Specifying Temporal Behaviour in Software Architectures for Groupware Systems

Timothy N. Wright[1], T.C. Nicholas Graham[2], and Tore Urnes[3]

[1] University of Canterbury, Private Bag 4800, Christchurch, New Zealand
tnw13@cosc.canterbury.ac.nz
[2] Queen's University, Kingston, Ontario, Canada K7L 3N6
graham@cs.queensu.ca
[3] Telenor Research and Development, P.O. Box 83, N-2007 Kjeller, Norway
tore.urnes@telenor.com

Abstract. This paper presents an example of how software architectures can encode temporal properties as well as the traditional structural ones. In the context of expressing concurrency control in groupware systems, the paper shows how a specification of temporal properties of the semi-replicated groupware architecture can be refined to three different implementations, each with different performance tradeoffs. This refinement approach helps in understanding the temporal properties of groupware applications, and increases confidence in the correctness of their implementation.

1 Introduction

Software architectures traditionally decompose systems into *components* responsible for implementing part of the system, and *connectors* enabling communication between these components. Components implement some part of the system's functionality, while connectors specify the form of intercomponent communication, for example, through method calls or events [28]. We refer to these as structural properties of the architecture.

In synchronous groupware applications, it is not only important to capture *how* components may communicate, but *when*. For example, in a multiuser video annotation system, it is important that all participants see and annotate the same frame [14]. In a shared drawing application, it is important that the drawing operations of participants do not conflict, for example with one person deleting a drawing object that another is moving. As the paper will show, such requirements on sequencing of updates and synchronization of shared state can be expressed as restrictions on when messages can be passed between components involved in an interaction.

This paper investigates how software architectures can specify temporal properties of an application as well as structural ones. From these temporal specifications, a variety of implementations can be derived, embodying different execution properties. This allows an approach where software architectures specify high level temporal properties of implementations, allowing architecture implementers to plug-replace any implementation meeting these properties.

P. Palanque and F. Paternò (Eds.): DSV-IS 2000, LNCS 1946, pp. 1−17, 2001.

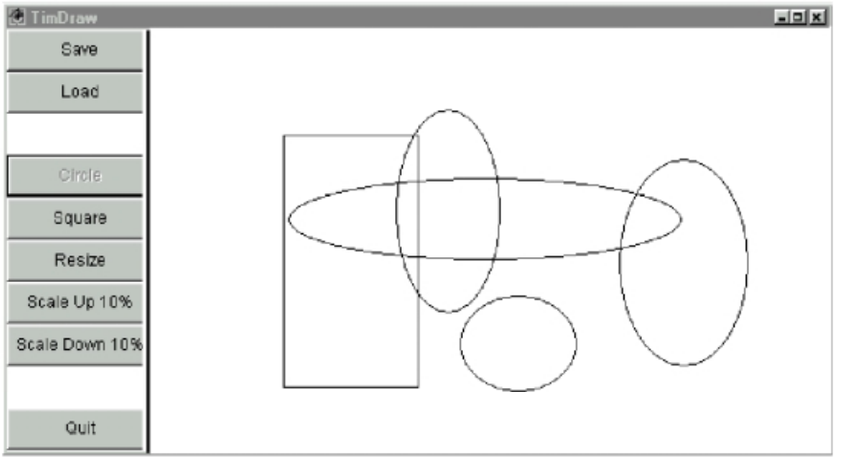

Fig. 1. A Groupware Drawing Program. This program was implemented in Java using the *TeleComputing Developer Toolkit* (TCD) [1].

As will be shown in the paper, the benefits of this approach are:

- Difficult temporal properties of groupware applications can be treated orthogonally to the application's functionality by embedding these properties in the software architecture;
- Premature commitment to algorithms implementing temporal properties can be avoided, as early design of the system focuses on desired behaviour rather than on algorithms implementing that behaviour;
- The process of specifying properties and refining implementations increases confidence in the correctness of the implementations and provides a clearer understanding of the temporal properties of the application.

In order to demonstrate this approach, we take the example of the implementation of concurrency control in a semi-replicated groupware architecture. We show how concurrency control properties can be encoded in the definition of the semi-replicated architecture itself. Specifically, we treat the problem of ensuring that transactions performed on shared data state are serializable, guaranteeing that operations performed by users do not conflict.

As we shall see in the paper, concurrency control algorithms are complex, and embody trade-offs of degree of consistency versus response time. It is therefore beneficial to separate the specification of the desired concurrency properties of an application from the concurrency control algorithm actually implementing it. To demonstrate this assertion, the paper is organized as follows. Section 2 describes the concurrency control problem in groupware, and introduces a simple groupware drawing tool as an example application. Section 3 introduces the widely used semi-replicated implementation architecture for groupware, and shows how it can be described to possess temporal properties ensuring correct concurrent behaviour. In order to show the flexibility of such a specification, sections 4 through 6 introduce the locking, Eager and adaptive concurrency control algorithms as implementations

refined from the semi-replicated architecture. These algorithms have all been implemented as part of the *TeleComputing Developer* (TCD) groupware development toolkit [1].

