The Journal of Logic and Algebraic Programming 78 (2009) 191–232

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Assisting the design of a groupware system – Model checking usability aspects of thinkteam☆

Maurice H. ter Beek [a], Stefania Gnesi [a], Diego Latella [a], Mieke Massink [a,*], Maurizio Sebastianis [b], Gianluca Trentanni [a]

[a] *Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', CNR Area della Ricerca, Via G. Moruzzi 1, 56124 Pisa, Italy*
[b] think3 *Inc., Via Ronzani 7/29, 40033 Bologna, Italy*

ARTICLE INFO

Available online 13 January 2009

Keywords:
Groupware
Concurrency
Formal methods
Verification
Model checking

ABSTRACT

Product Data Management (PDM) systems support the product/document management of design processes such as those typically used in the manufacturing industry. They allow enterprises to capture, organise, automate and share engineering information in an efficient way. The efficient handling of queries on product information and the uploading and downloading of families of related files for modification by designers are essential aspects of such systems. The efficiency of the system as perceived by its clients depends on its correct functioning, but also for a significant part on its performance aspects. In this article, we apply both qualitative and stochastic model-checking techniques to evaluate various usability and performance aspects of the thinkteam PDM system, and of several proposed extensions, thereby assisting the design phase of an industrial groupware system.

1. Introduction

Product Life-cycle Management (PLM) is the activity of managing a company's products across their life-cycles—from their conception, through design and manufacturing, to service and disposal—in the most effective way [1]. think3's thinkPLM is a suite of integrated PLM applications, built on thinkteam. thinkteam is think3's Product Data Management (PDM) application, catering the product/document management needs of design processes in the manufacturing industry. It allows enterprises to capture, organise, automate and share engineering information in an efficient way. thinkteam is used to manage data for products/documents undergoing constant changes as well as for occasional customisations, i.e. with long or short life-cycles. The current thinkteam setting consists of a number of clients that interact with one centralised Relational Data Base Management System (RDBMS) server. This RDBMS controls the storage and retrieval of data, such as Computer-Aided Design (CAD) files in a file-system-like repository, called the Vault. Access control is based on a 'retrial' principle: There is no queue (or reservation system) handling the client's requests for editing rights on a file.

☆ This work has been partially funded by the Italian project tocai.it, by the EU project Sensoria (IST-2005-016004), by the mini-project Faerus of the EU project Resist (IST-2006-026764) and by the Italian CNR/RSTL project XXL. thinkteam, think3 and thinkPLM are registered trademarks of think3 Inc. For details: http://www.think3.com.

* Corresponding author. Tel.: +39 0503152981.
E-mail addresses: maurice.terbeek@isti.cnr.it (M.H. ter Beek), stefania.gnesi@isti.cnr.it (S. Gnesi), diego.latella@isti.cnr.it (D. Latella), mieke.massink@isti.cnr.it (M. Massink), maurizio.sebastianis@think3.com (M. Sebastianis), gianluca.trentanni@isti.cnr.it (G. Trentanni).

1567-8326/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2008.11.004

thinkteam is a typical example of a groupware system, i.e. a multi-user computer system meant to assist people collaborating on a common project. Such systems are studied in the interdisciplinary research field of Computer Supported Cooperative Work (CSCW), which deals with the understanding of how people work together, and the ways in which computer technology can assist them [2]. Groupware is typically classified according to two dichotomies, viz. (1) whether its users work together at the same time (synchronous) or at different times (asynchronous) and (2) whether they work together in the same place (co-located) or in different places (dispersed). This is called the time space taxonomy by Ellis et al. [3]. thinkteam is an asynchronous and dispersed groupware system. Among the important design issues in groupware systems are data sharing, concurrency control and user awareness, where the latter should be understood as users having a sense of the (past, current, future) activities of other users—without direct communication—and using this as context for their own activities [4,5].

In this article, we address these and other issues in the context of thinkteam. More precisely, we use model checking to formalise and verify several design options for thinkteam extensions. However, we also apply model checking to 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. Recent years have seen an increasing interest in the use of model checking for the formal verification of (properties of) groupware [6,7,8] and publish/subscribe (pub/sub) systems [9,10,11,12]. In a pub/sub system, clients may publish files in a repository and subscribe themselves to automatically receive notifications on changes to files of their interest. One of the difficulties in this domain is that detailed models tend to generate very large state spaces due to the interleaving activity that comes with many asynchronously operating clients [11,13]. Our approach to use model-checking techniques in the groupware domain differs from these approaches. We generate small, abstract models that are intended to address specific usability-related groupware issues. We show that model checking can be of great help in an exploratory design phase, both for comparing different design options and for refining and improving the description of the proposed extensions. This way of using model checking is in support of a prototyping-like modelling technique. The focus is on obtaining in a relatively fast way an informed but perhaps somewhat approximative idea of the consequences, both qualitative and quantitative, of adding specific features to an existing groupware system. This is quite different from the traditional use of model checking as a technique to develop rather complete specifications, with the aim of reaching a maximal level of confidence in the correctness of complicated distributed algorithms. In this sense, our proposed use of model checking is somewhat resembling the idea of extreme programming [14]: Generating simple ad-hoc models of new features that are meant to be added to a system.

The work presented in this article is partly based on our previous publications on the application of model checking to the design of groupware systems, which we now briefly describe in chronological order. In [6], we have developed a rather abstract specification of that part of the protocol underlying the groupware toolkit Clock [15,16] that deals with concurrency and exclusive editing rights. In the same work we show, using the model checker Spin [17], that our specification satisfies a number of correctness properties, some of which were also mentioned in the work by Urnes [8] on Clock.

In [18,19], we have used the same model-checking technique to formalise and verify some concurrency and usability aspects of thinkteam, and of a first proposed extension of thinkteam with a lightweight and easy-to-use pub/sub event notification service. This service is intended to increase the users' awareness of the status of the development of the engineering product and the activities of the design team by intelligent data sharing. Whenever a thinkteam client *pub*lishes a file by importing it into the Vault, i.e. the relevant file repository, automatically all clients that are *sub*scribed to that file are notified via a multicast communication. We have analysed a number of qualitative correctness properties addressing concurrency, usability and awareness aspects. Pub/sub event notification de-couples the communication among users: A user publishing a document need not be concerned with whom the server will send a notification to, i.e. the users communicate through the server. Users need not actively participate in the notification in a synchronous way. In fact, the main strength of a pub/sub event notification service is said to be the "full de-coupling of the communicating participants in time, space and flow" [20]. The main results of the analyses of the pub/sub notification service are reported in Section 4. A related approach can be found in [21], where a case study in the automatic derivation of correct integration code for assembling a set of thinkteam's (software) components is reported.

Some usability issues, influenced more by the performance of a system than by its functional behaviour, cannot be analysed by qualitative model-checking techniques alone. In fact, we showed in [18,19] that the system is not starvation free, i.e. a client can be excluded from obtaining a file, simply because other clients competing for the same file are more 'lucky' in their attempts to obtain it. Such behaviour can be explained by the fact that clients are only provided with a file-access mechanism based on a 'retrial' principle. While analysis with qualitative model checking can be used to show that such a problem exists, it cannot be used to quantify the effect that it has on usability. In our case, the number of retries a client has to perform before obtaining a file is an important measure of the usability of the system. If this number is high, extending thinkteam with a waiting-list policy should be considered instead of the current simpler retry-based policy. The trade-off between these two design options was analysed in [22], where we used stochastic model-checking techniques. The main issues and results of such analysis experiments are reported in the present article in Section 5. Stochastic model checking is a relatively recent extension of qualitative model checking that allows for the analysis of qualitative properties of systems as well as performance- and dependability-related, i.e. quantitative, properties [23,24,25,26].

The original contribution of the present paper is presented in Section 6, where we use the same technique to model and analyse a new extension of thinkteam, concerning the replacement of the unique centralised Vault by a distributed set of replicated vaults and a preference list for vaults per client. The assignment of a suitable vault location has an effect on the performance of the system and therefore on the adequacy of the support it can provide to its clients. The properties we verify include the guarantee that files are modified by at most one client at a time, the expected waiting time for clients requesting a file for modification, the effect of workload conditions on the usability of the system and on the overhead in uploading and downloading files.

An important topic addressed in this article concerns the uptake of model-checking techniques by industry. Our industrial partner think3 had no previous experience with such analysis techniques. We show that our approach to start from small models that require little time to fully understand, but that nevertheless provide results that would be unfeasible to produce manually, and that can be used to generate performance diagrams directly related to issues of interest to the industry, can be successfully transferred to industry.

We begin this article with a description of thinkteam in Section 2, followed by a brief introduction to qualitative and stochastic model checking in Section 3. In Section 4, we present the analysis of the extension of thinkteam with a pub/sub event notification service. Subsequently, in Section 5, we present a stochastic model for the quantitative analysis of two different file access policies. In Section 6, we enrich the model of Section 5 to analyse thinkteam extended with multiple replicated vaults; the resulting model and analysis are the original contribution of the paper. Finally, we briefly describe the lessons learned from our experience with the use of qualitative and quantitative model checking in an industrial groupware setting in Section 7, and we draw some conclusions and give an outline of future work in Section 8.

2. Thinkteam

In this section, we present a brief overview of think3's PDM application thinkteam. For more information, we refer the reader to `http://www.think3.com`.

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 activity in terms of human, financial and material resources. Among the several non-design tasks that are required for the delivery of a final product to an enterprise's Manufacturing department, some are externally initiated by organisations 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 co-operation 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 (BOM), 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: For instance, 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.

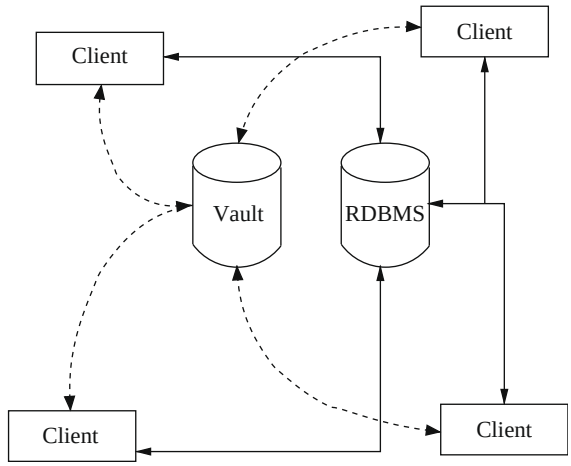


Fig. 1. The thinkteam structure. Document *checkIn/Out* is represented by dotted arrows; metadata operations are represented by solid ones.

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 these goals. As such it supports the whole product development life-cycle, bringing the right information to the right people at the right time. However, several extensions are currently proposed to further improve thinkteam, in particular its performance from the user's point of view when larger groups of CAD designers are using thinkteam in an ever more geographically distributed and potentially service-oriented way.

As we mentioned in Section 1, thinkteam is an asynchronous and dispersed groupware system. An additional difficulty that arises during the design of such groupware is their inherently distributed nature. This forces one to address issues like network communication, concurrency control and distributed notification. This has lead to the development of groupware toolkits that aid groupware developers with a series of programming abstractions aimed at simplifying the development of groupware applications. Examples include Rendezvous [27], GroupKit [28] and Clock [15,16]. In this article, we address a complementary approach to the design of the underlying protocols of such systems and their performance characteristics from a user's point of view.

2.1. Technical characteristics

Thinkteam is a three-tier data management system running on Wintel platforms (cf. also Fig. 1). The most typical installation scenario is a network of several desktop clients interacting with one centralised RDBMS server and one or more file servers grouped into a single Vault. 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. The dotted arrows in Fig. 1 denote document *checkIn* and *checkOut*, while solid arrows represent metadata operations. The internal communication between the Vault and the RDBMS are not explicitly presented in the figure. Below follows a general description of the operations of various (logical) thinkteam subsystems.

