A Case Study on the Automated Verification of Groupware Protocols

Maurice H. ter Beek
ISTI–CNR
Via G. Moruzzi 1
56124 Pisa, Italy
m.terbeek@isti.cnr.it

Mieke Massink
ISTI–CNR
Via G. Moruzzi 1
56124 Pisa, Italy
m.massink@isti.cnr.it

Diego Latella
ISTI–CNR
Via G. Moruzzi 1
56124 Pisa, Italy
d.latella@isti.cnr.it

Stefania Gnesi
ISTI–CNR
Via G. Moruzzi 1
56124 Pisa, Italy
s.gnesi@isti.cnr.it

Alessandro Forghieri
think3, Inc.
Via Ronzani 7/29
40033 Bologna, Italy
forghier@think3.com

Maurizio Sebastianis
think3, Inc.
Via Ronzani 7/29
40033 Bologna, Italy
sebastia@think3.com

ABSTRACT

We report on a fruitful combination of applying academic experience with formal modelling and verification techniques to an industrial case study. The goal of the case study was to investigate *a priori*, *i.e.* before implementation, the effects of adding a lightweight and easy-to-use publish/subscribe (event) notification service to thinkteam®—an asynchronous and dispersed groupware system which was developed by think3. Researchers from the Formal Methods and Tools (FM&&T) group of ISTI–CNR—with a longstanding experience in research on the development and application of formal methods, notations, and software tools for the specification, design, and verification of complex computer systems—therefore teamed up with think3—a global provider of integrated product development solutions that provides mechanical design and Product Data Management (PDM) software catering the product management needs of design processes in the manufacturing industry. The technical details of this joint research effort have been documented elsewhere, here we report on the lessons learned from this experience.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Experimentation,Verification

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

Keywords

Groupware, Model checking, Publish/subscribe notification, thinkteam

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]. Many concepts and techniques from computer science, such as concurrency control, need to be rethought in the groupware domain. This led to the development of new formal models like team automata, which were introduced explicitly for the description and analysis of groupware systems [2, 9, 16]. 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 was referred to as the time space taxonomy by Ellis *et al.* [10]. In Table 1 some example groupware systems have been classified.

groupware	*synchronous*	*asynchronous*
co-located	electronic whiteboards	version-control
dispersed	video-conferencing	electronic mail

Table 1: Example groupware systems.

An additional difficulty that arises when designing groupware is the inherently concurrent and distributed nature of such systems. This forces one to address issues like (network) communication, concurrency control, and distributed (event) notification, and makes groupware systems notoriously hard to analyse on paper or by traditional means such as testing and simulation. However, increased computing resources together with an enormous improvement in the

efficiency of model-checking algorithms in recent years, has facilitated the application of model checking to ever more complex systems of a concurrent and distributed nature. Model checking provides the automatic analysis of correctness properties of system designs in an efficient way. Such a verification can moreover be exhaustive, *i.e.* all possible input combinations and states can be taken into account, while counterexamples are generated 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 (the protocols underlying) groupware systems [3, 4, 5, 19, 22].

An essential ingredient of many a successful groupware system is awareness, which is a frequently used but seldom precisely defined notion from the field of CSCW [20]. Roughly speaking, it should be understood as the users of a groupware system 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 [8, 14]. The more aware the users are, the more they can coordinate their collaborative tasks in a fruitful manner. A widely used way to increase the awareness of the users of a groupware system is by enhancing it with a publish/subscribe (event) notification service. Such a service uses multicast communication to automatically inform those users that expressed an interest in ('are subscribed to') a particular kind of event whenever a user performs ('publishes') an event of this kind. Due to a potentially large number of users, it is fundamental to use subscription-based multicast communication rather than broadcast communication. Examples of applications of a publish/subscribe (event) 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 (event) notification decouples the communication among users: a user that publishes 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 publish/subscribe notification service is said to be the "full decoupling of the communicating participants in time, space and flow" [11]. Apart from these advantages, systems equipped with a publish/subscribe notification service are generally known as difficult to verify [12, 23]. One of the main reasons 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. In recent years there has been an increasing interest in the use of model-checking techniques for the formal verification of (properties of) systems equipped with a publish/subscribe notification service [4, 6, 7, 12, 21].

