Model Checking Publish/Subscribe Notification for thinkteam®

Maurice H. ter Beek [a] Mieke Massink [a] Diego Latella [a]
Stefania Gnesi [a] Alessandro Forghieri [b] Maurizio Sebastianis [b]

[a] *Istituto di Scienza e Tecnologie dell'Informazione, CNR*
Area della Ricerca di Pisa, Via G. Moruzzi 1, 56124 Pisa, Italy
{maurice.terbeek,mieke.massink,diego.latella,stefania.gnesi}@isti.cnr.it

[b] think3, *Inc.—European Headquarters, Via Ronzani 7/29, 40033 Bologna, Italy*
{alessandro.forghieri,maurizio.sebastianis}@think3.com

Abstract

This paper reports on the fruitful combination of academic experience with formal modelling techniques and industrial experience with requirements exploration. We study the addition of a publish/subscribe notification service to thinkteam, [1] a ready-to-use Product Data Management application developed by think3. thinkteam allows enterprises to capture, organise, automate, and share engineering product information and it is an example of an asynchronous and dispersed groupware system. We define an abstract specification (model) of the groupware protocol underlying thinkteam and augment it with a publish/subscribe notification service. Consequently, we show a number of important correctness properties of the thinkteam model, some of which are also relevant to groupware protocols in general. In particular, we show that by adding a publish/subscribe notification service to thinkteam, the user's awareness of the status of the development of the engineering product and the activities of the design team increases.

Key words: publish/subscribe notification, thinkteam, model checking, groupware, awareness, concurrency control.

1 Introduction

Computer Supported Cooperative Work (CSCW) is an interdisciplinary research field which deals with the understanding of how people work together, and the ways in which computer technology can assist them [13]. This technology mostly consists of multi-user computer systems called groupware (systems) [1,10]. Groupware is typically classified according to two dichotomies,

[1] thinkteam is a registered trademark of think3, Inc. For details: http://www.think3.com.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

viz. (1) whether its users work together at the same time (synchronous) or at different times (asynchronous) and (2) whether they work together in the same place (co-located) or in different places (dispersed). This is called the time space taxonomy by Ellis et al. [10]. In this paper we deal with thinkteam, which is an asynchronous and dispersed groupware system. Other examples include electronic mail, workflow, collaborative writing systems, and the version-control systems often used in software engineering to coordinate the changes made by multiple programmers to the same program.

Some important design issues in groupware systems are data sharing, user awareness, and concurrency control. In this paper we address these issues in the context of thinkteam. More precisely, we use model checking to formalise and verify a number of properties specifically of interest for the correctness of groupware protocols in general, i.e. not limited to the context of thinkteam. In recent years there has been an increasing interest in the use of model checking for the formal verification of (properties of) groupware [3,17,21] and publish/subscribe (pub/sub) systems [6,7,12,20].

thinkteam is think3's Product Data Management (PDM) application catering the product/document management needs of design processes in the manufacturing industry. Its main strengths are a rapid deployment and startup cycle, its flexibility, and a seamless integration with thinkdesign—think3's CAD solution—as well as with other third party products. thinkteam allows enterprises to manage the capturing, organising, automating, and sharing of engineering product information in an efficient way. In this paper we study the addition of a lightweight and easy-to-use pub/sub notification service to thinkteam. The goal of adding such a service to an application is to increase user awareness by intelligent data sharing: whenever a user publishes a document by sending it to a centralized repository, automatically all users that are subscribed to that document are asynchronously notified via a multicast communication. Due to a potentially large number of users, it is fundamental to use subscription-based multicast communication rather than broadcast communication. Other examples of applications of a pub/sub notification service include electronic auctions on the Internet and email alert services for new journal or book releases that many publishing houses offer nowadays.

Pub/sub notification decouples the communication among users: a user that publishes a document need not be concerned with whom the server will send a notification to, i.e. the users communicate through the server. Users need not actively participate in the notification in a synchronous way. In fact, the main strength of a pub/sub notification service is said to be the "full decoupling of the communicating participants in time, space and flow" [11]. Apart from these advantages, systems with a pub/sub notification service are generally difficult to verify [12,22]. The main reason for this is the inherent non-determinism in the order of notifications, which translates to a large number of possible interleavings and often results in a combinatorially too large number of possible system executions to verify.

Before presenting in detail the proposed pub/sub notification service for thinkteam, we define an abstract specification (model) of the thinkteam's underlying groupware protocol—which nevertheless covers faithfully its most important issues—and augment it with the pub/sub notification service. We show that this model is amenable to model checking by addressing the formalisation and verification of several issues of interest for the correctness of groupware protocols in general, i.e. not limited to those underlying thinkteam. In particular, we address key issues related to concurrency control and the issue of awareness through pub/sub notification. A related approach can be found in [19], where a case study in the automatic derivation of correct integration code for assembling a set of thinkteam's (software) components is reported.

Awareness is a frequently used, but seldom precisely defined notion from the field of CSCW [18]. Roughly speaking, it should be understood as users having a sense of the (past, current, future) activities of other users—without direct communication—and using this as context for their own activities [8,14]. The goal of increasing awareness is to help users coordinate their collaborative tasks. Concurrency control, on the other hand, is a well-known notion from computer science, referring to achieving the maximum degree of parallelism under a correctness criterion. Within groupware systems, the goal of concurrency control is to resolve conflicting user actions, while still allowing the users to perform their collaborative tasks in a tightly coupled manner [9].

