Model Checking Groupware Protocols

Maurice H. ter Beek Mieke Massink Diego Latella Stefania Gnesi
Istituto di Scienza e Tecnologie dell'Informazione, CNR
Area della Ricerca di Pisa, Via G. Moruzzi 1, 56124 Pisa, Italy
{terbeek,massink,latella,gnesi}@isti.cnr.it

Abstract. We show how model checking can be used for the verification of protocols underlying groupware systems. To this aim, we present a case study of those protocols underlying the Clock toolkit [1, 2] that are responsible for its concurrency control and distributed notification aspects. We abstract from the original specification of these protocols given in [3] in order to obtain a less detailed specification (model) that nevertheless covers many issues of interest. We show that this model is very well amenable to model checking by addressing the formalisation and verification of a number of important issues for the correctness of groupware protocols in general, i.e. not limited to those underlying Clock. In particular, we address data consistency through distributed notification, view consistency, absence of (user) starvation, and key issues related to concurrency control. As a result, we contribute to the verification of Clock's underlying groupware protocols, which was attempted in [3] with very limited success.

Keywords: groupware protocols, model checking, Clock toolkit, concurrency control, distributed notification

1 Introduction

Computer Supported Cooperative Work (CSCW for short) is concerned with understanding how people work together, and the ways in which computer technology can assist this cooperation [4]. By the nature of the field, this technology mostly consists of multi-user computer systems called groupware (systems) [5, 6]. In this paper we consider groupware allowing real-time collaboration, which is also called synchronous groupware. Examples include video conferencing, collaborative writing, and multi-user games.

An additional difficulty that arises during the design of synchronous groupware is the inherently distributed nature of such systems. This forces one to address issues like network communication, concurrency control, and distributed notification. This has lead to the development of groupware toolkits that aid groupware developers with a series of programming abstractions aimed at simplifying the development of groupware applications. Examples include Rendezvous [7], GroupKit [8], and Clock [1, 2]. In this paper we deal with the latter toolkit, which has been used to develop a number of groupware applications, such as a multi-user video annotation tool [9], the multi-user GroupScape HTML browser [10], a multi-user design rationale editor [11], and the ScenicVista user interface design tool [12].

In this paper we take a closer look at several of the groupware protocols underlying Clock. In particular, we focus on those concerned with the concurrency control and distributed notification aspects of Clock. The abstraction from details of the specification of the Clock protocol as given in [3], gives rise to a less detailed specification (model) of the concurrency control

and distributed notification aspects of the Clock that nevertheless covers faithfully many of the issues of interest. We show that this model is very well amenable to model checking by addressing the formalisation and verification of a number of issues specifically of interest for the correctness of groupware protocols in general, i.e. not limited to those underlying Clock. In particular, we address data consistency, view consistency, absence of (user) starvation, and key issues related to concurrency control. As a result we thus contribute to the verification of Clock's groupware protocols, which was attempted in [3] with very limited success.

Although time-performance issues are very important in groupware systems [13], the correctness of many of their underlying protocols is not critically depending on real time. In other words, the groupware protocols need to function correctly under whatever time assumptions are being made. This is mainly so because these groupware systems have often been designed for being used over the Internet, where the time performance that can be guaranteed is usually of the type 'best effort'. This means that much of the correctness of the groupware protocols can be analysed also with models that do not include real-time aspects. Of course this does not mean that real-time and performance aspects are not relevant to the design of groupware systems, to the contrary, but they need not necessarily be addressed in the same models as those being appropriate to verify correctness issues. In fact, abstracting from real-time and performance issues at first may make the difference between models that are computationally tractable and those that cannot be analysed with the help of automatised tools.

In this paper we show that with relatively simple models we can verify highly relevant properties of groupware protocols with currently freely available verification tools, such as the model checker Spin [14]. The properties we verify are mostly formalised as formulae of a Linear Temporal Logic (LTL for short) [15]. This has as an advantage that they are close to what are also called 'scenarios' that are often formulated (informally) during the initial phases of software design. In fact, LTL formulae reflect properties of typical—desired or undesired—behaviour (or uses) of the groupware system. Our future aim is to extend the models developed in this paper in order to cover also session management, various forms of replication and caching, and other concurrency control mechanisms.

We begin this paper with a brief description of the Clock toolkit and its underlying Clock protocol. We continue with an overview of the basic concepts of model checking and the model checker Spin, followed by a discussion of the specifications in Spin's input language Promela of some of Clock's groupware protocols. Subsequently we verify a number of core issues of the Clock protocol. Finally, we conclude with a discussion of future work.

2 The Clock Toolkit

In this section we present a brief overview of the Clock toolkit and its underlying Clock protocol. For more information on Clock or for obtaining Clock, we refer the reader to `www.cs.queensu.ca/~graham/clock.htm`.

The Clock toolkit is a high-level groupware toolkit that is supported by the visual ClockWorks [16] programming environment and which has a design-level architecture based on the Model-View-Controller (MVC for short) paradigm of [17]. According to this paradigm, an architecture organising interactive applications is partitioned into three separate parts: the Model implementing the application's data state and semantics, the View computing the graphical output of the application, and the Controller interpreting the inputs from the users. In Figure 1, the MVC architecture is depicted together with its communication protocol.

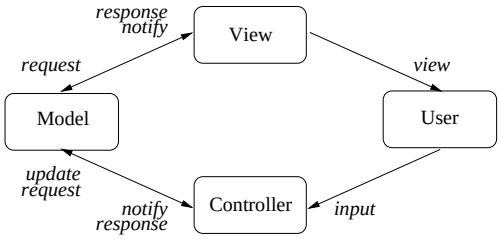

Figure 1: The MVC architecture and its communication protocol.

The Controller transforms an *input* from the User into an *update*, which it sends to the Model. In order to do so, it may need to obtain data from the Model by communicating via *request* and *response*. Upon receiving an *update*, the Model changes its data state and sends a *notify* to both the Controller and the View. The latter, upon receiving this *notify*, recomputes the display—for which it may need to obtain the new data state from the Model by communicating again via *request* and *response*—and eventually sends a *view* to the User.

In Clock's design-level architecture, the Model is situated on the server, while the View and the Controller are integrated and situated on each of the clients. The communication between the server and the clients is defined by a set of (communication) protocols, together called the Clock protocol. In [3], a simplified version—leaving out many implementation details— of the Clock protocol was formalised in the specification language Promela and an attempt was made to verify it with the model checker Spin [14]. Partly due to insufficient computing resources, however, it was impossible to verify the entire simplified Clock protocol. Consequently, an attempt was made to verify only a part of the Clock protocol still large enough to be relevant, viz. the part relevant to the issue of concurrency control and distributed notification. Unfortunately also this attempt was largely unsuccessful, i.e. the verification covered only 2% of the total state space. In this paper we continue the use of abstraction in order to obtain a Promela specification that is amenable to model checking, but which still models the core issues of the concurrency control and distributed notification aspects of the Clock protocol. As a result of this we aim at obtaining a Promela specification that can serve as a scenario or template specification for the verification of concurrency control and distributed notification issues in related groupware protocols.

2.1 The Clock Protocol

The functioning of the Clock protocol depends on the way it communicates with its environment. As depicted in Figure 2, its environment consists of a Session Manager communicating with the server and a set of Users communicating with the clients.

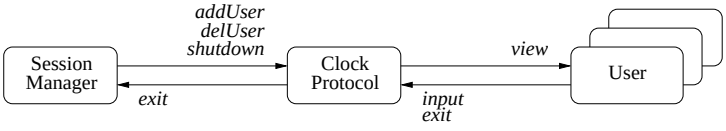

Figure 2: The Clock protocol embedded in its environment.

The Clock protocol consists of four protocols, viz. the MVC protocol, the Cache protocol, the Concurrency Control (CC for short) protocol, and the Replication protocol. Based on the MVC paradigm, the MVC protocol implements multi-user communication between the

server and its clients. The Cache protocol controls caches at the server and its clients in an attempt to reduce the time needed to access shared data. The CC protocol implements synchronisation of concurrent updates and controls the processing of user input and view recomputation. The Replication protocol, finally, controls local copies of selected shared data.