2 Motivation

To introduce the concurrency control problem and to motivate our approach of encoding temporal properties of applications in the software architecture, we present a simple groupware drawing program. As shown in figure 1, users may draw simple objects such as squares and circles on a shared canvas. Each user's actions are reflected in the canvases of other users in real time. In addition to standard editing operations, users may scale the entire diagram up or down, in increments of 10%.
In the implementation of the drawing program, a shared data structure (or *shared context*) contains the set of drawing objects. Figure 2 shows how operations for resizing and scaling objects are implemented. For example, a resize operation reads the object to be resized from the shared context, changes its size, and saves the object back to the shared context. Similarly, the scale operation scales each of the drawing objects in the shared context.

Figure 2 shows how concurrency problems can arise if two users simultaneously perform a resize and a scale operation. Here, the resize operation is performed while the scale is taking place, partially undoing the effect of the scale. This leaves the diagram in an inconsistent state, where the scale has been applied to all elements except the first. When two user actions lead to an inconsistent result, those actions are said to *conflict*. Concurrency control algorithms are designed to prevent the negative effects of conflicting actions.

2.1 Concurrency Control Styles

Concurrency control algorithms can be roughly divided into two classes – pessimistic and optimistic. Pessimistic schemes guarantee that when a participant in a groupware session attempts to modify the shared artifact, his/her actions will not conflict with the actions of other participants. This guarantee leads to intuitive user interface behaviour, but at the cost of responsiveness. Optimistic approaches, on the other hand, assume that actions will not conflict, and must detect and repair conflicts when they occur.

Resize object "1" to newSize	Scale entire diagram by k%
	`n=getNumberObjects()`
	`o1=getObjectAt("1")`
`s=getObjectAt("1")`	
`s.setSize(newSize)`	`o1.scale(k)`
	`setObjectAt("1",o1)`
`setObjectAt("1",s)`	`o2=getObjectAt("2")`
	...

Fig. 2. A resize operation conflicting with a scale operation**.**

Under pessimistic algorithms, update transactions resulting from user actions never fail. One way of achieving this property is to require clients to obtain a lock on the shared context before attempting to process a new user action [22]. This locking may reduce the potential for concurrent execution of clients and introduces networking overhead to obtain locks.

Under optimistic algorithms, update transactions may fail, potentially requiring work to be undone [16]. Optimistic algorithms improve performance by allowing client machines to process user actions in parallel.

Neither pessimistic nor optimistic approaches are suitable for every application. While optimistic approaches may provide better response times for short transactions that are inexpensive to undo [3,29], pessimistic algorithms are preferable in the following three cases:

- *Undo unacceptable:* In some applications, it is impossible to roll back user actions that are retroactively found to conflict with other actions. Examples of such actions include deleting a file or sending an email message.
- *Pessimistic faster:* To be effective, optimistic schemes rely on conflicts being rare, and the cost of undoing operations being inexpensive. Consider the scale operation of figure 2. This operation performs one read and write to the shared context for every drawing object. In a complex drawing with potentially tens or hundreds of objects, the scale operation is likely to conflict with an operation performed by some other user.
- *Optimistic unfair:* In a wide area network, some users may suffer longer latencies than others when accessing parts of the shared context. The actions of these users may be more likely to conflict than the actions of users with lower latency. Fairness may require that users with poor network connections use pessimistic concurrency control.

Concurrency control algorithms therefore embody tradeoffs in the desired behaviour of systems, but all provide the basic property of guaranteeing serializability of transactions carried out by participants in the groupware session. That is, the algorithm should never permit operations to conflict as in the example of figure 2. Our approach is therefore to encode this temporal property of transaction

serializability as part of the definition of the software architecture. We then show how these temporal properties can be implemented by both pessimistic and optimistic algorithms, and by a novel algorithm combining the two. This approach allows us to specify the desired temporal behaviour of the architecture (i.e., transaction serializability) separately from the algorithm used, avoiding premature commitment to a particular concurrency control algorithm.

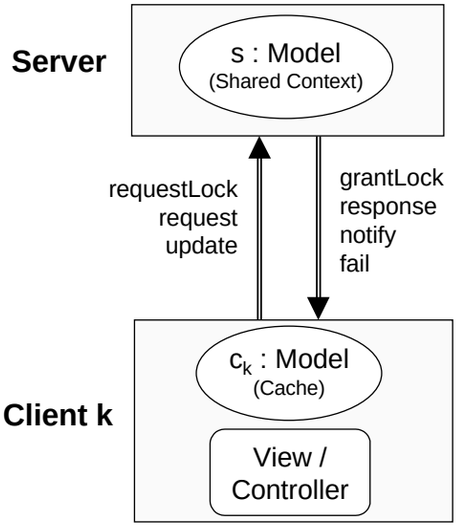

Fig. 3. The semi-replicated implementation architecture for groupware: A shared context is represented on a server machine. Clients contain a cache, a read-only replica of the shared context. Local context does not require concurrency control, and therefore is not represented. Writable replicas of the shared context are assumed to have no concurrency control, and therefore are also not represented.

