Model Checking Publish/Subscribe Notification for thinkteam®

Maurice H. ter Beek[1], Mieke Massink[1], Diego Latella[1], Stefania Gnesi[1], Alessandro Forghieri[2], and Maurizio Sebastianis[2]

[1] 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
[2] 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,[3] 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.

Keywords: publish/subscribe notification, thinkteam, model checking, asynchronous and dispersed groupware, awareness, concurrency control.

1 Introduction

Computer Supported Cooperative Work (CSCW for short) 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 [14]. This technology mostly consists of multi-user computer systems called groupware (systems) [1, 11]. Groupware is typically classified according to two dichotomies, viz. (1) whether its users are working together at the same time (synchronous) or at different times (asynchronous) and (2) whether they are working together in the same place (co-located) or in different places (dispersed). This is called the time space taxonomy by Ellis et al. [11]. In this paper we deal with thinkteam, which is an example of an asynchronous and dispersed groupware system.

[3] thinkteam is a registered trademark of think3, Inc. For details, see www.think3.com.

Other examples include electronic mail, workflow, collaborative writing systems, and the version-control systems which are 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 (*how is information shared and how is this indicated to the users?*), security (*how is information protected and the privacy of users guaranteed?*), user awareness (*how is information about a user's behaviour made available to other users and how do the other users consequently use this information?*), and concurrency control (*how are conflicting requests for information handled?*). In this paper we address data sharing, awareness, and concurrency control issues in the context of thinkteam. More precisely, we use model-checking techniques 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.

Increased computing resources together with an enormous improvement in the efficiency of model-checking techniques in recent years, has facilitated the application of model checking to ever more complex systems of concurrent and distributed nature. Many of the protocols underlying groupware and publish/subscribe 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. It is thus not surprising that there is an increasing interest in the use of model checking for the formal verification of (properties of) groupware [3, 20, 24] and publish/subscribe systems [5, 8, 13, 23].

thinkteam is think3's Product Data Management (PDM for short) 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. Technically, thinkteam is a three-tier Wintel application, which implements an object-based information management system using an underlying DBMS (Oracle, SQL Server, or access) for persistence and retrieval of the metadata (non-document data). Wintel is a commercially available remote control program for modems and ISDN. It can handle file transfer, application sharing and remote terminals.

In this paper we study the addition of a lightweight and easy-to-use publish/subscribe notification service to thinkteam. The goal of adding such a service to an application is to increase the awareness of its users by intelligent data sharing: whenever a user publishes a document or file by sending it to a centralized repository, automatically all users that are subscribed to that document or file 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 exam-

ples of applications of a publish/subscribe notification service include electronic auctions on the Internet and email alert services for new journal or book releases that many publishing houses offer nowadays.

Publish/subscribe notification services decouple the communication among users: a user that publishes a document or file does not need to be concerned with whom the server will consequently send a notification to, i.e. the users communicate through the server. Users thus need not actively participate in the notification in a synchronous way, which means that they need not interrupt their current activity for participating in this type of distributed event notification. In fact, the main strength of a publish/subscribe notification service is said to be the "full decoupling of the communicating participants in time, space and flow" [12]. Apart from these advantages, systems with a publish/subscribe notification service are generally difficult to verify [13, 25]. 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. This paper contains some of our initial results in verifying the addition of a pub/sub notification service to thinkteam.

Before presenting in detail the proposed publish/subscribe 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 publish/subscribe 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 the issue of awareness through publish/subscribe notification and key issues related to concurrency control. A related approach can be found in [22], 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 [21]. Roughly speaking, it should be understood as users having a sense of the (past, current, future) activities of other users—without the use of direct communication—and using this as a context for their own activities [9, 15]. 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, which refers to achieving the maximum degree of parallelism under a correctness criterion. Within groupware systems, the goal of concurrency control is to help resolve conflicting user actions, while still allowing the groupware system's users to perform their collaborative tasks in a tightly coupled manner [10].

Without marginalising the importance of time-performance issues for both groupware systems and publish/subscribe notification services [4, 8, 20, 23], the correctness of many of their underlying protocols is not critically depending on real time. After all, the protocols need to function correctly under a variety of time assumptions being made. This is mainly so because both groupware

systems and publish/subscribe notification services are often 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 their underlying protocols can be analysed also with models that do not include real-time aspects. Needless to say, this does not mean that real-time and performance aspects are not relevant to the design of groupware systems and publish/subscribe notification services, to the contrary, but they need not necessarily be addressed in the same models as those being appropriate to verify correctness issues. In fact, abstracting from real-time and performance issues at first may make the difference between models that are computationally tractable and those that cannot be analysed with the help of automatised tools.

In this paper we show that with relatively simple models we can verify highly relevant properties of groupware protocols with verification tools, such as the model checker SPIN [16]. The properties we verify are mostly formalised as formulae of a Linear Temporal Logic (LTL for short) [19]. This has the advantage that they are close to the 'scenarios' that are often informally formulated during the initial phases of software design. In fact, LTL formulae reflect properties of typical—desired or undesired—behaviour (or uses) of the groupware system. This makes it possible to evaluate design alternatives before their implementation and validation and not only afterwards. In this way, design errors, which constitute up to 40% of software errors and are among the most expensive ones to resolve when discovered only after the implementation phase, can be detected early on in the development, leading to considerable reductions in cost and quality improvements [17]. In fact, the specification (model) of thinkteam's underlying groupware protocol has been developed in close collaboration with think3. Moreover, it is think3's intention to use the specification (model) as basis for the planned implementation of a publish/subscribe notification service in thinkteam. As such, this paper thus reports on an ongoing cooperation between an academic and an industrial partner.

We begin this paper with a brief description of thinkteam and its underlying protocol, followed by an overview of the basic concepts of model checking and the model checker SPIN. We then continue by discussing the specification in SPIN's input language PROMELA of thinkteam's underlying protocol, after which we verify a number of core issues of the thinkteam protocol. Finally, we conclude with a discussion of future work.

2 thinkteam

In this section we present a brief overview of the use of thinkteam within the manufacturing industry. For more information or for obtaining thinkteam, we refer the reader to `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 the most resource intensive in terms of human, financial, and material resources. Among the several non-design tasks that concur to the delivery of a final product

to an enterprise's Manufacturing department, some are externally initiated by organizations such as the Sales or the 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.

Since the inception of PDM as a separate IT discipline, a few clear goals have been deemed a non-negotiable part of any successful implementation, viz.

- it must make significant contributions to the reduction of the overall conception/design/development cycle,
- it must supply a coherent and up-to-date view of the information,
- it must preserve the historical evolution of the information and of the events and actors that have affected it (traceability), and
- it must provide *easy* and *secure* access to all of the product-related information, both documental (vaulting) and non-documental (metadata).

To a large extent, thinkteam meets all these goals.

2.1 Technical Characteristics

thinkteam is a three-tier data management system running on Wintel platforms (cf. also Figure 1). The most typical installation scenario is a network of several desktop clients interacting with one centralized RDBMS server and one or more file servers. In this setting, 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. In what follows we give a general description of the operations of various (logical) thinkteam subsystems. The Graphical User Interface (GUI for short) is not described as it is not pertinent to the purpose of this paper.