In this paper we show that with relatively simple models we can verify highly relevant properties of groupware protocols with verification tools like the model checker Spin [15]. The properties we verify are mostly formalised as formulae of a Linear Temporal Logic (LTL) [16]. The specification (model) of thinkteam's underlying groupware protocol has been developed in close collaboration with think3 and it is their intention to use it as basis for the planned implementation of a pub/sub notification service in thinkteam. This paper thus reports on an ongoing cooperation between academy and industry.

We begin this paper with a brief description of thinkteam and its underlying protocol, followed by a discussion of the specification of this protocol in Spin's input language Promela. Subsequently we specify and verify a number of core issues of this protocol. Finally, we conclude with a discussion of future work.

2 thinkteam

In this section we present a brief overview of thinkteam. For more information we refer the reader to [4] and http://www.think3.com/products/tt.htm.

The design process in the manufacturing industry involves a vast number of activities. Product design is the most creative, but not necessarily the costliest or most resource intensive in terms of human, financial, and material resources. Among the non-design tasks involved with the delivery of a final product to an enterprise's Manufacturing department, some are externally initiated by organizations such as the Sales or Marketing departments, or by requests and

orders of individual customers (most often for companies working on order). Other tasks are initiated by the design office itself and require cooperation from suppliers, the Manufacturing department, and external consultants.

Design and non-design activities produce and consume information—both documental (CAD drawings, models, and manuals) and non-documental (Bill of Materials, reports, and workflow trails). It is the composition of this information that eventually activates the process that produces a physical object. Information mismanagement can, and often does, have direct impact on the cost structure of the manufacturing phase: e.g., having different part numbers for interchangeable items (a common mishap) causes unnecessary inventory bloat and increases the associated costs. An important part of the work of the design office goes into maintaining and updating projects that have been previously released: a historical view of the previous information is absolutely necessary for this. This is where PDM applications come into play.

2.1 Technical Characteristics

thinkteam is a three-tier data management system running on Wintel platforms (cf. Fig. 1). The most typical installation scenario is a network of desktop clients interacting with one centralized RDBMS server and one or more file servers. Components resident on each client node supply a graphical interface, metadata management, and integration services. Persistence services are achieved by building on the characteristics of the RDBMS and file servers. We now describe its vaulting subsystem, as it is relevant to our experiments.

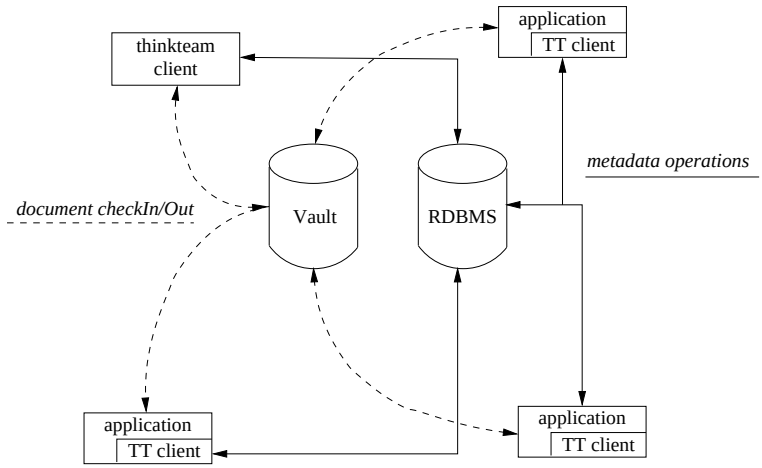

Fig. 1. The thinkteam structure.

The controlled storage and retrieval of document data in PDM applications is traditionally called vaulting, the vault being a file-system-like repository. The two main functions of vaulting are: (1) to provide a single, secure, and controlled storage environment, where the documents controlled by the PDM application are managed, and (2) to prevent inconsistent updates or changes to the document base, while still allowing the maximal access compatible with the

business rules. While the first function is subject to the implementation of the lower layers of the vaulting system, the second is implemented in thinkteam's underlying groupware protocol by a standard set of operations, viz.

get: extract a read-only copy of a document from the vault,

import: insert an external document into the vault,

checkOut: extract a copy of a document from the vault with the intent of modifying it (exclusive, i.e. only one checkout at a time is possible),

unCheckOut: cancel the effects of a previous checkout,

checkIn: replace an edited document in the vault (the document must previously have been checked out), and

checkInOut: replace an edited document in the vault, while at the same time retaining it as checked out.

It is important to note that access to documents (through the checkOut operation) is based on the "retrial" principle: there is no queue (or reservation system) handling the requests for editing rights on a document. Moreover, for the time being we consider the vault to reside on a sequential server.

thinkteam typically handles some 100,000 documents for 20-100 users. A user rarely checks out more than 10 documents a day, but she can keep a document checked out from anywhere between 5 minutes and several days.

2.2 Publish/Subscribe Notification

Adding a pub/sub notification service to thinkteam should solve a problem that commonly arises in connection with the usage of composite documents. This problem is a variant of the classic "lost update" phenomenon, depicted in Fig. 2, and arises when a client performs a *checkOut*/modify/*checkIn* cycle on a document that is used as reference copy by other clients.