All of these algorithms have been realized using the Dragonfly [1] implementation of the semi-replicated groupware architecture, in the TCD toolkit. In TCD, we exploit the separation of specification of temporal behaviour from its implementation, allowing concurrency control algorithms to be plug-replaced after the application has been developed.

3 The Semi-Replicated Architecture for Groupware

We model groupware systems using a semi-replicated architecture [15]. Semi-replicated systems are hybrid centralized/replicated systems, where all shared state is represented on a centralized component, some shared state is replicated to the clients, and private state is represented on the clients. Some shared state is replicated in the form of a read-only *client cache*.

Semi-replication is based on the *Model-View-Controller* (MVC) architecture for groupware development [20,15]. In MVC, the shared state underlying each participant's view is located in a *model*, a *controller* is responsible for mapping user actions onto updates to the model, and a *view* is responsible for updating the display

in response to changes in the model. MVC (and related architecture styles such as PAC* [5]) underlies a wide range of groupware development tools. Despite earlier suspicion that semi-replication is inherently inefficient [21], performance evaluation has shown this architecture to provide excellent response times, even over very wide area networks [29].

Figure 3 shows the elements of this model that are necessary to illustrate how concurrency control properties can be encoded within a software architecture connector. The figure further shows the set of messages allowing the client and server components to communicate. These messages are described in detail in section 3.2.

We assume that no concurrency control is applied to private state represented on clients (since there is no concurrent access to this state), and therefore omit local context from the model. We assume that the client cache is not writable by the client, and therefore can only be updated by the server. We further assume that any replicated state that is writable by the client has no concurrency control associated with it, and therefore need not be included in the model. Despite what may appear to be restrictive assumptions, this model describes the implementation architecture of a wide range of existing groupware development tools. (The following discussion is based on Phillips' survey of groupware development tools and their implementation architectures [24]).

Semi-replicated tools directly implementing this model (or subsets of the model) include Clock [29], TCD [1], Weasel [13], Suite [9], and Promondia [12]. GroupKit [25] is described by the model, as GroupKit environments implement shared state, and GroupKit provides no concurrency control for replicated shared data. Figure 3 also describes systems with replicated state under centralized coordination such as Habanero [6], Prospero [10], Ensemble [23] and COAST [27]. In these systems, a central component is responsible for concurrency control decisions, allowing the shared context to be modeled via a virtual server. Finally, the model describes fully centralized systems such as RendezVous [17], as the trivial case in which there is no replicated data at all.

Systems not described by the model include fully replicated systems using concurrency control algorithms based on roll-backs [8] or operation transforms [11]. Such fully replicated systems include DECAF [23], DreamTeam [26], Mushroom [19] and Villa [4].

Therefore, while this simplified treatment of the semi-replicated architecture does not cover every possible implementation of groupware, it describes a sufficiently large subset of current development tools to be interesting.

3.1 Encoding Concurrency Control in the Semi-replicated Architecture

In order to show how software architectures can encode temporal properties, we first formalize our simplified version of the semi-replicated architecture, and then define its concurrency control properties as restrictions over the treatment of messages.

As shown in figure 3, a groupware system consists of a set of client machines, each containing a cache, and a server machine containing shared state. Clients communicate with the server by issuing *requests* for information and *updates* that modify information. Parameters to requests and updates and responses to requests are all considered to be *values*.

Client and Server Components

We let $Client \subset \mathbb{N}$ represent a set of client machines. We define *Update*, *Request* and *Value* to be disjoint sets representing updates and requests made by the view/controller, and values returned by the model as the results of requests. We let $Time == \mathbb{N}$ represent time.

Model

A *Model* stores data. Models are queried via requests. The values of these requests may change over time.

$$Model == Time \times Request \rightarrow Value$$

If *m:Model* we write $m(t)$ to represent $\lambda r \bullet m(t,r)$,the snapshot of the model at time *t*.

As shown in figure 3, we let *s:Model* represent the shared context, and the family of functions c_k *:Model* represent a cache for each client $k \in Client$. When making requests, clients first consult their cache. If the response has not been cached (i.e., the request is not in the domain of the cache), the shared context is consulted. If used efficiently, a cache can considerably reduce the overhead of network communication [15]. We define a request function rq_k for each client $k \in Client$:

$$rq_k : Time \rightarrow Request \rightarrow Value$$
$$rq_k\,(t,r) ==$$
if $r \in \mathrm{dom}\,(c_k\,(t))$ then
$c_k\,(t,r)$
else
$s\,(t,r)$

View/Controller

The purpose of an MVC controller is to map user inputs onto updates to the model. In computing an update, the controller makes a set of requests to the model. We formalize the activity of the controller through an *update function*, which computes an update using values obtained from the model:

$$UpdateFn == \mathrm{seq}\ Value \rightarrow Update$$

An update *transaction* represents the application of an update function to values obtained through a sequence of requests executed at given times. Transactions originate from some client.

$$Transaction == Client \times UpdateFn \times \mathrm{seq}\,(Time \times Request)$$

The view/controller of each client can be thought of as executing a sequence of transactions. When a user performs an action, an update to the shared state is computed, based on values in the cache and shared context. When a client receives notification that the shared context has changed, it computes an update to the display.