Metadata management. thinkteam allows its users to manage representations of concrete entities (such as documents and components). These representations (often called business items or business objects) are described using an object model or meta-object model which can be customized by the end users, e.g. by changing the attributes pertaining to various types of object or by adding object

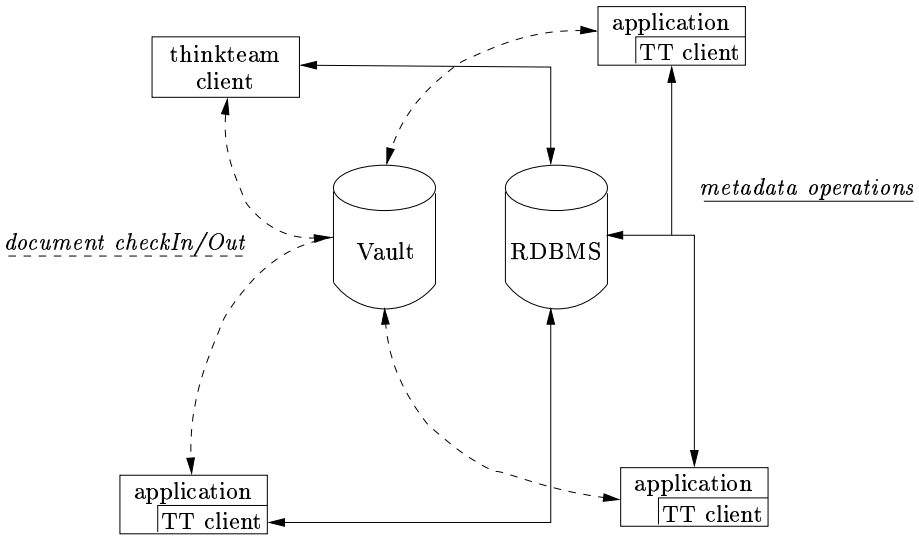

Fig. 1. The thinkteam structure.

types. Metadata management refers to operations on object instances and to the rules these operations obey to, as they are implemented in thinkteam. Typical operations are creation, attribute editing (e.g. adding/changing description, price, etc.), revisioning, changing state, connecting with other objects, and deletion.

DBMS interaction. thinkteam uses a RDBMS to persist and retrieve both its object model and the objects that are created during operation. DBMS interactions are fairly low level in nature and are completely transparent to end-users.

Vaulting. The controlled storage and retrieval of document data in PDM applications is traditionally called vaulting, where the vault is 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 that is compatible with the business rules. While the first function is the subject of the implementation of the lower layers of the vaulting system, the second function 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 above 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, thinkteam typically handles something like 100,000 documents for 20-100 users. A user rarely checks out more than 10 documents a day, but he or she can keep a document checked out from anywhere between 5 minutes and several days.

Integrations. thinkteam's data management functions would be unwieldy and require heavy manual intervention if it were not possible to automatically extract information from where it most often resides, i.e. in the documents themselves. Besides, it is almost always necessary to supplement the document production process with metadata extracted from the thinkteam repository and it is desirable that such operations can be performed automatically. To this end, thinkteam must integrate with the authoring applications responsible for document creation. This means thinkteam must be able to communicate with the application in order to mediate both its file operations (so the application can interact with the vaulting subsystem) and its metadata operations (so the document metadata can be merged with that present in its repository). Typical examples are automatic retrieval/storage of a file in its appropriate vaulting position, automatic capturing of data like document authorship, description, glossary keywords, etc., and automatic insertion (in the document body) of textual content referring to data in thinkteam's repository, e.g. for filling a drawing title block.

More complex interactions—whose need arises when integrating with CAD systems—concern mapping the structure of related/composite documents to structures of thinkteam, e.g. mechanical parts that are assembled in a single assembly must be mapped in a hierarchical product structure known as the Bill Of Materials, or BOM for short. Integration is typically achieved by developing software that is able to map a set of thinkteam operations to the application APIs, and that hooks commands of applications relevant to thinkteam (e.g. closing a file) in order to allow thinkteam to perform the appropriate action (e.g. putting a closed file back into the vault).

2.2 thinkteam at Work

thinkteam supports the CAD designer in various phases. An important area of thinkteam intervention is the overall industrialization part of a given project and involves activities that tend to be intensive with respect to the project metadata (attributes, BOM structure) while being light on the document management (and therefore vaulting) side. Vaulting capabilities are most frequently used—by a CAD designer—during some of the modelling phases, briefly described next.

Geometry information retrieval. The most usual design work in the manufacturing industry (thinkteam's prime target) involves the production of components that are part of more complex goods. The CAD models describing these products are called assemblies and are structured as composite documents referring to several (sometimes hundreds or thousands) individual model files. In this situation, most of the geometry data a designer deals with consists of reference material, i.e. parts surrounding the component he or she is actually creating or modifying. The designer needs to interact with this reference material in order to position, adapt, and mate to the assembly the part he or she is working with.

Most of the reference parts are normally production items subjected to PDM management and whose physical counterparts (model files) reside in the vault. The logical operation by which the designer gains access to them is the *get* operation discussed above, which is performed automatically as needed by the thinkteam/thinkdesign integration. This is the type of activity that happens most often and which is normally involved in all other activities listed below, as well as in many others not explicitly mentioned (such as visualization, printing, etc.).

Geometry (part) modification. Modifying an existing part is, in order of frequency, the second-most-used operation that a designer performs during his or her activity. As it is an already existing and managed part (i.e. it is already in the vault) the designer must express his or her intent to modify it with an explicit (exclusive) *checkOut* operation that prevents other attempts at modification by other users. A screenshot of this operation at work is depicted in Figure 2.

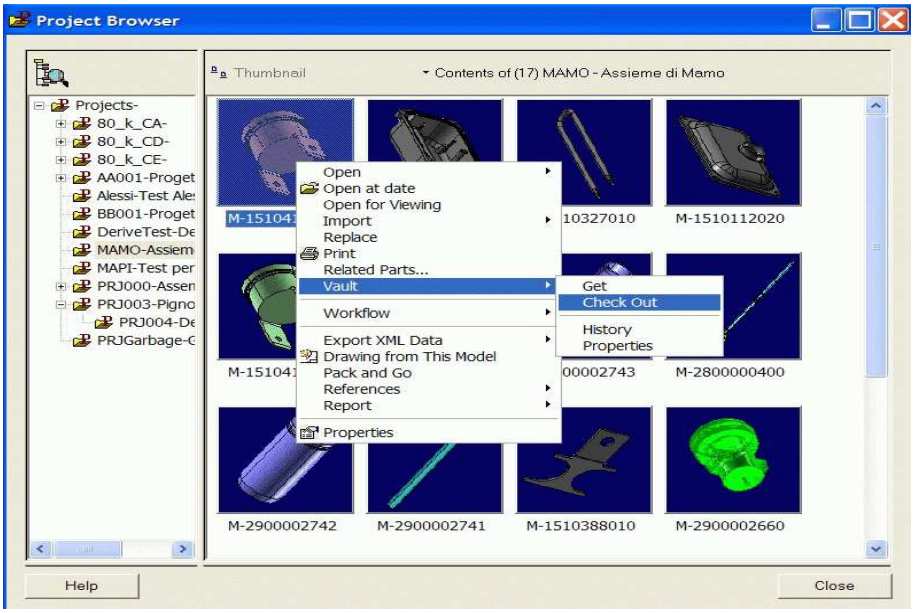

Fig. 2. A thinkteam user checks out a file from the vault.