Due to the size of the Clock protocol, we would undoubtedly run into the infamous state-explosion problem as soon as we would try to verify the full Clock protocol. Therefore, we abstract from the full Clock protocol and focus on those protocols that are fundamental to that aspect of the Clock protocol that we want to verify. Since in this paper we are interested in the concurrency control and distributed notification aspects of the Clock protocol rather than in its data aspects, we thus focus on the MVC protocol and the CC protocol. All protocols constituting the Clock protocol are implemented by one component on the server and one on each of the clients. In case of the MVC protocol this results in a Model component on the server and an integrated View/Controller (VC for short) component on each of the clients, while in case of the CC protocol this results in a Concurrency Controller (CC for short) component on the server and an Updater component on each of the clients.

In [3], two different mechanisms implementing the CC protocol are studied: the locking mechanism and the eager mechanism. The locking mechanism uses a single, system-wide lock that a client must acquire before it can process inputs and apply updates, thus guaranteeing a sequential application of updates. Moreover, no updates are allowed during view recomputation, i.e. the locking mechanism is more involving than the standard mutual exclusion paradigm. The eager mechanism, on the other hand, allows concurrent updates and update coalescing. To this aim, all updates that are in conflict with other concurrent updates are aborted and subsequently regenerated until they are handled. In this paper we focus on the locking mechanism, leaving the eager mechanism for future work.

Summarising, in this paper we thus address the MVC protocol and the CC protocol. Moreover, we assume that the CC protocol is implemented by the locking mechanism. Hence we consider the part of the Clock protocol and its environment as depicted in Figure 3.

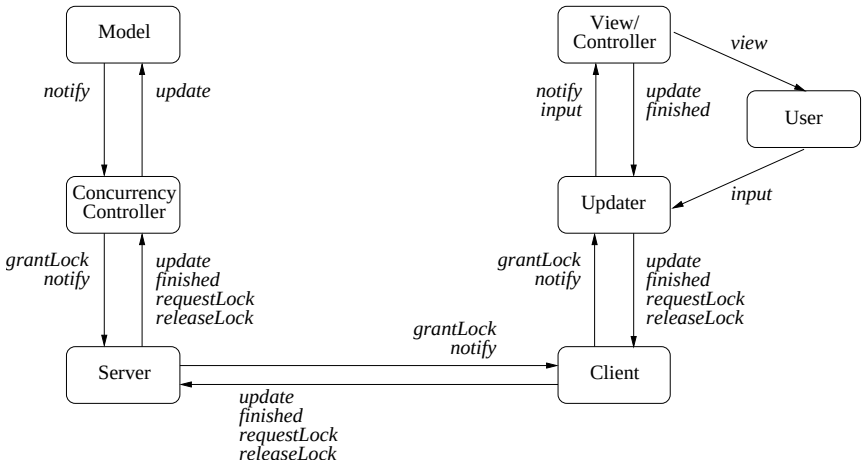

Figure 3: The part of the Clock protocol and its environment relevant to the locking mechanism.

From now on we shall refer to this part of the Clock protocol and its environment as the Model-View-Concurrency-Control (MVCC for short) protocol. A typical series of actions that can take place in the MVCC protocol is the following.

Upon receiving *input* from the User, the Updater tries to obtain the system-wide lock by sending a *requestLock* to the CC. The CC handles these lock requests in their order of

arrival by sending a *grantLock* back to the Updater. Only upon receiving a *grantLock*, the Updater is able to forward the original *input* to the VC. The VC transforms the *input* into an *update*, which it returns to the Updater. The latter returns the lock by attaching a *releaseLock* to the *update* and sending this back to the CC. The CC forwards the *update* to the Model, acknowledges the *releaseLock*, and is only now ready to handle the subsequent lock request. Upon receiving an *update*, the Model changes its data state and sends a *notify* back to each of the VC. Each VC, upon receiving a *notify*, recomputes the display, sends a *view* to its associated User, and sends *finished* back to the CC.

3 Model Checking the MVCC protocol

In this section we present the basic concepts of model checking and of the model checker Spin, followed by specifications in Spin's input language Promela of the MVCC protocol.

Model checking is an automatic technique for verifying whether or not a system design satisfies its specifications [18]. Such a verification is moreover exhaustive, i.e. all possible input combinations and states are taken into account. To avoid to run out of memory due to a state-space explosion—which would make an exhaustive verification impossible—a simplified model is used, which captures the core of the system design while abstracting from unnecessary details.

One of the best known and most successful model checkers is Spin, which was developed at Bell Labs during the last two decades [14]. It offers a spectrum of verification techniques, ranging from partial to exhaustive verification. It is freely available through `spinroot.com` and it is very well documented. Apart from these obvious advantages we have chosen to use Spin in this paper because of the aforementioned earlier attempt at verifying a simplified version of the Clock protocol with Spin in [3], which moreover contains a specification of that simplified Clock protocol in Spin's input language Promela in one of its Appendices.

Promela is a non-deterministic C-like specification language for modelling finite-state systems communicating through channels [14]. Formally, specifications in Promela are built from processes, data objects, and message channels. Processes are the components of the system, while the data objects are its local and global variables. The message channels, finally, are used to transmit data between processes. Such channels can be local or global and they can be FIFO buffered—for modelling asynchronous communication—or handshake (a.k.a. rendezvous)—for modelling synchronous communication. Assume that processes A and B are connected by a channel aToB. Then A can send a message *m* to B over this channel by executing the statement `aToB!m`. If aToB is a buffered channel and its buffer is not full, then *m* is stored in the buffer until B executes `aToB?m` and thereby receives *m* from A over this channel. This is an example of asynchronous communication between A and B. If, on the other hand, aToB is a handshake channel, then the two above executions must be synchronised, i.e. aToB can pass but not store messages. This is an example of synchronous communication. For more detailed information on Promela, we refer the reader to [14].

Promela specifications can be fed to Spin, together with a request to verify certain correctness properties. Spin then converts the Promela processes into finite-state automata and on-the-fly creates and traverses the state space of a product automaton over these finite-state automata, in order to verify the specified correctness properties. Spin is able to verify both safety and liveness properties. Safety properties are those that the system under scrutiny may not violate, whereas liveness properties are those that it must satisfy. Such properties either

formalise whether or not certain states are reachable, or whether or not certain executions can occur. A typical safety property one usually desires is the absence of deadlock states, i.e. states from which there is no possibility to continue the execution that led to these states.

There are several ways of formalising correctness properties in Promela, the following two of which we shall use in this paper. First, we may add *basic assertions* to a Promela specification. Subsequently, we can verify their validity by running Spin. As an example, consider that we want to be sure that no lock has been granted the moment in which we are to grant a lock request. Consider moreover that there is a boolean variable writelock, which is set to `true` every time a lock request is granted. Then we can add the basic assertion

```
assert(writeLock == false)
```

to the Promela specification just before a lock is granted and let Spin verify whether there are any assertion violations. In case Spin concludes that this assertion may be violated, it also presents a counterexample. Otherwise it simply reports that there are no assertion violations.

Secondly, we may add *progress labels* to the Promela specification, which mark a specific point in the specification. Subsequently, we can use such labels to formulate an LTL property and test its validity by running Spin. LTL is an extension of predicate logic allowing one to express assertions about behaviour in time, without explicitly modelling time. Spin accepts formulae in LTL that are constructed on the basis of atomic propositions (including `true` and `false`), the Boolean connectives ! (negation), && (and), || (or), $->$ (implication), and $<->$ (equivalence), and the temporal operators $[]$ (always), $<>$ (eventually), and `U` (until). Given a sequence σ of states from the behaviour of a system, the formula $[]p$ is `true` if the property p always remains `true` in every state of σ, the formula $<>\ p$ is `true` if the property p eventually becomes `true` in at least one state of σ, and the formula $p\,\mathtt{U}\,q$ is `true` if the property p remains `true` in the states of σ until the property q becomes `true` in a state of σ. For more detailed information on LTL, we refer the reader to [15].

As an example, consider that we want to guarantee that the Promela specification of the MVCC protocol excludes starvation of its users, i.e. we want to know whether or not a user can always eventually provide input. Then we can add the progress label

```
doneInput
```

to the Promela specification directly following a user input, i.e. just after the statement

```
userToUpdater[id]!input
```

in the User process. In this statement, `id` identifies the array index of the userToUpdater array of (buffered) channels. Subsequently we can formulate the LTL formulae

$$[]\ <>\ \text{User[pid]@doneInput},$$

where pid is the *process instantiation number* of the User process about which we want to know whether or not every sequence of states from the behaviour of the Promela specification of the MVCC protocol contains a state in which this user's state is the progress label `doneInput`. Starting with 0, Spin assigns—in order of creation—a unique pid to each process it creates, which can be used in LTL formulae for process identification. Finally, we can verify the validity of the above LTL formulae by running Spin. Again, when Spin concludes that this statement is not valid, then it also presents a counterexample. Otherwise it simply reports that the statement is valid. Note that in this paper we generally explain a formula in words only after we have stated it formally.

