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. The enormous improvements in the efficiency of model-checking techniques in recent years facilitates their application to ever more complex systems of concurrent and distributed nature. Many of the protocols underlying groupware systems need to deal with those aspects as well, which makes them notoriously hard to analyse on paper or by traditional means such as testing and simulation. Model checking allows for the automatic analysis of correctness and liveness properties in an exhaustive and time-efficient way, generating counterexamples in case certain properties are found not to be satisfied. In this paper 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 network communication, concurrency control, and distributed notification aspects. In particular, we address key issues related to concurrency control, data consistency, view consistency, and absence of (user) starvation. 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]. Our focus is on groupware allowing real-time collaboration, which is also called synchronous groupware. Examples include video conferencing, collaborative writing, and multi-user games. Synchronous groupware is inherently distributed in nature and the design of its underlying protocols thus needs to address the intricated behaviour based on network communication, concurrency control, and distributed notification. This has led 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 [6], GroupKit [7], and Clock [1, 2].

In this paper we look at the Clock toolkit, mainly because its underlying protocols have been formally specified in [3]. Clock has been used to develop a number of groupware applications, such as a multi-user video annotation tool [8], the multi-user GroupScape HTML browser [9], a multi-user design rationale editor [10], and the ScenicVista user interface design tool [11]. We analyse several of Clock's underlying groupware protocols, among which those concerned with its concurrency control and distributed notification aspects. The analyses of the correctness and performance of these protocols with traditional techniques, such as testing and simulation, may not reveal all possible problems and are usually very time consuming. We thus propose the use of model checking, which has recently become mature enough to address problems of industrial size. Model checking is an automated technique for verifying whether a logical property holds for a finite-state model of a distributed system. In [3], an attempt was made to use model checking for verifying the correctness of Clock's protocols. Due to the level of detail of that specification and the capabilities of model-checking tools at that time, the state space was too large to handle and the attempt thus had very limited success. We revisit this work by developing a more abstract specification (model) of the concurrency control and distributed notification aspects of Clock that nevertheless covers faithfully many issues of interest. We show that this model is very well amenable to model checking by addressing the formalisation and verification of several issues specifically of interest for the correctness of groupware protocols in general, i.e. not limited to those underlying Clock. We focus on key issues related to concurrency control, data consistency, view consistency, and absence of (user) starvation. As a result we thus contribute to the verification of Clock's protocols.

In a broader context, our work shows that with relatively simple models one can verify highly relevant properties of groupware protocols with currently freely available verification tools, such as the model checker Spin [12]. The properties we verify are mostly formalised as formulae of a Linear Temporal Logic (LTL for short) [13], reflecting 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.

Although time-performance issues are very important in groupware systems [14], 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.

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 an overview of the Clock toolkit and its underlying protocols. For more information or to obtain Clock, cf. `www.cs.queensu.ca/~graham/clock.htm`.

The Clock toolkit is a high-level groupware toolkit that is supported by the visual Clock-Works [15] programming environment and which has a design-level architecture based on the Model-View-Controller (MVC for short) paradigm of [16]. 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.

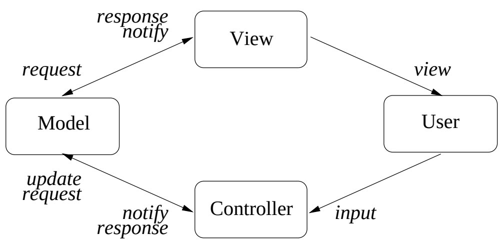

Figure 1: The MVC architecture and its communication protocol.

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 version of this protocol capturing its behavioural aspects but leaving out many implementation details was formalised in the specification language Promela and an attempt was made to verify it with the model checker Spin [12]. Partly due to insufficient computing resources, however, it was impossible to verify the entire protocol. Consequently, an attempt was made to verify only a part of the protocol still large enough to be relevant, viz. the part relevant to concurrency control and distributed notification. Unfortunately also this attempt was largely unsuccessful as only 2% of the total state space was covered. We make some further abstractions 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. Since these issues are present in many groupware systems, we aim at providing a sort of reference specification that can be used as starting point for the specification of variants of the protocols considered here.

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.

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

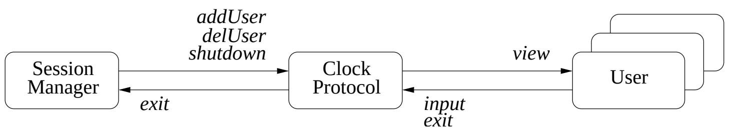

Figure 2: The Clock protocol embedded in its environment.