In this paper we report on the lessons that we have learned from a case study on the application of formal modelling and verification techniques during the requirements analysis phase of a software design. The goal of this case study was to investigate *a priori*, *i.e.* before any implementation, the effects of adding a lightweight and easy-to-use publish/subscribe (event) notification service to thinkteam—an asynchronous and dispersed groupware system which was developed by think3. To this aim, first thinkteam's underlying groupware protocol was formally specified, after which the model checker Spin [15] was used to verify a number of important properties related to thinkteam's concurrency control and awareness aspects in an automated way. Most properties were formalised as formulae of a Linear Temporal Logic (LTL) [18], which has as advantage that these formulae are close to describing the kind of 'scenarios' that are often informally formulated during the requirements analysis phase of software design. In fact, LTL formulae reflect properties of typical—desired or undesired—behaviour (or uses) of the system under scrutiny. This thus 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 implementation [17]—can indeed be detected early on in the development, leading to considerable reductions in cost and serious improvements of quality. To the best of our knowledge, this case study is one of the first successful applications of model checking to the verification of publish/subscribe notification services in a groupware setting. The precise verifications that were performed, together with their outcome, are described in full detail in [4]. Here we instead focus on the lessons that can be learned from this experience.

We begin this paper with an overview of the basic concepts of model checking, the model checker Spin, its input language Promela, and LTL. We then present our case study, whose aim is to model and verify the addition of a publish/subscribe notification service to thinkteam's underlying groupware protocol, followed by the lessons we learned from this case study. Finally, we conclude with a discussion of future work.

2. METHODS AND TOOLS

In this section we give a brief overview of the methods and tools that we have used for our case study, *i.e.* the model checker Spin, its input language Promela, and LTL.

Spin is one of the best known and most successful model checkers. It offers a spectrum of verification techniques, ranging from partial to exhaustive verification, and it 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. Spin has been developed at Bell Labs during the last two decades. For more detailed information, we refer to `www.spinroot.com`.

Promela is a non-deterministic C-like specification language for modelling finite-state systems communicating by channels. 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 FIFO buffered—to model asynchronous communication—or handshake (a.k.a. 'rendezvous')—to model synchronous communication. For more detailed information on Promela, we refer to [15].

Promela specifications can be checked for deadlocks and/or against correctness properties specified as LTL formu-

lae by means of SPIN. LTL is an extension of predicate logic allowing one to express assertions about behaviour in time, without explicitly modelling time. For more detailed information on LTL, we refer to [18]. 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. An important feature of many a model checker, including SPIN, is that counterexamples are generated in case certain properties are found not to be satisfied. These counterexamples can consequently be used for 'debugging' the PROMELA specification.

3. thinkteam

thinkteam is think3's cooperative 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 Computer-Aided Design (CAD) solution—as well as with other third-party products. thinkteam allows enterprises to capture, organise, automate, and share engineering product information in an efficient way. For more information, we refer to `www.think3.com/products/tt.htm`.

3.1 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 desktop clients interacting with one centralized Relational Database Management System (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) is not described as it is not relevant for the purpose of this paper.

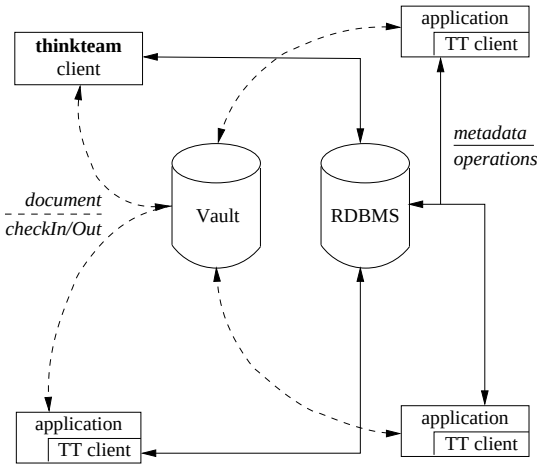

Figure 1: The thinkteam structure.