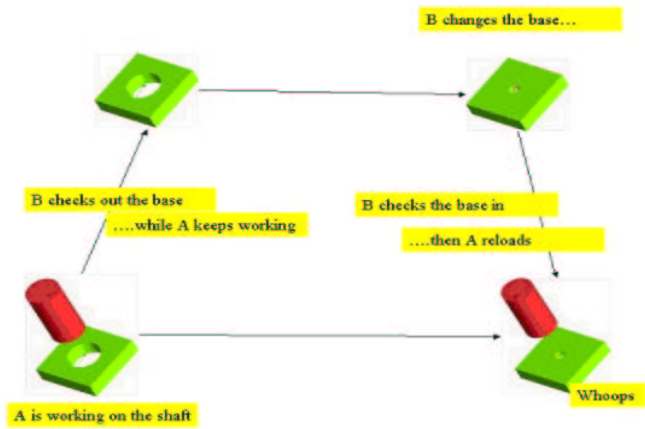

Fig. 2. The "lost update" phenomenon.

Note that, in order to maximize concurrency, a *checkOut* in thinkteam

creates an exclusive lock for write access but not for read access. It is thus possible for clients to gain read access to documents that are checked out by others. An automatic solution of this conflict is not easy, as it is critically related to the type, nature, and scope of the changes that will be performed on the document. Moreover, standard but harsh solutions—like maintaining a dependency relation between documents and use it to simply lock all documents depending on the document being checked out—are out of the question for think3, as they would cause these documents to be unavailable for too long periods of time. For thinkteam, the preferred solution is thus to leave it to the users to resolve such conflicts. However, a pub/sub notification service would provide the means to supply the clients with adequate information by

- informing the client who checks out a document of existing outstanding reference copies, and
- notifying the copy holders upon *checkOut* and *checkIn* of the document.

In this paper a pub/sub notification service is added to the protocol underlying thinkteam. More precisely, the service which think3 proposed to add actually is more refined than the one described above. All users subscribed to a document are notified whenever a user extracts this document from the repository for editing purposes. Furthermore, as soon as the user finishes editing and publishes the document in the repository, this causes an update on this document to all users that are subscribed to it. Hence not only those holding a read-only copy of the document recieve up-to-date information on its status, but all users that are registered for the specific document.

2.3 The thinkteam Protocol

The functioning of thinkteam is defined by its underlying multi-user communication schema, called the thinkteam protocol. In this paper we abstract from the complete thinkteam protocol and focus on the vaulting operations. We thus abstract from the RDBMS system and all its related operations. The model of the thinkteam protocol used in this paper is depicted in Fig. 3.

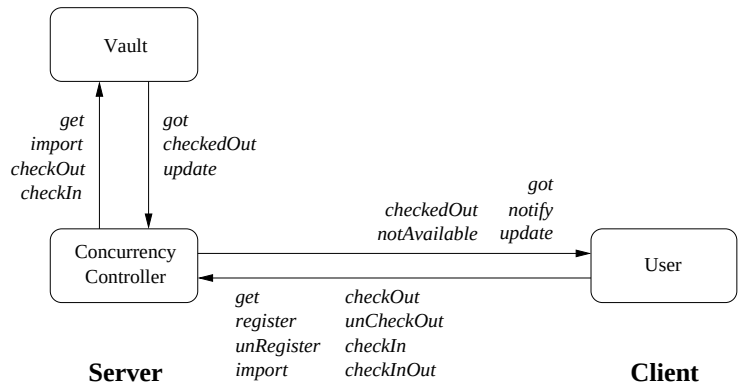

Fig. 3. The thinkteam protocol.

This model is composed of three components, viz. the Vault, the Concurrency Controller (CC), and the User. While the User is located on the client side, the Vault and the CC can be found on the server side. The messages that can be sent from one component to another are those described in Section 2.1, completed with the messages *got*, *checkedOut*, *notAvailable*, *notify*, and *update*, whose functioning we now explain. A user requesting a read-only copy of a file via a *get* is answered by a *got*, while a user requesting editing rights for a file via a *checkOut* is answered by a *checkedOut* or a *notAvailable*, depending on the availability of the requested file. In this way, the "direction" of a message is clear from its name. Moreover, all users that are registered for a file receive a *notify* the moment in which this particular file is checked out by another user, while they receive an *update* as soon as that user has sent either an *unCheckOut*, a *checkIn*, or a *checkInOut* to the CC.

Typical series of actions that take place in the thinkteam protocol are the following. A user can indicate the desire to extract a file from the vault by sending a *checkOut* to the CC. Upon receiving this action, the CC checks whether this file is available or whether it is locked as the result of a *checkOut* by another user. If the file is not locked, then the CC sends it to the user that requested it via a *checkedOut*; otherwise the user receives a *notAvailable*. Instead of extracting a file, a user can always request a read-only copy of a file by sending a *get* to the CC, which the CC responds to with a *got*. At any time, the user can insert a new file into the vault by sending it to the CC via an *import*. Finally, the user that has extracted a file has three options, viz.