When the designer is ready to publish his or her work, he or she will release it to the system by issuing an explicit *checkIn* command, which frees the model for modification. Were the designer to change his or her mind, then he or she may choose to issue an *unCheckOut* command, which frees the model but discards any changes that have occurred since the *checkOut*. Finally, he or she may issue a *checkInOut* if he or she wants to release an intermediate version of the model to the system, but does not yet want to release it for further modifications by others. All these actions require explicit action on the part of the user and are exposed via suitable parts of thinkteam's GUI in thinkdesign.

Geometry (part) creation. Lastly, a designer may create a completely new component and insert it into the system. As the part will initially be created outside the system vault, an *import* operation is required to register it with thinkteam. In this case a special environment (called Save Into Project, or SIP for short) is provided, combining metadata/vaulting operations for speedily registering several changes and modifications.

2.3 Publish/Subscribe Notification

The addition of a publish/subscribe notification service to thinkteam should allow the solution of a problem that commonly arises in connection with the usage of composite documents and which is a variant of the classic "lost update" phenomenon. This phenomenon, depicted in Figure 3, arises when one client performs a checkout/modify/checkin cycle on a document that may be used as reference copy by other clients.

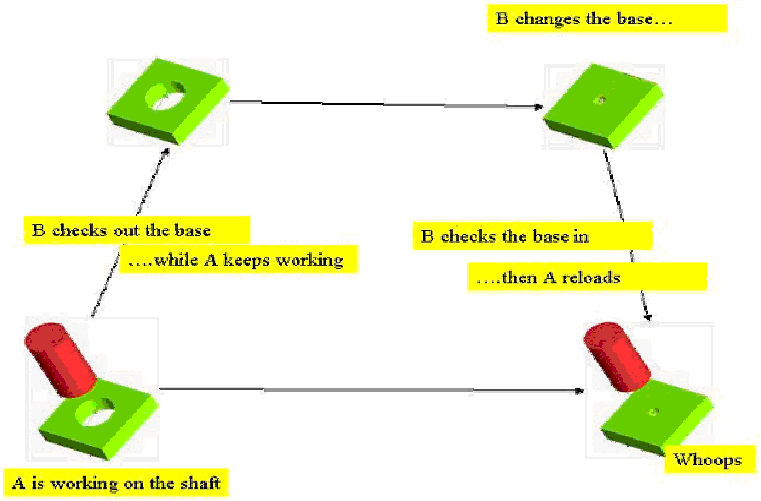

Fig. 3. The "lost update" phenomenon.

At this point it is important to note that in order to maximize concurrency, checkout operations in thinkteam create exclusive locks only for write access but not for read access. It is therefore possible for clients to gain read access to documents that are checked out by other clients. While an automatic solution of the conflict is not possible—because it is critically related to the type, nature, and scope of the changes that will be performed on the document—a publish/subscribe notification facility 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 publish/subscribe notification service is added to the protocol underlying thinkteam. More precisely, the service which think3 proposed to add to thinkteam actually is more refined than the service 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.

2.4 The thinkteam Protocol

The functioning of thinkteam is defined by its underlying multi-user communication schema, which we call the thinkteam protocol. In this paper we use an abstract specification (model) of thinkteam's underlying groupware protocol—which nevertheless covers faithfully its most important aspects—augmented with the publish/subscribe notification service that think3 is planning to add to thinkteam. We abstract from the complete thinkteam protocol and focus on the communication schema that is fundamental to the two aspects of the thinkteam protocol that we want to verify, viz. the concurrency control and the newly added publish/subscribe notification aspects. We thus abstract completely from the RDBMS system and all its related operations. The abstract model of the thinkteam protocol used in this paper is depicted in Figure 4.

The abstract model is composed of three components, viz. the Vault, the Concurrency Controller (CC for short), 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 explain next. A user that requests 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 directly clear from the name of the message. Moreover, all users

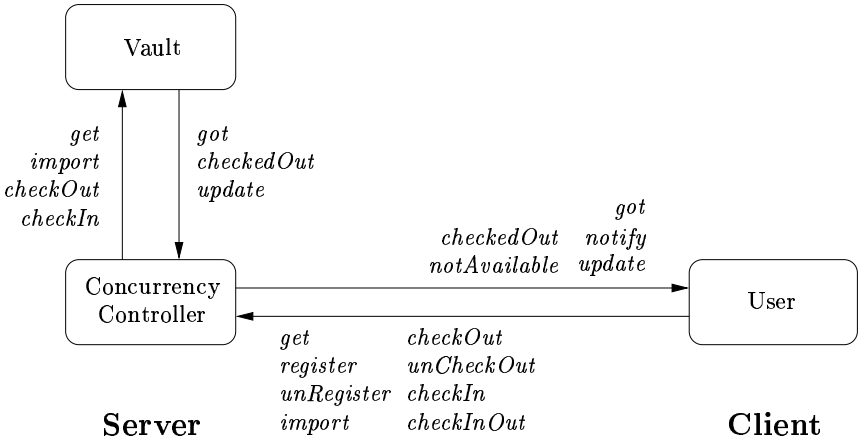

Fig. 4. The thinkteam protocol.

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.

Some typical series of actions that can take place in the thinkteam protocol are the following. A user can indicate that he or she wants 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 an extraction by another user. If the file is not locked, then the CC sends it to the user that requested it via a *checkedOut*, while 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, upon which the CC sends it 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.

1. modify the file and then put it back into the vault by sending it via a *checkIn* to the CC,
2. refrain from modifying the file and simply return the file as it was by sending an *unCheckOut* to the CC, or
3. insert a modified version of the file into the vault—while keeping the file in his or 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 publish/subscribe 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 he or she receives a *notify* whenever another user extracts this file from the vault. Similarly, he or she receives an *update* whenever another user inserts (publishes) a file in the vault.

3 Model Checking the thinkteam Protocol

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

Model checking is an automatic technique to verify whether a concurrent system design satisfies its specifications [6]. Such a verification is moreover exhaustive, i.e. all possible input combinations and states are taken into account.

One of the best known and most successful model checkers is SPIN, which was developed at Bell Labs during the last two decades [16]. It offers a spectrum of verification techniques, ranging from partial to exhaustive verification, is freely available through `www.spinroot.com`, and is very well documented.

PROMELA is a non-deterministic C-like specification language for modelling finite-state systems communicating through channels [16]. Formally, specifications in PROMELA are built from processes, data objects, and message channels. Processes are the components of the system, while the data objects are its local and global variables. The message channels, finally, are used to transmit data between processes. Such channels can be local or global and they can be FIFO buffered—for modelling asynchronous communication—or handshake (a.k.a. rendezvous)—for modelling synchronous communication. Assume that processes A and B are connected by a channel aToB. Then A can send a message m to B over this channel by executing the statement `aToB!m`. If aToB is a buffered channel and its buffer is not full, then m is stored in the buffer until B executes `aToB?m` and thereby receives m from A over this channel. This is an example of asynchronous communication between A and B. If, on the other hand, aToB is a handshake channel, then the two above executions must be synchronised, i.e. aToB can pass but not store messages. This is an example of synchronous communication. For more detailed information on PROMELA, we refer the reader to [16].