3.1 The Promela Specification

In this section we discuss the Promela specification of the MVCC protocol which we intend to validate in the next section. Our starting point is the Promela specification of the aforementioned simplified Clock protocol as given in one of the Appendices of [3]. The source code of this specification was generously provided by the author himself.

Recall that our focus on concurrency control and distributed notification aspects of the Clock protocol has led to the MVCC protocol as an abstraction of the Clock protocol. In more detail, from the Promela specification as given in one of the Appendices of [3] we have omitted the environment processes SessionManager and SessionUpdater, the simple caching processes ServerCache and ClientCache, and the replication processes Replicator and Replica, as well as all global variables, messages, and channels associated to these processes. Since none of these interfered with either the concurrency control algorithm underlying the CC protocol or the distributed notification algorithm underlying in the MVC protocol, their removal does not alter the behaviour of these algorithms. Obviously, the described reduction of the total number of processes, data objects, and message channels reduces the state-space and thereby the risk to run into a state-space explosion.

Subsequently we have modified the resulting Promela specification of the MVCC protocol even further—wherever this was possible without changing its meaning—in order to reduce the size of the state space as well as that of the state vector to a greater extent. The state vector is used by Spin to uniquely identify a system state and contains information on the global variables, the channel contents, and for each process its local variables and its process counter. Minimising its size thus results in less bytes that Spin needs to store for each system state and thereby further reduces the risk to run out of memory. Finally, in [14] it is noted that next to the total number of processes, data objects, and message channels in a Promela specification, the most common reason for running out of memory is the buffersize of buffered channels. The most important further modifications that we have performed on the Promela specification of the MVCC protocol are the following.

1. We have reduced the number of processes and channels by integrating the Server and Client processes into the CC and Updater processes, respectively. In Figure 3 we can see that this is a valid abstraction, since the Server and Client processes are nothing more than message-passing processes. Therefore, integrating them with the CC and Updater processes does not alter the meaning of the specification. This obviously reduces both the size of the state space and that of the state vector.

2. We have reduced the number of buffered channels by replacing them as much as possible by handshake channels. This reduces the number of interleaving steps and thus the size of the state space. It moreover reduces the channel contents and thus the state vector.

3. We have further reduced the number of interleaving steps by grouping assignments into atomic blocks as much as possible. More precisely, all administrative statements (such as the updating of bits or booleans) have been grouped into `d_step`s. As a result, they are treated as one deterministic sequence of code that is executed indivisibly, i.e. as if it were one single statement. This obviously reduces the size of the state space, where all interleaving executions are considered.

Hence we consider the modified MVCC protocol as depicted in Figure 4 in case of two users, where buffered (handshake) channels are depicted as dashed (solid) arrows.

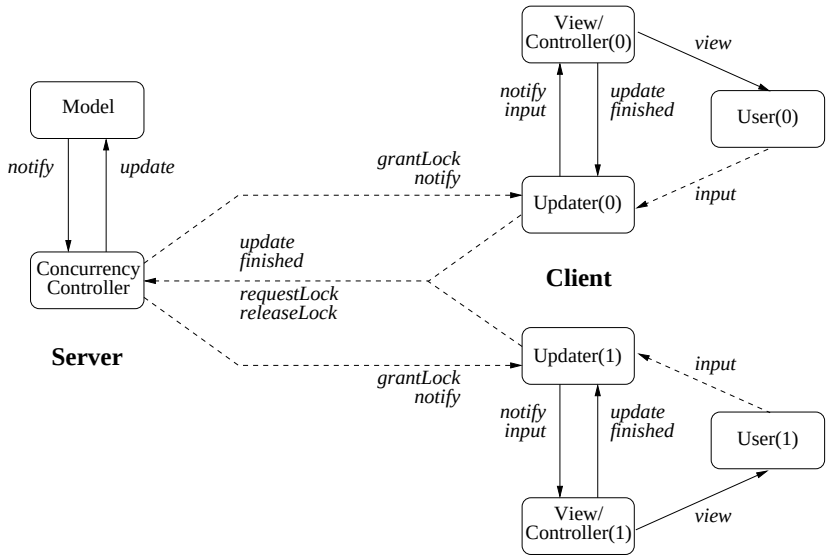

Figure 4: The modified MVCC protocol in case of two users.

It is important to note the buffered channel that is shared by the two Updaters and which connects them to the CC, as it regulates FIFO scheduling of the lock requests from the two Users. The blocks connected by arrows labelled with messages in Figure 4 represent processes communicating by sending variables through channels in the complete Promela specification, which is given in Appendix A. For ease of reading, we have added comments to the source code. Furthermore, we have added some basic assertions and progress labels for verification purposes.

4 Validation with Spin

In this section we show that the abstractions which we have applied to the Promela specification of [3] are sufficient for being able to verify a number of core issues of the concurrency control aspects of the MVCC protocol with Spin.

All verifications reported in this paper have been performed by running Spin Version 4.0.4 on a SUN® Netra™ X1 workstation with 1000 Megabytes of available physical memory.

First and foremost we have let Spin perform a full statespace search for invalid endstates, which is Spin's formalisation of deadlock states. This resulted in the following output.

```
(Spin Version 4.0.4 -- 12 April 2003)
       + Partial Order Reduction

Full statespace search for:
       never-claim             - (not selected)
       assertion violations    - (disabled by -A flag)
       cycle checks            - (disabled by -DSAFETY)
       invalid endstates       +

State-vector 160 byte, depth reached 4539, errors: 0
  38473 states, stored
  34009 states, matched
  72482 transitions (= stored+matched)
  17041 atomic steps
```

```
hash conflicts: 918 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
6.463  equivalent memory usage for states (stored*(State-vector+overhead))
4.384  actual memory usage for states (compression: 67.84%)
       State-vector as stored = 106 byte + 8 byte overhead
2.097  memory used for hash-table (-w19)
0.320  memory used for DFS stack (-m10000)
6.718  total actual memory usage

real       2.0
user       1.9
sys        0.0
```

We see that it now takes Spin just two seconds to conclude that there are no deadlocks. We moreover see that the Promela specification we have used has a 160 byte state vector, whereas that of [3] has a 332 byte state vector. The abstractions that we have applied have thus reduced the state vector with more than a factor 2. From this we gain a lot of confidence w.r.t. the verifiability of future extensions of this Promela specification, e.g. by incorporating session management, the Replication and Cache protocols, or concurrency control based on the eager mechanism. First, however, we want to assure that the MVCC protocol satisfies some properties that are fundamental to concurrency control based on the locking mechanism.

4.1 Correctness Properties

In [3], several correctness criteria that the CC protocol must satisfy have been formulated, covering both safety and liveness properties. Since these properties should also be satisfied by the MVCC protocol, we now briefly recall them.

Data consistency. The server's shared data state must be kept consistent. Any user input that is processed by a client must thus lead to a complete update of the shared data state, which in its turn must result in a notification of this update to all clients.

View consistency. No updates are allowed during view recomputation. Any user's view recomputation must have finished before any further update of the shared data state can take place.

Absence of starvation. Every user input must eventually be responded to. Any user's input must thus result in a lock request, which eventually should be granted.

We also add several core properties of concurrency control based on the locking mechanism.

Concurrency control. Every lock request must eventually be granted, at any moment in time only one user may be in the possession of a lock, and every obtained lock must eventually be released.

In the subsequent sections we verify all of the above properties for the modified MVCC protocol in case of two users by using Spin, the Promela specification given in Appendix A, and an extension of the latter which we will discuss shortly. This shows that verifications of the optimised Promela specification of the MVCC protocol are very well feasible with the current state of the art of available model checking tools such as Spin.

4.2 Concurrency Control

In this section we verify three core properties of locking-based concurrency control.

The first property is that every lock request is eventually granted. To verify this we have added the progress labels

`doneRequestLock` and `doneGrantLock`

to the Promela specification of the Updater just after the statements where the Updater sends a *requestLock* to the CC and where it receives a *grantLock* from the CC, i.e. just after

`updaterToCC!requestLock,id` and `ccToUpdater[id]?grantLock`,

respectively, and then we have let Spin run verifications of the LTL formulae

$$[]\,(\text{Updater[pid]@doneRequestLock} -> <>\ \text{Updater[pid]@doneGrantLock}),$$

where pid is 3 (for Updater(0)) or 6 (for Updater(1)). We thus verify whether or not it is always the case that whenever an updater has requested a lock on behalf of a user, then it eventually grants this user a lock. It takes Spin just over a quarter of an hour to conclude that the above LTL formulae are valid.