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 full Clock protocol, we abstract from it and focus on those protocols that are fundamental to that aspect of the Clock that we want to verify. Since in this paper we are interested in its concurrency control and distributed notification aspects rather than in its data aspects, we thus focus on the MVC 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 only providing mutual exclusion. 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, we thus address the MVC and the CC protocol, and we assume that the latter is implemented by the locking mechanism. Hence we consider the part of the Clock protocol and its environment as depicted in Figure 3.

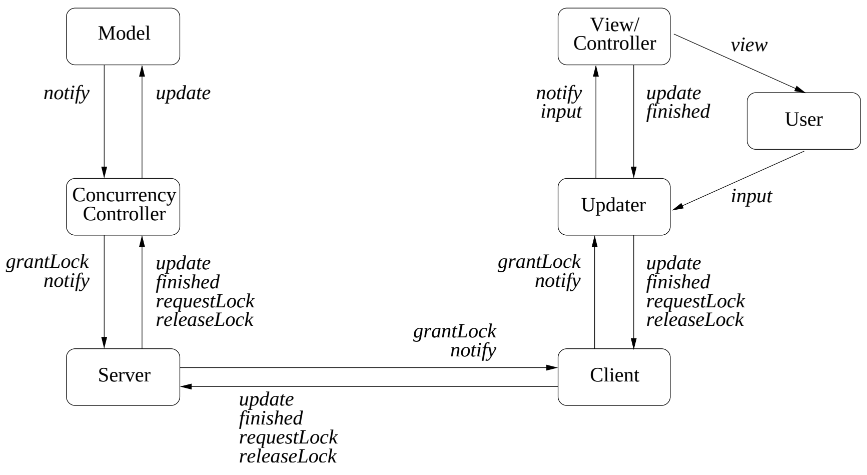

Figure 3: The part of the Clock protocol and its environment relevant to the locking mechanism.

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 such 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* and returns it 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 next 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 an overview of the abstractions we have applied to the specification of [3].

Model checking is an automated technique for verifying whether a logical property holds for a finite-state model of a distributed system [17]. Such verifications are moreover exhaustive, i.e. all possible input combinations and states are taken into account—a task that is impossible to undertake by hand. This makes model checking ideally suited for verifying whether a system design satisfies its specifications. To avoid to run out of memory due to a state-space explosion—which would make an exhaustive verification impossible—usually a model of the system is used that abstracts as much as possible from implementation details, while still capturing the essence of the behavioural aspects of the system.

One of the best known and most successful model checkers is Spin, which was developed at Bell Labs during the last two decades [12]. 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 also because of the aforementioned earlier attempt at verifying a version of the Clock protocol with Spin in [3], which moreover contains a specification of it in Spin's input language Promela.

Promela is a non-deterministic C-like specification language for modelling finite-state systems communicating through channels [12]. 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 above two 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 [12].

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 certain states are reachable, or whether 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 specification just before a lock is granted and let Spin verify whether there are any assertion violations. If Spin concludes that this assertion may be violated, then it also presents a counterexample showing the model's undesired behaviour. Otherwise it simply reports that there are no assertion violations, meaning that the property is satisfied by the model.

Secondly, we may add *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). A system computation is modelled by a sequence of states and the *behaviour* of a system is the set of all such sequences. 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. A system (Promela model) satisfies a formula if and only if the formula is `true` for all sequences of its behaviour. For more detailed information on LTL, we refer the reader to [13].

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 a user can always eventually provide input. The User process is specified as follows in Promela.

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

It consists of a `do`-loop with two statements that can be chosen non-deterministically every time the loop is entered. The first models the user sending an *input* to the Updater, while the second models the user receiving a *view* update from the VC. Here `id` identifies the entry of the userToUpdater and vcToUser arrays of (buffered) channels. Imagine that we add the label `doneInput` to this specification directly following a user input, i.e. just after the statement `userToUpdater[id]!input`. Then 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 every sequence of states from the behaviour of the specification of the MVCC protocol contains a state in which this user's state is the 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 (one for each user) by running Spin. Again, if Spin concludes that this statement is not valid, then it also presents a counterexample showing a computation of the system during which the particular user is never able to perform an *input*. Otherwise it simply reports that the statement is valid.