PROMELA specifications can be checked against correctness properties specified as Linear Temporal Logic (LTL for short) formulae by means of SPIN. SPIN 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 formalise either reachability of states 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 *assertions* to a PROMELA specification and 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 assertion

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

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

Secondly, we may formulate an LTL property and test its validity against the PROMELA specification, possibly enriched with specific *labels* identifying relevant points of process executions, by running SPIN. LTL is an extension of predicate logic allowing one to express assertions about behaviour in time, without explicitly modelling time. SPIN accepts formulae in LTL that are constructed on the basis of atomic propositions (including `true` and `false`), the Boolean connectives ! (negation), && (and), || (or), -> (implication), and <-> (equivalence), and the temporal operators [] (always), <> (eventually), and `U` (until). Given a sequence σ of states from the behaviour of a system starting in the initial state, 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 `U` q is `true` if the property p remains `true` in the states of σ until the property q becomes `true` in a state of σ. For more detailed information on LTL, we refer the reader to [19].

As an example, consider that we want to guarantee that the PROMELA specification of the thinkteam protocol is such that a user is always able to *get* a file if he or she so wishes. Then we can add the labels

`todoGet` and `doneGet`

to the User process of the PROMELA specification directly *before* and *after*, respectively, the moment in which the user may send a *get* to the CC, i.e. encompassing the statement

```
userToCC!get,id.
```

In this statement, `id` identifies the particular user that sends a *get* to the CC via the userToCC (handshake) channel. We can then formulate the LTL formulae

[] (User[pid]@todoGet -> <> User[pid]@doneGet),

where pid is the *process instantiation number* of the User process about which we want to know whether all sequences of states from the complete behaviour of the PROMELA specification of the thinkteam protocol that contain a state in which this user's state is the label `todoGet` also contain a state (further in time) in which this user's state is the label `doneGet`. Starting with 0, SPIN assigns—in order of creation—a unique pid to each process it creates, which can be used in LTL formulae for process identification. Finally, we can verify the validity of the above LTL formulae by running SPIN. Again, when SPIN concludes that this statement is not valid, then it also presents a counterexample. Otherwise it simply reports that the statement is valid.

3.1 The Promela Specification

In this section we discuss the PROMELA specification of the thinkteam protocol that can be found in Appendix A. In particular, we list the assumptions that we have made in our specification, as well as the improvements that have resulted from detailed discussions with think3.

Recall that our focus on the concurrency control and publish/subscribe notification aspects of the thinkteam protocol has led to the abstract model described in Section 2.4. On top of the list of abstractions described there, we have made several more assumptions in our specification. The reason for most of these additional assumptions is to reduce the size of the state space, as well as that of the state vector, as much as possible. The state vector is used by SPIN to uniquely identify a system state and contains information on the global variables, the channel contents, and for each process its local variables and its process counter. Minimising its size thus results in less bytes that SPIN needs to store for each system state and thereby further reduces the risk to run out of memory. Finally, as noted in [16], 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. With this in mind, the most important assumptions that we have made in the PROMELA specification of the thinkteam protocol are the following.

1. 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 small buffer, without immediately causing a state-space explosion.[4]
2. 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.
3. 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.
4. 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*.
5. 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.

[4] Handshake channels `userToCC` and `ccToUserAdmin[numUsers]` can be turned into channels with buffer 2 and 1, respectively, without creating deadlocks in any case upto 4 users.

This enabled us to detect a number of ambiguities and unclear aspects of the design that think3 has in mind of the kind of publish/subscribe notification service they want to add to thinkteam. think3 had no previous experience with model checking. Some central questions that were addressed in these meetings, and there answers, are as follows.

When exactly are which requests enabled? Originally we had prohibited a user to do a *get* after it had already initiated a *checkOut* on the same file. However, think3 said that a *get* must be allowed also after a *checkedOut* has been obtained. Such a *get* should be seen as a kind of "re-get".

What is the exact semantics of requests? Originally we had allowed a user to (un)register for a file for which it actually possessed editing rights. think3 however said that this must be prohibited. We do allow a user to (un)register for a file as long as it is waiting for a response to the *checkOut* it sent for this file. After all, this response might also be negative.

How are simultaneous requests handled? As mentioned before, access to documents currently is based on the "retrial" principle: there is no queue (or reservation system) handling simultaneous requests for a document. However, think3 has expressed interest in equipping thinkteam with a document reservation system.

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 concerning the awareness and concurrency control aspects of the thinkteam protocol with SPIN.

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.

First and foremost we have let SPIN perform a full statespace search for invalid endstates, which is SPIN's formalisation of deadlock states. In case of 2 users, this resulted in the following output.

```
(Spin Version 4.1.3 -- 24 April 2004)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim             - (none specified)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 84 byte, depth reached 4423, errors: 0
   20686 states, stored
   13533 states, matched
```

```
   34219 transitions (= stored+matched)
       9 atomic steps
hash conflicts: 10 (resolved)
(max size 2^23 states)

Stats on memory usage (in Megabytes):
1.986   equivalent memory usage for states (stored*(State-vector+overhead))
0.902   actual memory usage for states (compression: 45.40%)
        State-vector as stored = 32 byte + 12 byte overhead
33.554  memory used for hash table (-w23)
3.200   memory used for DFS stack (-m100000)
37.574  total actual memory usage

real        1.6
user        1.3
sys         0.1
```

For 2 users it thus takes SPIN just over 1 second to conclude that there are no deadlocks. In case of 3 users, on the other hand, the following output was obtained.

```
(Spin Version 4.1.3 -- 24 April 2004)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim             - (none specified)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 108 byte, depth reached 434033, errors: 0
1.86628e+06 states, stored
2.51414e+06 states, matched
4.38041e+06 transitions (= stored+matched)
      12 atomic steps
hash conflicts: 175055 (resolved)
(max size 2^23 states)

Stats on memory usage (in Megabytes):
223.953 equivalent memory usage for states (stored*(State-vector+overhead))
65.055  actual memory usage for states (compression: 29.05%)
        State-vector as stored = 23 byte + 12 byte overhead
33.554  memory used for hash table (-w23)
16.000  memory used for DFS stack (-m500000)
114.783 total actual memory usage

real     3:12.6
user     3:06.5
sys         1.3
```

For 3 users it thus takes SPIN just over 3 minutes to conclude that there are no deadlocks.