Before we continue it is important to note that the above formulae are trivially valid if the left-hand side of the implication is always `false` in every run, i.e. if the Updater never passes the `doneRequestLock` label. Therefore, we have let Spin also run verifications of the LTL formulae

$$!\,(<>\ \text{Updater[pid]@doneRequestLock}),$$

where pid is 3 (for Updater(0)) or 6 (for Updater(1)). We thus verify whether or not there exists a run in which the left-hand side of the implication eventually becomes `true` by verifying whether or not the Updater can ever pass the `doneRequestLock` label. It takes Spin just a split second to conclude that the above LTL formulae are not valid. It moreover provides counterexamples which show that the Updater can eventually pass the `doneRequestLock` label. Though never mentioned specifically, for all formulae in the sequel that contain a logical implication we have verified that the left-hand side of this implication can indeed actually become `true` in at least one run.

The second property is that at any moment in time, the CC may have granted at most one lock. To verify this we have added the basic assertion

```
assert(writeLock == false)
```

to the Promela specification of the CC just before it is about to grant a lock to a user, i.e. just before the statement

`ccToUpdater[id]!grantLock`,

and then we have let Spin run a verification on assertion violations. We thus verify whether or not it is always the case that the boolean variable `writeLock` is false (indicating that no user currently has a lock in its possession) the moment in which the CC is about to grant a user a lock by sending *grantLock* to the updater associated to this user. Note that `writelock` is set to true by the CC directly after it has sent *grantLock*. In just a few minutes Spin concludes that the above basic assertion is never violated, which proves that the property is valid.

The third property is that every obtained lock is eventually released. To verify this we have added the progress label

```
doneReleaseLock
```

to the Promela specification of the Updater just after the statement where the Updater sends a *releaseLock* to the CC, i.e. just after

```
updaterToCC!releaseLock,id,
```

and then we have let Spin run verifications of the LTL formulae

$$[]\,(\text{Updater[pid]@doneGrantLock} -> <>\ \text{Updater[pid]@doneReleaseLock}),$$

where pid is 3 (for Updater(0)) or 6 (for Updater(1)). We thus verify whether or not it is always the case that whenever an updater has obtained a lock on behalf of a user, then it eventually releases this lock. Again, it takes Spin just a few minutes to conclude that the above LTL formulae are valid as well.

The verifications performed in this section show that the concurrency control aspects of the MVCC protocol are well designed. They moreover satisfy the core properties of concurrency control based on the locking mechanism, as specified in the previous section.

4.3 Data Consistency

In this section we verify data consistency, which is an important property of groupware systems in general and the MVCC protocol in particular. Data consistency not only requires the server's data state to be kept consistent, but each update of the shared data state should moreover be communicated to all clients through distributed notification. Any user input processed by a client must thus lead to a complete update of the shared data state, which in its turn must result in a notification of this update to all clients.

Unfortunately, the Promela specification we have used so far does not contain enough information for verifying data consistency. This is because the addition of a progress label

```
doneUpdate
```

to the Promela specification of the Model just after the statement where it receives an *update* from the CC, i.e. just after

```
ccToModel?update,_,
```

where _ matches any value, would not have allowed us to conclude which user's input caused this update. To nevertheless verify data consistency, we have extended the Promela specification only for verification purposes with a user ID. This ID identifies the user that has *caused* an update and is sent along with all actions involved in the resulting distributed notification of this update, i.e. *update*, *notify*, and *finished*. The complete Promela specification extended with this user ID is given in Appendix B. Consequently, we have added the progress labels

```
doneUpdate0
```
and
```
doneUpdate1
```

to the extended Promela specification of the Model just after the statement

```
ccToModel?update,ID
```

in the form of an `if`-statement which guarantees that `doneInput0` is passed if the `ID` in `ccToModel?update,ID` equals 0 (for User(0)), whereas `doneInput1` is passed if it equals 1 (for User(1)). Analogously, we have added the progress labels

`doneNotify0` and `doneNotify1`

to the extended Promela specification of the Updater just after the statement where it receives a *notify* from the CC, i.e. just after

`ccToUpdater[id]?notify,ID`,

in the form of an `if`-statement which guarantees that `doneNotify0` is passed if the `ID` in `ccToUpdater[id]?notify,ID` equals 0 (for User(0)), whereas `doneNotify1` is passed if it equals 1 (for User(1)).

Subsequently, to verify that any user input that is processed by a client must lead to a complete update of the shared data state, we have added the progress label

`doneInput`

to the extended Promela specification of the Updater just after the statement where it receives user input, i.e. just after

`userToUpdater[id]?input`,

and then we have let Spin run verifications of the LTL formulae

$$[\,](\text{Updater[pid]@doneInput} -> <> \text{Model[1]@doneUpdateX}),$$

where pid is 3 and X is 0 (for Updater(0) corresponding to User(0)) or pid is 6 and X is 1 (for Updater(1) corresponding to User(1)), while 1 is the pid of the Model. We thus verify whether or not it is always the case that whenever an updater processes a user input, then the Model eventually updates the shared data state. It takes Spin about ten minutes to conclude that the above LTL formulae are valid.

Finally, to verify that any update of the shared data state in its turn results in a notification of this update to all clients, we have let Spin run verifications of the LTL formulae

$$[\,](\text{Model[1]@doneUpdateX} -> ((<> \text{Updater[3]@doneNotifyX}) \,\&\&\, (<> \text{Updater[6]@doneNotifyX}))),$$

where 1 is the pid of the Model, 3 is the pid of Updater(0), and 6 is the pid of Updater(1), while X is 0 (for Updater(0)) or 1 (for Updater(1)). We thus verify whether or not it is always the case that whenever the Model updates the shared data state on behalf of one of the users, then all updaters eventually receive a notification of the update for that user. It takes Spin just over half an hour to conclude that also the above LTL formulae are valid.

The verifications performed in this section show that the distributed notification aspects of the MVCC protocol are well designed and that data consistency is guaranteed.

4.4 View Consistency

In this section we verify view consistency rather than data consistency as another important property of groupware systems in general and the MVCC protocol in particular. These two properties are related, but the focus now lies on what a user sees on his or her screen.

In [3], view consistency is defined as excluding updates during view recomputation. Hence any user's view recomputation must have finished before any further update of the shared data state can occur (and trigger a new view recomputation). However, a user's input is based on what he or she sees on his or her screen. Therefore, we believe it to be equally important for groupware systems in general and the MVCC protocol in particular that input should not be based on an outdated view. Hence any user's view recomputation based on an earlier input should have finished before this user can provide further input.

Initially, we verify that any user's view recomputation must have finished before any further update of the shared data state can take place. To do so, we have used the temporal operator `U` (until) to prohibit the CC to forward an *update* to the Model for the second time before both user's views have been recomputed as a result of the first time it has forwarded an *update* to the Model. We thus needed to distinguish the progress label indicating that the CC *has* in fact forwarded an *update* from the one indicating that it *does not do so again* until it has received a *finished* from the VCs of both users. Therefore we have added to the extended Promela specification of the CC the progress labels

`doneInputY` and `doneInputY2`,

where `Y` is 0 (for an *update* from Updater(0)) or 1 (for an *update* from Updater(1)), just after the statement where the CC sends an *update* to the Model, i.e. just after

`ccToModel!update,id`,

and the progress labels

`doneFinishedXY`,

where `X` is 0 (for a *finished* from Updater(0)) or 1 (for a *finished* from Updater(1)) and `Y` is 0 (for a *finished* resulting from an *update* from Updater(0)) or 1 (for a *finished* resulting from an *update* from Updater(1)), just after the statement where it sends a *finished* to the CC, i.e. just after

`updaterToCC!finished,id,ID`.

Consequently, we have let Spin run verifications of the LTL formulae

$$[]\,(\text{CC[2]@doneUpdateY2} -> (((!\,\text{CC[2]@doneUpdateY})\,\mathtt{U}\,\text{CC[2]@doneFinished0Y})\,\&\&\,((!\,\text{CC[2]@doneUpdateY})\,\mathtt{U}\,\text{CC[2]@doneFinished1Y}))),$$

where 1 is the pid of the CC and Y is 0 (for an *update* from Updater(0)) or 1 (for an *update* from Updater(1)). It takes Spin just over one and a half hour to conclude that the above LTL formulae are valid.