2.1.1. RDBMS Interaction

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

2.1.2. 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:

- to provide a single, secure and controlled storage environment where the documents controlled by the PDM application are managed, and
- 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 former is the subject of the implementation of the lower layers of the vaulting system, the latter is implemented in thinkteam's underlying protocol by a standard set of operations made available to the clients, viz. those listed in Table 1. It is important to note that access to files (through the above *checkOut* operation) is based on a 'retrial' principle: There is no queue (or reservation system) handling the requests for editing rights on a file.

Table 1
thinkteam user operations.

Operation	Effect
get	Extract a read-only copy of a file from the Vault
import	Insert an external file into the Vault
checkOut	Extract a copy of a file 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 preceding *checkout*
checkIn	Replace an edited file in the Vault (the file must previously have been checked out)
checkInOut	Replace an edited file in the Vault, while at the same time retaining it as checked out

2.2. Thinkteam *at work*

Thinkteam supports CAD designers in various design phases. An important area of thinkteam intervention is the overall industrialisation part of a given project and involves activities that tend to be intensive w.r.t. the project's 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 the modelling phases, briefly described next.

2.2.1. 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 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 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. 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 visualisation, printing, etc.).

2.2.2. Geometry (part) modification

Modifying an existing part is, in order of frequency, the second-most-used operation a designer performs during her activity. Given that the part already exists (i.e. it is already in the Vault) the designer must express the intent to modify it with an explicit (write exclusive) *checkOut* operation that prevents attempts at modification by other users. A screen-shot of the invocation of this operation in the Graphical User Interface (GUI) is depicted in Fig. 2.

When designers are ready to publish their work, they will release it to the system by issuing an explicit *checkIn* command, which frees the model for modification. Were the designers to change their mind, then they might choose to issue an *unCheckOut* command, which frees the model but discards any changes that have occurred since the *checkOut*. Finally, they may issue a *checkInOut* if they want to release an intermediate version of the model to the system, but do 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. Note that when a file has been checked out, but before it has been checked in, the file can still be accessed by other designers in read-only mode, by means of *get* operations.

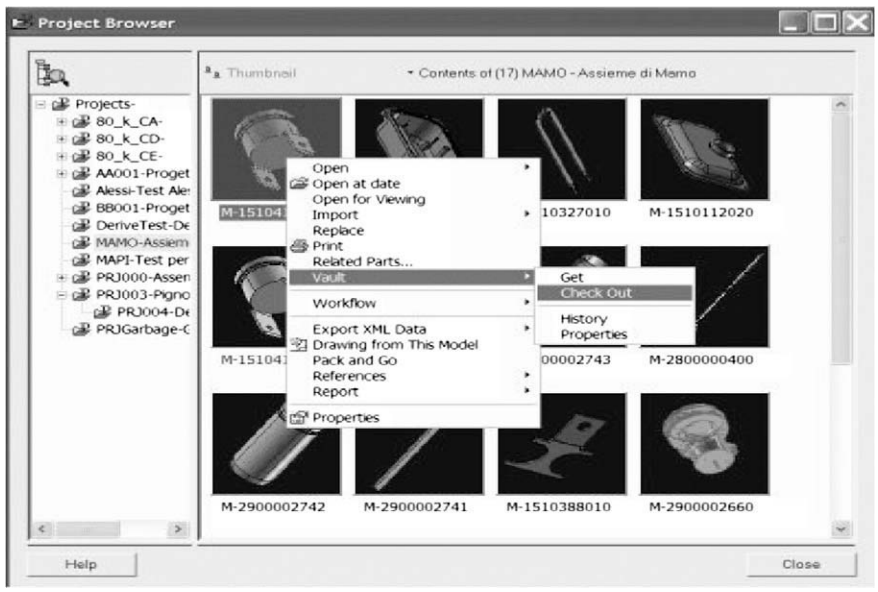


Fig. 2. A thinkteam user checks out a file from the Vault.

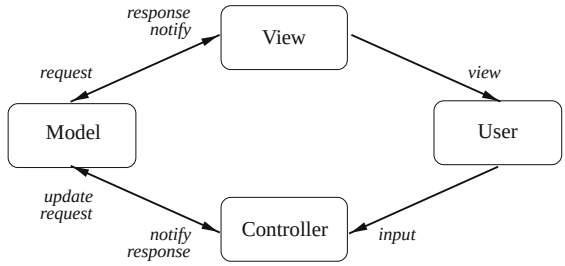


Fig. 3. The MVC architecture and its communication protocol.

2.2.3. 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, the special environment Save Into Project (SIP) is provided, combining metadata/vaulting operations to speedily register changes and modifications.

2.3. Architecture

Thinkteam's design-level architecture is based on the Model-View-Controller (MVC) paradigm of [29]. According to this paradigm, an architecture organising interactive applications is partitioned into three separate parts, viz.

- the Model, implementing the application's data state and semantics,
- the View, computing the graphical output of the application, and
- the Controller, interpreting the inputs from the users.

In Fig. 3, the MVC architecture is depicted together with its communication protocol. The Controller transforms an *input* from the User, e.g. a *checkOut* request, into an *update*, which it sends to the Model. In order to do so, it may need to obtain data from the Model, e.g. whether the requested file is locked, by communicating via *request* and *response*. Upon receiving an *update*, the Model changes its data state and sends a *notify* to both the Controller and the View. The latter, upon receiving this *notify*, re-computes the display—for which it may need to obtain the new data state from the Model by communicating again via *request* and *response*—and eventually sends a *view* to the User.

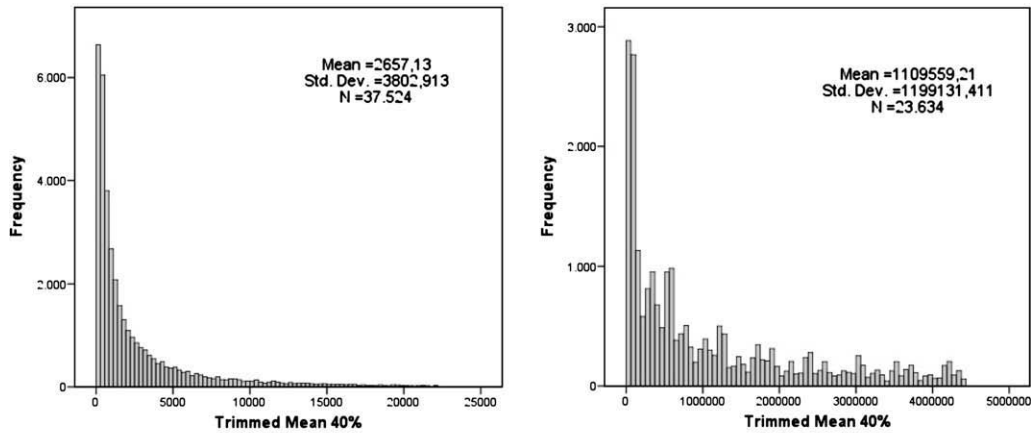
In thinkteam's design-level architecture, the Model and the Controller are integrated and situated on the server, while a View is situated on each of the clients. The communication between the server and the clients is defined by a set of (communication) protocols.

2.4. Characteristics of use

Thinkteam handles something like a few hundred thousand files for some 20–100 users. In order to get more realistic data on the particular use that clients make of the system, think3 has provided us with a cleaned-up log-file, comprising all activity (in terms of the operations listed in Table 1) of one of the manufacturing industries using thinkteam from 2002 through 2006, for us to analyse. This log-file contains, for each operation, its time stamp (in the format day-month-year and hour-minute-second), the name of the user that performed it and the file the operation refers to. In this way, each line in the log-file represents an atomic access to the Vault. The format of the log-file is easy to handle, but it contains a really huge amount of data (792,618 Vault accesses by 104 users regarding 183,492 files). Moreover, think3 has improved its logging mechanism during the years. For these reasons, we have restricted our analysis to the year 2006. The aim of our analysis was to obtain some insight on the timing issues concerning the duration of editing sessions and the occupancy of files.

The data of 2006 concerns 83 users collaborating on a total of 181,535 files, 23,134 of which were checked out at least once during the year. The remaining files were used exclusively as reference material, e.g. downloaded in read-only fashion by means of *get* operations. A total of 65 users turned out to be involved in editing sessions. We present the analysis of a subset of the data that is directly relevant for the models that will be presented in Sections 5 and 6. These concern the duration of editing sessions (i.e. the time that has passed between a *checkOut* and a *checkIn* of a file by the same user) and the duration of periods in which files were not locked (i.e. the time that has passed between a *checkIn* and a *checkOut* of the same file by possibly different users). Instead, the number of times that a user unsuccessfully tries to *checkOut* a locked file (i.e. checked out by another user) has not been explicitly logged, so little can be said about that. However, it is possible to obtain an indirect approximation of the number of users that compete for access to the same file by analysing the number of users that modify the same file during the investigated period.

In Fig. 4, we present the data that we obtained after a series of operations performed on the log-file to filter out the logging of irrelevant operations such as the numerous *get* operations (recall that this is the most-frequently-used operation), operations that could be traced back to system administrator interventions and some further anomalous log operations.



(a) Duration of edit sessions. (b) Duration of file inter-access time.

Fig. 4. (a) and (b) Histograms of the log-file analyses reported in this section.

Table 2
Number of files edited by at least two users.

Nr. files	5077	1407	301	79	24	22	8	8	6	10	3	1	1	1
Nr. users	2	3	4	5	6	7	8	9	10	11	12	13	14	17

These graphs have been produced with SPSS v15.0 [30] by computing the so-called *mean trimmed* 40% (i.e. discarding the lower and higher 20% of the scores and taking the mean of the remaining scores).

Fig. 4a shows a histogram of the distribution of the duration of editing sessions. On the *x*-axis, the time is presented in seconds. The histogram thus shows all sessions except for 20% of the shortest and 20% of the longest sessions, meaning that sessions of less than 111 s (i.e. approximately 2 min) and more than 22,256 s (i.e. approximately 370 min) have been removed. This has been done because the log-file contained many very short sessions that were not corresponding to real user editing sessions, but rather to automatic system operations that were also logged and that feature in the log-file as very short *checkOut-checkIn* sessions. We see that the mean duration trimmed 40% of an edit session is 2,657 s, so approximately 45 min. It is easy to see that most sessions tend to be rather short.

In Fig. 4b a histogram is shown of the duration of intervals during which files, that were involved in at least two editing sessions, were *not* locked (i.e. taken in *checkOut*). These data give an impression of the time that passes between Vault accesses to the same file by possibly different users. Each bar in this histogram corresponds to a duration of about 17 h. Many of the intervals fall into the first few periods, indicating that there are many cases in which files were used rather intensively.

Finally, Table 2 shows the number of files that were edited by more than one user in 2006. A further analysis of these files shows that it is quite common that multiple users are editing the same file on the same day, and that there are also days in which upto 8 users are accessing the same file on the same day or on adjacent days. We are aware of the fact that the log analysis is only covering one year of data collected at one particular client, and may therefore not be completely representative of a typical thinkteam user. However, the log data are real observations and do provide information on an example of actual use of the system which can help to put modelling results into the proper perspective.

3. Modelling and analysis techniques

Traditional software-engineering methods like testing and simulation alone, would be quite time consuming and not very suitable to provide the information needed to design the thinkteam extensions we mentioned in Section 1. The main reason for this is that the concurrency aspects of groupware system like thinkteam (with users located in geographically distributed places that collaborate in an asynchronous but co-ordinated way) play a major role in the important design issues for such systems (data sharing, user awareness and concurrency control to guarantee mutually exclusive file modification rights). Formal methods supported by tools, such as model checkers, would in principle be more suitable, even though their application in the groupware domain has not been very widespread.