(i) modify the file and then put it back into the vault by sending it via a *checkIn* to the CC,

(ii) refrain from modifying the file and simply return the file as it was by sending an *unCheckOut* to the CC, or

(iii) insert a modified version of the file into the vault—while keeping the file in her possession for further editing—by sending the CC a *checkInOut* (in which case the file remains locked for other users, but they can always obtain a read-only version of the file by means of a *get* operation).

Then there are some new features that concern the pub/sub notification service. A user can subscribe (unsubscribe) to a file by sending the CC a *register* (*unRegister*), whereas a user is registered automatically as the result of a *get*. If a user is subscribed to a certain file, then she receives a *notify* whenever another user extracts this file from the vault. Similarly, she receives an *update* whenever another user inserts (publishes) a file in the vault.

3 Model Checking the thinkteam Protocol

In this section we discuss some basics of the model checker Spin and the specification of the thinkteam protocol in Spin's input language Promela.

Promela is a non-deterministic C-like specification language for modelling finite-state systems communicating through channels [15]. 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. Channels can be local or global and FIFO buffered—for modelling asynchronous communication—or handshake—for modelling synchronous communication.

Promela specifications can be model checked by Spin against correctness properties specified as LTL formulae [16]. Spin converts the Promela processes into finite-state automata (FSA) and on-the-fly creates and traverses the state space of a product automaton over these FSA, in order to verify the specified correctness properties. There are several ways to formalise correctness properties in Promela, two of which we use in this paper. First, we add *assertions* to a Promela specification and verify their validity by running Spin. Secondly, formulate LTL properties and test their validity against the Promela specification, possibly enriched with specific *labels* identifying relevant points of process executions, by running Spin.

3.1 The Promela *Specification*

The Promela specification of the thinkteam protocol can be found in [4]. Here we list the assumptions on which it is based, as well as the improvements that have resulted from detailed discussions with think3.

Next to the abstractions described in Section 2.3, we have made several assumptions in our specification in order to reduce the size of both the state space and the state vector, which is used by Spin to uniquely identify a system state. The most important assumptions are as follows.

(i) The transmission time of messages between user and server is very fast with respect to the interarrival time between requests from different users. This results in a very low probability of competing requests. Therefore we have chosen to mainly use handshake channels for communication. Furthermore, initial verifications with Spin have shown that each of these handshake channels can be replaced by a channel with a buffer of size at most 2 without directly causing a state-space explosion for upto 4 users.

(ii) At any moment in time there is only one file (file 0) in the vault, hence the *import* of a file by a user currently is not modelled.

(iii) The administrative user actions *notify* and *update* are always enabled. To achieve this, the User process has an associated UserAdmin process, which does nothing else than receiving these actions.

(iv) A *get* by a user is responded to by the CC without allowing further interleavings, while interleavings are allowed to take place before the CC responds to a user's *checkOut*.

(v) No message is ever lost. We come back to this in the sequel.

During interactive design sessions with think3, including both physical meetings and meetings by means of groupware systems like teleconferencing and email, we have used Spin in various ways to present the behaviour of our specification. Examples include simulation, message sequence charts, and counterexamples. This enabled us to detect a number of ambiguities and unclear aspects of the design that think3 has in mind of the kind of pub/sub notification service they want to add to thinkteam. think3 had no previous experience with model checking. The central questions we addressed in these meetings are: "When exactly are which requests enabled?", "What is the exact semantics of requests?", and "How are simultaneous requests handled?".

4 Validation with Spin

In this section we show that the abstractions which we have applied to the thinkteam protocol are sufficient to allow for the verification of a number of correctness properties of the thinkteam protocol with Spin. [2]

First we have let Spin perform a full statespace search for invalid endstates, which is Spin's formalisation of deadlock states, in case of 2-4 users. The results are summarised in Table 1, with the runtime given as hours:minutes:seconds.

users	state vector	depth reached	errors	memory used	runtime	flags
2	84 byte	4423	0	37.574 Mbytes	1.3	
3	108 byte	434033	0	114.783 Mbytes	3:06.5	
4	132 byte	10484899	0	916.095 Mbytes	8:18:36.5	-DMA=28

Table 1
Results of full statespace searches for invalid endstates.

In case of 4 users, the available physical memory was insufficient. However, after disabling the explicit *register* (but still allowing implicit registration by means of a *get*) and enabling Spin's *minimized automaton procedure* with 28 as the maximal depth of the graph that is constructed for the minimized automaton representation (cf. [15] for details) no deadlocks were found—while a full state-space search was accomplished.

The reported results give a good impression of the fast-growing number of interleavings in applications of this kind and, consequently, of the difficulties in obtaining exhaustive verifications of relevant properties. This is one of the major reasons for some of the unsuccessful applications of model checking to groupware systems in the past [21].

[2] All verifications reported in this paper have been performed by running Spin Version 4.1.3 on a SUN® Netra™ X1 workstation with 1,000 Megabytes of available physical memory.

4.1 Correctness Properties

In [3], several correctness criteria that groupware protocols must satisfy have been formulated, covering both safety and liveness properties. Clearly some of these properties, such as those regarding the locking-based concurrency control mechanism, should also be satisfied by the thinkteam protocol. However, the thinkteam protocol must also satisfy some specific properties that are related to the pub/sub notification service. The set of properties which we intend to address in the forthcoming sections is as follows.