Next we verify that any user's view recomputation based on an earlier input must have finished before this user can provide further input. To do so, we have again used the temporal operator `U` (until), this time however to prohibit a user to provide input (i.e. pass the

`doneInput` progress label) before both user's views have been recomputed (i.e. both have passed the `doneView` progress label). We thus needed to distinguish the progress label indicating that a user *has* in fact provided input from the one indicating that it *does not do so again* until both users have passed the `doneView` progress label. Therefore we have added the progress label

`doneInput2`

to the Promela specification of the User just after the progress label

`doneInput`

and then we have let Spin run verifications of the LTL formulae

$$[]\,(\text{User[pid]@doneInput2} \rightarrow ((!\,\text{User[5]@doneInput})\,\&\&\,(!\,\text{User[8]@doneInput}))\,\mathtt{U}\,(\text{User[5]@doneView}\,\&\&\,\text{User[8]@doneView})),$$

where pid is 5 (for User(0)) or 8 (for User(1)), while 5 and 8 are the pids of User(0) and User(1), respectively. It takes Spin just a split second to conclude that these formulae are not valid! It in fact presents counterexamples, one of which we have sketched in Figure 5 in the form of a message sequence chart.

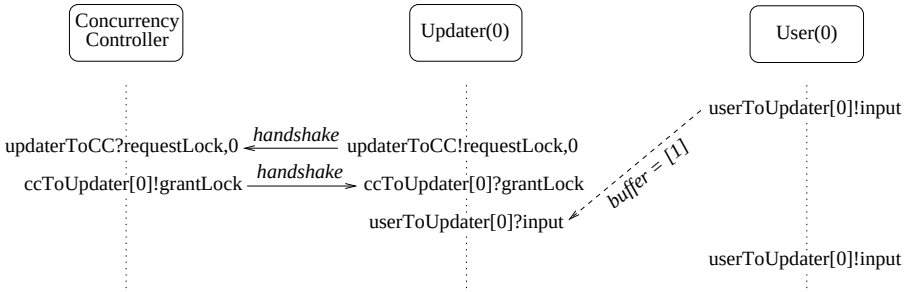

Figure 5: The message sequence chart of a counterexample for view consistency.

This message sequence chart describes the following problem. At a certain moment in time, User(0) provides an input by storing an *input* message in the buffered channel (with buffersize 1) connecting User(0) with its associated Updater(0). This Updater(0) consequently participates in two handshakes with the CC, the first one to request the lock and the second one to obtain the lock. Now that Updater(0) has obtained the lock, it reads the *input* message from the aforementioned buffered channel, thereby emptying its buffer. At this moment, User(0) may thus fill the buffer again with an *input* message, which is by definition based on a view that has not been updated w.r.t. User(0)'s own input.

The verifications performed in this section show that view consistency is guaranteed when it is understood as the prohibition of updates during view recomputation. However, a user's input may be based on an outdated view.

4.5 Absence of Starvation

A further desirable property of any groupware system in general and the MVCC protocol in particular is that none of its users can be excluded forever, i.e. every user input must

eventually be responded to. In this section we verify this absence of (user) starvation and the fact that any user's input must result in a lock request, which eventually should be granted.

The main question we thus ask ourselves here is whether or not each user can always provide input if it wishes to do so. To verify this, we have let Spin run verifications of the LTL formulae

$$[\,]\ <>\ \text{User[pid]@doneInput},$$

where pid is 5 (for User(0)) or 8 (for User(1)). We thus verify whether or not it is always the case that a user always eventually sends *input* to its associated Updater. Unfortunately, in just a split second Spin concludes that the above LTL formulae are not valid. It moreover presents counterexamples. More precisely, it finds cyclic behaviour in which one of the user processes can never again send *input* to its associated updater, after having done so just once in the very beginning.

It is important to note that even when we have used Spin's weak fairness notion, Spin concluded that the above formulae are not valid. In fact, it can still be the case that a user is continuously busy updating its view through the statement `vcToUser[id]?view` rather than performing an input through the statement `userToUpdater[id]!input`. We come back to this issue in the next section.

We continue by verifying that any user's input must result in a lock request, which eventually should be granted. From Subsection 4.2 we know that every lock request is eventually granted. To verify that a user input eventually results in a lock request, we have let Spin run verifications of the LTL formulae

$$[\,]\,(\text{User[pid1]@doneInput} -> <>\ \text{Updater[pid2]@doneRequestLock}),$$

where pid1 is 5 and pid2 is 3 (for User(0) and Updater(0)) or pid1 is 8 and pid2 is 6 (for User(1) and Updater(1)). We thus verify whether or not it is always the case that whenever a user provides input, then its associated updater eventually requests a lock. It takes Spin just over ten seconds to conclude that the above LTL formulae are not valid. This does not come as a surprise, however, because the cyclic behaviour in the above counterexample in fact is such that the updater that is associated to the user that can never again send *input* after having done so once in the beginning, is continuously busy with operations related to the updating of its associated user's view. These operations are the result of the other user's continuous stream of *input*.

The verifications performed in this section show that a user input need not be responded to. As a result, we were not able to guarantee the absence of user starvation and thus neither the fact that any user's input should lead to a lock request. In the next section we show that under a proper condition, absence of user starvation can be guaranteed.

4.6 Spin and Fairness

In the previous section we have seen that, given the Promela specification of the MVCC protocol given in Appendix A, Spin does not allow one to conclude that a user can always provide input in the MVCC protocol, not even when using weak fairness. Instead, Spin can continuously favour the execution of the user's view action `vcToUser[id]?view` over that of the user's input action `userToUpdater[id]!input`.

Recall that we have specified the User process as follows in Promela.

```
proctype User(byte id)
{
  do
  :: userToUpdater[id]!input;
  :: vcToUser[id]?view;
  od
}
```

The reason for this is the way fairness is implemented in Spin, viz. at the process level rather than at the statement level. Consequently, a computation is said to be weakly fair if every process that is continuously enabled from a particular point in time will eventually be executed after that point. Note that this does not guarantee that every (infinitely often) enabled statement of such a process will eventually be executed after that point. This is due to the fact that such a process may contain more than one statement that is continuously enabled from a particular point in time and in order for this process to be weakly fair it suffices that one of these statements will eventually be executed after that point. In this section we discuss one possible solution to overcome this problem and thus enforce fairness on the statement level.

In [14] it is suggested to enforce weak fairness by specifying the desired fairness constraint c as an LTL formula and consequently verifying whether or not a specification satisfies a property p under the condition that it satisfies c. Rather than verifying p one thus verifies $c -> p$. LTL is sufficiently expressive for specifying fairness constraints of this type.

To overcome the problem we encountered above, we thus had to specify a constraint which guarantees that both users are equally given the possibility to perform input. To this aim, we have added the progress labels

`checkInput` and `checkNotify`

to the Promela specification of the VC just after the statements where it receives an *input* or a *notify* from the Updater, i.e. just after

`updaterToVC[id]?input` and `updaterToVC[id]?notify`,

respectively, and then we have let Spin run verifications of the LTL formulae

$$((\,[\,] <>\, \text{VC[pid1]@checkNotify}) \,\&\&\, ([\,] <>\, \text{VC[pid1]@checkInput})) -> [\,] <>\, \text{User[pid2]@doneInput},$$

where pid1 is 4 and pid2 is 5 (for VC(0) and User(0)) or pid1 is 7 and pid2 is 8 (for VC(1) and User(1)). We thus verify whether or not a user can always provide input under the condition that its associated VC checks for update notifications and user input in a fair way, i.e. it always eventually checks for a *notify* from the corresponding updater and it always eventually checks for an *input* from the corresponding updater. Hence, we restrict our verification to those computations in which the VC satisfies a kind of progress condition under which both actions that it is able to receive, are in fact received infinitely often. We know that such computations exist because, as said before, we have verified for all formulae in this paper that the left-hand sides of implications can indeed actually become `true` in at least one run. It takes Spin just about twenty minutes to conclude that the above LTL formulae are valid.

The verifications performed in this section show that absence of user starvation can be guaranteed by adding a proper constraint as a preamble to the LTL formula expressing the

absence of user starvation. Such a constraint could reflect a possible implementation strategy that guarantees fair treatment of enabled options. It may sometimes be quite hard to express the necessary constraints in LTL. In fact, it is our experience that most of the time spent on the verifications performed in this paper actually went into formulating the proper LTL formulae.