In case of 4 users, finally, the available physical memory was insufficient for a full statespace search for invalid endstates. However, by disabling the explicit *register* (but thus still allowing implicit registration for a file by means of a *get*) and by enabling SPIN's *minimized automaton procedure* with 28 as the maximal depth of the graph that is constructed for the minimized automaton representation (cf. [16] for details), it takes SPIN just over 8 hours (during which it reaches a depth of 10484899—a factor of almost 25 more than in the case of 3 users) to generate the following output.

```
(Spin Version 4.1.3 -- 24 April 2004)
        + Partial Order Reduction
        + Compression
        + Graph Encoding (-DMA=28)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 132 byte, depth reached 10484899, errors: 0
Minimized Automaton:    4197939 nodes and 2.11151e+07 edges
4.60979e+07 states, stored
8.48633e+07 states, matched
1.30961e+08 transitions (= stored+matched)
      15 atomic steps
hash conflicts: 0 (resolved)
(max size 2^23 states)

Stats on memory usage (in Megabytes):
6638.093 equivalent memory usage for states (stored*(State-vector+overhead))
453.906  actual memory usage for states (compression: 6.84%)
33.554   memory used for hash table (-w23)
420.000  memory used for DFS stack (-m15000000)
916.095  total actual memory usage

real  8:36:28.7
user  8:18:36.5
sys        41.6
```

The results of the full statespace searches for invalid endstates reported in this section are summarised in Table 1. In this table, the runtime is given as hours:minutes:seconds.

These 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 the unsuccessful application of model checking on groupware systems in the past [24].

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.

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 publish/subscribe 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 it 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 Appendix A. This shows 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. 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 the thinkteam protocol's locking-based concurrency control mechanism, as 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 any 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—it instead responds with

a *notAvailable*. To verify this property we add a number of user-specific labels to the PROMELA specification of the CC process, viz. (with `X` $\in \{0, 1, 2\}$)

`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 sends a *checkedOut* to user `X=id`, and
`doneNotAvailableX` directly follows `ccToUser[id]!notAvailable`, by which the CC sends a *notAvailable* to user `X=id`.

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*:

$$[]\,(\text{CC[2]@doneCheckOut0} \rightarrow <>\,(\text{CC[2]@doneCheckedOut0} \,||\, \text{CC[2]@doneNotAvailable0})).$$

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.

Before we continue it is important to note that the above formulae are trivially valid if the left-hand sides of the implications are always `false` in every run, i.e. if the CC never passes the `doneCheckOutX` labels. To this aim, we also let SPIN run verifications of the LTL formulae

$$!\,(<>\,\text{CC[2]@doneCheckOut0})$$

as well as of analogous versions for users 1 and 2. We thus verify whether there exist runs in which the left-hand sides of the implications eventually become `true` by verifying whether the CC can ever pass the `doneCheckOutX` labels. It takes SPIN just a split second to conclude that the above LTL formulae are *not valid*. It moreover provides counterexamples which show that the CC can eventually pass the `doneCheckOutX` labels. Though never mentioned specifically, for all formulae in the sequel that contain a logical implication we have verified that the left-hand side of this implication can indeed actually become `true` in at least one run.

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 that file. 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 PROMELA specification of the CC process the moment in which it is about to grant a lock to a user by sending it a *checkedOut*, i.e. immediately preceding

`writeLock = true; ccToVault!checkOut,id`. Consequently, we 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 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 states that every lock on a file must eventually be released. In the thinkteam protocol, the CC releases the lock on file 0 when it receives either a *checkIn* or an *uncheckOut* from the user to whom it had last granted the lock via *checkedOut*. We thus need to verify that every *checkedOut* is eventually followed by a *checkIn* or an *unCheckOut* from the same user to whom the *checkedOut* was sent. To verify this property we add two more user-specific labels to the PROMELA specification of the CC process, viz. (with `X` $\in \{0, 1, 2\}$)

`doneCheckInX` directly follows `userToCC?checkIn,id`, by which the CC receives a *checkIn* from user `X=id`, and

`doneUnCheckOutX` directly follows `userToCC?unCheckOut,id`, by which the CC receives an *unCheckOut* from user `X=id`.

Subsequently, we 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. We thus verify whether it is always the case that whenever the CC has granted a lock request on file 0 to a user through a *checkedOut*, then this user eventually releases this lock by sending the CC a *checkIn* or an *unCheckOut*. It takes SPIN just a split second to conclude that the above 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*, or *checkInOut* ad infinitum. This gave us the idea to re-run SPIN but this time with its weak fairness option enabled.

A run or computation of SPIN 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.

It takes SPIN just a split second to conclude that even with weak fairness enabled, the above LTL formulae are *not valid*. Again the provided counterexamples

are clear: a user that holds the lock on file 0 can endlessly perform *checkInOut* and thus never release the lock on file 0. This is an unavoidable property of the thinkteam protocol. In practice this undesirable situation is avoided in thinkteam by a superuser or system administrator that can be contacted by a certain user with the request to "convince" another user towards releasing the file it currently 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 this means that the CC may 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 PROMELA 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 through the atomic sequence of statements labelled `Update`, i.e. directly preceding `Update: atomic`. We then let SPIN run a verification on assertion violations and as such we thus verify whether it is always the case that the boolean variable `writeLock` is `true` (indicating that a user currently has a lock in its possession) the moment in which the CC is about to update all registered users of the fact that the user that currently has a file locked in its possession, has published an intermediate version of the file in the vault. In just over 3 minutes SPIN concludes that the above basic assertion is never violated, which proves that the property is valid.

4.3 Awareness

In this section we verify the three properties dealing with user awareness through the thinkteam protocol's publish/subscribe notification service, as formulated in Section 4.1.

No illegal notify (update). The first property states that a user does not receive a *notify* nor an *update* if it is not registered for the file these messages refer to. In other words, 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*. This is not enough, though, since a user could *unRegister* in between the *get* or *register* and the *notify* (*update*). Therefore we moreover require that in between an *unRegister* and a *notify* (*update*), no *get* or *register* may take place.

To this aim, we add a couple of labels to the PROMELA specification of the User process, viz.

`doneRegister` directly follows `userToCC!register,id`, by which user `id` sends a *register* to the CC and

`doneUnRegister` directly follows `userToCC!unRegister,id`, by which user `id` sends an *unRegister* to the CC.

We furthermore add several labels to the PROMELA specification of the UserAdmin process, viz.

`doneNotify` directly follows `ccToUserAdmin[id]?notify`, by which the UserAdmin process associated to user `id` receives a *notify* from the CC and
`doneUpdate` directly follows `ccToUserAdmin[id]?update`, by which the UserAdmin process associated to user `id` receives an *update* from the CC.

Subsequently, we let SPIN run verifications of the LTL formulae

$$!(!(\text{User[3]@doneGet} \,||\, \text{User[3]@doneRegister})\ \mathtt{U}\ \text{UserAdmin[4]@doneLab}) \;\&\&\; [](\text{User[3]@doneUnRegister} \rightarrow !(!(\text{User[3]@doneGet} \,||\, \text{User[3]@doneRegister})\ \mathtt{U}\ \text{UserAdmin[4]@doneLab})),$$

where doneLab is doneNotify or doneUpdate. Moreover, we also let SPIN run 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* or an *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 another number of user-specific labels to the PROMELA specification of the CC process, viz. (with $\mathtt{X} \in \{0, 1, 2\}$)

`doneGetX` directly follows `userToCC?get,id`, by which the CC receives a *get* from user `id`,
`doneRegisterX` directly follows `userToCC?register,id`, by which the CC receives a *register* from user `id`,
`doneUnRegisterX` directly follows `userToCC?unRegister,id`, by which the CC receives an *unRegister* from user `id`, and
`doneNotifyX` directly follows `ccToUserAdmin[ID]!notify`, by which the CC sends a *notify* to (registrated) user `ID`.