Model checking is an automatic technique which can be used to verify whether a system design satisfies its requirements specification [31]. Such a verification is moreover exhaustive, i.e. all possible input combinations and states are taken into account. One of the few early attempts at using model-checking techniques in the groupware domain, e.g. to model and analyse the protocols underlying Clock [15,16], were not very encouraging due to the generation of a huge state space. To avoid running out of memory due to such a state-space explosion—which would make an exhaustive verification unfeasible—it is

necessary to use more abstract models, which capture only the core of the system design while abstracting from unnecessary details. As said before, this is the approach we follow in this article.

We will use two model-checking techniques. One technique has been specifically designed to deal with large state spaces and is suitable for the analysis of qualitative (correctness) properties, providing also counterexamples in the form of message sequence charts. The other technique has been specifically designed for the analysis of quantitative aspects of a system that may provide information on its expected usability. From an industrial point of view, both aspects are often closely related and both may provide necessary complementary information on the system under design. We briefly describe both model-checking techniques in the next sections.

3.1. Qualitative model checking

One of the best known and most successful qualitative model checkers is SPIN, which was developed at Bell Labs during the last two decades [17]. It offers a spectrum of verification techniques, ranging from partial to exhaustive verification. It is freely available through `spinroot.com` and it is very well documented. Apart from these obvious advantages we have chosen to use SPIN because of the aforementioned earlier attempt at verifying a simplified version of the Clock protocol with SPIN in [8], which moreover provides a complete specification of the Clock protocol in SPIN's input language PROMELA.

PROMELA is a non-deterministic C-like specification language for modelling finite-state systems communicating through channels [17]. Formally, specifications in PROMELA are built from processes, data objects and message channels. Processes are the active components of the system, while the data objects are its local and global variables. Message channels are used to transmit data between processes. They can be local or global. Each channel is characterised by the type of messages which can be sent/received via the channel and by its length. Channels with non-zero length are used to model asynchronous communication and behave like FIFO buffers, while zero-length ones model synchronous communication (rendezvous). For further details we refer the reader to [17]. PROMELA specifications can be given as input to SPIN, together with a request to verify certain correctness properties. SPIN then converts the PROMELA processes into finite-state automata and on the fly creates and traverses the state space of a product automaton over these finite-state automata, in order to verify the specified correctness properties. SPIN can be used to verify both safety and liveness properties. Intuitively, a safety property asserts that "nothing bad happens"; examples of safety properties are *absence of deadlock* or *mutual exclusion*. On the other hand, a liveness property is one which asserts that "something good" will eventually happen; examples of liveness properties are *absence of individual starvation* or *responsiveness* [32,17].

There are several ways of formalising correctness properties in PROMELA, the following two of which we shall use in this article. First, *basic assertions* may be added to a PROMELA specification. Subsequently, their validity can be verified by running SPIN. For instance, consider that we want to be sure that a file is not locked the moment in which a lock request for that file is going to be granted. Consider moreover that there is a boolean variable `writeLock`, which is `true` every time a lock request is granted. Then the basic assertion `assert(writeLock == false)` can be added to the PROMELA specification just before a lock is granted and SPIN can be used to verify whether there are assertion violations. If an assertion is violated, a counterexample is automatically generated.

SPIN accepts Linear Temporal Logic (LTL) formulae as requirement specifications to be checked over PROMELA models. The syntax for LTL formulas which we will use in the present paper is given below[1]:

$$\Phi ::= a \mid \neg \Phi \mid \Phi \vee \Phi \mid \Phi \,\mathtt{U}\, \Phi$$

Atomic propositions a include `true` and user-defined properties expressed in a C-like notation. Moreover, the special proposition `P[n]@label` can be used, where `label` is a label introduced in the PROMELA model and marks a specific point in process `P`. The proposition holds in every state of the model where `P` is at point `label`. Index `n` is the process identifier and it is useful for distinguishing different instances of the same process(type) active in a system model. The way SPIN assigns such an identifier is rather rigid and often makes formulae not so easy to understand. Consequently in the body of the paper we use the simplified notation `P@label` and, when there are different instances of `P` to be taken into consideration we just use the notational convention `P0`, `P1`, etc. Negation ($\neg$) and disjunction ($\vee$) are standard and can be combined in the usual way in order to get other logical connectives, like conjunction ($\wedge$), implication ($-\!>$), equivalence ($<\!-\!>$). Given a system run, i.e. a sequence σ of states from the behaviour of a system, the formula $\Phi_1 \,\mathtt{U}\, \Phi_2$ holds if there exists a state in σ where the formula Φ_2 holds and the formula Φ_1 holds in the states of σ until Φ_2 holds. Formula $\Diamond\Phi$ ($\Box\Phi$, respectively) abbreviates $\mathtt{true}\,\mathtt{U}\,\Phi$ ($\neg\Diamond\neg\Phi$, respectively). For a more detailed and formal introduction on LTL, we refer the reader to [17,33].

3.2. Quantitative model checking

In the past, functional and performance analysis of systems were considered two separate areas of research and practice. The latest developments in model checking, extending system verification to performance and dependability aspects, have

[1] The actual syntax accepted by SPIN is slightly different. For the sake of readability, in the body of the present paper we prefer to use standard LTL syntax (while in Appendix A canonical SPIN syntax is used).

led to integrated qualitative and quantitative analysis techniques, even if the very useful generation of counterexamples for such model checkers is still under development [34].

In this article, we use the probabilistic symbolic model checker PRISM [25,35] which supports, among others, the verification of Continuous Stochastic Logic (CSL) properties over Continuous Time Markov Chains (CTMCs) [36]. These CTMCs can be generated by high-level description languages, among which the Performance Evaluation Process Algebra PEPA [37], for which PRISM provides a front end which translates PEPA specifications into PRISM native language descriptions. The PRISM modelling language is a simple state-based language based on the Reactive Modules formalism of Alur and Henzinger [38]. A CTMC is automatically generated by PRISM from a system model description in the native language. PRISM checks the satisfaction of CSL properties for given states in the given model and provides feedback on the calculated probabilities of such states where appropriate. It uses symbolic data structures (i.e. variants of Binary Decision Diagrams).

The branching-time temporal logic CSL [39,40] is a stochastic variant of the well-known Computational Tree Logic CTL [41]. Let I be an interval on the real line, let p be a probability value and let $\bowtie$ be a comparison operator, i.e. $\bowtie \in \{<, \leq, \geq, >\}$. The syntax and informal semantics of CSL are given in the table below:

State formulae

$$\Phi ::= a \mid \neg \Phi \mid \Phi \vee \Phi \mid \mathcal{S}_{\bowtie p}(\Phi) \mid \mathcal{P}_{\bowtie p}(\varphi)$$

$\mathcal{S}_{\bowtie p}(\Phi)$	: probability that Φ holds in steady state is $\bowtie p$
$\mathcal{P}_{\bowtie p}(\varphi)$	: probability that a path fulfils φ is $\bowtie p$

Path formulae

$$\varphi ::= X^I \Phi \mid \Phi \mathcal{U}^I \Phi$$

$X^I \Phi$	: next state is reached at time $t \in I$ and fulfils Φ
$\Phi \mathcal{U}^I \Psi$	: Φ holds along path until Ψ holds at $t \in I$

The meaning of atomic propositions (a), negation ($\neg$) and disjunction ($\vee$) is standard. Using these operators, other boolean operators can be defined in the usual way. In the variant of CSL used in PRISM, the probability bound $\bowtie\ p$ can be replaced by $=?$, which denotes that one is looking for the value of the probability rather than verifying whether it respects a certain bound. The intervals $I = [0, t]$ and $I = [t, \infty)$ are usually written as $\leq t$ and $\geq t$, resp.

It is worth pointing out that the CTL formulae $A\varphi$ (i.e. all paths fulfil φ) and $E\varphi$ (i.e. there exists a path that fulfils φ) coincide, under proper fairness constraints, with the degenerate CSL formulae $\mathcal{P}_{\geq 1}(\varphi)$ and $\mathcal{P}_{>0}(\varphi)$ [42,40]. Consequently, PRISM can also be used to check qualitative, functional, properties of stochastic models, a feature we will use in the sequel.

The basis for the definition of CTMCs are exponential distributions of random variables. The parameter which completely characterises an exponentially distributed random variable is its *rate* λ, which is a positive real number. A real-valued random variable X is exponentially distributed with rate λ (written EXP(λ)) if the probability of X being at most t, i.e. Prob($X \leq t$), is $1 - e^{-\lambda t}$ if $t \geq 0$ and is 0 otherwise, where t is a real number. The expected value of X is λ^{-1}. Exponentially distributed random variables enjoy the *memory-less property*, i.e. Prob($X > t + t' \mid X > t$) = Prob($X > t'$), for $t, t' \geq 0$.

CTMCs have been extensively studied in the literature (a comprehensive treatment can be found in [36], while we suggest [43] for a gentle introduction). For the purposes of this article, it suffices to recall that a CTMC $\mathcal{M}$ is a pair $(\mathcal{S}, \mathbf{R})$, where $\mathcal{S}$ is a finite set of *states* and $\mathbf{R} : \mathcal{S} \times \mathcal{S} \to \mathbf{R}_{\geq 0}$ is the *rate matrix*. The rate matrix characterises the transitions between the states of $\mathcal{M}$. The probability that a transition will be taken from state s within time t is $1 - e^{-\sum_{s' \in \mathcal{S}} \mathbf{R}(s,s') \cdot t}$, while the probability that such a transition leads to state s'' is $\mathbf{R}(s, s'') / \sum_{s' \in \mathcal{S}} \mathbf{R}(s, s')$. We would like to point out that the traditional definition of CTMCs does not include self-loops, i.e. transitions from a state to itself. On the other hand, the presence of such self-loops does not alter the standard analysis techniques (e.g. transient and steady-state analyses) and self-loops moreover turn out to be useful when model checking CTMCs [44]. We thus allow them in this article.

In PEPA, systems can be described as interactions of *components* that may engage in *activities* in much the same way as in other process algebras. Components reflect the behaviour of relevant parts of the system, while activities capture the actions that the components perform. A component may itself be composed of components. The specification of a PEPA activity consists of a pair *(action type, rate)* in which *action type* symbolically denotes the type of the action, while *rate* characterises the *exponential* distribution of the activity duration. The PEPA expressions used in this article have the following syntax:

$$P ::= (\alpha, r).P \mid P + P \mid P \bowtie_L P \mid X$$

The basic mechanism to construct behavioural expressions is through prefixing. Component $(\alpha, r) \cdot P$ carries out activity (α, r), with action type α and duration Δt determined by rate r. The average duration is given by $1/r$ since, by definition, Δt is an exponentially distributed random variable, with rate r. After performing the activity, the component behaves as P. Component $P + Q$ models a system that may behave either as P or as Q, representing a *race condition* between components. The co-operation operator $P \bowtie_L Q$ defines the set of action types L on which components P and Q must synchronise (or *co-operate*); both components proceed independently with any activity not occurring in L. The notation $||$ is often used instead of $\bowtie_\emptyset$. The expected duration of a co-operation of activities $\alpha \in L$ is a function of the expected durations of the corresponding

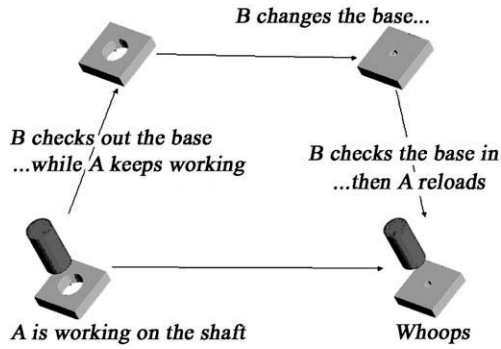


Fig. 5. The 'lost update' phenomenon.

activities in the components. Roughly speaking, it corresponds to the longest one (the actual definition can be found in [37], where the interested reader can find all formal details of PEPA). A special case is the situation in which one component is passive (i.e. has the special rate -) w.r.t. the other component. In this case the total rate is determined by that of the active component only. The behaviour of process variable X is that of P, provided that a definition $X = P$ is available for X.