5 Conclusion

In this paper we have shown how model checking can be used for the verification of protocols underlying groupware systems. More precisely, we have presented a case study on the formalisation and verification of those protocols underlying the Clock toolkit, that are responsible for its concurrency control and distributed notification aspects. The correctness properties that we have verified in this paper are related to important groupware issues such as data consistency, view consistency, absence of (user) starvation, and concurrency control. As a result, we contribute to the verification of some of Clock's underlying groupware protocols, which was attempted in [3] with very limited success.

In the future we plan to verify other interesting properties after extending the model developed in this paper in order to cover also session management, various forms of replication and caching, and other concurrency control mechanisms. Regarding the Clock toolkit this can be achieved by incorporating some of its components that we have abstracted from in this paper, i.e. the Cache protocol and the Replication protocol from the Clock protocol, the part of the CC protocol regarding the eager mechanism, and the Session Manager from Clock's environment. For the development of such extensions of the specification we moreover plan to take into consideration other modelling techniques, in particular compositional ones like process algebras and team automata [19]. The combination of compositionality together with powerful abstraction notions supported by a sound algebraic theory (e.g. congruences and equational laws) not only makes process algebras well suited for protocol modelling, but also gives opportunities for effectively tackling the complexity of the analysis. Furthermore, nowadays several model checkers are available for process algebras (e.g. JACK [20], CADP [21], CWB-NC [22], and μCRL [23]). Team automata were introduced explicitly for the description and analysis of groupware systems and their interconnections [24, 25] and were shown to be useful in a variety of groupware settings [26, 27]. A key feature of team automata is the intrinsic flexibility of their synchronisation operators. In [28] it was shown that constructing team automata according to certain natural types of synchronisation guarantees compositionality. Moreover, in [29] some preliminary work on model checking team automata using Spin was carried out.

Finally, an important component of groupware analysis has to do with performance and real-time issues. Consequently we plan to carry out experimentation with quantitative extensions of modelling frameworks (e.g. timed-, probabilistic-, and stochastic-automata), related specification languages (e.g. stochastic process algebras), and proper support tools for verification and formal dependability assessment (e.g. stochastic model checking [30, 31] and formal specification-driven discrete simulation tools [32]).

6 Acknowledgements

Maurice ter Beek has been partially supported by an ERCIM postdoctoral fellowship. Moreover, all four authors have been partially funded by the Italian Ministery MIUR "Special Fund

for the Development of Strategic Research" under CNR project "Instruments, Environments and Innovative Applications for the Information Society", sub-project "Software Architecture for High Quality Services for Global Computing on Cooperative Wide Area Networks".

The authors are grateful to Tore Urnes for sharing his Promela specification of part of the Clock protocol with us.

References

[1] T.C.N. Graham, T. Urnes, and R. Nejabi, Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware, Proceedings 9th ACM Symposium on User Interface Software and Technology (UIST'96), Seattle, WA (M. Brown and R. Rao, eds.), ACM Press, New York, NY (1996) 1–10.

[2] T. Urnes and T.C.N. Graham, Flexibly Mapping Synchronous Groupware Architecture to Distributed Implementations, Design, Specification and Verification of Interactive Systems'99 (D.J. Duke and A. Puerta, eds.), Springer Computer Science, Springer-Verlag, Wien (1999) 133–148.

[3] T. Urnes, Efficiently Implementing Synchronous Groupware, Ph.D. thesis, Department of Computer Science, York University, Toronto (1998).

[4] J. Grudin, CSCW: History and Focus, IEEE Computer **27, 5** (1994) 19–26.

[5] C.A. Ellis, S.J. Gibbs, and G.L. Rein, Groupware: Some Issues and Experiences, Communications of the ACM **34, 1** (1991) 38–58.

[6] R.M. Baecker (ed.), Readings in Groupware and Computer Supported Cooperation Work: Assisting Human-Human Collaboration, Morgan Kaufmann Publishers, San Mateo, CA (1992).

[7] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, and W. Wilner, The Rendezvous Architecture and Language for Constructing Multi-User Applications, ACM Transactions on Computer-Human Interaction **1, 2** (1994) 81–125.

[8] M. Roseman and S. Greenberg, Building Real Time Groupware with GroupKit, A Groupware Toolkit, ACM Transactions on Computer-Human Interaction **3, 1** (1996) 66–106.

[9] T.C.N. Graham and T. Urnes, Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces, Proceedings 19th International Conference on Software Engineering (ICSE'97), Boston, MA (R. Taylor, ed.), ACM Press (1997) 172–182.

[10] T.C.N. Graham, GroupScape: Integrating Synchronous Groupware and the World Wide Web, Proceedings International Conference on Human-Computer Interaction (INTERACT'97), Sydney, Australia (S. Howard, J. Hammon, and G. Lindgaard, eds.), Chapman & Hall, London (1997) 547–554.

[11] M. Sage and Ch. Johnson, Pragmatic Formal Design: A Case Study in Integrating Formal Methods into the HCI Development Cycle, Design, Specification and Verification of Interactive Systems'98 (P. Markopoulos and P. Johnson, eds.), Springer Computer Science, Springer-Verlag, New York (1998) 134–155.

[12] J. Brown and S. Marshall, Sharing Human-Computer Interaction and Software Engineering Design Artifacts, Proceedings 8th Australian Computer-Human Interaction Conference (OzCHI'98), Adelaide, Australia (P. Calder and B. Thomas, eds.), IEEE Computer Society Press (1998) 53–60.

[13] C. Papadopoulos, An Extended Temporal Logic for CSCW, The Computer Journal **45, 4** (2002) 453–472.

[14] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison Wesley Publishers, Reading, MA (2003).

[15] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems—Specification, Springer-Verlag, Berlin (1992).

[16] T.C.N. Graham, C.A. Morton, and T. Urnes, ClockWorks: Visual Programming of Component-Based Software Architectures, Journal of Visual Languages and Computing **7, 2** (1996) 175–196.

[17] G.E. Krasner and S.T. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming **1, 3** (1988) 26–49.

[18] E.M. Clarke Jr., O. Grumberg, and D.A. Peled, Model Checking, MIT Press, Cambridge, MA (1999).

[19] M.H. ter Beek, Team Automata—A Formal Approach to the Modeling of Collaboration Between System Components, Ph.D. thesis, Leiden Institute of Advanced Computer Science, Leiden University, Leiden (2003).

[20] A. Bouali, S. Gnesi, and S. Larosa. The Integration Project for the JACK Environment. Bulletin of the EATCS **54** (1994) 207–223. Cf. also `fmt.isti.cnr.it/jack/`

[21] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, CADP: A Protocol Validation and Verification Toolbox, Proceedings 8th International Conference on Computer-Aided Verification (CAV'96), New Brunswick, NJ (R. Alur and T.A. Henzinger, eds.), Lecture Notes in Computer Science **1102**, Springer-Verlag, Berlin (1996) 437–440. Cf. also `www.inrialpes.fr/vasy/cadp/`

[22] R. Cleaveland and S. Sims, The NCSU Concurrency Workbench, Proceedings 8th International Conference on Computer-Aided Verification (CAV'96), New Brunswick, NJ (R. Alur and T.A. Henzinger, eds.), Lecture Notes in Computer Science **1102**, Springer-Verlag, Berlin (1996) 394–397. Cf. also `www.cs.sunysb.edu/~cwb/`

[23] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol, μCRL: A Toolset for Analysing Algebraic Specifications, Proceedings 13th International Conference on Computer-Aided Verification (CAV 2001), Paris, France (G. Berry, H. Comon, and A. Finkel, eds.), Lecture Notes in Computer Science **2102**, Springer-Verlag, Berlin (2001) 250–254. Cf. also `www.cwi.nl/~mcrl/`

[24] C.A. Ellis, Team Automata for Groupware Systems, Proceedings International ACM SIGGROUP Conference on Supporting Group Work (GROUP'97), Phoenix, Arizona (S.C. Hayne and W. Prinz, eds.), ACM Press, New York, NY (1997) 415–424.

[25] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg, Synchronizations in Team Automata for Groupware Systems, Computer Supported Cooperative Work—The Journal of Collaborative Computing **12, 1** (2003) 21–69.