Concurrency control. (1) Every lock request must eventually be responded to, (2) at any moment in time and for every file, only one user may possess a lock on that file, (3) every lock on a file must eventually be released, and (4) a lock on a file is not released as the result of a *checkInOut*.

Awareness. (1) A user does not receive either a *notify* or an *update* if she is not registered for the file these messages refer to, (2) every *checkOut* must eventually result in a *notify* to all (and only those) users that are registered for the file being checked out, and (3) every *unCheckOut*, *checkIn*, and *checkInOut* must eventually result in an *update* to all (and only those) users that are registered for the file to which these message refer.

Denial of service. No user can be denied a service forever.

In the subsequent sections we analyse all of the above properties for the thinkteam protocol in case of 3 users by using Spin and the Promela specification given in [4]. Conceptually, this number of users covers many of the interesting combinations such as, e.g., the case in which one user wants to edit a file to which only one of the remaining two users is subscribed. Note that in the following section we always first state a formula, followed by its explanation.

4.2 Concurrency Control

In this section we verify the four core properties of the thinkteam protocol's locking-based concurrency control mechanism, formulated in Section 4.1.

Respond to lock. The first property states that every lock request must eventually be responded to. In the thinkteam protocol, the CC handles a user's *checkOut* as a lock request for file 0 and it either grants the lock by responding with a *checkedOut* or—if the lock cannot be granted—with a *notAvailable*. To verify this property we add a number of user-specific labels to the specification of the CC process, viz. `doneCheckOutX` directly follows `userToCC?checkOut,id`—by which the CC receives a *checkOut* from user `X=id`—, `doneCheckedOutX` directly follows `ccToUser[id]!checkedOut`—by which the CC responds to user `X=id` by sending her a *checkedOut*—, and `doneNotAvailableX` directly follows `ccToUser[id]!notAvailable`—by which the CC responds to user `X=id` by sending her a *notAvailable*. These labels allow us to formulate, e.g., that whenever the CC has received a lock

request on file 0 by user 0 via a *checkOut*, then it eventually responds by sending that user either a *checkedOut* or a *notAvailable*:

```
[] (CC[2]@doneCheckOut0 ->
        <> (CC[2]@doneCheckedOut0 || CC[2]@doneNotAvailable0)).
```

The 2 in this formula is the (unique) *process instantiation number* of the CC. Starting with 0, Spin assigns—in order of creation—such a number to each process it creates, which can be used in LTL formulae for process identification.

We let Spin run verifications of this LTL formula as well as of analogous versions for users 1 and 2. It takes Spin just over fifteen minutes to conclude that the above LTL formulae are valid.

Though never mentioned specifically, for all formulae in the sequel that contain a logical implication we have verified that the left-hand side can indeed become `true` in at least one run. Moreover, in the sequel we will not spell out the exact points in the Promela specfication where labels have been added, but we will simply list them and assume the positions to be clear from the description. Their exact positions can be found in [4].

Unique lock/file. The second property states that at any moment in time and for every file, only one user may possess a lock on that file. Given a file, the CC may thus have granted at most one lock for it. In the thinkteam protocol this means that at any moment in time, only one user may have extracted file 0 through a *checkOut*. To verify this property we add the basic assertion `assert(writeLock == false)` to the specification of the CC process when it is about to grant a lock to a user by sending her a *checkedOut*. We then let Spin run a verification on assertion violations. As a result, we 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 setting `writeLock` to `true` and sending *checkedOut* to this user. In about 3 minutes Spin concludes that the above basic assertion is never violated, which proves that the property is valid.

Release file+lock. The third property is that every lock on a file must eventually be released. The CC releases a lock when it receives a *checkIn* or an *uncheckOut* from the user it last granted the lock via a *checkedOut*. Hence we must verify that every *checkedOut* is eventually followed by a *checkIn* or an *unCheckOut* from the same user to whom it sent a *checkedOut*. So we add two user-specific labels to the specification of the CC process, viz. `doneCheckInX` and `doneUnCheckOutX`—by which the CC receives a *checkIn* (*unCheckOut*) from user `X`—and let Spin run verifications of the LTL formula

```
[] (CC[2]@doneCheckedOut0 ->
        <> (CC[2]@doneCheckIn0 || CC[2]@doneUnCheckOut0))
```

as well as of analogous versions for users 1 and 2. It takes Spin just a split second to conclude that these LTL formulae are *not valid*. The provided counterexamples are clear: the CC can endlessly be kept busy by the users that do not possess the lock on file 0; these users repeat an alternation of *get*, *register*, *unRegister*, *checkOut*, and *checkInOut* ad infinitum. This led to the idea to re-run Spin with its weak fairness option enabled.

A run or computation of Spin is called *weakly fair* if every process that is continuously enabled from a particular point in time will eventually be executed after that point. This does not guarantee that every (infinitely often) enabled statement of such a process will eventually be executed after that point: the process may contain more than one statement that is continuously enabled from a particular point in time and in order for it to be weakly fair it suffices that one of them will eventually be executed after that point.

Again it takes Spin just a second to show that the above formulae are *not valid*. The counterexamples are clear: a user holding the lock can endlessly perform *checkInOut* and thus never release the lock. This is an unavoidable property of the thinkteam protocol. In thinkteam practice this situation is avoided by a superuser or system administrator that a user can contact with the request to "convince" another user to release the file she has checked out.