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:

$$[]\,!((\text{CC[2]@doneGet0} \,||\, \text{CC[2]@doneRegister0}) \;\&\&\; (<>\,(\text{CC[2]@doneCheckedOut1} \,||\, \text{CC[2]@doneCheckedOut2})) \;\&\&\; (!\,\text{CC[2]@doneUnRegister0}\ \mathtt{U}\ ((\text{CC[2]@doneCheckedOut1} \,||\, \text{CC[2]@doneCheckedOut2}) \;\&\&\; []\,!\,\text{CC[2]@doneNotify0}))).$$

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 the above formulae are valid.

Update if registered. The third property states that 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 messages refer. To verify this property we add several more user-specific labels to the PROMELA specification of the CC process, viz. (with `X` $\in \{0, 1, 2\}$)

`doneCheckedInX` directly follows `vaultToCC?update,id`, by which the CC receives confirmation from the Vault of the fact that the file, which the CC received from user `id` by `userToCC?checkIn,id` and which the CC forwarded to the Vault by `ccToVault!checkIn,id`, has indeed been inserted into the Vault,

`doneCheckedInOutX` directly follows `vaultToCC?update,id`, by which the CC receives confirmation from the Vault of the fact that the file, which the CC received from user `id` by `userToCC?checkInOut,id` and which the CC forwarded to the Vault by `ccToVault!checkIn,id`, has indeed been inserted into the Vault, and

`doneUpdateX` directly follows `ccToUserAdmin[ID]!update`, by which the CC sends an *update* to (registrated) user `ID`.

In a similar way as above, these labels allow us to formulate, e.g., that it may never be the case that user 1 or user 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:

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

where

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

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 think-team protocol in particular is that its users cannot be denied a service forever.

In Section 3, e.g., we formulated the property that a user is always able to *get* a file if he or she so wishes. To this aim, we augmented the User process of the PROMELA specification with the label `todoGet` directly *before* and the label `doneGet` directly *after* the moment in which the user may send a *get* to the CC, i.e. encompassing the statement `userToCC!get,id`, and we formulated the LTL formulae

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

where pid equals 3 (for user 0), 5 (for user 1), or 7 (for user 2). Subsequently, 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, as the latter is continuously kept busy by the other users. 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 that it 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 system administrator or superuser 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, in which the runtime is again given as hours:minutes:seconds. We recall that all verifications have been performed for the case of 3 users.

The verifications performed in this paper show that the concurrency control and awareness aspects of the thinkteam protocol completed with a publish/subscribe notification service are well designed. We have seen, however, that according to the thinkteam protocol a user is not obliged to ever return a file that it has checked out to the Vault. In thinkteam this situation is dealt with by means of a system administrator or superuser which can intervent in such sitations and force the particular user to return the file it has checked out to the vault.

property	verification	state vector	depth reached	errors	memory used	runtime
CC-1	Respond to lock	112 byte	3147677	0	473.209 Mb	16:54.1
CC-2	Unique lock/file	108 byte	434033	0	114.783 Mb	3:06.0
CC-3	Release file+lock	116 byte	7348	1	193.862 Mb	0.9
CC-4	Keep file locked	108 byte	434033	0	114.783 Mb	3:06.0
AW-1a	No illegal notify	112 byte	3071518	0	539.769 Mb	21:22.1
AW-1b	No illegal update	112 byte	3057025	0	558.508 Mb	22:45.4
AW-2	Notify if registered	112 byte	3338868	0	967.955 Mb	39:22.2
AW-3	Update if registered	112 byte	4183223	0	925.049 Mb	38:57.6
DoS	Denial of Service	116 byte	1801	1	193.759 Mb	0.4

Table 2. Results of the verifications performed in this paper.

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 publish/subscribe 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 the thinkteam protocol is well designed with respect to these aspects. To the best of our knowledge, this is one of the first successful applications of exhaustive model checking techniques to the verification of publish/subscribe notification services in a groupware setting.

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 publish/subscribe notification services. In fact, it is think3's intention to use them as basis for their planned implementation of a publish/subscribe notification service in thinkteam. In this respect the related approach of [22] 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 (reported in this paper) 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 publish/subscribe notifi-

cation 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 2.3 that one of the reasons for think3's desire to add a publish/subscribe notification service to thinkteam was to be able to solve the variant of the "lost update" phenomenon depicted in Fig. 3. 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.

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, 18] and formal specification-driven discrete simulation tools [7]).

6 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".

References

1. R.M. Baecker (ed.), *Readings in Groupware and Computer Supported Cooperation Work—Assisting Human-Human Collaboration*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.
2. C. Baier, B.R. Haverkort, H. Hermanns, and J.-P. Katoen, Automated Performance and Dependability Evaluation Using Model Checking. In *Performance Evaluation of Complex Systems: Techniques and Tools—Performance'02 Tutorial Lectures, LNCS* 2459, Springer-Verlag, Berlin, 2002, 261–289.
3. M.H. ter Beek, M. Massink, D. Latella, and S. Gnesi, Model Checking Groupware Protocols. In *Cooperative Systems Design—Scenario-Based Design of Collaborative Systems* (F. Darses, R. Dieng, C. Simone, and M. Zacklad, eds.), *Frontiers in Artificial Intelligence and Applications* 107, IOS Press, Amsterdam, 2004.
4. M. Caporuscio, 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.