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 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.

RDBMS interaction. thinkteam uses a RDBMS to persist and retrieve both its object model and the objects created during operation. RDBMS interactions are fairly low level in nature and 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 per document 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 currently based on the 'retrial' principle: there is no queue (nor a reservation system) handling the requests for editing rights on a document.

3.2 thinkteam at Work

thinkteam supports the CAD designer in various phases. An important area of thinkteam intervention is the overall industrialization part of a project and involves activities that tend to be intensive w.r.t. the project metadata, 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 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 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 she is working with.

Most reference parts are normally production items subjected to PDM management and whose physical counterparts (the 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 her activity. Since it is an already existing and managed part (*i.e.* it is already in the vault), the designer must express 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 is given in Figure 2.

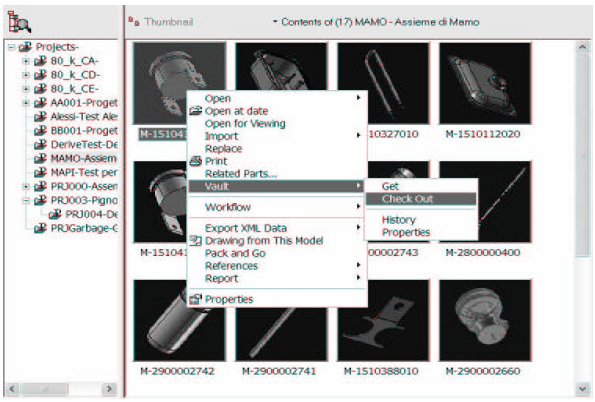

Figure 2: A thinkteam user checks out a file.

When the designer is ready to publish her work, she will release it to the system by issuing an explicit *checkIn* command, which frees the model for modification by others. Were the designer to change her mind, then she may choose to issue an *unCheckOut* command, which frees the model but discards any changes that have occurred since the *checkOut*. Finally, she may issue a *checkInOut* if 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 (SIP) is provided, combining metadata/vaulting operations for speedily registering several changes and modifications.

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

3.3 Adding Publish/Subscribe Notification

think3 originally planned the addition of a publish/subscribe notification service to thinkteam in order to solve 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, sketched in Figure 3, may come into play when a client performs a *checkOut*/modify/*checkIn* cycle on a document that may be used as reference copy by other clients.

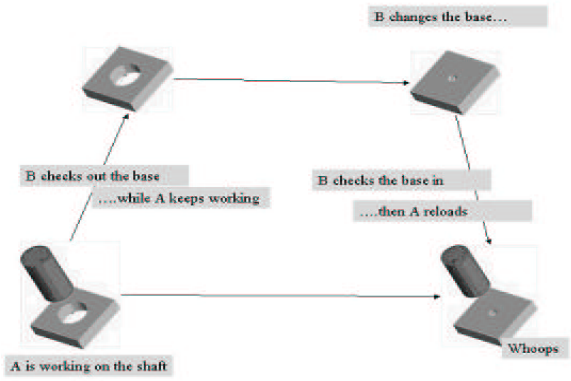

Figure 3: The 'lost update' phenomenon.

It is important to recall 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 easy—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.

The service which think3 proposed to add to thinkteam actually is more refined than the service just described. 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 receive up-to-date information on its status, but so do all users that are registered for the specific document.