4. An event notification service for thinkteam

The addition of a pub/sub event notification service to thinkteam was originally proposed 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 Fig. 5, 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.

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

- inform a client issuing a *checkOut* of any outstanding reference copies, and
- notify the copy holders of every *checkOut*/*checkIn* of the original document.

The service which think3 proposed to add actually is more refined than the one described above. All users subscribed to a document are notified whenever a user extracts this document from the Vault for editing purposes. Moreover, as soon as the user finishes editing and publishes the document in the Vault again, 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 all users that are registered for the specific document. The material in this section is partly based on our earlier work on the analysis of a pub/sub extension to thinkteam presented in [45,18,19].

4.1. The thinkteam protocol

Thinkteam's functioning is defined by its underlying multi-user communication schema, which we will refer to as the thinkteam protocol. We define a model of thinkteam's groupware protocol which covers faithfully all aspects which are relevant for our analysis and we extend it with the pub/sub event notification service. In particular, we focus on the communication schema and, as a result, we abstract completely from the RDBMS and the details of all its related operations. The resulting model of the extended thinkteam protocol that we use is depicted in Fig. 6. The model has been validated both with thorough discussion with—and revision by—think3 experts, supported also by SPIN simulations, and by positive model-checking outcomes.

The general model is composed of three components, viz. the `Vault`, the Concurrency Controller process (`CC`) and the `User`. In the actual verification sessions we report about in this paper we configured the model with 3, and sometimes 4, instances of the `User`. While the `User` is assumed located on the client side, the `Vault` and the `CC` are assumed located 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 *register*, *unRegister*, *notify*, *update*, *got*, *checkedOut* and *notAvailable*, whose functioning we explain next. The first four messages concern the added pub/sub event notification service. `User`s can explicitly subscribe (unsubscribe) themselves to a file by sending a *register* (*unRegister*) to the `CC`. Furthermore, the *get* operation is altered such that a user

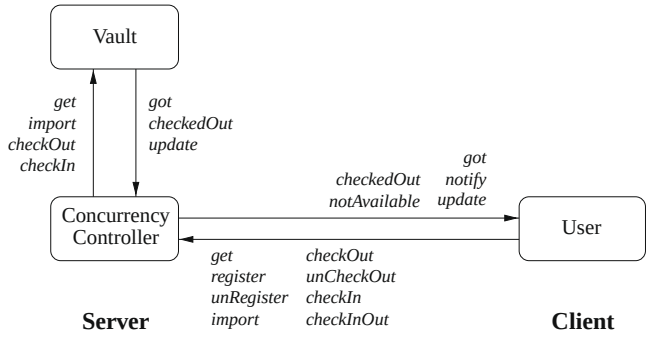


Fig. 6. The augmented thinkteam protocol.

issuing it is implicitly registered for the relevant file. If users are subscribed to a certain file, then they are sent a *notify* by the CC whenever another user either extracts this file from the Vault via a *checkOut* or keeps a file checked out via a *checkInOut*, in which case this *notify* is preceded by an *update*. Similarly, they receive 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 this 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 CC. Upon receiving this action, 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 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) to a file by sending a *register* (*unRegister*) to CC. Rather than extracting a file, a User can always request a read-only copy of a file by sending a *get* to CC, upon which 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.

- modify the file and then put it back into the Vault by sending it via a *checkIn* to CC,
- refrain from modifying the file and simply return the file as it was by sending an *unCheckOut* to CC, or
- insert a modified version of the file into the Vault—while keeping the file in her possession for further editing—by sending it to CC via a *checkInOut* (in which case the file remains locked for other users, who can however 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, whereas only in the latter case CC moreover sends a *notify* to all those other users—indicating that the file remains locked—and it does so *after* it has sent the *update*. Finally, a User can always decide to insert a new file into the Vault by sending it to CC via an *import*.

4.2. Model checking the thinkteam protocol

In this section we present the PROMELA specification and some of the results of the verification of concurrency control properties and properties that are related to user awareness. In particular, we show that by adding a pub/sub event notification service to thinkteam, the users' awareness of the status of the development of the engineering product and the activities of the design team increases, which helps users to avoid the lost-update problems. In fact the pub/sub mechanism automatically informs subscribed users about changes to relevant files.

4.2.1. The PROMELA specification

The full and commented PROMELA specification of the augmented thinkteam protocol is given in Appendix A.[2] Here we list the assumptions on which it is based and provide a justification where necessary.

Besides the modelling decisions described in Section 4.1, we made several assumptions in our specification in order to reduce the size of both the state space and the state vector (used by SPIN to uniquely identify a system state). The most important assumptions, and their impact on the PROMELA specification, are:

1. At any moment, there is only one file (called file 0) in the Vault, hence the *import* of a file by a user currently is not modelled. This may seem a strong limitation of the model, but the properties of interest focus on mutual exclusive write access of users interested in modifying the *same* file and related notification actions. This justifies this abstraction.
2. The administrative user actions *notify* and *update* are always enabled. To achieve this, process User has an associated UserAdmin process, which does nothing else than receiving the *notify* and *update* messages.

[2] This is a slightly improved version of the specification in [45].

3. After a `User` has sent a *get* to the `CC` it 'actively waits' for an answer, i.e. the user activity is suspended, whereas users can be involved in further interactions while waiting for the `CC` to respond to their *checkOut*. This is modelled by means of a specific PROMELA input command. This assumption has been made to considerably reduce the size of the state space, without interfering with relevant correctness properties.
4. No message is ever lost. Message loss is dealt with by lower-level communication protocols.
5. The transmission time of messages between user and server is considerably smaller than the inter-arrival 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, which reduces considerably the memory requirements for verification. Further verifications with SPIN indeed showed that replacing these handshake channels by channels with a small buffer, in order to model a more asynchronous communication, still leads to feasible memory requirements while preserving the correctness of the verified properties [45]. This justifies this abstraction.

Under the above assumptions we obtained a PROMELA model on which the *exhaustive* verification of a number of correctness properties of the thinkteam protocol with SPIN has been possible. All reported verifications were performed by running SPIN v4.1.3 on a SUN® workstation with 1000 Mb of available physical memory. The details of most of the verifications that we performed can be found in [45,18]. Here we summarise their outcome.

4.2.2. Deadlock detection

We performed a so-called full state-space search for invalid endstates, which is SPIN's formalisation of deadlock states, in case of 2–4 users. The results are summarised in Table 3.

In case of 4 `Users`, the available physical memory proved to be insufficient. However, after disabling the explicit *register* (but still allowing implicit registration by means of a *get*) and enabling SPIN's *minimised automaton procedure* with 28 as the maximal depth of the graph that is constructed for the minimised automaton representation (cf. [17] for details) no deadlocks were found—while a full state-space search was accomplished.

The reported results give a good impression of the fast-growing state-space size in applications of this kind and, consequently, of the difficulties in obtaining exhaustive verifications of relevant properties. This is one of the major reasons for some of the unsuccessful applications of model checking to groupware systems in the past [8] and the main motivation for the assumptions introduced in the previous section.

4.2.3. Verified properties

Further properties relevant to the correctness of the thinkteam protocol with a pub/sub event notification service can be divided into three classes: classic concurrency control properties, potential denial of service (starvation) and awareness-related properties. The verification of most of these properties has been described in detail in our earlier work [45,18]. Here we summarise the results and illustrate the formalisation of some of the less common awareness properties. Four basic concurrency control properties have been verified:

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*

The awareness properties focus on verifying whether the protocol deals properly with the notifications to the users so that they are in the position to properly keep track of the developments of the manufact design that is directly relevant to them:

AW-1a	A user does not receive a *notify* if she is not registered for the file the *notify* refers to
AW-1b	A user does not receive an *update* if she is not registered for the file the *update* refers to
AW-2a	Every *checkOut* must eventually lead to a *notify* to all users registered for the file that the *checkOut* refers to
AW-2b	Every *checkInOut* must eventually lead to a *notify* to all users registered for the file that the *checkInOut* refers to
AW-3	Every *unCheckOut*, *checkIn* and *checkInOut* must eventually lead to an *update* to all users registered for the file the messages refer to

Finally, we verified one denial of service property:

DoS	No user can be denied a service forever

Table 3
Results of full state-space searches for deadlocks.

Users	State vector	Depth reached	Deadlock	Memory used	Flags
2	84 byte	4423	No	37.574 Mbytes	
3	108 byte	483303	No	129.529 Mbytes	
4	132 byte	10484899	No	916.095 Mbytes	-DMA=28

Obviously some of these properties, like those regarding the locking-based concurrency control mechanism, should also be satisfied by the basic, not yet extended thinkteam protocol. The awareness criteria, however, are specifically related to the pub/sub event notification service.

4.2.4. Awareness

Below we illustrate the formalisation and verification of some of the aforementioned awareness properties. Though never mentioned specifically, for all formulae in the sequel that contain a logical implication we have verified that the left-hand side can indeed become `true` in at least one system run. Moreover, in the sequel we will not spell out the exact points in the PROMELA specification where specific *labels* identifying relevant points of process executions have been added, but we will simply list them and assume their positions to be clear from the description. Otherwise, cf. the full PROMELA specification given in Appendix A.

No illegal notify (update). Criterion AW-1a(b) states that users do not receive a *notify* (*update*) if they are not registered for the file these messages refer to, i.e. the user does not receive any 'illegal' *notify* (*update*). We thus need to verify that a *notify* (*update*) is preceded by either a *get* or a *register*; moreover, we have to verify that this is the case whenever an *unRegister* takes place. Notice that both the above requirements must be fulfilled; in fact, a run in which a *get* or *register* is followed by an *unRegister*, which in turn is followed by *notify*, and such that there is no *get* or *register* between such a *unRegister* and the *notify* would satisfy the first but not the second requirement. On the other hand, a run where no *unRegister* at all takes place would trivially satisfy the second requirement. Identical considerations apply w.r.t. *update*. The first requirement is an instance of the typical *precedence* pattern: the requirement that a precedes b is expressed by $\neg(\neg a \,\mathtt{U}\, b)$, i.e. it is never the case that b can occur without a having taken place; if we furthermore want the precedence to be satisfied whenever c happens, then we have to use the additional invariant $\Box(c \rightarrow \neg(\neg a \,\mathtt{U}\, b))$, which gives the pattern for the second requirement. In order to instantiate a, b, and c properly, we added several labels to the specification of the `User` process, viz. `doneGet`, `doneRegister` and `doneUnRegister`—right after the `User` has sent a *get* (*register*, *unRegister*, resp.) to the `CC`. We furthermore added two labels to the specification of the `UserAdmin` process, viz. `doneNotify` and `doneUpdate`—right after it has received a *notify* (*update*) from `CC`. The formula that expresses Criterion AW-1a with reference to the first user is then:

$$\neg(\neg \mathit{GetOrRegister}\ \mathtt{U}\ \mathit{Notify}) \wedge \Box(\mathit{UnRegister} \rightarrow \neg(\neg \mathit{GetOrRegister}\ \mathtt{U}\ \mathit{Notify}))$$

where *GetOrRegister* stands for `User@doneGet` ∨ `User@doneRegister`, *Notify* stands for `UserAdmin@doneNotify`, and *UnRegister* stands for atomic proposition `User@doneUnRegister`. A similar formula can be verified for Criterion AW-1b, using label `doneUpdate` instead of label `doneNotify`. SPIN takes about 20 min to verify that the formula is satisfied. We verified analogous versions of these LTL formulae for the other users.