Conflicts

The temporal property of interest in this architecture is that transactions do not conflict with each other. Intuitively, two transactions conflict if the modifications to the shared context performed by one transaction cause inconsistencies in the state being used by another transaction. For example, in figure 2, the resize and scale transactions conflict because the resize transaction changes values being used by the scale transaction, with the result that only some of the diagram elements are scaled. More precisely, a transaction conflicts if the update value that would be obtained from executing the transaction at the time the transaction is to be applied differs from the value of the transaction as computed. That is:

$$conflict : Transaction \times Time \rightarrow Bool$$

is defined as:

$$\begin{aligned} &conflict\ ((k, u, <(t_1, r_1), \ldots, (t_n, r_n)>), t) \\ &\quad == u\ (<rq_k\ (t_1, r_1), \ldots, rq_k\ (t_n, r_n)>) \\ &\quad \neq u\ (<s\ (t, r_1), \ldots, s\ (t, r_n)>) \end{aligned}$$

Within the semi-replicated architecture, we define that no conflicting transaction is committed; that is, that no conflicting transaction is permitted to modify the shared context. We encode this simply as the property that if *tr:Transaction* is committed at time *t:Time*, then:

$$\neg conflict\ (tr, t)$$

Implementations of the semi-replicated architecture must therefore ensure that no conflicting transaction is committed. To achieve this, a *pessimistic* concurrency controller ensures that transactions are computed only at times they will not conflict, while an *optimistic* concurrency controller detects when a transaction conflicts, and rolls it back instead of committing it.

It should be noted that this definition of *conflict* ensures that all committed transactions are serializable [22]. For some applications where unintuitive behaviour resulting from conflicts may be tolerable, this definition may in fact be too restrictive [16,22].

3.2 Implementation of Semi-replication

As shown in figure 3, client and server machines communicate via a set of messages. In sections 4 and 5, we describe how these messages are used to implement the concurrency control requirement of this architecture. These implementations all require the architecture's specification, while providing different performance tradeoffs. First, we informally specify the meaning of the messages themselves:

- The network separating the server and clients is assumed to be lossless and fifo. That is, messages are assumed to arrive at their destination, and if two messages are sent to the same destination, they arrive in the order in which they are sent. Clients are assumed to process messages atomically and in sequence.

- If the server receives the message `requestLock`, it eventually replies with the message `grantLock`.
- If the server receives the message *request* (*r*) at time $t \in Time$, where $r \in Request$, then the server responds with `response` (*r*, *s* (*t*,*r*)).
- If the server receives and commits an update `update` (*u*, *tr*), where $u \in Update$ and $tr \in Transaction$, then the shared context is modified. Committing an update is the only action that modifies the shared context; therefore if the server commits no updates between times t_1 and t_2, then $\forall\, t_a, t_b : t_1 .. t_2 \bullet s(t_a) = s(t_b)$.
- If the server receives the update `update` (*u*, *tr*), it may issue the message `fail` to indicate that the update has not been committed, and must be recomputed.
- If the server sends the message `notify` at time t_1 and the client receives the message at time t_2, then the client cache may be updated to values provided by the notification message: $\forall r \in \mathrm{dom}(c_k(t_2)) \bullet c_k(t_2, r) = s(t_1, r)$
- If client *k* receives the message `response` $(r, s(t_1, r))$ at time t_2, where $t_2 \in Time$ and $r \in Request$, then the cache is updated so that $c_k(t_2, r) = s(t_1, r)$
- No other messages modify the client cache. That is, if *k* receives no messages between times t_1 and t_2, then $\forall\, t_a, t_b : t_1 .. t_2 \bullet c_k(t_a) = c_k(t_b)$.

The last section showed how the temporal property that transactions not conflict can be encoded as part of the definition of the semi-replicated architecture. In the following sections, we show how this property can be refined to a wide range of implementations. This allows developers using the semi-replicated architecture to reason about the temporal properties of their application without having to make early commitment to a particular concurrency control algorithm.

4 Locking

One standard approach to implementing pessimistic concurrency control is to require clients to obtain a lock before computing a transaction [16]. We first formally define locking, and then specify how locking is implemented. We then show that the locking algorithm satisfies the temporal properties required by the semi-replicated architecture.

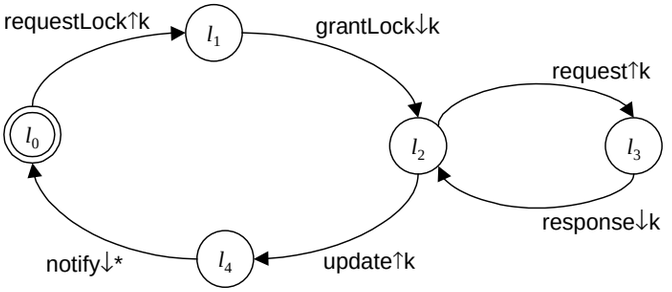

Fig. 4. Implementation of Locking Concurrency Control.