4. AUTOMATED VERIFICATION

In this section we briefly describe the application and results of using formal modelling and verification techniques to investigate the addition of a publish/subscribe notification service to the protocol underlying thinkteam. The full details can be found in [4]. First, an abstract PROMELA specification of thinkteam's underlying groupware protocol, augmented with a publish/subscribe notification service, is defined. Subsequently it is shown that the applied abstractions are sufficient to enable the use of SPIN for the automated verification of a number of correctness properties concerning the concurrency control and awareness aspects of the augmented thinkteam protocol.

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 Mb of available physical memory.

4.1 The thinkteam Protocol

thinkteam's functioning is defined by its underlying multi-user communication schema, which we call the thinkteam protocol. In [4] we defined an abstract specification (model) of thinkteam's underlying groupware protocol, which nevertheless covers faithfully its most important aspects, and we augmented it with the publish/subscribe notification service that think3 intends to add to thinkteam. We abstracted from the full thinkteam protocol but focused on the communication schema that is fundamental to the two aspects of this protocol that we wanted to verify, *viz.* its concurrency control and awareness aspects. We thus abstracted completely from the RDBMS and all its related operations. The abstract model of the augmented thinkteam protocol that we used is depicted in Figure 4.

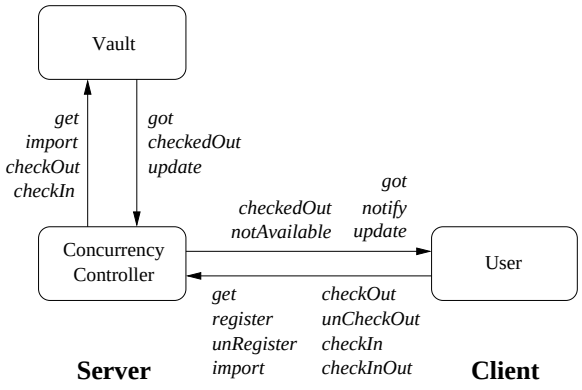

Figure 4: The augmented thinkteam protocol.

This abstract 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 Subsection 3.1, completed with the messages *register*, *unRegister*, *notify*, *update*, *got*, *checkedOut*, and *notAvailable*, whose functioning we explain next. The first four messages concern the added publish/subscribe notification service. A user can explicitly subscribe (unsubscribe) herself to a file by sending a *register* (*unRegister*) to the CC. Furthermore, the *get* operation is altered such that a user issuing it is implicitly registered for the respective file. If a user is subscribed to a certain file, then she is sent a *notify* by the CC whenever another user extracts this file from the vault. Similarly, she receives an *update* from the CC whenever another user inserts (publishes) a file in the vault. Finally, to make the 'direction' of a message clear from the name of that message, a user that requests a read-only copy of a file through a *get* receives the copy by a *got*, while a user requesting editing rights for a file through a *checkOut* either receives the file by a *checkedOut* or receives a *notAvailable*, depending on the file's availability.

Some typical series of actions that can take place in the augmented thinkteam protocol are the following. A user can indicate that 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* and it moreover sends a *notify* to all other users registered for that file, while otherwise (*i.e.*, if the file is locked) the user receives a *notAvailable*. Recall that at any moment in time, a user can explicitly subscribe (unsubscribe) herself to a file by sending a *register* (*unRegister*) to the CC. Rather than 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 her the file via a *got* and moreover implicitly registers this user for the file. 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 her possession for further editing—by sending it to the CC via 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).

In either of these cases, the CC sends an *update* to all other users that are registered for this extracted file. Finally, a user can always decide to insert a new file into the vault by sending it to the CC via an *import*.

4.2 The PROMELA Specification

The complete PROMELA specification of the aforementioned augmented thinkteam protocol can be found in [4]. Here we only list some of the assumptions that we made in our specification. The reason for these assumptions is to reduce as much as possible the size of the state space, as well as that of the state vector. The state vector is used by SPIN to uniquely identify a system state and contains information on the global variables, the channel contents, and for each process its local variables and its process counter. Minimising its size thus results in less bytes that SPIN needs to store for each system state and thereby further reduces the risk to run out of memory during formal verifications. Finally, as noted in [15], 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 made in the PROMELA specification of the augmented thinkteam protocol are the following.