5. M. Caporuscio, 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.
6. E.M. Clarke Jr., O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, Cambridge, MA, 1999.
7. P.R. D'Argenio, J.-P. Katoen, and E. Brinksma, General Purpose Discrete Event Simulation Using ⌂. In *Proceedings PAPM'98*, Università degli Studi di Verona, 1998, 85–102.
8. X. Deng, M.B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh, Model-Checking Middleware-Based Event-Driven Real-Time Embedded Software. In *Proceedings FMCO'02, LNCS* 2852, Springer-Verlag, Berlin, 2002, 154–181.
9. P. Dourish and V. Bellotti, Awareness and Coordination in Shared Workspaces. In *Proceedings CSCW'92*, ACM Press, New York, 107–114.
10. C.A. Ellis and S.J. Gibbs, Concurrency Control in Groupware Systems. In *Proceedings SIGMOD'89*, ACM Press, New York, 399–407.
11. C.A. Ellis, S.J. Gibbs, and G.L. Rein, Groupware—Some Issues and Experiences. *Communications of the ACM* 34, 1 (1991), 38–58.
12. P.Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, The Many Faces of Publish/Subscribe. *ACM Computing Surveys* 35, 2 (2003), 114–131.
13. D. Garlan, S. Khersonsky, and J.S. Kim, Model Checking Publish-Subscribe Systems. In *Proceedings SPIN'03, LNCS* 2648, Springer-Verlag, Berlin, 2003, 166–180.
14. J. Grudin, CSCW: History and Focus. *IEEE Computer* 27, 5 (1994), 19–26.
15. C. Gutwin, M. Roseman, and S. Greenberg, Supporting Awareness of Others in Groupware. In *Companion Proceedings SIGCHI'96*, ACM Press, New York, 1996, 205–215.
16. G.J. Holzmann, *The SPIN Model Checker—Primer and Reference Manual*, Addison Wesley Publishers, Reading, MA, 2003.
17. P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann, Qualitätssicherung Software-basierter technischer Systeme—Problembereiche und Lösungsansätze. *Informatik Spektrum* 21, 5 (1998), 249–258.
18. M.Z. Kwiatkowska, G. Norman, and D. Parker, Probabilistic Symbolic Model Checking with PRISM—A Hybrid Approach. In *Proceedings TACAS'02, LNCS* 2280, Springer-Verlag, Berlin, 2002, 52–66.
19. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems—Specification*, Springer-Verlag, Berlin, 1992.
20. C. Papadopoulos, An Extended Temporal Logic for CSCW. *The Computer Journal* 45, 4 (2002), 453–472.
21. K. Schmidt, The Problem with 'Awareness'—Introductory Remarks on 'Awareness in CSCW'. *Computer Supported Cooperative Work—The Journal of Collaborative Computing* 11, 3-4 (2002), 285–298.
22. M. Tivoli, P. Inverardi, V. Presutti, A. Forghieri, and M. Sebastianis, Correct Components Assembly for a Product Data Management Cooperative System. In *Proceedings CBSE'04, LNCS* 3054, Springer-Verlag, Berlin, 2004, 84–99.
23. S. Tripakis and S. Yovine, Timing Analysis and Code Generation of Vehicle Control Software using Taxys. In *Proceedings RV'01, ENTCS* 55, 2, Elsevier Science Publishers, Amsterdam, 2001, 174–183.
24. T. Urnes, Efficiently Implementing Synchronous Groupware, Ph.D. thesis, Department of Computer Science, York University, Toronto (1998).
25. L. Zanolin, C. Ghezzi, and L. Baresi, An Approach to Model and Validate Publish/Subscribe Architectures. In *Proceedings SAVCBS'03*, Technical Report 03-11, Department of Computer Science, Iowa State University, 2003, 35–41.