For each client $k \in Client$, we define a *lock* function that specifies whether the client holds a lock between times t_1 and t_2.

$$lock_k : Time \times Time \rightarrow Bool$$

If client k holds a lock over a time interval, the cache is synchronized with the shared context, and the model does not change during the interval:

$$lock_k(t_1, t_2) \Rightarrow (\forall t : t_1 .. t_2 \bullet rq_k(t) = s(t)) \wedge (\forall t_a, t_b : t_1 .. t_2 \bullet s(t_a) = s(t_b))$$

If a lock is held while a transaction is carried out and applied, the transaction will not conflict. Theorem 1 therefore guarantees that locking implements the temporal requirements of the semi-replicated architecture.

Theorem 1. If $k \in Client$, $tr=(k, u, <(t_1, r_1), \ldots, (t_n, r_n)>) \in Transaction$, $t_a=\min(<t_1, \ldots, t_n>)$ and $t_b=\max(<t_1, \ldots, t_n>)$, then

$$lock_k(t_a, t_b) \Rightarrow \neg conflict(tr, t_b)$$

Proof. Follows directly from the definitions of *lock* and *conflict*.

4.1 Implementation of Locking

Figure 4 shows the implementation of locking from the point of view of a server. The implementation is expressed as a finite state machine, starting in state l_0. The notation `m`$\uparrow k$ specifies that client k sends message `m` to the server; `m`$\downarrow k$ indicates that the server sends `m` to client k, and `m`$\downarrow *$ specifies that the server multicasts `m` to all clients.

To carry out a transaction using locking concurrency control, the client sends a `requestLock` message to the server requesting a lock. If no other client holds a lock, the server responds with a `grantLock` message; otherwise, the lock request is queued. The client may issue any number of `request` messages, receiving corresponding `response` messages. Request/response pairs may be entered in the client cache. The client ends the transaction by sending an `update` message. The update is performed on the server.

A `notify` message is sent to all clients, instructing them to resynchronize their caches. Notification may take many forms. Simple notification simply invalidates all cache entries. More targeted notification (such as the *presend* caching scheme [15]) specifies exactly which cache entries have been invalidated by the update.

The `update` message implicitly releases the lock.

This implementation of locking guarantees that a client holds a lock from the time the `grantLock` message is received by the client until the time the `update` message is committed by the server.

Theorem 2. If client k receives the message `grantLock`$\downarrow k$ at time t_1, and the server receives the message `update`$\uparrow k$ at time t_2, and the server has in the meantime passed only through states l_2 and l_3, then $lock_k(t_1, t_2)$.

Proof. We must show that between t_1 and t_2, *(i)* the shared context does not change, and *(ii)* the cache remains synchronized with the shared context. *(i)* Between t_1 and t_2, the server remains in states l_2 and l_3. In these states, the value of the shared context does not change. Therefore $\forall t_a, t_b : t_1 .. t_2 \bullet s(t_a) = s(t_b)$. *(ii)* At t_1, either the client has just received a `notify` message from the server, or is in its initial state. Therefore $rq_k(t_1) = s(t_1)$. In states l_2 and l_3, the server issues only `response` messages. Assume the server issues `response` $(r, s(t_a,r))$ at time $t_a \in t_1 .. t_2$, and the message is received by k at time $t_b \in t_1 .. t_2$. Then k may update the cache so that $c_k(t_b,r)=s(t_a,r)$, with the result that $rq_k(t_b,r) = s(t_a,r)$. However, from *(i)* we know that $s(t_b)=s(t_a) \Rightarrow s(t_b,r)=s(t_a,r)$, so the cache has remained synchronized. Therefore, $\forall t : t_1 .. t_2 \bullet rq_k(t) = s(t)$.

5 Eager

The locking approach of the last section provided a direct implementation of the temporal properties specified in the semi-replicated architecture. Locking is interesting in cases where transactions are long (and therefore likely to generate conflicts), or cases where unrolling conflicting transactions is expensive (or impossible).

Our second approach is to compute the update transaction and apply it only if it is determined not to conflict. This form of concurrency control is optimistic in the sense that updates are computed in the hope that they will not conflict. However, updates are not committed until they are known not to conflict. This differs from purely optimistic algorithms (such as rollback approaches [3,19]), in which updates may also be optimistically committed.

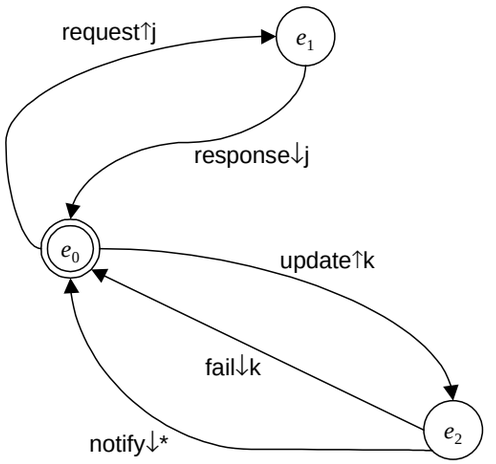

Fig. 5. Implementation of Eager Concurrency Control.