Notify if registered. Criterion AW-2a states that every *checkOut* must eventually result in a *notify* to all users registered for the file these messages refer to. Let us consider a generic state of a run; this would be a *bad* state w.r.t. the criterion if in this state a user, say `User0`, has registered itself and a *checkOut* is then performed by one of the other two users, `User1` or `User2`, which is not followed by a notification to `User0` despite this user had not unregistered (after its registration and) before the *checkOut* has taken place. In order to capture such a bad behaviour we added some user-specific labels to the specification of the `CC` process, viz. `doneGet0`, `doneRegister0` and `doneUnRegister0`—where `CC` receives a *get*, a *register* and an *unRegister* from `User0`—`doneCheckedOut1`, `doneCheckedOut2`—where `CC` responds to `User1` and `User2` respectively by sending them a *checkedOut*—and `doneNotify0`—by which the `CC` sends a *notify* to `User0`. The above bad behaviour is captured by the following formula:

$$\mathit{Registered0} \wedge (\neg \mathit{UnRegistered0}\ \mathtt{U}\ (\mathit{checkOut1or2} \wedge \Box \mathit{NoNotified0}))$$

where *Registered0* stands for proposition `CC@doneGet0` ∨ `CC@doneRegister0`, *UnRegistered0* stands for `CC@doneUnRegister0`, *checkOut1or2* stands for the proposition `CC@doneCheckedOut1` ∨ `CC@doneCheckedOut2`, and *NoNotified0* stands for ¬`CC@doneNotify0`. Let *AW2aBadFor0* stand for the above LTL formula. Then Criterion AW-2a w.r.t. `User0` corresponds to the LTL formula $\Box\neg$*AW2aBadFor0*.

The formulae for Criterion AW-2a w.r.t. `User1` and `User2` are similar to the above one and make use of further corresponding labels. We verified the above and analogous versions of this LTL formula in which the users change roles. SPIN verifies each formula in approximately 40 min. All the formulae are satisfied. In a pretty similar way we verified that also Criterion AW-2b holds.

Table 4
Results of the verifications discussed in Section 4.

	Property	State vector	Depth reached	Errors	Memory used
CC-1	Respond to lock	112 byte	3147677	0	473.209 Mbytes
CC-2	Unique lock/file	108 byte	434033	0	114.783 Mbytes
CC-3	Release file+lock	116 byte	7348	1	193.862 Mbytes
CC-4	Keep file locked	108 byte	434033	0	114.783 Mbytes
AW-1a	No illegal notify	112 byte	3071518	0	539.769 Mbytes
AW-1b	No illegal update	112 byte	3057025	0	558.508 Mbytes
AW-2a	Notify if registered	112 byte	3338868	0	967.955 Mbytes
AW-2b	Notify if registered	112 byte	3945506	0	894.278 Mbytes
AW-3	Update if registered	112 byte	4038761	0	745.900 Mbytes
DoS	Denial of Service	116 byte	1801	1	193.759 Mbytes

Update if registered. Criterion AW-3 states that every *unCheckOut*, *checkIn* and *checkInOut* must eventually result in an *update* to all users that are registered for the file these messages refer to. The schema for the formalisation of this criterion is the same as for Criterion AW-2a, namely, w.r.t `User0`, $\Box\neg$*AW3BadFor0*, where formula *AW3BadFor0* is the expected one:

$$\mathit{Registered0} \wedge (\neg \mathit{UnRegistered0}\ \mathtt{U}\ (\mathit{OR} \wedge \Box \mathit{NoUpdate0}))$$

where *NoUpdate0* stands for $\neg$(`CC@doneUpdate0` $\vee$ `CC@doneIOUpdate0`), *OR* stands for the following proposition:

`CC@doneUnCheckOut1` $\vee$ `CC@doneUnCheckOut2` $\vee$ `CC@doneCheckedIn1` $\vee$
`CC@doneCheckedIn2` $\vee$ `CC@doneCheckedInOut1` $\vee$ `CC@doneCheckedInOut2`

and proper labels are inserted in the relevant points of process `CC`. As before, similar formulae are defined w.r.t. `User1` and `User2`. Verification shows that also these formulae are satisfied.

4.3. Summary

All criteria listed in Section 4.2.3 were verified in either [45,18] or Section 4.2.4. Table 4 gives an overview of all the verification results.
The verifications show that the concurrency control and awareness aspects of the extended thinkteam protocol are—largely—well designed. However, the following two criteria turned out not to be satisfied.

Release file + lock. The thinkteam protocol does not oblige a user to ever return a file to the `Vault` that she has previously checked out. Therefore, a user that holds the lock on file 0 can endlessly perform *checkInOut* actions and never release this lock. This is inherent to the thinkteam protocol. In practice this undesirable situation is avoided by the intervention of a system administrator that a user can contact with the request to 'convince' another user towards releasing the file she currently has checked out.

Denial of Service. The `CC` can endlessly be kept busy by one of the users so that other users never get their turn. Also this is 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. In Section 5.4 we analyse the option of adding a queue for handling pending requests and compare an access policy based on retrial with one based on a waiting list.

5. A waiting-list access policy for thinkteam

As we have seen in Section 4.3, the problem with the denial of service property is that it may be the case that one of the users might never get its turn to, e.g., perform a *checkOut*, simply because the system is continuously kept busy by the other users, while this user did express a desire to perform a *checkOut*. As said before, such behaviour forms an integral part of the thinkteam protocol. This is because access to documents is based on the 'retrial' principle: thinkteam currently has no queue or reservation system handling concurrent requests for a document. Before simply extending thinkteam with a reservation system, a complementary quantitative analysis could provide further insight in the following usability issues:

- how often, on average, do users have to express their requests before they are satisfied, and
- under which system conditions (number of users, file processing time, etc.) would such a reservation system really improve usability.

Below we will address these issues.

5.1. Stochastic model of thinkteam*—the retry-based access policy*

In this section, we describe a stochastic model of thinkteam for its current retry-based access policy, which we later analyse using stochastic model-checking techniques. We consider the case that there is only one file, the (exclusive) access to which

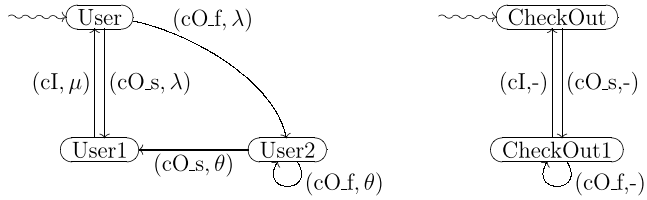


Fig. 7. From left to right: automata of the `User` and the `CheckOut` components.

```
% The definition of the User component consists of the definition for
% state User (which is the initial state), given below:
User = (cO_s,lambda).User1 + (cO_f,lambda).User2;

% the definition for state User1, given below:
User1 = (cI,mu).User;

% and the definition for state User2, given below:
User2 = (r_s,theta).User1 + (r_f,theta).User2;
```

Fig. 8. PEPA model for the `User` component.

is handled by a `CheckOut` component. Basically, we abstract from the file identity and we model only the fact that when a user attempts to perform a *checkOut*, this may either complete successfully or fail due to the fact that the requested file is already checked out. We are not interested, in this model, in describing the parallel access to different files and we furthermore assume that a user takes at most one file at a time in *checkOut* for modification.

We use the stochastic process algebra PEPA to specify the model. In Fig. 7, the models of the `User` and the `CheckOut` components are shown graphically as stochastic automata for the mere purpose of presentation.

The corresponding formal PEPA model is presented in Appendix B. The PEPA model corresponding to the `User` component is reported also in Fig. 8. The correspondence is straightforward and based on the *state-oriented* style of process specification: There is a distinct process defining equation for each state, and the body of such definitions are the expected ones.

As usual, a user, after performing activities which do not require access to the relevant file can express interest in checking out the file by performing a *checkOut*. This is modelled by state `User`, the initial state of the `User` component. The *checkOut* operation can either result in the user being granted access to the file or in the user being denied access because the file is currently already checked out by another user. In Fig. 7, the successful execution of a *checkOut* is modelled by activity (cO_s, λ), leading to state `User1`, while a failed *checkOut* is modelled by (cO_f, λ), leading to state `User2`. If the user does not obtain the file, then she may retry to obtain it, modelled by activity (cO_s, θ) in case of a successful retry and by (cO_f, θ) in case the *checkOut* failed. The *checkIn* operation, finally, is modelled by activity (cI, μ). In the sequel, we will often call λ (θ, μ, resp.) the request (retry, edit, resp.) rate. The `CheckOut` component takes care that only one user at a time can have the file in her possession. To this aim, it simply keeps track of whether the file is checked out (state `CheckOut1`) or not (state `CheckOut`, the initial state of the component).

The system with three users is modelled by the following PEPA term:

$$(\texttt{User}||\texttt{User}||\texttt{User}) \bowtie_{\{\texttt{cO}_\texttt{s},\texttt{cO}_\texttt{f},\texttt{cI}\}} \texttt{CheckOut}$$

The PEPA specifications are accepted as input by PRISM and then translated into the PRISM language. The resulting specification is given in Appendix C. From such a specification, PRISM automatically generates a CTMC with 19 states and 54 transitions that can be found in [46].

We use CSL to formalise and analyse in the above model various usability issues concerning the retry-based access policy used in thinkteam. In this context, it is important to fix the time units one considers. We choose hours as our time unit. For instance, if $\mu = 5$ this means that a typical user keeps the file in its possession for $60/5 = 12$ min on average. In the next sections we show the analyses we performed. All experiments have been performed by running PRISM v2.0 on an ordinary PC, taking only a few seconds of CPU time each. The iterative numerical method used for the computation of probabilities was Gauss-Seidel and the accuracy 10^{-6}. For details concerning these options, we refer the reader to [35] and to `http://www.prismmodelchecker.org`.

5.2. Analyses of performance properties

We first analyse the probability that a user that has requested the file and is now in 'retry mode' (state `User2` of the `User` component) obtains the requested file within the next five hours. This property can be formalised in CSL as

$$\mathcal{P}_{=?}\,([\ \text{TRUE}\ \mathcal{U}^{\leq 5}\ \mathit{User1}\ \{\mathit{User2}\}\ ]), \tag{P0}$$

where atomic proposition *User1* (*User2*, resp.) holds whenever component `User` is in state `User1` (`User2`, resp.).[3]

Formula P0 must thus be read as follows: "what is the probability that path formula TRUE $\mathcal{U}^{\leq 5}$ *User1* is satisfied for state `User2`?".

The results for Formula P0 are presented in Fig. 9a for request rate $\lambda = 1$ (i.e. on average a user requests the file 1 h after its last successful access), retry rate θ taking values 1, 5 and 10 (i.e. in 1 h a user on average retries one, five or 10 times), edit rate $\mu = 1$ (i.e. a user, on average, keeps the file checked out for 1 h) and for different numbers of users ranging from 1 to 10. The edit rate in this model is close to the mean value obtained from the log-file analysis in Section 2.4. The retry rate is a best guess, whereas the number of users considered is in line with what has been found in the log-file analysis.

Clearly, with an increasing number of users the probability that users get their file within the time interval is decreasing. On the other hand, with an increasing retry rate and a fixed edit rate the probability for a user to obtain the requested file within the time interval is increasing. Further results could easily be obtained by model checking for different rate parameters that may characterise different profiles of use of the same system. In particular, this measure could be used to evaluate under which circumstances (e.g. when it is known that only a few users will compete for the same file) the retry-based access policy would give satisfactory results from a usability point of view.