Keep file locked. The fourth property states that a lock on a file is not released as the result of a *checkInOut*. In the thinkteam protocol, the CC may thus not change the value of `writeLock` (which is `true`) as the result of a *checkInOut*, i.e. the checked out file remains checked out/locked. To verify this property we add the basic assertion `assert(writeLock == true)` to the specification of the CC process after it has updated the vault by sending it a *checkIn* (as the result of a *checkInOut* received by the user) and before it updates the users of this fact. We let Spin run a verification on assertion violations and thus verify whether it is always the case that `writeLock` is `true` the moment in which the CC is about to update all registered users of the fact that the user that currently has a file in its possession, has published an intermediate version of it in the vault. In some 3 minutes Spin concludes that this basic assertion is never violated, proving that the property is valid.

4.3 Awareness

In this section we verify the three properties dealing with awareness through the thinkteam protocol's pub/sub service, formulated in Section 4.1.

No illegal notify (update). The first property states that a user does not receive a *notify* (*update*) if she is not registered for the file these messages refer to, i.e. the user does not receive any "illegal" *notify* (*update*). We thus need to verify that every *notify* (*update*) is preceded by either a *get* or a *register*. However, since a user could *unRegister* in between the *get* or *register* and

the *notify* (*update*), we moreover require that in between an *unRegister* and a *notify* (*update*), no *get* or *register* takes place. To this aim, we add several labels to the specification of the User process, viz. `doneGet`, `doneRegister`, and `doneUnRegister`—by which the User sends a *get* (*register*, *unRegister*) to the CC. We furthermore add two labels to the specification of the UserAdmin process, viz. `doneNotify` and `doneUpdate`—by which it receives a *notify* (*update*) from the CC. We then let Spin run verifications of the LTL formulae

```
! ( ! (User[3]@doneGet || User[3]@doneRegister) U UserAdmin[4]@doneLab)
                    && [] (User[3]@doneUnRegister ->
! ( ! (User[3]@doneGet || User[3]@doneRegister) U UserAdmin[4]@doneLab)),
```

where doneLab is doneNotify or doneUpdate. We also ran verifications of analogous versions of these LTL formulae for the other users. Stated differently, we verify whether it may be the case that a user receives a *notify* (*update*) without currently being registered for the file these messages refer to. It takes Spin just over twenty minutes to conclude that the above formulae are valid.

Notify if registered. The second property states that every *checkOut* must eventually result in a *notify* to all (and only those) users that are registered for the file being checked out. To verify this property we add some more user-specific labels to the specification of the CC process, viz. `doneGetX`, `doneRegisterX`, and `doneUnRegisterX`—by which the CC receives a *get* (*register*, *unRegister*) from user `X`—and `doneNotifyX`—by which the CC sends a *notify* to user `X`. These labels allow us to formulate, e.g., that it may never be the case that user 0 is (still) registered for file 0 the moment in which either user 1 or user 2 checks out file 0, but user 0 nevertheless is not notified:

```
[] ! ((CC[2]@doneGet0 || CC[2]@doneRegister0) && 
   ( <> (CC[2]@doneCheckedOut1 || CC[2]@doneCheckedOut2)) && 
      ( ! CC[2]@doneUnRegister0 U ((CC[2]@doneCheckedOut1 || 
            CC[2]@doneCheckedOut2) && [] ! CC[2]@doneNotify0))).
```

We let Spin run verifications of this LTL formula as well as of analogous versions in which the users change roles. It takes Spin almost forty minutes to conclude that these formulae are valid.

Update if registered. The third property states that every *unCheckOut* (*checkIn*, *checkInOut*) must eventually result in an *update* to all (and only those) users that are registered for the file these messages refer to. To verify this property we add some more user-specific labels to the specification of the CC process, viz. `doneCheckedInX` and `doneCheckedInOutX`—by which the CC receives confirmation from the Vault of the fact that the file, which the CC received from user `X` and forwarded to the Vault, has indeed been inserted into

the Vault—and `doneUpdateX`—by which the CC sends an *update* to user `X`. As before, these labels allow us to formulate, e.g., that it may never be the case that user 1 or 2 sends an *unCheckOut*, a *checkIn*, or a *checkInOut* for file 0 to the CC, while user 0 is not currently registered for file 0, and that user 0 does eventually get updated for file 0, without meanwhile registering for file 0:

$$[]\,!\,((\mathrm{CC}[2]@\mathrm{doneGet0}\,||\,\mathrm{CC}[2]@\mathrm{doneRegister0})\,\&\&\,(<>\,\mathrm{OR})\,\&\&\,(!\,\mathrm{CC}[2]@\mathrm{doneUnRegister0}\,\mathtt{U}\,(\mathrm{OR}\,\&\&\,[]\,!\,\mathrm{CC}[2]@\mathrm{doneUpdate0}))),$$

where

$$\mathrm{OR} = (\mathrm{CC}[2]@\mathrm{doneUnCheckOut1}\,||\,\mathrm{CC}[2]@\mathrm{doneUnCheckOut2}\,||\,\mathrm{CC}[2]@\mathrm{doneCheckedIn1}\,||\,\mathrm{CC}[2]@\mathrm{doneCheckedIn2}\,||\,\mathrm{CC}[2]@\mathrm{doneCheckedInOut1}\,||\,\mathrm{CC}[2]@\mathrm{doneCheckedInOut2}).$$

We let Spin run verifications of this LTL formula as well as of analogous versions in which the users change roles. It takes Spin almost forty minutes to conclude that these formulae are valid.