We call this approach *Eager*, an algorithm implementing optimistic update computation with pessimistic update application. Section 5.2 explains how Eager can be implemented efficiently, while providing automatic, fine-grained conflict

detection. Eager concurrency control can provide significantly better performance than locking for transactions typically found in groupware applications, and is particularly appropriate to use over wide area networks [29].

The Eager algorithm (and its optimization) are sufficiently complex that its temporal properties are not obvious from reading the algorithm itself. The architectural specification, however, shows clearly what properties the algorithm has. Additionally, our approach of refining the algorithm from this specification lends confidence to the correctness of Eager's implementation.

A sufficient condition to determine whether a transaction conflicts is to examine the current values of the requests which were used to compute the update. If the requests have not changed value, the update does not conflict. The *fail* function determines whether a transaction may conflict at time *t*:

$$fail : Transaction \times Time \rightarrow Bool$$

Letting $tr=(k, u, <(t_1, r_1), \ldots, (t_n, r_n)>) \in Transaction$, we define

$$fail\ (tr, t) == \exists\ i:1..n \bullet rq_k\ (t_i, r_i) \neq s(t, r_i)$$

This condition is conservative, in that some non-conflicting transactions may fail.

Theorem 3 demonstrates that Eager concurrency control satisfies the temporal properties of the semi-replicated architecture:

Theorem 3. Let *t*:*Time* and *tr*:*Transaction*. Then

$$\neg fail\ (tr,t) \Rightarrow \neg conflict\ (tr,t)$$

Proof: Let $tr=(k, u, <(t_1, r_1), \ldots, (t_n, r_n)>) \in Transaction$. Then

$$\begin{aligned}
&\neg fail\ (tr,t)\\
&\quad \Rightarrow \forall\ i:1..n \bullet rq_k\ (t_i, r_i) = s(t, r_i)\\
&\quad \Rightarrow u\ (rq_k\ (t_1, r_1),\ldots, rq_k\ (t_n, r_n)\\
&\qquad\qquad = u\ (s(t_1, r_1),\ldots,s(t_n, r_n))\\
&\quad \Rightarrow \neg\ conflict\ (tr,t)
\end{aligned}$$

5.1 Implementation of Eager Concurrency Control

Figure 5 shows how Eager concurrency control is implemented at the server. From a start state of e_0, the server can handle requests from any client *j*, and responds with the appropriate response message.

If the server receives the message `update` (*u*, *tr*) at time *t*, for $tr \in Transaction$ and $u \in Update$, then from state e_2, the server must determine whether to commit the update. If *fail* (*tr*, *t*), the server issues the `fail` message. Otherwise, the server commits the update and issues a `notify` to all clients.

Theorem 4. If the server receives the message `update`(*u*,*tr*) at time *t*, where $tr \in Transaction$ and $u \in Update$, then if *conflict*(*tr*,*t*), the server does not commit the update.

Proof. State e_2 only commits *u* if ¬*fail* (*tr*,*t*). By theorem 3, ¬*fail* (*tr*,*t*) ⇒ ¬*conflict* (*tr*,*t*).

5.2 Optimization

Directly computing the *fail* function is expensive, as complete information on the transaction is required. Passing the transaction information over a network can be expensive in bandwidth, and in marshalling and unmarshalling. Computing whether the values of requests have changed places load on the server machine. Eager concurrency control can be optimized, making it substantially faster than locking concurrency control in a wide area context.

Our approach of refining implementations from architectural specifications allows us to demonstrate that this optimized algorithm still satisfies the temporal properties of the semi-replicated architecture.

First, we assign integer id's to cache entries. Let *CacheId* == ℕ. Then assume the existence of a function

$$h : Request \times Value \rightarrow CacheId$$

with the property that $\forall r_1, r_2 \in Request, v_1, v_2 \in Value \bullet h(r_1, v_1) = h(r_2, v_2) \Rightarrow r_1 = r_2 \wedge v_1 = v_2$. We then define a new version of the *fail* function that operates over cache id's. If $tr = (k, u, <(t_1, r_1), \ldots, (t_n, r_n)>) \in Transaction$, $t \in Time$, then

$$fail'(tr,t) == \exists i : 1..n \bullet h(r_i, rq_k(t_i, r_i)) \neq h(r_i, s(t, r_i))$$

h can be implemented efficiently on the clients by having the cache assign an integer id to each of its entries. A server cache [15] can perform the same function on the server. The `update` message can then simply pass integer cache id's rather than the values of the requests themselves. Computing *fail'* involves integer comparisons over the cache id's, rather than recomputing and comparing request values.