3.1 The Promela Specification

In this section we discuss the Promela specification of the MVCC protocol of which we intend to validate a number of properties in the next section. Our starting point is the Promela specification of the aforementioned simplified Clock protocol as given in [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 Clock has led to the MVCC protocol as an abstraction of the Clock protocol. From the Promela specification given in [3] we have thus omitted the parts concerning session management, various forms of replication and caching, and the eager concurrency control mechanism. Since none of these parts interfered with either the concurrency control algorithm underlying the CC protocol or the distributed notification algorithm underlying the MVC protocol, their removal does not alter the behaviour of these algorithms. These changes led to a reduction of the total number of processes, data objects, and message channels, and thus to a reduction of the state space and thereby the risk to run out of memory during verification.

Subsequently we have worked on a further reduction of the state vector. 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 fewer bytes that Spin needs to store for each system state.

Finally, in [12] 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 where possible. More precisely, all administrative statements (e.g. updating bits or booleans) have been grouped into `d_step`s. These are then treated as one deterministic sequence of code that is executed indivisibly, i.e. as if it were one statement. This thus reduces the size of the state space, where all interleaving executions are considered.

The resulting Promela specification 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. Summarising, we thus 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. 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 represent processes communicating by sending variables through channels in the complete Promela specification, which can be found in [18].

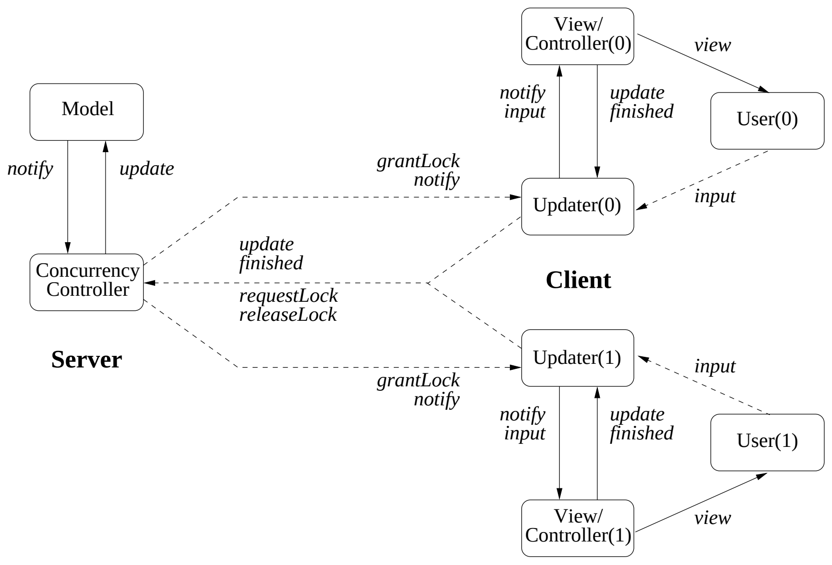

Figure 4: The modified MVCC protocol in case of two users.

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 verifying a number of core issues of the MVCC protocol with Spin. All verifications have been performed by running Spin Version 4.0.4 on a SUN® Netra™ X1 workstation with 1 Gigabyte of available physical memory.

First we have let Spin perform a full state-space search for invalid endstates, which is Spin's formalisation of deadlock states. It took Spin just two seconds to conclude that there are no deadlocks, which is an enormous reduction compared to the hour it took in [3] to cover only 2% of the total state space. From this we gain a lot of confidence w.r.t. the verifiability of future extensions of our Promela specification, e.g. by adding the protocols concerning session management, various forms of replication and caching, or concurrency control based on the eager mechanism. To begin with, however, we want to assure that the MVCC protocol satisfies some core properties of 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 re-computation must have finished before another update of the shared data state takes 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; only one user at a time may be in the possession of a lock; every obtained lock must eventually be released.

In the subsequent sections we verify all the above properties for the modified MVCC protocol in case of two users by using Spin, the Promela specification given in [18], 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. This gives us more confidence on the correct design of the MVCC protocol than, e.g., traditional analysis techniques such testing and (symbolic) simulation, which usually cover only a part of the reachable state space.