1. The transmission time of messages between user and server is very fast w.r.t the interarrival time between requests from different users. This results in a very low probability of competing requests. Therefore we chose to mainly use handshake channels for communication. Furthermore, initial verifications with SPIN have shown that replacing each of these handshake channels by a channel with a small buffer, still leads to feasible memory requirements [4].

2. At any moment in time there is only one file (called 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. After a user has sent a *get* to the CC it 'actively waits' for an answer, *i.e.* she cannot participate in further interleavings, whereas a user can be involved in further interleavings while waiting for the CC to respond to her *checkOut*.

5. No message is ever lost.

4.3 Validation with SPIN

The abstractions that we applied to the thinkteam protocol are sufficient for verifying a number of correctness properties concerning the concurrency control and awareness aspects of the augmented thinkteam protocol with SPIN. The details of the verifications that we performed can be found in [4]. Here we summarise the outcome. Table 2 presents the results of full statespace searches for invalid endstates, which is SPIN's formalisation of deadlock states. The runtime is given as hours:minutes:seconds.

users	state vector	depth reached	deadlock	memory used	runtime
2	84 byte	4423	no	37 Mb	0:0:01
3	108 byte	434033	no	114 Mb	0:03:06
4	132 byte	10484899	no	916 Mb	8:18:36

Table 2: Results of the search for deadlocks.

These results give a good impression of the fast-growing number of interleavings in groupware applications like thinkteam and, consequently, of the difficulties in obtaining exhaustive verifications of relevant properties of such groupware applications. This is one of the main reasons for some of the unsuccessful applications of model checking to groupware systems in the past [22].

Subsequently, the following correctness criteria for a groupware protocol with a publish/subscribe notification service were formulated in [4].

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

Awareness. (AW-1) A user does not receive (a) a *notify* or (b) an *update* if it is not registered for the file these messages refer to; (AW-2) every *checkOut* must eventually lead to a *notify* to all (and only those) users that are registered for the checked out file; (AW-3) every *unCheckOut*, *checkIn*, and *checkInOut* must eventually lead to an *update* to all (and only those) users that are registered for the file these message refer to.

Denial of service. (DoS) No user can be denied a service forever.

Note that these criteria include both safety and liveness properties. These properties were verified in [4] for the augmented thinkteam protocol in case of 3 users by using SPIN and the given PROMELA specification. 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. The results of these verifications are summarised in Table 3, in which the runtime is given as minutes:seconds.

property	state vector	depth reached	valid	memory used	runtime
CC-1	112 byte	3147677	yes	473 Mb	16:54
CC-2	108 byte	434033	yes	114 Mb	3:06
CC-3	116 byte	7348	no	193 Mb	0:01
CC-4	108 byte	434033	yes	114 Mb	3:06
AW-1a	112 byte	3071518	yes	539 Mb	21:22
AW-1b	112 byte	3057025	yes	558 Mb	22:45
AW-2	112 byte	3338868	yes	967 Mb	39:22
AW-3	112 byte	4183223	yes	925 Mb	38:57
DoS	116 byte	1801	no	193 Mb	0:01

Table 3: Results of the verifications.

The verifications show that the concurrency control and awareness aspects of the thinkteam protocol completed with a publish/subscribe notification service are—for the most part—well designed. However, two properties turned out to be invalid.

(CC-3) Since the thinkteam protocol does not oblige a user to ever return a file that it has checked out to the Vault, a user that holds the lock on file 0 can thus endlessly perform *checkInOut* and never release the lock on file 0. This property is inherent to the thinkteam protocol. In practice this undesirable situation is avoided 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.

(DoS) The CC can endlessly be kept busy by one of the users so that other users thus never get their turn. Also this property forms an integral part of the thinkteam protocol. This is due to the fact that 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.