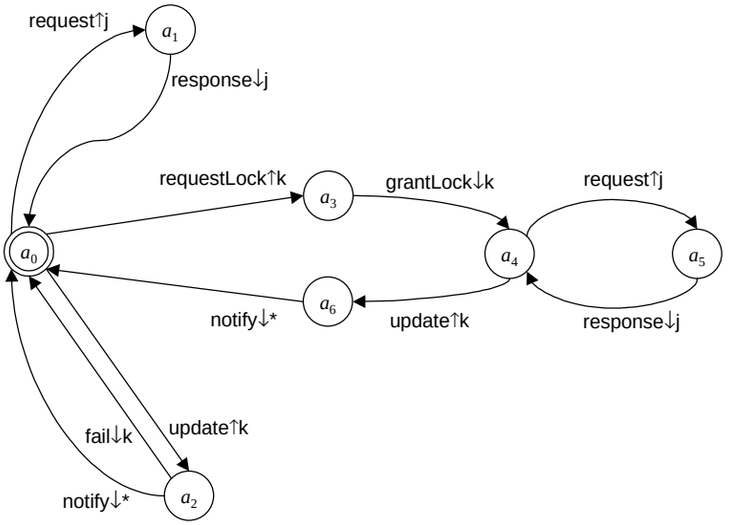

Fig. 6. Implementation of Adaptive Concurrency Control.

Theorem 5. Let *t*:*Time* and *tr*:*Transaction*. Then

$$fail'(tr,t) \Rightarrow fail\ (tr,t)$$

Proof. Follows directly from the definitions of *fail*, *fail'* and *h*.

The implementation of Eager concurrency control can therefore be optimized by substituting the computation of *fail* in e_2 with *fail'*. This implementation is used in the Clock [29] and TCD [1] groupware development toolkits.

6 Adaptive Concurrency Control

As discussed in section 2.1, neither pessimistic nor optimistic concurrency control is appropriate for all applications, and in fact a single application should be able to combine both forms of concurrency control. This section describes how locking and Eager concurrency control can be combined to a single adaptive algorithm. This algorithm has the following properties:

- Clients can decide on a per-transaction basis whether to use locking or Eager concurrency control.
- The concurrency controller is associated with the model, and automatically adapts to the concurrency control scheme being used by the clients.
- Multiple transactions can be processed in parallel, where some are locking and some Eager.
- Adaptive concurrency control places no overheads on either the locking or Eager algorithms. Eager transactions may be computed even if another client holds a lock on the model (but may not commit while the lock is in place.)

This example shows that the approach of specifying temporal properties in architectures not only permits the specification of existing, well-understood algorithms, but also supports the development of new algorithms.

The remainder of this section describes the implementation of adaptive concurrency control.

6.1 Implementation of Adaptive Concurrency Control

The server implementation of adaptive concurrency control is shown in figure 6. This implementation simply combines the finite state machines of figures 4 and 5. From start state a_0, if a client requests a lock, locking concurrency control is used for that client's transaction. Otherwise, requests and updates are processed using the Eager method. The one change in the locking algorithm is that from state a_4, any client can make a request, not just the client holding the lock. This does not affect the client holding the lock, as requests do not change the value of the shared context. This change allows Eager transactions to be computed concurrently with locking transactions; however, updates resulting from Eager transactions may not be committed until after the lock is released.

Adaptive concurrency control ensures that no conflicting transaction is committed:

Theorem 6. Assume that clients compute update transactions either using a locking pattern (where the client makes no requests until a `grantLock` message is received) or an Eager pattern (where the client does not request a lock). Then adaptive concurrency control ensures that if the server receives the message `update` (*u*,*tr*) at time *t*, where $u \in Update$ and $tr \in Transaction$, then if *conflict*(*tr*,*t*), the server does not commit the update.

Proof. Apply the same argument as used in theorems 2 and 4.

From this example, we therefore see it is possible to refine the implementation of a novel concurrency control algorithm permitting both pessimistic and optimistic transactions to be executed in parallel.

7 Conclusion

This paper has introduced the concept that software architectures can encode temporal properties of software systems as well as the traditional structural properties. We have shown an example of such properties in the context of specifying concurrent behaviour of clients and server in the semi-replicated groupware architecture. We have shown that both pessimistic and optimistic concurrency control algorithms can be refined from the required temporal behaviour, as well as a novel adaptive scheme permitting both optimistic and pessimistic transactions to execute in parallel.

This approach allows us to treat architectures as specifications both of the structural and temporal properties of interactive systems. The architecture specifies high level properties of the system's behaviour, while a developer is free to implement such behaviour in any way he/she chooses. This represents a departure from the common approach in groupware, where rather than the specifying the behavioural properties of an application, developers commit early to a particular concurrency control algorithm.

The main weakness of our approach in its current form is that our model of the semi-replicated architecture is simplified, permitting no concurrency control over the replicated data. This does not allow us to treat the operation transform class of concurrency control algorithms such as dOpt [11], CCU [7] and ORESTE [18]. Interesting future work will be to extend our description of semi-replication to address this shortcoming. Additionally, we are currently extending this architectural approach to areas other than concurrency control. This involves the development of a full range of architectural primitives embodying temporal properties accounting for the non-zero latency of real networks, and accounting for architectural evolution over time resulting from session management.