4.2 Concurrency Control

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

First it must be the case that every lock request is eventually granted. To verify this we have added the label `doneRequestLock` to the specification of the Updater at the point where it sends a *requestLock*, accompanied by the `id` of the user requesting it, to the CC (`updaterToCC!requestLock,id`) and the label `doneGrantLock` at the point where it receives a *grantlock* from the CC (`ccToUpdater[id]?grantLock`). Consequently, 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 it is always the case that whenever an updater has requested a lock on behalf of a user, it eventually grants this user a lock. It takes Spin just a few minutes to conclude that these formulae are valid.

Secondly, it must be the case that the CC may have granted only one lock at a time. Therefore, the basic assertion `assert(writeLock == false)` was added to the CC's specification there where it sends the Updater a *grantLock* (`ccToUpdater[id]!grantLock`) and then we have let Spin run a verification on assertion violations. We thus verify whether 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 this *grantLock*. Again, in just a few seconds Spin concludes that the above basic assertion is never violated.

Thirdly, it must be the case that every obtained lock is eventually released. To verify this we have added the label `doneReleaseLock` to the specification of the Updater at the point where it sends the CC a *releaseLock* (`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 it is always the case that whenever an updater has obtained a lock on a user's behalf, then it eventually releases this lock. It takes Spin a few minutes to conclude that also these formulae are valid.

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 specification used so far does not contain enough information to verify data consistency. This is due to the fact that adding a label `doneUpdate` to the specification of the Model there where it receives an *update* from the CC (`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 our 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 [18]. We have added the labels `doneUpdate0` and `doneUpdate1` to the extended specification of the Model just after the statement `ccToModel?update,ID` in the form of an `if`-statement, guaranteeing 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)). Likewise, we have added the labels `doneNotify0` and `doneNotify1` to the extended specification of the Updater at the point where it receives a *notify* from the CC (`ccToUpdater[id]?notify,ID`) in the form of an `if`-statement which guarantees that `doneNotify0` is passed if the `ID` in the statement `ccToUpdater[id]?notify,ID` equals 0 (for User(0)), whereas `doneNotify1` is passed if it equals 1 (for User(1)).

Subsequently, in order 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 label `doneInput` to the extended specification of the Updater at the point where it receives an *input* from the User (`userToUpdater[id]?input`) and then we have let Spin verify 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 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 several 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

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

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 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 about a quarter of 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 another update of the shared data state occurs (and triggers 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 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 specification of the CC the labels `doneInputY` and `doneInputY2`, where `Y` is 0 (for an *update* from Updater(0)) or 1 (for an *update* from Updater(1)), in this order at the point where the CC sends an *update* to the Model (`ccToModel!update,id`) and the 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)), to the extended specification of the Updater at the point where it sends a *finished* to the CC (`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 almost an hour to conclude that these 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`, this time however to prohibit a user to provide input (i.e. pass the `doneInput` label) before both user's views have been recomputed (i.e. both have passed the `doneView` label). We thus needed to distinguish the 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` label. Therefore we have added the label `doneInput2` to the Promela specification of the User just after the label `doneInput` and then we have let Spin run verifications of the LTL formulae

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

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 few seconds to conclude that these formulae are not valid! It in fact presents counterexamples, one of which we have sketched in Figure 5.

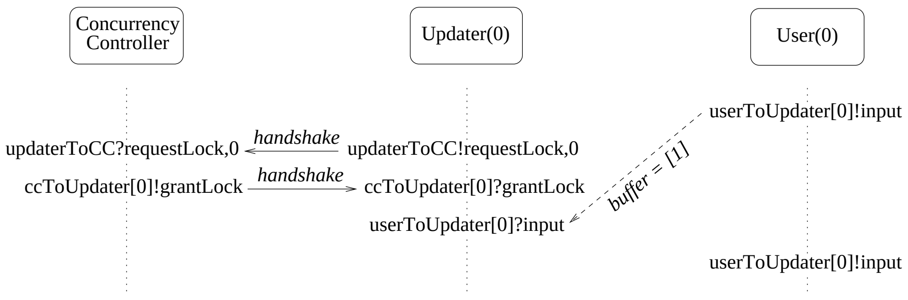

Figure 5: The message sequence chart of a counterexample for view consistency.

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 is thus whether each user can infinitely often 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 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. Even if we use Spin's weak fairness notion, defined below, then the above formulae are not valid. 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 now verify that any user's input must result in a lock request, which eventually should be granted. We already know that every lock request is eventually granted. To verify that a user input eventually results in a lock request, we have let Spin verify 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 it is always the case that whenever a user provides input, then its associated updater eventually requests a lock. It takes Spin just several seconds to conclude that the above LTL formulae are not valid. This does not come as a surprise, 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 (that are the result of the other user's continuous stream of *input*s).