5. LESSONS LEARNED

In the context of an Italian research project, researchers from the FM&&T group of ISTI–CNR and from the European Headquarters of think3 teamed up to model and verify the addition of a publish/subscribe notification service to thinkteam's underlying groupware protocol. The FM&&T group has a longstanding experience in research on the application of software tools to specify and verify system designs, in particular with model-checking techniques. think3, on the other hand, has a strong competence in the application domain but no previous experience with model checking. The concurrent and distributed nature of thinkteam and the fact that SPIN is freely available and very well documented, are the main reasons for using SPIN in the case study described in this paper.

During interactive design sessions with think3, which included both physical meetings and meetings by means of groupware systems like teleconferencing and electronic mail, think3 has required a basic knowledge of SPIN. In fact, we have been able to use SPIN in various ways to present the behaviour of the specification (model) of thinkteam's underlying groupware protocol. Examples include simulation, message sequence charts, and counterexamples. This caused the detection of a number of ambiguities and unclear aspects of the design that think3 had in mind of the kind of publish/subscribe notification service they intend to add to thinkteam. Examples include the two properties (CC-3, DoS) that were shown to be invalid in the previous section, as well as the numerous questions that were addressed during these interactive meetings. To give an idea of the type of questions, we now list three of them, with their answers.

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 'reget'.

What are the exact semantics of requests? Originally, we had allowed a user to (un)register for a file for which it 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 may 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.

The specification (model) of thinkteam's underlying groupware protocol has thus been developed in close collaboration with think3. Moreover, it is think3's intention to use this specification as basis for their planned implementation of a publish/subscribe notification service in thinkteam. This obviously has the advantage of guaranteeing the correctness of the properties that were found to hold for this specification in the previous section. For think3 this experience in model checking a specification before actually implementing it, has been a true eye opener. They recognized the inherent concurrency aspects of groupware applications as well as their intricate behaviour. Moreover, the relatively simple and abstract high-level models that we developed turned out to be of great help to focus on the key issues of the development of the interface aspects of thinkteam, before turning to the more detailed design and implementation issues. In fact, think3 has expressed the intention to install SPIN in order to get more acquainted with it and—ultimately—to acquire the skills to perform automated verification of the (groupware) protocols underlying their software.

6. CONCLUSIONS AND FUTURE WORK

This paper reports on the lessons learned in the course of an ongoing project of applying academic experience with formal modelling—and with model checking in particular—to an industrial case study. The goal of the case study was to investigate the effects of adding a publish/subscribe notification service to think3's cooperative PDM solution thinkteam, an example of an asynchronous and dispersed groupware system. To this aim, we specified thinkteam's underlying groupware protocol in PROMELA, after which SPIN was used to verify a number of important properties related mostly to thinkteam's concurrency control and awareness aspects. The outcome showed that many ambiguities could be removed, leading to a design in which more confidence can be put w.r.t. the addressed aspects. In fact, it is think3's intention to use the developed (specification) model as basis for their planned implementation of a publish/subscribe notification service in thinkteam. To the best of our knowledge, the case study reported on in this article is one of the first successful applications of exhaustive model-checking techniques to the verification of publish/subscribe notification services in a groupware setting.

In the future we intend to increase the number of files that can be handled by our specification (model) of the augmented thinkteam protocol. We also intend to further investigate the consequences of abandoning the 'retrial' principle w.r.t. document access and introduce a document reservation system instead. The most obvious way to model this in PROMELA 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 [4]. Currently, a thinkteam user that registers itself for a document is informed of the current status of that document only when its status changes. We intend to extend thinkteam's publish/subscribe notification service in such a way that the user who checks out a document is informed automatically of any existing outstanding reference copies of this document.

Finally, we recall that one of the reasons for think3's desire to augment thinkteam with a publish/subscribe notification service is to be able to solve the variant of the 'lost update' phenomenon depicted in Figure 3. However, 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.