4.4 Denial of Service

A further desirable property of any groupware system in general and the thinkteam protocol in particular is that its users cannot be denied a service forever. A user should, e.g., always be able to *get* a file if she so wishes. To verify this property, we augment the specification of the User with the label `todoGet` directly *before* the statement by which the user may send a *get* to the CC (and we recall that directly *after* this statement we put the label `doneGet`). Subsequently we formulate the LTL formulae

$$[]\,(\mathrm{User}[\mathrm{pid}]@\mathrm{todoGet} -> <>\,\mathrm{User}[\mathrm{pid}]@\mathrm{doneGet}),$$

where pid equals 3 (for user 0), 5 (for user 1), or 7 (for user 2). Next we let Spin run verifications of these LTL formulae with its weak fairness option enabled. 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 users can never get its turn to send a *get* to the CC, because the latter is continuously kept busy by the other users, while this user nevertheless expressed the desire to send the CC a *get*. Such behaviour, in which the CC is kept busy by one of the users and other users thus never get their turn, forms an integral part of the thinkteam protocol as it is defined in this paper. This is because, as mentioned before, in thinkteam access to documents is based on the "retrial" principle: there currently is no queue (or reservation system) handling simultaneous requests for a document. However, think3 has expressed interest in considering a document reservation system in a future version of thinkteam.

Similar to the cyclic behaviour described above, it can be shown that a user is not obliged to ever return a file to the Vault which she has checked out. In the thinkteam protocol, a user is simply never forced to undertake any action whatsoever. Such behaviour is similar to that discussed in Section 4.2 for the case of releasing a lock and is dealt with in a similar way in thinkteam by means of a superuser or system administrator which can, e.g., force certain users to eventually return files to the vault.

4.5 Summary

In Sections 4.2-4.4 we have used the model checker Spin to verify the set of correctness properties listed in Section 4.1. The results of these verifications are summarised in Table 2, with the runtime again given as hours:minutes:seconds.

verified property	state vector	depth reached	errors	memory used	runtime
Respond to lock	112 byte	3147677	0	473.209 Mbytes	16:54
Unique lock/file	108 byte	434033	0	114.783 Mbytes	3:06.0
Release file+lock	116 byte	7348	1	193.862 Mbytes	0.9
Keep file locked	108 byte	434033	0	114.783 Mbytes	3:06.0
No illegal notify	112 byte	3071518	0	539.769 Mbytes	21:22.1
No illegal update	112 byte	3057025	0	558.508 Mbytes	22:45.4
Notify if registered	112 byte	3338868	0	967.955 Mbytes	39:22.2
Update if registered	112 byte	4183223	0	925.049 Mbytes	38:57.6
Denial of Service	116 byte	1801	1	193.759 Mbytes	0.4

Table 2
Results of the verifications performed in this paper.

In this paper we show that verifications of the Promela specification of the thinkteam protocol are very well feasible with the current state of the art of available model-checking tools such as Spin. Moreover, the results show that the concurrency control and awareness aspects of the thinkteam protocol completed with a pub/sub notification service are well designed. We have seen, however, that the thinkteam protocol does not oblige a user to ever return a file she has checked out to the Vault. In thinkteam this situation is dealt with by means of a superuser or system administrator which can intervent and force a user to return the file she has checked out to the vault.

5 Conclusions and Future Work

This paper is the result of ongoing work on applying academic experience with formal modelling—and with model checking in particular—to an industrial case study. The goal of this case study was to investigate the effects of adding a pub/sub notification service to think3's PDM solution thinkteam. To this aim, we have first specified thinkteam's underlying groupware protocol in Promela, after which we have used Spin to verify a number of important

properties related to thinkteam's concurrency control and awareness aspects. The outcome has shown that many ambiguities could be removed, leading to a design in which more confidence can be put with respect to the addressed aspects. To the best of our knowledge, our approach is among the first successful applications of exhaustive model-checking techniques to the verification of pub/sub notification services in a specific groupware setting. Other success stories, focussing more on a middleware setting, are described in [7,12]. In [7] a new model checker is introduced which, by ruling out certain infeasible interleavings, is shown to be capable of verifying a realistically complex system that uses pub/sub event notification. In [12] a generic, parameterised framework for model checking pub/sub systems is defined, complete with the automatic generation of model-checking code.

The specification (model) we have developed in this paper and its related correctness properties may serve as a basis for the formal modelling and verification of other variants of groupware systems or pub/sub notification services. In fact, it is think3's intention to use them as basis for their planned implementation of a pub/sub notification service in thinkteam. In this respect the related approach of [19] can be of use, as it studies the automatic derivation of correct integration code for assembling a set of thinkteam's (software) components, starting from a set of formally specified requirements.