A bit more complicated measure can be obtained with some additional calculations involving steady-state analysis (by means of model checking). Fig. 9c shows the average number of retrials per file request for request rate $\lambda = 1$, retry rate θ taking values 5 and 10, edit rate μ ranging from 1 to 10 and for 10 users. The measure has been computed as the average number of retries that take place over a certain system observation period of time T, after equilibrium has been reached, divided by the average number of requests during T. We computed the average number of retries (requests) as follows. First, we computed the steady-state probability p (q) of the user being in 'retry mode' ('request mode', resp.), i.e. $p \stackrel{\text{def}}{=} \mathcal{S}_{=?}$ (*User2*) ($p \stackrel{\text{def}}{=} \mathcal{S}_{=?}$ (*User*), resp.). The fraction of time the user is in 'retry mode' ('request mode') is then given by $T \times p$ ($T \times q$). The average number of retries (requests) is then $\theta \times T \times p$ ($\lambda \times T \times q$). Hence the measure of interest is $(\theta \times p)/(\lambda \times q)$.

It is easy to observe in Fig. 9c that the number of retrials decreases considerably when the edit rate is increased (i.e. when the users, on average, keep the exclusive access to a file for a shorter period of time). We also note that a relatively high edit rate is needed to obtain an acceptably low number of retries in the case of 10 users that regularly compete for the same file. The effect on the average number of retries is even better illustrated in Fig. 9b, where with a similar approach as outlined above the average number of retries per file request is presented for request rate $\lambda = 1$, retry rate θ taking values 5 and 10, fixed edit rate $\mu = 5$ and various numbers of users. Clearly, the average number of retries per file request increases sharply when the number of users increases.

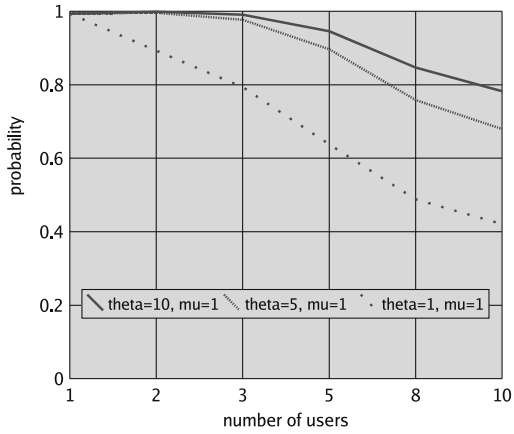
5.3. An abstract version of the retry-based model

For comparing the retry-based model with the waiting-list one which we will describe later, in Section 5.4, we are mainly interested in the *number* of users which are retrying when the relevant file is assigned to another user; more specifically, we are interested in the number of `User` components which are in the 'retry' state `User2`, and in the current status of the file, i.e. if it is checked out (state `CheckOut1` of component `CheckOut`) or not (state `CheckOut` of the component). In order to describe the kind of abstraction we are interested in, we first give an intuitive description of the information which is attached to the states of the PRISM CTMC associated to a PEPA model.[4] Strictly speaking, we are dealing with *state-labelled CTMCs* [40]—as is standard practice for stochastic model checking. A state-labelled CTMC is a triple $(\mathcal{S}, \mathbf{R}, L)$, where L is a function associating each state $s \in \mathcal{S}$ to a label in some label set. Intuitively, each state s of the CTMC resulting from a PEPA model, being a *global* system state, corresponds to an element of the Cartesian product of the sets of states of the components of the model. This correspondence can be encoded in the label of s. For instance, the initial state of the PEPA model of the retry-based policy can be thought of as labelled by $\langle$`User`, `User`, `User`, `CheckOut`$\rangle$; similarly, the state reached after an unsuccessful *checkOut* performed by, say, the third user while the second user is working on the relevant file and the first user is performing activities which do not require access to the file, is labelled by $\langle$`User`, `User1`, `User2`, `CheckOut1`$\rangle$. In the following we briefly describe the abstraction we need; more details can be found in [46]. The abstraction of interest can be performed in two steps. We first translate the CTMC $\mathcal{M} = (\mathcal{S}, \mathbf{R}, L)$ of the PEPA model into a CTMC $\mathcal{M}' = (\mathcal{S}', \mathbf{R}', L')$ where each state label is replaced by one containing only the relevant information. Then, as a second step, we minimise $\mathcal{M}'$ w.r.t. strong Markovian bisimulation equivalence [37], in order to get a smaller model to analyse.[5] $\mathcal{M}'$ is easily defined by letting $\mathcal{S}' \stackrel{\text{def}}{=} \mathcal{S}$ and $\mathbf{R}' \stackrel{\text{def}}{=} \mathbf{R}$ and, for each $s \in \mathcal{S}'$, $L'(s) = \langle x, y \rangle$ where x is 0 if `CheckOut` is an element of $L(s)$, i.e. the file is not checked out, and is 1 if `CheckOut1` is an element of $L(s)$, i.e. the file is checked out, while y is the number of elements of $L(s)$ equal to `User2`, i.e. the number of users which are in the retry state. Formally, $L'(s)$ is defined as $L'(s) \stackrel{\text{def}}{=} \langle \mathit{is}(\texttt{CheckOut1}, c), \mathit{Nretry}(u_1, u_2, u_3, c) \rangle$ where tuple $\langle u_1, u_2, u_3, c \rangle$ is the original label of s, i.e. $\langle u_1, u_2, u_3, c \rangle \stackrel{\text{def}}{=} L(s)$ and functions *Nretry* and *is* are defined in the expected way:

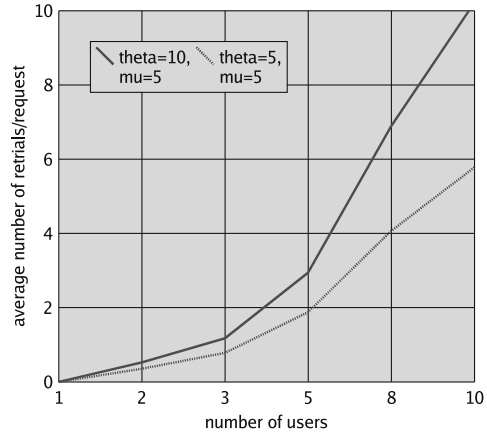
[3] The actual formula accepted by PRISM is given in Appendix C. It refers to the model representation in the PRISM modelling language, where different states of component `P` are represented by different values of variable `P_STATE`.

[4] As we already mentioned before, the actual information in PRISM CTMCs refers to the PRISM input language rather than to the PEPA one; here we refer directly to PEPA for the sake of readability, although at the cost of some technical imprecision.

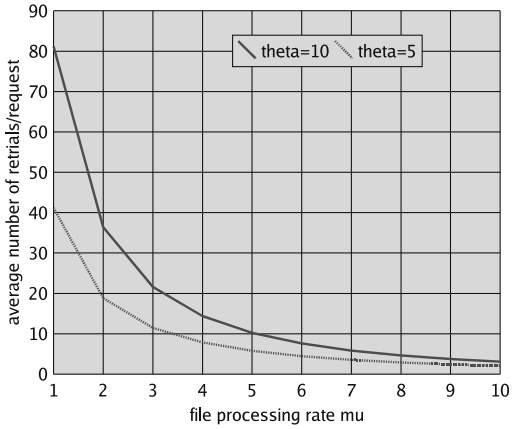
[5] We recall here that steady-state properties are preserved by strong Markovian bisimulation equivalence [47].



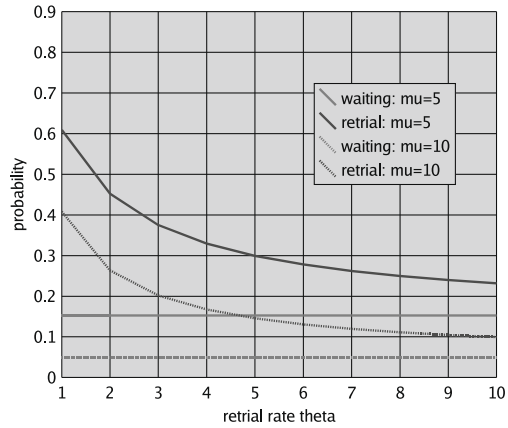
(a) Probability for users in 'retry mode' to checkOut requested file in next 5 h.



(b) Average number of retrials per file request for a varying number of users.



(c) Average number of retrials per file request in case of 10 users.



(d) The retry-based access policy vs. the waiting-list access policy.

Fig. 9. (a)–(d) Results of the analyses performed in this section. All rates are per hour.

$$Nretry(v_1, v_2, v_3, v_4) \stackrel{\text{def}}{=} is(\texttt{User2}, v_1) + is(\texttt{User2}, v_2) + is(\texttt{User2}, v_3)$$

$$is(w_1, w_2) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } w_1 = w_2 \\ 0 & \text{if } w_1 \neq w_2 \end{cases}$$

Given the relatively small size of the model we performed the transformation by hand. The resulting, reduced, CTMC is given in Fig. 10.[6]

Note that when a `User` inserts the file taken in *checkOut* back into the Vault via a *checkIn* activity, the `CheckOut` component does allow another user to *checkOut* the file but this need not be a user that has tried before to obtain the file. In fact, a race condition occurs between the request and retry rates associated to the *checkOut* activity (cf. states labelled by $\langle 0, 1\rangle$ and $\langle 0, 2\rangle$). Note also that once the file is checked in, it is *not* immediately granted to another user, even if there are users that have expressed their interest in obtaining the file. In such a situation, the file will remain unused for a period of time which is exponentially distributed with rate $2\theta + \lambda$ in the case that two users are retrying to get the file and $2\lambda + \theta$ if their is only one user retrying.

[6] Notice that the CTMC obtained by removing the self-loops from that of Fig. 10 is frequently used in the theory of *retrial queues* [48,49,36].

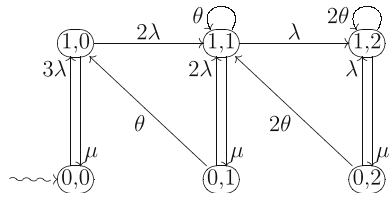


Fig. 10. CTMC for the retry-based access policy.

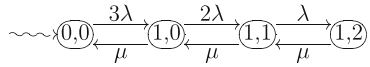


Fig. 11. CTMC for the waiting-list access policy.

5.4. Stochastic model of thinkteam—the waiting-list access policy

In contrast with our model for thinkteam's current retry-based access policy of Section 5.2, we now assume that a user's file request is put into a FIFO queue when the file is already in use by another user. The moment in which the file becomes available, the first user in this FIFO queue obtains the file. This implies the following changes w.r.t. the model of Section 5.2. Since a user no longer retries to obtain the file after an initial unsuccessful attempt to *checkOut* the file, the new `User` component has two states only, viz. state `User2` is removed from the `User` component as given in Fig. 7. Moreover, since the `CheckOut` component now implements a FIFO policy, the new corresponding component, i.e. component `FIFO` in the new PEPA model specification, must keep track of the number of users in the FIFO queue. The full specifications of the new `User` and `FIFO` components are given in Appendix D and their translations into the PRISM language are given in Appendix E. From these specifications, PRISM automatically generates a CTMC with 16 states and 30 transitions that can be found in [46].

We apply a translation also to this CTMC and minimise the result, getting the CTMC of Fig. 11. Also in this case the result labels are pairs $\langle x, y\rangle$ where x has the same meaning as in Section 5.3 while y is the number of users in the queue. Notice that states $\langle 0, 1\rangle$ and $\langle 0, 2\rangle$ no longer occur. This is due to the fact that, once the file is checked in, it is immediately granted to another user, viz. the first in the FIFO queue. The fact that we consider request rate λ, edit rate μ, one file and three users means that we are dealing with a M|M|1|3 *queuing system* [43,36]. The CTMC of Fig. 11 is the CTMC underlying this queuing system and this type of CTMC is also called a *birth-death process* [43,36].

