed isolation, sharing may occur. To handle abort propagation in such cases, compensating transactions [10] must be provided to explicitly discard the effect of committed results. Hence, abort dependencies among transactions must also be defined. Of course, to be able to fulfil the durability requirement, all transaction operations must be logged. This means that either a possible underlying DBMSs does the logging or a supported mechanism describing the transaction execution is provided.

<table>
<thead>
<tr>
<th>Functional relationship</th>
<th>Full Durability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactional relationship</td>
<td>Flat or nested structure</td>
</tr>
<tr>
<td>Inter- and intra-transaction dependencies:</td>
<td>begin, commit, and abort</td>
</tr>
<tr>
<td>Correctness criteria</td>
<td>Depending on isolation, see Table 3</td>
</tr>
<tr>
<td>Applied policies</td>
<td>Applied on correctness criteria</td>
</tr>
</tbody>
</table>

Table 4 Durability and relevant elements (Boldface means the dependencies are mandatory)

4. Dynamic re-specification of transactional behaviour

A widely accepted solution to overcome the diverse and dynamic nature of cooperative applications is to have transaction models that not only can be adapted to different needs, but also can be refined to meet new requirements while transactions are being executed.

The advantages of the ability to modify and adjust the specification of transactional behaviour during runtime are evident. First, we do not need to know in advance the complete set of actions to be carried out, as extensions and adjustments can be accomplished while the involved transactions are being scheduled or executed. Moreover, as users can do part of the specification themselves – as their activities go along, they have better control on the execution of their (trans)actions, thus making them able to choose appropriate actions based on the current needs.

To achieve all this, we need the distinction in Figure 2. In addition, there is a need to separate the execution specification into two main parts: fixed execution specification and modifiable execution specification, as depicted in Figure 3. In this way, predictable execution primitives can be collected in a fixed part, whereas those that may have to be modified during runtime are in the modifiable part.

4.1. Discussion and analysis

The building blocks of the execution specification that affect the transactional behaviour are: 1) transaction constraints, which are used to control and manage the effects of the execution of transactions on other transactions; 2) management operations, which are used to control and manage the initiation, termination, and dynamic restructuring of transactions; 3) advanced operations, which are used to specify operations at higher abstraction levels than read and write, and 4) a transaction execution descriptor, which contains all information on the execution of transactions as well as information related to model specification.

4.1.1. Effect management using transaction constraints

The effect of a transaction on other transactions depends on the interaction among the involved transactions. It can be managed by defining constraints that explicitly determine the interleavings that are a) forbidden – i.e., conflicts, b) allowed – i.e., permits, and c) mandatory – i.e., demands. They constitute the basic units of user managed correctness criteria. Table 5 gives an overview of such constraints. Their relationship to existing approaches is explained in Section 6.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Relevant when</th>
<th>Not Needed when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflicts</td>
<td>To hinder interference</td>
<td>Possible to predict a set of actions</td>
</tr>
<tr>
<td>Permits</td>
<td>To allow flexible sharing</td>
<td>Isolation is relaxed, and conflicts are defined or locking is used</td>
</tr>
<tr>
<td>Demands</td>
<td>To ensure correct execution</td>
<td>Always relevant when isolation is relaxed</td>
</tr>
</tbody>
</table>

Table 5 Overview of effect management

Conflicts – as in [11] – are constraints identifying operations that are not allowed to execute concurrently. Their main purpose is to hinder interference by forbidding specific interleavings to occur – i.e., hindering operations that do not obey specific conflicts rules from executing. In this sense, conflicts can be seen as an analogy to locks.

So, when are conflicts relevant? Conflicts provide a higher abstraction level than locking. Therefore, whenever such an abstraction level is required, the definition of conflicts is encouraged. This also means that as long as it is possible to predict the set of actions to be carried out within the cooperative activities, conflicts should be used. Moreover, since conflicts may be customised to the application needs, they may generally allow a higher degree of flexibility than standard locking protocols – e.g., 2PL [12].

But, should conflicts be fixed or modifiable? To restrict the level of flexibility to a manageable level, conflicts are by default fixed. This implies that if conflicts are relevant,
they must be specified before runtime. To cope with the limitations caused by the dependency on a complete a priori knowledge of actions, conflicts may be accompanied with a locking protocol allowing users to issue locks on their behalf as their activities are in progress, without following a strict lock acquisition rules. Such type of locks is called user-controlled locks, similar to those utilised in existing cooperative systems – e.g., BSCW\(^1\). Our user-controlled locks differ from existing ones in that instead of totally relying on users in issuing all locks, they serves as a supplement to conflicts, and the freedom may be restricted to a manageable level. Hence, if the actual conflicts are incomplete we may apply user controlled locks. But, if conflicts are too restrictive, permits may be defined.

**Permits** – as in [5] – are constraints defining relationships among operations that are normally conflicting – e.g., according to the conflict rules – but, for a specific set of transactions, are allowed to appear, anyhow. Thus, its main purpose is to allow controlled but flexible sharing.