A The Complete Promela Specification

```
/* Macros */

#define numUsers 3
#define numFiles 1  /* several variables below should be defined per file if numFiles > 1 */

/* Handshake and other communication channels */

mtype =
{
  get, got, checkOut, checkedOut, notAvailable, unCheckOut,                    /* User */
  import, checkIn, checkInOut,                                               /* publish */
  register, unRegister,                                                    /* subscribe */
  update, notify,                                                        /* notification */
};

/* Channels between client and server */

chan userToCC = [0] of {mtype, byte};
chan ccToUser[numUsers] = [1] of {mtype};
chan ccToUserAdmin[numUsers] = [0] of {mtype};

/* Internal server channels */

chan ccToVault = [0] of {mtype, byte};
chan vaultToCC = [0] of {mtype, byte};

/* Client processes */

proctype User(byte id)
{
  byte edit[numFiles], registered[numFiles];                     /* User status file 0 */
  bool waitingForCheckedOut = false;                             /* status of checkOut */

  do
  :: (!waitingForCheckedOut) ->                /* if User didn't try to checkout file 0, */
       todoGet: skip;                               /* (label for verification purposes) */
       userToCC!get,id;                               /* then it may send get to CC and */
       doneGet: skip;                               /* (label for verification purposes) */
       ccToUser[id]?got;                          /* as soon as it receives got from CC, */
       registered[0] = true                              /* it is registered for file 0 */
  :: (!registered[0] && !edit[0]) ->  /* if User didn't register for nor checkout file 0, */
       registered[0] = true;           /* then it may want to be registered for file 0 by */
       userToCC!register,id;                                  /* sending register to CC */
       doneRegister: skip                           /* (label for verification purposes) */
  :: (registered[0] && !edit[0]) ->  /* if User did register for but not checkout file 0, */
       registered[0] = false;      /* then it may not want to be registered for file 0 by */
       userToCC!unRegister,id;                               /* sending unregister to CC */
       doneUnRegister: skip;                        /* (label for verification purposes) */
  :: (!edit[0] && !waitingForCheckedOut) ->   /* if User didn't (try to) checkout file 0, */
       waitingForCheckedOut = true;          /* then it may start waitingForCheckedOut by */
       userToCC!checkOut,id                                   /* sending checkOut to CC */
  :: (edit[0]) ->                           /* if User has checkedOut file 0, then it may */
       if
       :: userToCC!unCheckOut,id ->                  /* either send UnCheckOut to CC and */
          edit[0] = false                                     /* no longer edit file 0, */
       :: userToCC!checkIn,id ->                           /* or send checkIn to CC and */
          edit[0] = false                                     /* no longer edit file 0, */
       :: userToCC!checkInOut,id                           /* or send checkInOut to CC */
       fi                                                   /* (and still edit file 0) */
  :: ccToUser[id]?checkedOut ->           /* as soon as User receives checkedOut from CC, */
       d_step{edit[0] = true;                                 /* it may edit file 0 and */
       waitingForCheckedOut = false}                    /* thus stop waitingForCheckedOut */
  :: ccToUser[id]?notAvailable ->       /* as soon as User receives notAvailable from CC, */
       waitingForCheckedOut = false                /* then it stops waitingForCheckedOut */
/* TEMPORARILY DISABLED AS LONG AS numFiles = 1 :                                        */
/*  :: userToCC!import,id;                                    User sends import to CC and */
```

```
/*                                                    a file is added to numFiles */
  od
}

proctype UserAdmin(byte id)
{
  do
  :: ccToUserAdmin[id]?notify;                          /* User receives notify from CC */
       doneNotify: skip                                /* (label for verification purposes) */
  :: ccToUserAdmin[id]?update ->                        /* User receives update from CC */
       doneUpdate: skip                                /* (label for verification purposes) */
  od
}

/* Server processes */

proctype ConcurrencyController()
{
  byte id, ID, registered[numUsers];                  /* registration per User for file 0 */
  bool writeLock = false;                                       /* status lock for file 0 */

  do
  :: userToCC?get,id ->                                  /* upon receiving get from User, */
       if
       :: (id == 0) ->
            doneGet0: skip                             /* (label for verification purposes) */
       :: (id == 1) ->
            doneGet1: skip                             /* (label for verification purposes) */
       :: (id == 2) ->
            doneGet2: skip                             /* (label for verification purposes) */
       fi;
       registered[id] = true;                             /* User is registered for file 0, */
       ccToVault!get,id;                                     /* CC sends get to Vault, and, */
       vaultToCC?got,id;                                  /* upon receiving got from Vault, */
       ccToUser[id]!got                                             /* CC sends got to User */
  :: userToCC?register,id ->                           /* upon receiving register from User, */
       if
       :: (id == 0) ->
            doneRegister0: skip                        /* (label for verification purposes) */
       :: (id == 1) ->
            doneRegister1: skip                        /* (label for verification purposes) */
       :: (id == 2) ->
            doneRegister2: skip                        /* (label for verification purposes) */
       fi;
       registered[id] = true                                 /* User is registered for file 0 */
  :: userToCC?unRegister,id ->                       /* upon receiving unRegister from User, */
       if
       :: (id == 0) ->
            doneUnRegister0: skip                      /* (label for verification purposes) */
       :: (id == 1) ->
            doneUnRegister1: skip                      /* (label for verification purposes) */
       :: (id == 2) ->
            doneUnRegister2: skip                      /* (label for verification purposes) */
       fi;
       registered[id] = false                     /* User is no longer registered for file 0 */
  :: userToCC?checkOut,id ->                     /* whenever CC receives checkOut from User: */
       if
       :: (id == 0) ->
            doneCheckOut0: skip                        /* (label for verification purposes) */
       :: (id == 1) ->
            doneCheckOut1: skip                        /* (label for verification purposes) */
       :: (id == 2) ->
            doneCheckOut2: skip                        /* (label for verification purposes) */
       fi;
       if
       :: !writeLock ->                       /* (1) if there is no writeLock on file 0, then */
            assert(writeLock == false);             /* (assertion for verification purposes) */
            writeLock = true;                                /* CC sets writeLock on file 0, */
```

```
          ccToVault!checkOut,id;                        /* sends checkOut to Vault, and, */
          vaultToCC?checkedOut,id;              /* upon receiving checkedOut from Vault, */
          ccToUser[id]!checkedOut;                 /* sends checkedOut to User (id), and */
          if
          :: (id == 0) ->
               doneCheckedOut0: skip               /* (label for verification purposes) */
          :: (id == 1) ->
               doneCheckedOut1: skip               /* (label for verification purposes) */
          :: (id == 2) ->
               doneCheckedOut2: skip               /* (label for verification purposes) */
          fi;
          ID = 0;
          do
          :: (ID < numUsers) ->
               if
               :: (ID != id && registered[ID]) ->       /* to each registered User, not */
                    ccToUserAdmin[ID]!notify;           /* equal to id, CC sends notify */
                    if
                    :: (ID == 0) ->
                         doneNotify0: skip         /* (label for verification purposes) */
                    :: (ID == 1) ->
                         doneNotify1: skip         /* (label for verification purposes) */
                    :: (ID == 2) ->
                         doneNotify2: skip         /* (label for verification purposes) */
                    fi
               :: else -> skip
               fi;
               ID++
          :: else -> break
          od
     :: else ->                           /* (2) if there is a writeLock on file 0, then */
          ccToUser[id]!notAvailable;                   /* CC sends notAvailable to User */
          if
          :: (id == 0) ->
               doneNotAvailable0: skip             /* (label for verification purposes) */
          :: (id == 1) ->
               doneNotAvailable1: skip             /* (label for verification purposes) */
          :: (id == 2) ->
               doneNotAvailable2: skip             /* (label for verification purposes) */
          fi;
     fi
:: userToCC?unCheckOut,id ->                     /* upon receiving unCheckOut from User, */
     if
     :: (id == 0) ->
          doneUnCheckOut0: skip                    /* (label for verification purposes) */
     :: (id == 1) ->
          doneUnCheckOut1: skip                    /* (label for verification purposes) */
     :: (id == 2) ->
          doneUnCheckOut2: skip                    /* (label for verification purposes) */
     fi;
     writeLock = false;                              /* CC sets writeLock to false, and */
     goto Update                                        /* updates all registered users */
:: userToCC?checkIn,id ->                           /* upon receiving checkIn from User, */
     if
     :: (id == 0) ->
          doneCheckIn0: skip                       /* (label for verification purposes) */
     :: (id == 1) ->
          doneCheckIn1: skip                       /* (label for verification purposes) */
     :: (id == 2) ->
          doneCheckIn2: skip                       /* (label for verification purposes) */
     fi;
     ccToVault!checkIn,id;                                 /* CC sends checkIn to Vault, */
     writeLock = false;                                /* sets writeLock to false, and, */
     vaultToCC?update,id;               /* upon receiving update (by User id) from Vault, */
     if
     :: (id == 0) ->
          doneCheckedIn0: skip                     /* (label for verification purposes) */
     :: (id == 1) ->
```

```
            doneCheckedIn1: skip                    /* (label for verification purposes) */
       :: (id == 2) ->
            doneCheckedIn2: skip                    /* (label for verification purposes) */
       fi;
       goto Update                                       /* updates all registered users */
  :: userToCC?checkInOut,id ->                    /* upon receiving checkInOut from User, */
       ccToVault!checkIn,id;                             /* CC sends checkIn to Vault and */
       vaultToCC?update,id;             /* upon receiving update (by User id) from Vault, */
       if
       :: (id == 0) ->
            doneCheckedInOut0: skip                 /* (label for verification purposes) */
       :: (id == 1) ->
            doneCheckedInOut1: skip                 /* (label for verification purposes) */
       :: (id == 2) ->
            doneCheckedInOut2: skip                 /* (label for verification purposes) */
       fi;
       assert(writeLock == true);                /* (assertion for verification purposes) */
Update:ID = 0;                                           /* updates all registered users: */
       do
       :: (ID < numUsers) ->
            if
            :: (ID != id && registered[ID]) ->           /* to every registered User, not */
                 ccToUserAdmin[ID]!update;               /* equalling id, CC sends update */
                 if
                 :: (ID == 0) ->
                      doneUpdate0: skip             /* (label for verification purposes) */
                 :: (ID == 1) ->
                      doneUpdate1: skip             /* (label for verification purposes) */
                 :: (ID == 2) ->
                      doneUpdate2: skip             /* (label for verification purposes) */
                 fi
            :: else -> skip
            fi;
            ID++
       :: else -> break
       od
/* TEMPORARILY DISABLED AS LONG AS numFiles = 1 :                                        */
/*  :: userToCC?import,id ->                              upon receiving import from User, */
/*       ccToVault!import,id                                     CC sends import to Vault */
  od
}

proctype Vault()
{
  byte id;

  do
  :: ccToVault?get,id ->                                  /* upon receiving get from CC, */
       vaultToCC!got,id                                        /* Vault sends got to CC */
  :: ccToVault?checkOut,id ->                          /* upon receiving checkOut from CC, */
       vaultToCC!checkedOut,id                           /* Vault sends checkedOut to CC */
  :: ccToVault?checkIn,id ->                            /* upon receiving checkIn from CC, */
       vaultToCC!update,id                         /* Vault sends update (by User id) to CC */
/* TEMPORARILY DISABLED AS LONG AS numFiles = 1 :                                        */
/*  :: ccToVault?import,id                                  Vault receives import from CC */
  od
}

/* Initialisation process */

init
{
  byte Users = 0;

  atomic
  {
    run Vault();                                                              /* pid = 1 */
    run ConcurrencyController();                                              /* pid = 2 */
```

```
    run User(Users);                                          /* User(0) with pid = 3 */
    run UserAdmin(Users);                                   /* UserAdmin(0) with pid = 4 */
    Users++;
    do
    :: (Users < numUsers) ->
         run User(Users);             /* User(1) with pid = 5, User(2) with pid = 7, etc. */
         run UserAdmin(Users);   /* UserAdmin(1) with pid = 6, UserAdmin(2) with pid = 8, */
         Users++                                                                 /* etc. */
    :: else -> break
    od
  }
}
```