5.5. Retry-based versus waiting-list access policy

It is interesting to compare the two models w.r.t. the probability that there are users waiting to obtain the file as a result of a *checkOut* request. To obtain this measure, we compute the probabilities for at least one user not being granted the file after asking for it, i.e. the steady-state probability p to be in a state in which at least one user has performed a *checkOut* but did not obtain the file yet. In the retry-based access policy this concerns states labelled by $\langle 0, 1\rangle$, $\langle 0, 2\rangle$, $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$ of the CTMC Retrial in Fig. 10, whereas in the waiting-list access policy this concerns states $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$ of the CTMC FIFOqueue in Fig. 11. Hence this can be expressed as the pseudo-CSL steady-state formula[7]:

$$p \stackrel{\text{def}}{=} \mathcal{S}_{=?}\,([\ (\texttt{Retrial\_STATE} = \langle 0,1\rangle)\ \vee\ (\texttt{Retrial\_STATE} = \langle 0,2\rangle)\ \vee\ (\texttt{Retrial\_STATE} = \langle 1,1\rangle)\ \vee\ (\texttt{Retrial\_STATE} = \langle 1,2\rangle)\ ])$$

for the retry-based access policy, while for the waiting-list one the formula is

$$p \stackrel{\text{def}}{=} \mathcal{S}_{=?}\,([\ (\texttt{FIFOqueue\_STATE} = \langle 1,1\rangle)\ \vee\ (\texttt{FIFOqueue\_STATE} = \langle 1,2\rangle)\ ]).$$

We point out here that having used the transformed CTMC results in very simple properties which state the measures of interest. Would we have used the original CTMCs, then we would have had to take into account all possible combinations of one or two users being in the retry state, leading to much more complex CSL formulae. This would become intractable when models with more than three users are considered.

The results of our comparison are presented in Fig. 9d for request rate $\lambda = 1$, retry rate θ (only for the retry-based access policy of course) ranging from 1 to 10, edit rate μ taking values 5 and 10 and, as before, three users.

It is easy to see that, as expected, the waiting-list access policy outperforms the retry-based one in all the cases we considered: The probability to be in one of the states $\langle 1, 1\rangle$ or $\langle 1, 2\rangle$ of the CTMC of the waiting-list access policy is always lower than the probability of the CTMC of the retry-based access policy to be in one of the states $\langle 0, 1\rangle$, $\langle 0, 2\rangle$, $\langle 1, 1\rangle$ or $\langle 1, 2\rangle$.

[7] Given the simplicity of the reduced CTMCs of Figs. 10 and 11, these formulae refer directly to their specification in the PRISM language.

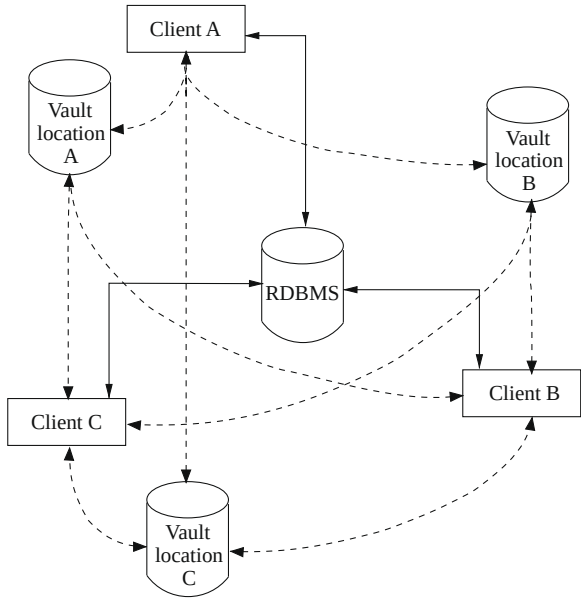


Fig. 12. Thinkteam with replicated Vaults. Document *checkIn/Out* is represented by dotted arrows; metadata operations are represented by solid ones.

Note that for large θ the probability in case of the retry-based access policy is asymptotically approaching that of the waiting-list one, of course given the same values for λ and μ. While we did verify this for values of θ upto 10^9, it is of course extremely unrealistic to assume that a user performs 10^9 retries per hour. We can therefore conclude that the time that a user has to wait 'in the long run' for the file after it has performed a *checkOut* is always less when using the waiting-list access policy than when using the retry-based access policy. Furthermore, while increasing the retry rate does bring the results for the retry-based access policy close to those for the waiting-list one, it takes highly unrealistic retry rates to reach a difference between the two policies that is insignificantly small.

6. Multiple replicated vaults for thinkteam

A further extension of thinkteam that think3 is considering is the addition of a service-oriented functionality, viz. that of multiple replicated Vaults. These Vaults reside in a number of locations that are assumed to be geographically distributed (cf. Fig. 12). thinkteam is assumed to have persistent data on the status of the replicated Vaults and on the status of all files, i.e. whether a file is currently checked out by a designer or available for modification.

When designers query thinkteam in this new setting, e.g. for a copy of a file, thinkteam typically responds by sending them the address of the 'best possible' Vault location. Ideally, this is the designer's preferred Vault location (e.g. with a good connection in terms of bandwidth), while a second-best location is assigned if the preferred location is down or has a too high workload. If, on the other hand, the most recent *checkIn* of the requested file was performed by the same designer, then the local version of the file can be used, thus saving a *checkOut*.

When designers have obtained a Vault address, they may *checkOut* the file, edit it and eventually *checkIn* the file, again with a strong preference for their preferred Vault location. After each *checkIn*, the related location informs thinkteam that the file has been uploaded. Afterwards, thinkteam updates the status of the file, i.e. removes its lock, and makes it available for other designers requesting it. This communication also transfers the status information of the Vault locations to thinkteam. Neither the communications between the Vault locations needed to keep them consistent nor those between the Vault locations and thinkteam are represented in Fig. 12. In the model of the system considered in this article, we do not explicitly address the communications between Vaults but assume that they are kept consistent using suitable algorithms. The communication between the Vaults and thinkteam will be modelled explicitly.

6.1. Stochastic model of thinkteam

The model that we consider, and which is described in detail in Appendix F, is composed of three Vault locations (components `Va`, `Vb` and `Vc`) that contain identical file repositories, three explicitly modelled Clients (components `CA`, `CB` and `CC`) that compete for the same file and the thinkteam application (component `TT`). Each Vault location is connected to `TT` and they communicate their status to `TT` systematically. Interesting aspects of the status of a Vault location for the purpose of performance analysis are, e.g., its workload, availability (i.e. being up or down) and the bandwidth available to the various

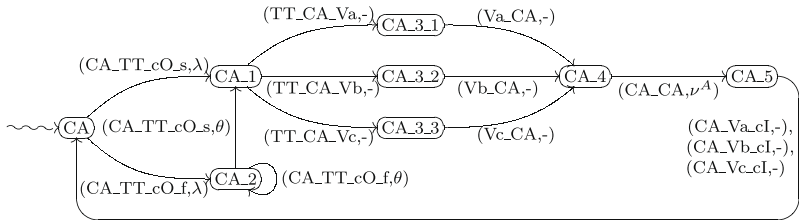


Fig. 13. Client CA.

Clients. TT keeps a record of the status of all files, i.e. whether a file is locked (because it is checked out by a Client) or available for download and modification.

The current model is based on the following assumptions.

1. The bandwidth between a Client and a Vault is constant and each Client prefers downloading and uploading files from the Vault with the best connection. At times this connection may be down, however, in which case a Client will use the next preferred Vault.
2. Each Client has a static preference list with the preferred Vault order.
3. The three explicitly modelled Clients do not influence significantly the overall performance of the full system, which includes many active Clients modelled only implicitly by means of the responsiveness characteristics of the various Vaults. Our aim is to analyse a number of correctness and usability aspects of the system from these three Clients' point of view.
4. We only consider a subset of the operations available to thinkteam Clients, viz. the most important ones: *checkOut* and *checkIn*. This keeps the model relatively simple. Further operations can easily be added at a later stage.
5. TT is not enabled to inform a Client of the fact that the local version of the file is the most up-to-date if TT notices that the most recent *checkIn* of the requested file was performed by that Client. This is future work.
6. System availability is not addressed; i.e. the possibility that all three Vaults are unavailable is not modeled. Consequently a user *always* gets the address of a Vault from thinkteam.

Also in this case we abstract from the specific file identity and model only the fact that when a Client tries to perform a *checkOut*, she may succeed or fail due to the locking status of the file under the assumption that, in general, three Clients are interested in the same file. Instead, we focus on the interactions of different Clients with different vaults and on their impacts on system behaviour in terms of functional and performance properties. We assume furthermore that Clients have at most one file checked out at any time.

For a compact presentation, in the following we present the behaviour of the components graphically as abstract forms of stochastic automata in Figs. 13 and 14, which have also been used in discussions with our colleagues from think3. The labels of the states and transitions will play an important role in the next section, when discussing the model's analyses. The transition labels are of the general form ⟨from⟩_⟨to⟩_⟨action⟩, in which the part ⟨from⟩_⟨to⟩ indicates the direction of the information flow between processes (e.g. CA_TT denotes a communication from CA to TT) while the ⟨action⟩ part indicates a specific action (e.g. cO_s for successful *checkOut*, cO_f for failed *checkOut* and cI for *checkIn*).

Client process. We describe in detail the behaviour of CA (cf. Fig. 13), that of CB and CC being similar. Initially, in state CA, with rate λ the Client performs a request to TT to download a file for modification. This request is successful if the file is available (CA_TT_cO_s, λ) and fails if the file is currently being edited by another Client (CA_TT_cO_f, λ). In case the request is successful, TT provides the address of the 'best possible' Vault location to the Client (e.g. TT_CA_Va means that Client CA receives the address of Vault Va).

The policy to assign a Vault location is kept very simple in this model: Clients receive the address of their preferred Vault location (the first on their preference list) with highest static probability. They receive the address of different Vault locations with lower probabilities, thus modelling the fact that the preferred Vault location is not always available, be it due to a high workload or due to temporary unavailability. These probabilities can be tuned in order to better match the performance characteristics of the planned system. Indications for such probabilities are obtained, e.g., from the log-file analysis of the expected performance characteristics of the single centralised Vault currently used in thinkteam (cf. Section 2.4).

When Client CA has obtained the address of a Vault location, the Client can *download* (e.g. from Vault A by (Va_CA,-)) the requested file, then edit the file for about $1/\nu^A$ time units while in state CA_4, which is left with transition (CA_CA,ν^A) and, finally, *upload* the file to a Vault by means of a *checkIn* (e.g. to Vault A by (CA_Va_cI,-)), following a preference list as for downloading, and return to the initial state CA. Actions with a rate that is indicated by '-' are passive, i.e. the rate value is established during synchronisation—in this case between the Client process and the Vault process, with the Vault process determining its value.

If the Client's request for a file fails, a series of retry actions is started (state CA_2), with rate θ, to obtain the file at a later moment. After a number of successive failed requests, the Client eventually performs a successful request and moves to state CA_1 (we recall here that PRISM assumes a fair process semantics, i.e. the probability of any set of unfair runs amounts to 0).

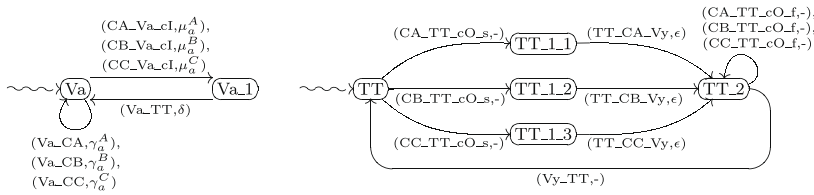


Fig. 14. From left to right: Vault Va and TT.

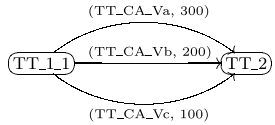


Fig. 15. Close-up of transitions from state TT_1_1 to state TT_2.