Initial permits can be defined before runtime to allow a set of transactions to override a possible conflicts definition. This means that based on permits, some executing transactions performing conflicting operations may be granted access to the same data, but others are prohibited. Further, due to the interactive form of transactions, it is not always possible to predict a priori the interaction or interleaveings among transactions that are needed during runtime. Therefore, the ability to modify permits during runtime is vital. This includes addition and removal of permits. New permits may, for instance, be needed when new transactions are initiated. This is normally ok, since it will only affect future execution of actions. Conversely, a situation in the cooperative process may cause the permission to share being irrelevant or undesirable. Therefore, the ability to temporarily or permanently remove some specified permits must also be considered. However, removal of permits may implies that interleavings that already have appeared become illegal according to the current conflict rules. This, in turn, may result in invalidation of already performed actions, thus wasting a lot of work. So, before a user remove a specific permits relationship, he/she has to consider the cost of invalidation of actions.

**Demands** are constraints specifying sequences of operations or steps that must appear for the actual transactions to produce a correct execution. Considering the required dynamic behaviour of transactions, specification of demands should be modifiable. Normally, only part of such a specification can be done before runtime for the same reasons as for permits. In this sense, modification of demands includes removals and additions of constraints. Removal of demands means that required actions are no longer needed. This is normally ok since it will not affect already performed or committed actions.

Addition means that new actions are inserted to the demands specifications. This may be needed when new transactions are initiated, or when new steps are required. Unlike removal, inserting new constraints may impose some difficulties. For instance, new actions may appear between two already executed steps – i.e., the addition comes too late. For this reason, addition of new constraints is restricted to involve actions that have not been executed yet – checked using the execution descriptor. But, if a user anyway choose to add new steps between already executed actions, he/she must be aware that such actions may have to be aborted, and re-executed to reflect the new constraints.

### 4.1.2. Management of transaction execution

Management operations consist of primitives that are used to manage and control the initiation, termination and dynamic restructuring of transactions. Initiation of transactions is done by issuing begin. To terminate initiated transactions, either commit or abort is issued. Due to the required user control, transaction terminations are usually interactive. But, the effects of a termination on other transactions depend on specified dependencies – i.e., commit and abort dependencies. If a user terminates a transaction he/she will also be told which other associated transactions will be terminated.

The concept of dynamic restructuring of transactions has been pointed out as an important primitive to allow cooperation among transactions [13]. The concept can be realised in several different ways, with possibly different purposes (see Section 6). We stress the use of dynamic restructuring of transactions to realise a specified transaction structure – e.g., spawning, and to allow transactions to delegate responsibilities to other transactions during runtime. To this extent, the relevance of dynamic restructuring depends on the defined structure for the actual transactions. In other words, full restructuring can only be achieved if the involved transactions have a nested structure, as defined by the characteristics specification. This means that if a transaction is flat then only operation transfer is allowed, since spawning is irrelevant. If the transaction, on the other hand, is nested then dynamic restructuring may be accomplished in two ways. Operations can be transferred to existing sub-transactions. Or, if a target doesn’t exist, creation of a new subtransaction precedes the intended operation transfer.

### 4.1.3. Support for advanced operations

The aim of advanced operations is to allow specification of operations at an abstraction level higher than, but based on, read and write. Such an approach is useful because of its ability to exploit the operation semantics to increase concurrency [14]. Moreover, abstraction beyond read and write is required for advanced applications, such as software engineering environments, and the like. Modelling software engineering activities – e.g., compile, edit, and so on – with read and write is generally a complex task. Using advanced operations, we may increase the modularity, thereby simplifying the modelling task.

Thus, one of the main goals of advanced operations is to further improve the usability of transactions within advanced applications. However, there are issues that need further consideration. An important one is whether the definition of advanced operations during runtime should be allowed or not. As for transaction constraints, it is desirable

---

\(^1\)See http://bscw.gmd.de.
to allow runtime definition. Recall that a complete set of tasks is not always possible to predict in advance. New operations may be required as the actual activities go along. The complexity of introducing new advanced operations during runtime may, however, impose some difficulties. First, introduction of new operations may require the definition of new conflicts. However, referring to Section 4.1.1, conflicts cannot be modified. Therefore, runtime definition of new operations is only allowed if it does not require new conflicts. Further, since no new conflicts exist, new permits are only necessary if user-controlled locks are used. Finally, introduction of new demands may be required as new steps will be executed. The restrictions that this may imply are that new operations added to the actual demands should not appear between already executed steps (see also Section 4.1.1).

Summarising, most of the advanced operations must be specified before the execution of transactions, but new operations may be introduced during runtime given that they do not need to modify the actual conflicts.

5. Supporting heterogeneity

A main requirement for cooperative work is support for heterogeneity. To our knowledge, only few existing frameworks for tailoring transaction models to specific applications allow resource managers other than DBMSs. This may be a shortcoming since users in the cooperative environment may want to access information or data from several places not necessarily residing in a database. As an example, data needed during a collaboration process may be on a web-server, regular file systems, as well as heterogeneous database systems, among others. For this reason, openness is crucial.

![Figure 4 CAGISTrans high-level architecture](image)

To cope with both openness and other reasons, we have chosen to implement our system supporting the CAGISTrans framework as a middleware to bridge user-applications and existing resource management systems (see Figure 4). Therefore, we do not have to build the system from scratch as with e.g., TSME.