In the future we intend to augment the number of files (currently set to one) that can be handled by our specification of the thinkteam protocol. Furthermore, we intend to further investigate the consequences of abandoning the "retrial" principle with respect to document access and introduce a document reservation system instead. The most obvious way to model this is by replacing the handshake channels from the users to the CC with buffered channels. While this obviously increases the total number of interleavings in our specification, initial verifications have shown that this still leads to feasible memory requirements. Finally, a thinkteam user that registers itself for a document is currently informed of the current status of that document only when its status changes. We plan to extend thinkteam's pub/sub notification service in such a way that the user who checks out a document is informed automatically of existing outstanding reference copies of this document.

We recall from Section 2.2 that one of the reasons for think3's desire to add a pub/sub notification service to thinkteam was to be able to solve the variant of the "lost update" phenomenon depicted in Fig. 2. It is important to note that the addition of such a service to thinkteam only partially solves this phenomenon, viz. nothing is solved in case a *notify* or an *update* does not reach its destination. A possible solution to overcome this would be to enhance the *notify*'s and *updates*'s with a sequence number. This would enable a user to realize that a *notify* or an *update* got lost and can undertake action to remedy this problem, e.g. by requesting the missing information from the CC. A different solution would be to try and reduce the possibility of losing messages by sending redundant copies of each *notify* and *update* to the user,

thereby reducing the chances of these actions not reaching their destination. The latter solution would create much overhead, though. This is a topic worth further investigation, in particular because the specific human interactions inherent to groupware protocols make this problem different from the well-studied issue of message loss in network protocols.

We conclude by noting that 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 support tools for verification and formal dependability assessment (e.g. stochastic model checking [2] and formal specification-driven discrete simulation tools).

Acknowledgements

This research has 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".

We thank the four anonymous referees for their suggestions which have improved this paper.

References

[1] Baecker, R.M. (ed.), "Readings in Groupware and Computer Supported Cooperation Work", Morgan Kaufmann, 1992.

[2] Baier, C., B.R. Haverkort, H. Hermanns, and J.-P. Katoen, *Automated Performance and Dependability Evaluation Using Model Checking*, "Performance Evaluation of Complex Systems—Performance'02 Tutorial Lectures", LNCS **2459**, Springer-Verlag, 2002, 261–289.

[3] ter Beek, M.H., M. Massink, D. Latella, and S. Gnesi, *Model Checking Groupware Protocols*, "Cooperative Systems Design", IOS Press, 2004, 179–194.

[4] ter Beek, M.H., M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis, *Model Checking Publish/Subscribe Notification for* thinkteam, Technical Report 2004-TR-20, ISTI-CNR, 2004.
URL: http://fmt.isti.cnr.it/WEBPAPER/TRTT.ps.

[5] Caporuscio, M., A. Carzaniga, and A.L. Wolf, *Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications*, IEEE Transactions on Software Engineering **29, 12** (2003), 1059–1071.

[6] Caporuscio, M., P. Inverardi, and P. Pelliccione, *Formal Analysis of Clients Mobility in the Siena Publish/Subscribe Middleware*, Technical Report, Department of Computer Science, University of L'Aquila, 2002.

[7] Deng, X., M.B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh, *Model-Checking Middleware-Based Event-Driven Real-Time Embedded Software*, Proc. FMCO'02, LNCS **2852**, Springer-Verlag, 2002, 154–181.

[8] Dourish, P., and V. Bellotti, *Awareness and Coordination in Shared Workspaces*, Proc. CSCW'92, ACM Press, 1992, 107–114.

[9] Ellis, C.A., and S.J. Gibbs, *Concurrency Control in Groupware Systems*, Proc. SIGMOD'89, ACM Press, 1989, 399–407.

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

[11] Eugster, P.Th., P. Felber, R. Guerraoui, and A.-M. Kermarrec, *The Many Faces of Publish/Subscribe*, ACM Computing Surveys **35, 2** (2003), 114–131.

[12] Garlan, D., S. Khersonsky, and J.S. Kim, *Model Checking Publish-Subscribe Systems*, Proc. SPIN'03, LNCS **2648**, Springer-Verlag, 2003, 166–180.

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

[14] Gutwin, C., M. Roseman, and S. Greenberg, *Supporting Awareness of Others in Groupware*, Companion Proc. CHI'96, ACM Press, 1996, 205–215.

[15] Holzmann, G.J., "The SPIN Model Checker", Addison Wesley, 2003.

[16] Manna, Z., and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems—Specification", Springer-Verlag, 1992.

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

[18] Schmidt, K., *The Problem with 'Awareness'*, Computer Supported Cooperative Work—The Journal of Collaborative Computing **11, 3-4** (2002), 285–298.

[19] Tivoli, M., P. Inverardi, V. Presutti, A. Forghieri, and M. Sebastianis, *Correct Components Assembly for a Product Data Management Cooperative System*, Proc. CBSE'04, LNCS **3054**, Springer-Verlag, 2004, 84–99.

[20] Tripakis, S., and S. Yovine, *Timing Analysis and Code Generation of Vehicle Control Software using Taxys*, Proc. RV'01, ENTCS **55, 2**, 2001, 174–183.

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

[22] Zanolin, L., C. Ghezzi, and L. Baresi, *An Approach to Model and Validate Publish/Subscribe Architectures*, Proc. SAVCBS'03, Technical Report 03-11, Department of Computer Science, Iowa State University, 2003, 35–41.