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 however be guaranteed.

4.6 Spin and Fairness

In the previous section we have seen that, given the specification of the MVCC protocol in [18], 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 statement `vcToUser[id]?view` updating a user's view over that of the statement `userToUpdater[id]!input` performing a user's input (recall the specification of the User process from Section 3). The reason for this is the way fairness is implemented in Spin, viz. at the process 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 [12] it is suggested to enforce weak fairness by specifying the desired fairness constraint c as an LTL formula and consequently verifying whether 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 encountered above, we had to specify a constraint guaranteeing that both users are equally given the possibility to perform input. Therefore we have added the label `checkInput` to the specification of the VC there where it receives an *input* from the Updater (`updaterToVC[id]?input`) and the label `checkNotify` there where it receives a *notify* from the Updater (`updaterToVC[id]?notify`). 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 a user can always provide input under the condition that its VC checks for update notifications and user input in a fair way, i.e. whenever it checks for a *notify* from an updater, then it always eventually also checks for an *input* from the corresponding user. It takes Spin just over three quarters of an hour 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 guaranteeing fair treatment of enabled options.

5 Conclusion

In this paper we have shown that 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 concurrency control, data consistency, view consistency, and absence of (user) starvation. As a result, we contribute to the verification of some of Clock's underlying groupware protocols, which was attempted earlier in [3] with very limited success.

In the future we plan to prove 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 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. The combination of compositionality and 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 [19], CADP [20], and CWB-NC [21]). Team automata were introduced explicitly for the description and analysis of groupware systems and their interconnections [22] and were shown to be useful in a variety of groupware settings [23]. A key feature of team automata is the intrinsic flexibility of their synchronisation operators. In [24] it was shown that constructing team automata according to certain natural types of synchronisation guarantees compositionality. Moreover, in [25] 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 [26]).

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", subproject "Software Architecture for High Quality Services for Global Computing on Cooperative Wide Area Networks".

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

References

[1] T.C.N. Graham, T. Urnes, and R. Nejabi, Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware, Proceedings UIST'96, ACM Press, New York, NY (1996) 1–10.

[2] T. Urnes and T.C.N. Graham, Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations, Proceedings DSVIS'99, 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.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.

[7] 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.

[8] T.C.N. Graham and T. Urnes, Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces, Proceedings ICSE'97, ACM Press (1997) 172–182.

[9] T.C.N. Graham, GroupScape: Integrating Synchronous Groupware and the World Wide Web, Proceedings INTERACT'97, Chapman & Hall, London (1997) 547–554.

[10] M. Sage and Ch. Johnson, Pragmatic Formal Design: A Case Study in Integrating Formal Methods into the HCI Development Cycle, Proceedings DSVIS'98, Springer-Verlag, New York (1998) 134–155.

[11] J. Brown and S. Marshall, Sharing Human-Computer Interaction and Software Engineering Design Artifacts, Proceedings OzCHI'98, IEEE Computer Society Press (1998) 53–60.

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

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

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

[15] 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.

[16] 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.

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

[18] M.H. ter Beek, M. Massink, D. Latella, and S. Gnesi, Model Checking Groupware Protocols. Technical Report 2003-TR-61, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 2003. Available through `fmt.isti.cnr.it`

[19] 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/`

[20] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu, CADP: A Protocol Validation and Verification Toolbox, Proceedings CAV'96, LNCS **1102**, Springer-Verlag, Berlin (1996) 437–440. Cf. also `www.inrialpes.fr/vasy/cadp/`

[21] R. Cleaveland and S. Sims, The NCSU Concurrency Workbench, Proceedings CAV'96, LNCS **1102**, Springer-Verlag, Berlin (1996) 394–397. Cf. also `www.cs.sunysb.edu/~cwb/`

[22] 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.

[23] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg, Team Automata for Spatial Access Control, Proceedings ECSCW 2001, Kluwer Academic Publishers, Dordrecht (2001) 59–77.

[24] M.H. ter Beek and J. Kleijn, Team Automata Satisfying Compositionality, Proceedings FME 2003, LNCS **2805**, Springer-Verlag, Berlin (2003) 381–400.

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

[26] 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, LNCS **2459**, Springer-Verlag, Berlin (2002) 261–289.