Because of this, CAGISTrans inherits several advantages from the well-known middleware approach. First, increased support for distribution is achieved by means of web-support. CAGISTrans allows information and data that are administrated by the transactional framework to be accessible from the Internet. Thus, people can work together independent of their geographical location. Second, increased resource availability is gained by allowing resources administrated by the CAGISTrans system to reside on repository types other than databases. This makes it possible to access richer information and data types than those supported by database systems. Such data may be of interest or even critical for the accomplishment of the situated cooperative process. Third, database independence is attained since CAGISTrans is developed as a middleware rather than a complete transaction management system – e.g., an advanced TP-monitor. This is based on the observation that the transaction specification and execution can be accomplished independent of the underlying repository support. This also enables us to port the transaction models to different type of systems, thus extending the usability of transaction models to wider application areas. To further address such portability as well as our dynamic support we have developed a transaction specification language based on XML – the eXtensible Markup Language\(^1\).

The main open question concerning this approach is the performance issue. As can be inferred from the discussion in Section 4, execution of such transactions may introduce some overhead. This could be a bottleneck regarding transaction execution speed. Nevertheless, in the context of cooperative work, transactions are generally long-lived. Therefore, extra seconds needed for management and control purposes would, in the global picture, be insignificant.

6. Relation to other work

The following presents and discusses relevant features of existing models and frameworks with those of CAGISTrans.

Starting with transaction oriented workspaces, the concept has been used extensively in COO [15] and TransCoop [16], among others. COO uses temporary, shared sub-data bases (scratch-pads) for data exchange and integration work. Whereas, TransCoop focuses on exchange of operations instead of exchanging data between private and public workspaces, and correctness control is handled through history validation and merging mechanisms. Our workspace model differs from these in the use of a nested workspace structure that applies generic nesting levels – to regulate the degree of sharing. To the authors’ knowledge, existing approach are restricted to two levels – i.e., private and public, and three levels – i.e., public, semi-public, and private. In addition, our approach utilizes user defined constraints – cf., conflicts, permits, and demands – to allow controlled sharing and coordinated workspace accesses, as well as to ensure correctness.

The inter-transaction dependency concept was originally proposed in the ACTA [17] framework for reasoning about and synthesising (new) transaction models. Intra-transaction dependencies were used in the TSME [6] framework. CAGISTrans adopts both concepts, but our main focus is on more controlled and finer abort management scheme.

User-defined correctness criteria in e.g., the Cooperative Transaction Hierarchy are based on patterns and conflicts [11], where they were represented as state-machines. The concepts of demands and conflicts in CAGISTrans are

\(^1\)See http://www.xml.org.
built on these concepts. However, unlike patterns, demands are represented as directed graphs. Further, while a complex set of patterns has to be defined in advance, without any possibility for redefinition during runtime, demands can be modified while the actual transactions are being executed. Moreover, our concept of conflicts is defined as relationships rather than with a state machine. In this way, when transactions are to be validated, instead of checking the state machine, which is usually complex, CAGISTrans utilises more simple table inquiries. Finally, permits were originally proposed in ASSET [5]. Their use have been further extended in CAGISTrans to provide more controlled sharing by accompanying it with conflicts and demands.

Dynamic restructuring was originally proposed and implemented in the Split and Join Transaction model [13]. Dynamic restructuring in CAGISTrans is similar to this approach. The differences lie in the way how the restructuring is performed and the constraints that are applied on executing transactions. CAGISTrans realises dynamic restructuring by combining the notion of delegation and transaction spawning. And, while split and join model applies serialisability as a correctness criterion, CAGISTrans allows application customised criteria.

7. Concluding remarks

This paper has presented a new framework for customising and tailoring transaction models to application needs. The main contributions of this work are the identification of the useful features of existing models and the way we compose, organise and realise these to suit both the dynamic and heterogeneous nature of cooperative environments.

We have built our framework on existing transaction models and transactional frameworks by extracting beneficial features from these. Using this as a starting point, we have attempted to further extend the models and frameworks to address the remaining problems within transactional support for cooperative work. To address the issue of heterogeneity we have chosen to adopt a middleware-based approach rather than building a stand-alone system from scratch. Such an approach has allowed the use of the CAGISTrans framework in a wide range of applications, where openness and portability are crucial.

Major parts of the CAGISTrans framework has been implemented as proof of concept prototypes [18,19,20]. These have been tested regarding usability and practicality. The experiences from this are that the framework is capable of supporting the basic aspects of dynamic transactions, allowing users to specify a model and execute their transactions in a flexible but controlled manner. Typical applications of our framework range from software engineering to conference management applications. Nevertheless, there is a need to further improve the user support in terms of ease of use, which will be part of our future research.

Acknowledgements: We would like to thank Rune Selvåg, Mufrid Krilic, and Lars Killingdal for their contributions in the implementation of the CAGISTrans prototypes. We would also like to thank our colleagues Roger Midstraum and Lasse Natvig for their comments at an early stage of this work.

References