Acknowledgments. This research was carried out by the authors at the Software Technology Laboratory of Queen's University, and was partially supported by the Natural Science and Engineering Research Council (NSERC). The work greatly benefited from the work of Gary Anderson in the TeleComputing Developer toolkit, and from numerous discussions with Greg Phillips.

8 References

1. G.E. Anderson, T.C.N. Graham, and T.N. Wright. Dragonfly: Linking conceptual and implementation architectures of multiuser interactive systems. In *Proc. ICSE 2000*, 2000.
2. R.M. Baecker, editor. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration.* Morgan Kaufmann Publishers, 1993.
3. G. Banavar, K. Miller, and M. Ward. Adaptive views: Adapting to changing network conditions in optimistic groupware. In *Proc. Euro-PDS '98*, 1998.
4. S. Bhola, B. Mukherjee, S. Doddapaneni, and M. Ahamad. Flexible batching and consistency mechanisms for building interactive groupware applications. In *18th International Conference on Distributed Computing Systems (ICDCS)*, 1998.
5. G. Calvary, J. Coutaz, and L. Nigay. From single-user architectural design to PAC*: A generic software architecture model for CSCW. In *Proc. CHI '97*, pages 242-249. ACM Press, 1997.
6. Chabert, E. Grossman, L. Jackson, S. Pietrowicz, and C. Seguin. *Java object sharing in Habanero.* CACM, 41(6):69-76, June 1998.
7. G.V. Cormack. *A calculus for concurrent update.* Research report CS-95-06, University of Waterloo, 1995. Available from ftp://cs-archive.uwaterloo.ca.
8. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design. Addison-Wesley, second edition*, 1994.
9. P. Dewan and R. Choudhary. A high-level and flexible framework for implementing multiuser user interfaces. *ACM TOIS*, 10(4):345-380, October 1992.
10. P. Dourish. Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. In *Proc. ACM CSCW '96.* ACM Press, 1996.
11. C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *Proc. SIGMOD '89*, pages 399-407. ACM Press, 1989.
12. U. Gall and F.J. Hauck. Promondia: A Java-based framework for real-time group communication on the Web. In *Proceedings of the 6th World Wide Web Conference*, Santa Clara, CA. April 7--11. Published as Computers, Networks and ISDN 29(8/13). Elsevier Science Publishers B. V. (North-Holland), 1997.
13. T.C.N. Graham and T. Urnes. Relational views as a model for automatic distributed implementation of multi-user applications. In *Proc. ACM CSCW '92*, pages 59-66. ACM Press, 1992.
14. T.C.N. Graham and T. Urnes. Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces. In *Proc. ICSE 19*, pages 172-182. ACM Press, 1997.
15. T.C.N. Graham, T. Urnes, and R. Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proc. ACM UIST '96*, pages 1-10. ACM Press, 1996.
16. S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proc. ACM CSCW '94*, pages 207-217. ACM Press, 1994.
17. R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, and W. Wilner. The Rendezvous language and architecture for constructing multi-user applications. *ACM TOCHI*, 1(2):81-125, June 1994.
18. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In Proc. 13th International Conference on Distributed Computing Systems (ICDCS), pages 195--202, 1993.
19. T. Kindberg, G. Coulouris, J. Dollimore, and J. Heikkinen. Sharing objects over the Internet: The Mushroom approach. In *Proceedings of IEEE Global Internet '96* (Mini-conference at GLOBECOM '96, London, England, Nov. 20-21). IEEE ComSoc, 1996.

20. G.E. Krasner and S.T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *JOOP*, 1(3):26-49, August/September 1988.
21. J.C. Lauwers and K.A. Lantz. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *Proc. CHI '90*, (also in [2]), pages 303-311. ACM Press, 1990.
22. J.P. Munson and P. Dewan. A concurrency control framework for collaborative systems. In *Proc. ACM CSCW '96*, pages 278-287. ACM Press, 1996.
23. R.E. Newman-Wolfe, M.L. Webb, and M. Montes. Implicit locking in the Ensemble concurrent object-oriented graphics editor. In J. Turner and R. Kraut, editors, *Proc. ACM CSCW '92*, pages 265-272. ACM Press, 1992.
24. W.G. Phillips. *Architectures for synchronous groupware*. Technical Report 1999-425, Department of Computing and Information Science, Queen's University, May 1999.
25. M. Roseman and S. Greenberg. Building real time groupware with GroupKit, a groupware toolkit. ACM TOCHI, 3(1):66-106, March 1996.
26. J. Roth and C. Unger. Dreamteam - a platform for synchronous collaborative applications. In Th. Herrmann and K. Just-Hahn, editors, *Groupware und organisatorische Innovation (D-CSCW'98)*, pages 153-165. B.G. Teubner Stuttgart, Leipzig, 1998.
27. Schuckmann, L. Kirchner, J. Schummer, and J.M. Haake. Designing object-oriented synchronous groupware with COAST. In *Proc. ACM CSCW '96*. ACM Press, 1996.
28. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
29. T. Urnes and T.C.N. Graham. Flexibly mapping synchronous groupware architectures to distributed implementations. In *Proc. DSVIS'99*, pages 133-148, 1999.