[26] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg, Team Automata for CSCW, Proceedings 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems, Berlin, Germany (H. Weber, H. Ehrig, and W. Reisig, eds.), Fraunhofer Institute for Software and Systems Engineering, Berlin (2001) 1–20.

[27] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg, Team Automata for Spatial Access Control, Proceedings 7th European Conference on Computer Supported Cooperative Work (ECSCW 2001), Bonn, Germany (W. Prinz, M. Jarke, Y. Rogers, K. Schmidt, and V. Wulf, eds.), Kluwer Academic Publishers, Dordrecht (2001) 59–77.

[28] M.H. ter Beek and J. Kleijn, Team Automata Satisfying Compositionality, Proceedings 12th International Symposium of Formal Methods Europe (FME 2003), Pisa, Italy (K. Araki, S. Gnesi, and D. Mandrioli, eds.), Lecture Notes in Computer Science **2805**, Springer-Verlag, Berlin (2003) 381–400.

[29] M.H. ter Beek and R.P. Bloem, Model Checking Team Automata for Access Control. Unpublished manuscript, 2003.

[30] M.Z. Kwiatkowska, G. Norman, and D. Parker, Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach, Proceedings 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002), Grenoble, France (J.-P. Katoen and P. Stevens, eds.), Lecture Notes in Computer Science **2280**, Springer-Verlag, Berlin (2002) 52–66.

[31] C. Baier, B.R. Haverkort, H. Hermanns, and J.-P. Katoen, Automated Performance and Dependability Evaluation Using Model Checking, Performance Evaluation of Complex Systems: Techniques and Tools—Performance 2002 Tutorial Lectures (M.C. Calzarossa and S. Tucci, eds.), Lecture Notes in Computer Science **2459**, Springer-Verlag, Berlin (2002) 261–289.

[32] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma, General Purpose Discrete Event Simulation Using  , Proceedings 6th International Workshop on Process Algebras and Performance Modeling (PAPM'98), Nice, France (C. Priami, ed.), Università degli Studi di Verona (1998) 85–102.

A The Complete Promela Specification

```
/* Macros */

#define maxUsers 2                 /* all bits below must become bytes when maxUsers > 2 */

/* Handshake and other communication channels */

mtype =
{
  input, view,                                                              /* User */
  requestLock, grantLock, releaseLock,                                /* locking mechanism */
  update, notify,                                                      /* data consistency */
  finished                                                             /* view consistency */
};

/* Channels between environment and client layer */

chan userToUpdater[maxUsers] = [1] of {byte};    /* not [0] due to nempty test in Updater */
chan vcToUser[maxUsers] = [0] of {byte};

/* Internal client layer channels */

chan updaterToVC[maxUsers] = [0] of {byte};
chan vcToUpdater[maxUsers] = [0] of {byte};

/* Channels between client layer and server layer */

chan updaterToCC = [6] of {byte, bit};       /* not [0] due to FIFO scheduling in Updater */
chan ccToUpdater[maxUsers] = [1] of {byte};                /* not [0] due to atomic in CC */

/* Internal server layer channels */

chan ccToModel = [0] of {byte, bit};
chan modelToCC = [0] of {byte, bit};

/* Environment processes */

proctype User(byte id)
{
  do
  :: userToUpdater[id]!input;                              /* User sends Input to Updater */
     doneInput: skip;                        /* (progress label for verification purposes) */
     doneInput2: skip                        /* (progress label for verification purposes) */
  :: vcToUser[id]?view;                                    /* User notes new View from VC */
     doneView: skip                          /* (progress label for verification purposes) */
  od
}

/* Server layer processes */

proctype Model()
{
  byte id;

  do
  :: ccToModel?update,_ ->                    /* Upon receiving any User Update from CC, */
       atomic
       {
         id = 0;
         do
         :: (id < maxUsers) ->
              modelToCC!notify,id;            /* Model responds with Notify to each User */
              id++
         :: else -> break
         od
       }
  od
}
```

```
proctype ConcurrencyController()
{
  chan lockQ = [maxUsers] of {bit};
  mtype msg;
  bool writeLock = false;
  byte id, numNotified = 0;

  do               /* if at least one user requested the lock and there is no writeLock, */
  :: (nempty(lockQ) && numNotified == 0 && !writeLock) ->                    /* then CC */
       lockQ?id;                            /* checks which User first did a requestLock, */
       assert(writeLock == false);               /* (assertion for verification purposes) */
       ccToUpdater[id]!grantLock;           /* sends a grantLock for the User to Updater, */
       writeLock = true                                 /* and sets writeLock to be true */
  :: updaterToCC?msg,id ->
       if
       :: (msg == update) ->                  /* Upon receiving User Update from Updater */
            ccToModel!update,id;              /* CC forwards User Update to Model and CC */
            atomic
            {
              id = 0;
              do
              :: (id < maxUsers) ->
                   modelToCC?notify,id;              /* waits for Notify from Model, 1 a */
                   ccToUpdater[id]!notify;           /* User, forwards Notify to Updater */
                   numNotified++;                    /* and finally augments numNotified */
                   id++
              :: else -> break
              od
            }
       :: (msg == finished) ->            /* Upon receiving Finished from Updater, CC has */
            numNotified--                 /* 1 more User with View recomputation Finished */
       :: (msg == requestLock) ->            /* CC notes User id requestLock from Updater */
             lockQ!id                        /* and hence must handle User id requestLock */
       :: (msg == releaseLock) ->                    /* CC notes releaseLock from Updater */
            writeLock = false                      /* and hence sets writeLock to false */
       fi
  od
}

/* Client layer processes */

proctype ViewController(byte id)
{
  do
  :: updaterToVC[id]?input ->                   /* Upon receiving User Input from Updater */
       checkInput: skip;                    /* (progress label for verification purposes) */
       vcToUpdater[id]!update                    /* VC responds by returning a User Update */
  :: updaterToVC[id]?notify ->              /* Upon receiving a Notify of Update from VC, */
       checkNotify: skip;                   /* (progress label for verification purposes) */
       vcToUser[id]!view;                   /* VC informs both User (by sending new View) */
       vcToUpdater[id]!finished             /* and Updater of View recomputation Finished */
  od
}

proctype Updater(byte id)
{
  mtype msg;
  bool canProcessInput = true, waitingForLock = false;

  do                                                    /* Updater notes User Input and */
  :: (canProcessInput && !waitingForLock && nempty(userToUpdater[id])) ->
       updaterToCC!requestLock,id;                      /* does the requestLock from CC */
       doneRequestLock: skip;               /* (progress label for verification purposes) */
       d_step{canProcessInput = false;              /* It cannot process more User Input */
       waitingForLock = true}                       /* if waiting for lock to be granted */
  :: vcToUpdater[id]?msg ->
       if                                        /* Updater deals with message from VC */
       :: (msg == update) ->                          /* Updater notes Update from VC */
            updaterToCC!update,id;                         /* so forwards User Update */
```

```
            updaterToCC!releaseLock,id;                        /* and a releaseLock to CC */
            doneReleaseLock: skip;          /* (progress label for verification purposes) */
       :: (msg == finished) ->                            /* Updater notes Finished from VC */
            updaterToCC!finished,id;                         /* and it thus informs CC that */
            canProcessInput = !waitingForLock                /* View recomputation Finished */
       fi
  :: ccToUpdater[id]?msg ->
       if                                                 /* Updater deals with message from CC */
       :: (msg == grantLock) ->                          /* as Updater notes grantLock from CC, */
            doneGrantLock: skip;            /* (progress label for verification purposes) */
            userToUpdater[id]?input;                    /* it finally receives Input from User */
            doneInput: skip;                /* (progress label for verification purposes) */
            updaterToVC[id]!input;                      /* Updater forwards User Input to VC & */
            waitingForLock = false                      /* is hence no longer waiting for lock */
       :: (msg == notify) ->                                  /* Updater notes Notify from CC */
            updaterToVC[id]!msg                               /* & thus forwards Notify to VC */
       fi
  od
}

/* Initialisation process */

init
{
  byte Users = 0;

  atomic
  {
    run Model();                                                          /* pid = 1 */
    run ConcurrencyController();                                          /* pid = 2 */
    run Updater(Users);                                  /* Updater(0) with pid = 3 */
    run ViewController(Users);                                /* VC(0) with pid = 4 */
    run User(Users);                                        /* User(0) with pid = 5 */
    Users++;
    do
    :: (Users < maxUsers) ->
         run Updater(Users);                             /* Updater(1) with pid = 6 */
         run ViewController(Users);                           /* VC(1) with pid = 7 */
         run User(Users);                                   /* User(1) with pid = 8 */
         Users++
    :: else -> break
    od
  }
}
```