7. ACKNOWLEDGEMENTS

The research reported in this paper 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".

8. REFERENCES

[1] R. M. Baecker, editor. *Readings in Groupware and Computer Supported Cooperation Work—Assisting Human-Human Collaboration*. Morgan Kaufmann, San Mateo, CA, 1992.

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

[3] M. H. ter Beek, M. Massink, D. Latella, and S. Gnesi. Model Checking Groupware Protocols. In F. Darses, R. Dieng, C. Simone, and M. Zacklad, editors, *Cooperative Systems Design—Scenario-Based Design of Collaborative Systems*, volume 107 of *Frontiers in Artificial Intelligence and Applications*, pages 179–194. IOS Press, Amsterdam, 2004.

[4] M. H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis. Model Checking Publish/Subscribe Notification for thinkteam. Technical Report 2004-TR-20, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 2004.

[5] M. H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, and M. Sebastianis. Automated Verification of Groupware Protocols. *ERCIM News—Special: Automated Software Engineering*, 58:33–35, July 2004.

[6] 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.

[7] X. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh. Model-Checking Middleware-Based Event-Driven Real-Time Embedded Software. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proceedings FMCO'02*, volume 2852 of *LNCS*, pages 154–181. Springer-Verlag, Berlin, 2002.

[8] P. Dourish and V. Bellotti. Awareness and Coordination in Shared Workspaces. In J. Turner and R. Kraut, editors, *Proceedings CSCW'92*, pages 107–114. ACM Press, New York, 1992.

[9] C. A. Ellis. Team Automata for Groupware Systems. In S. C. Hayne and W. Prinz, editors, *Proceedings GROUP'97*, pages 415–424. ACM Press, New York, 1997.

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

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

[12] D. Garlan, S. Khersonsky, and J. S. Kim. Model Checking Publish-Subscribe Systems. In T. Ball and S. K. Rajamani, editors, *Proceedings SPIN'03*, volume 2648 of *LNCS*, pages 166–180. Springer-Verlag, Berlin, 2003.

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

[14] C. Gutwin, M. Roseman, and S. Greenberg. Supporting Awareness of Others in Groupware. In M.J. Tauber, editor, *Companion Proceedings SIGCHI'96*, pages 205–215. ACM Press, New York, 1996.

[15] G. J. Holzmann. *The SPIN Model Checker—Primer and Reference Manual*. Addison Wesley, Reading, MA, 2003.

[16] J. Kleijn. Team Automata for CSCW – A Survey –. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication-Based Systems—Advances in Petri Nets*, volume 2472 of *LNCS*, pages 295–320. Springer-Verlag, Berlin, 2003.

[17] P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann. Qualitätssicherung Software-basierter Technischer Systeme—Problembereiche und Lösungsansätze. *Informatik Spektrum*, 21(5):249–258, 1998.

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

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

[20] K. Schmidt. The Problem with 'Awareness'—Introductory Remarks on 'Awareness in CSCW'. *Computer Supported Cooperative Work—The Journal of Collaborative Computing*, 11(3-4):285–298, 2002.

[21] S. Tripakis and S. Yovine. Timing Analysis and Code Generation of Vehicle Control Software using Taxys. In K. Havelund and G. Rosu, editors, *Proceedings RV'01*, volume 55(2) of *ENTCS*, pages 174–183. Elsevier, Amsterdam, 2001.

[22] T. Urnes. *Efficiently Implementing Synchronous Groupware*. PhD thesis, Department of Computer Science, York University, Toronto, 1998.

[23] L. Zanolin, C. Ghezzi, and L. Baresi. An Approach to Model and Validate Publish/Subscribe Architectures. In M. Barnett, S. H. Edwards, D. Giannakopoulou, and G. T. Leavens, editors, *Proceedings SAVCBS'03*, Technical Report 03-11, pages 35–41. Department of Computer Science, Iowa State University, 2003.