B The Complete Promela Specification Extended with a User ID

```
/* Macros */

#define maxUsers 2                 /* all bits below must become bytes when maxUsers > 2 */

/* Handshake and other communication channels */

mtype =
{
  input, view,                                                                  /* User */
  requestLock, grantLock, releaseLock,                                /* locking mechanism */
  update, notify,                                                      /* data consistency */
  finished                                                             /* view consistency */
};

/* Channels between environment and client layer */

chan userToUpdater[maxUsers] = [1] of {byte};    /* not [0] due to nempty test in Updater */
chan vcToUser[maxUsers] = [0] of {byte};

/* Internal client layer channels */
```

```
chan updaterToVC[maxUsers] = [0] of {byte, bit};
chan vcToUpdater[maxUsers] = [0] of {byte, bit};

/* Channels between client layer and server layer */

chan updaterToCC = [6] of {byte, bit, bit};  /* not [0] due to FIFO scheduling in Updater */
chan ccToUpdater[maxUsers] = [1] of {byte, bit};          /* not [0] due to atomic in CC */

/* Internal server layer channels */

chan ccToModel = [0] of {byte, bit};
chan modelToCC = [0] of {byte, bit, bit};

/* Environment processes */

proctype User(byte id)
{
  do
  :: userToUpdater[id]!input;                              /* User sends Input to Updater */
     doneInput: skip;                        /* (progress label for verification purposes) */
     doneInput2: skip                        /* (progress label for verification purposes) */
  :: vcToUser[id]?view;                                    /* User notes new View from VC */
     doneView: skip                          /* (progress label for verification purposes) */
  od
}

/* Server layer processes */

proctype Model()
{
  byte id;
  bit ID;

  do
  :: ccToModel?update,ID ->              /* Upon receiving an Update of User ID from CC, */
       if
       :: (ID == 0) ->
            doneUpdate0: skip              /* (progress label for verification purposes) */
       :: (ID == 1) ->
            doneUpdate1: skip              /* (progress label for verification purposes) */
       fi;
       atomic
       {
         id = 0;
         do
         :: (id < maxUsers) ->                               /* Model responds with Notify */
              modelToCC!notify,id,ID;                        /* to each User and indicates */
              id++                                           /* also which User did Update */
         :: else -> break
         od
       }
  od
}

proctype ConcurrencyController()
{
  chan lockQ = [maxUsers] of {bit};
  mtype msg;
  bool writeLock = false;
  byte id, numNotified = 0;
  bit ID;

  do               /* if at least one user requested the lock and there is no writeLock, */
  :: (nempty(lockQ) && numNotified == 0 && !writeLock) ->                      /* then CC */
       lockQ?id;                               /* checks which User first did requestLock, */
       assert(writeLock == false);                /* (assertion for verification purposes) */
       ccToUpdater[id]!grantLock,id;           /* sends grantLock for the User to Updater, */
       writeLock = true                                    /* and sets writeLock to be true */
  :: updaterToCC?msg,id,ID ->
```

```
      if
      :: (msg == update) ->                  /* Upon receiving User Update from Updater */
           ccToModel!update,id;              /* CC forwards User Update to Model and CC */
           if
           :: (ID == 0) ->
                doneUpdate0: skip;         /* (progress label for verification purposes) */
                doneUpdate02: skip         /* (progress label for verification purposes) */
           :: (ID == 1) ->
                doneUpdate1: skip;         /* (progress label for verification purposes) */
                doneUpdate12: skip         /* (progress label for verification purposes) */
           fi;
           atomic
           {
             id = 0;
             do
             :: (id < maxUsers) ->                          /* waits for each User for */
                  modelToCC?notify,id,ID;                   /* Notify from Model which */
                  ccToUpdater[id]!notify,ID;                /* it forwards to Updater, */
                  numNotified++;                            /* plus User ID of Update, */
                  id++                                      /* & augments numNotified  */
             :: else -> break
             od
           }
      :: (msg == finished) ->            /* Upon receiving Finished from Updater, CC has */
           if
           :: (id == 0 && ID == 0) ->
                doneFinished00: skip       /* (progress label for verification purposes) */
           :: (id == 0 && ID == 1) ->
                doneFinished01: skip       /* (progress label for verification purposes) */
           :: (id == 1 && ID == 0) ->
                doneFinished10: skip       /* (progress label for verification purposes) */
           :: (id == 1 && ID == 1) ->
                doneFinished11: skip       /* (progress label for verification purposes) */
           fi;
           numNotified--                 /* 1 more User with View recomputation Finished */
      :: (msg == requestLock) ->            /* CC notes User id requestLock from Updater */
           lockQ!id                         /* and hence must handle User id requestLock */
      :: (msg == releaseLock) ->                    /* CC notes releaseLock from Updater */
           writeLock = false                        /* and hence sets writeLock to false */
      fi
  od
}

/* Client layer processes */

proctype ViewController(byte id)
{
  bit ID;

  do
  :: updaterToVC[id]?input,id ->              /* Upon receiving User Input from Updater */
       checkInput: skip;                    /* (progress label for verification purposes) */
       vcToUpdater[id]!update,id             /* VC responds by returning a User Update */
  :: updaterToVC[id]?notify,ID ->           /* Upon receiving a Notify of Update from VC, */
       checkNotify: skip;                   /* (progress label for verification purposes) */
       vcToUser[id]!view;                   /* VC informs both User (by sending new View) */
       vcToUpdater[id]!finished,ID          /* and Updater of View recomputation Finished */
  od
}

proctype Updater(byte id)
{
  mtype msg;
  bool canProcessInput = true, waitingForLock = false;
  bit ID;

  do                                                    /* Updater notes User Input and */
  :: (canProcessInput && !waitingForLock && nempty(userToUpdater[id])) ->
       updaterToCC!requestLock,id,id;                   /* does the requestLock from CC */
       doneRequestLock: skip;              /* (progress label for verification purposes) */
```

```
       d_step{canProcessInput = false;              /* It cannot process more User Input */
       waitingForLock = true}                       /* if waiting for lock to be granted */
  :: vcToUpdater[id]?msg,ID ->
       if                                          /* Updater deals with message from VC */
       :: (msg == update) ->                             /* Updater notes Update from VC */
            updaterToCC!update,id,id;                      /* so forwards User Update */
            updaterToCC!releaseLock,id,id;                 /* and a releaseLock to CC */
            doneReleaseLock: skip;          /* (progress label for verification purposes) */
       :: (msg == finished) ->                   /* Updater notes from VC that User ID */
            updaterToCC!finished,id,ID;          /* Finished View recomputation, hence */
            canProcessInput = !waitingForLock;   /* it tells CC & User canProcessInput */
       fi
  :: ccToUpdater[id]?msg,ID ->
       if                                          /* Updater deals with message from CC */
       :: (msg == grantLock) ->                   /* as Updater notes grantLock from CC, */
            doneGrantLock: skip;            /* (progress label for verification purposes) */
            userToUpdater[id]?input;             /* it finally receives Input from User */
            doneInput: skip;                /* (progress label for verification purposes) */
            updaterToVC[id]!input,id;            /* Updater forwards User Input to VC & */
            waitingForLock = false               /* is hence no longer waiting for lock */
       :: (msg == notify) ->                          /* Updater notes Notify from CC */
            if
            :: (ID == 0) ->
                 doneNotify0: skip          /* (progress label for verification purposes) */
            :: (ID == 1) ->
                 doneNotify1: skip          /* (progress label for verification purposes) */
            fi;
            updaterToVC[id]!msg,ID                      /* & thus forwards Notify to VC */
       fi
  od
}

/* Initialisation process */

init
{
  byte Users = 0;

  atomic
  {
    run Model();                                                          /* pid = 1 */
    run ConcurrencyController();                                          /* pid = 2 */
    run Updater(Users);                                  /* Updater(0) with pid = 3 */
    run ViewController(Users);                               /* VC(0) with pid = 4 */
    run User(Users);                                        /* User(0) with pid = 5 */
    Users++;
    do
    :: (Users < maxUsers) ->
         run Updater(Users);                             /* Updater(1) with pid = 6 */
         run ViewController(Users);                          /* VC(1) with pid = 7 */
         run User(Users);                                   /* User(1) with pid = 8 */
         Users++
    :: else -> break
    od
  }
}
```