Vault process. In the following we describe component Va (cf. Fig. 14), the behaviour of Vb and Vc being very similar. Each Vault (location) can receive *download* operations from a Client (for Va and client, say CB, by (Va_CB,γ_a^B)) with rate γ_a^B corresponding to the average download time $1/\gamma_a^B$ for that specific Client and Vault, alternated with *checkIn* operations (again, in the case of CB, CB_Va_cI,μ_a^B) with rate μ_a^B. After each *checkIn*, the Vault informs TT (by (Va_TT,δ)) that the file has been *uploaded* to the Vault. In this way, TT can update the status of the file, i.e. remove the lock and make it available to other Clients. The same communication also models the transfer of the status information of the Vaults to TT.

TT **process.** The behaviour of TT is modelled as follows (cf. Fig. 14). Initially, TT waits for file requests from Clients that it may honour (e.g. from CA by action (CA_TT_cO_s,-)). In case of such a successful file request, TT assigns a Vault to the Client, using the assignment policy described above.

This policy can be modelled in a stochastic way by creating a race condition between the different assignments in the following way. If one wants that Client A is assigned Vault A in approximately 50% of the cases, Vault B in approximately 33% and Vault C in approximately 16% of the cases, one can choose suitable rates that reflect this. For instance, state TT_1_1 has *three* outgoing transitions, labelled by (TT_CA_Va,300), (TT_CA_Vb,200) and (TT_CA_Vc,100), as is shown in a close-up of that fragment of the model in Fig. 15.

The total exit rate from state TT_1_1 is thus $300 + 200 + 100 = 600$, and the probability that Vault A is assigned to Client A is then $300/600 = 0.5$. These relatively high exit rates model the fact that Vault assignment is very fast as compared to the other activities.[8] Actions such as (TT_CA_Vb,ϵ) model assigning a Vault, locking the file and sending the address of Vault B to Client A. Any further request for the same file is explicitly denied (e.g. to Client B by (CB_TT_cO_f),-) until TT has received a message from a Vault indicating that the file has been uploaded (e.g. for Vault B by (Vb_TT,-)). TT is then back in its initial state, ready to accept further requests for the file.

Full specification. The specification of thinkteam is completed by the parallel composition of the three Client components (CA, CB, CC) the three Vault components (Va, Vb, Vc), and the TT component, as follows (with X=A,B,C, y=a,b,c and z=f,s):

$$(\mathtt{CA}||\mathtt{CB}||\mathtt{CC}) \bowtie_{\{\mathtt{CX\_TT\_cO\_z},\mathtt{TT\_CX\_Vy},\mathtt{CX\_Vy\_cI},\mathtt{Vy\_CX}\}} (\mathtt{TT} \bowtie_{\{\mathtt{Vy\_TT}\}} (\mathtt{Va}||\mathtt{Vb}||\mathtt{Vc})).$$

The complete PEPA specification and its translation into the PRISM language are given in Appendices F and G, resp. From such a specification, PRISM automatically generates a CTMC with 104 states and 330 transitions.

Note that we have restricted the model to the investigation of the performance characteristics for three Clients competing for the same file during approximately the same period. More detailed analyses of the log-file might provide more information on the typical and maximal number of Clients that usually compete for the same file. The model can easily be extended with a limited number of explicitly modelled Clients, along the same lines as in Section 5.

6.2. Analysis of the model of thinkteam with multiple replicated vaults

All the analyses reported in this section were performed with the stochastic model checker PRISM v3.1.1 on an ordinary PC and, in the case of a model with three Clients, took a negligible amount of CPU time.

The rate values used for the analyses are given in Table 5 except when explicitly stated. These rates should be read considering that the letters in the subscripts refer to the names of the Vaults while those in the superscripts indicate the Clients. Hence μ_b^A, e.g., is the upload rate between Vault B and Client A, which equals the download rate γ_b^A between Vault B and Client A.

[8] For reasons of space and readability, these details are abstracted from in Fig. 14: Only a nominal indication ϵ of the relevant rates is given.

Table 5
Rate values used for the functional analyses.

	μ_a^A	μ_a^B	μ_a^C	μ_b^A	μ_b^B	μ_b^C	μ_c^A	μ_c^B	μ_c^C							ϵ_2	ϵ_3
λ	γ_a^A	γ_a^B	γ_a^C	γ_b^A	γ_b^B	γ_b^C	γ_c^A	γ_c^B	γ_c^C	ν^A	ν^B	ν^C	θ	δ	ϵ	2ϵ	3ϵ
0.1	60	20	40	40	60	20	20	40	60	0.3	0.3	0.3	6	100	100	200	300

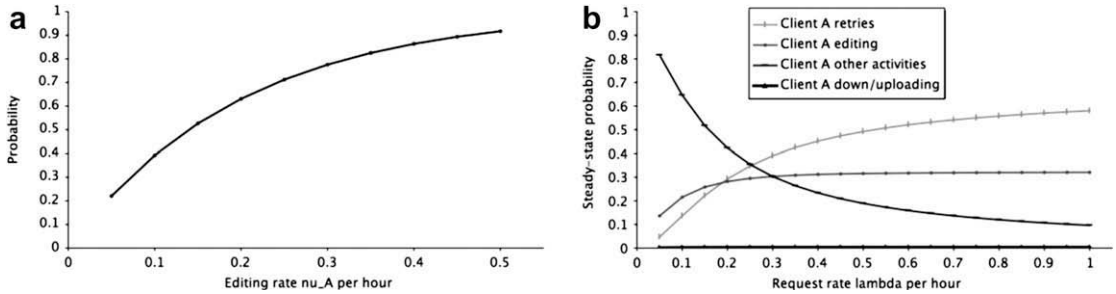


Fig. 16. (a) Probability for a Client to *checkIn* a file within 5 h after the *checkOut*. (b) Percentage of time Client A spends on activities (varying file inter-access times).

The time units in this model are hours. This means for instance that rate $\nu^A = 0.3$ models that on average Client A spends $60/0.3 = 200$ min editing a file, while rate $\theta = 6$ models that on average a Client retries to *checkOut* a locked file every $60/6 = 10$ min.

6.2.1. Analyses of functional properties

Before undertaking performance analyses of the model, it is important to gain confidence in its correctness by verifying several qualitative properties. We have verified, e.g., the absence of deadlocks, various progress properties and mutual exclusion of the right to edit a file. Two examples of such properties are the following.

Whenever CA successfully manages to *checkOut* a file from Va, eventually a *checkIn* of that file is performed. This property is captured by the following pseudo-CSL formula:

$$\mathcal{P}_{\geq 1}\,([\ \text{TRUE}\ \mathcal{U}\ \mathit{CA\_5}\ \{\mathit{CA\_3\_1}\}\ ]), \tag{P1}$$

where atomic proposition *CA_5* (*CA_3_1*, resp.) holds whenever CA is in state CA_5 (CA_3_1, resp.). Verification with PRISM confirmed Formula P1 to hold for the model.

The probability should be (at most) zero that two (or more) Clients will eventually obtain permission to modify the same file at the same time. This property can be formalised in pseudo-CSL as follows:

$$\mathcal{P}_{\leq 0}\,([\text{TRUE}\ \mathcal{U}\ (\mathit{PermittedAB} \vee \mathit{PermittedAC} \vee \mathit{PermittedBC})]), \tag{P2}$$

where *PermittedAB* stands for the proposition $\mathit{CA\_1} \wedge \mathit{CB\_1}$ with *CA_1* (*CB_1*, resp.) holding whenever component CA is in state CA_1 (CB is in state CB_1, resp.). *PermittedAB* thus means that both CA and CB have been permitted to edit the file. *PermittedAC* and *PermittedBC* are defined similarly. Verification with PRISM confirmed that Formula P2 holds for the model.

6.2.2. Analyses of performance properties

In this section we show some performance aspects of the model, and in particular some usability aspects seen from the perspective of the Clients.

Swiftness of returning file. Formula P1 above only shows that Clients eventually upload the file (after they downloaded it). The following pseudo-CSL formula can be used to quantify this aspect, for client A (for clients B and C the formulae are similar):

$$\mathcal{P}_{=?}\,([\ \text{TRUE}\ \mathcal{U}^{\leq 5}\ \mathit{CA\_5}\ \{\mathit{CA\_3\_1}\}\ ]), \tag{P3}$$

i.e. what is the probability that within 5 h after successfully downloading the file (state CA_3_1), CA is in state CA_5 ready to upload the file? The results are presented in Fig. 16a for edit rate ν^A varying from 0.05 to 0.5, i.e. from an average of 20 down to 2 h of editing activity. As expected, the less time a Client spends editing, the higher the probability that the file is returned within 5 h.

Behaviour on the long run. One of the parameters that influences the way Clients use thinkteam, and which therefore influences the time they spend on different activities, is the average time between required accesses to the same file. The change in average time spent on different activities by a typical Client when varying this inter-access time can be well-observed in Fig. 16b: For each activity, the percentage of time Client A spends on that activity is shown, for various values of λ, which has a strong impact on the request rate.

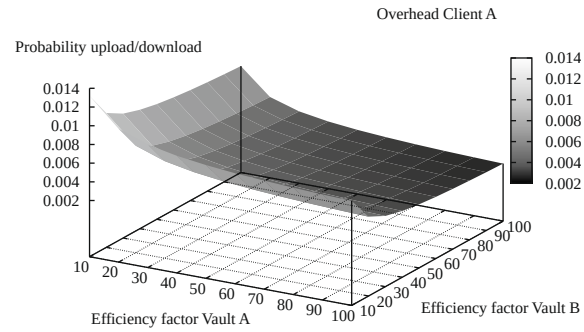


Fig. 17. Steady-state probability for a Client to be down- or uploading files.

We see that when λ is very low, most time is spent on other activities, very little on down- and uploading and on retrying. This pattern changes considerably as λ increases. The time spent waiting for the file (retrying) increases sharply and, obviously, a Client spends much less time on other activities. She also spends more time editing the file, but this increase is not so sharp.

The family of properties analysed to obtain the results shown in Fig. 16b are simple steady-state properties of the form $\mathcal{S}_{=?}$ ([a]) where proposition *a* holds when the component of interest is in the relevant state. For instance the formula

$$\mathcal{S}_{=?}\,([\ CA\_2\ ]), \tag{P4}$$

CA_2 indicates the steady-state probability of Client A retrying in order to obtain a file, given that *CA_2* holds whenever `CA` is in state `CA_2`, to indicate that Client A is retrying to obtain a file.

Time spent downloading and uploading files. The time Clients spend down- and uploading files depends largely on the bandwidth of their connection to the Vaults, the size of the files and the workload of the Vaults. Fig. 17 shows the effect that a change in workload of Vaults A and B has on the percentage of time Client A spends down- and uploading files. On the *x*-axis an efficiency factor for Vault A is shown, ranging from 10 to 100, which multiplies the download rates γ_a^X, with X = A, B, C, of Vault A for Clients A, B and C, which are initially set to 0.6, 0.2 and 0.4, resp. Likewise on the *y*-axis for Vault B, multiplying the download rates of that Vault for Clients A, B and C, which are initially set to 0.4, 0.6 and 0.2, resp. All other rates are as in Table 5. On the *z*-axis the steady-state probability for a Client to be down- or uploading files is shown.

Hence, the higher the efficiency factor, the faster the Vaults perform, and the lower the workload. Indeed, as expected, we see that the probability that Client A spends time on down- and uploading on the long run is smallest when both Vaults (A and B) are working optimally. We also observe that if only Vault B has a high workload (i.e. a low efficiency factor), and thus performs slower, then this influences the time that Client A spends down- and uploading. This is because part of the time Client A downloads from (and uploads to) Vault B. Note also that this percentage does not decrease much after a certain performance of the Vaults has been reached: This occurs more or less at efficiency factor 40-50, for the parameter settings chosen for the analysis. The results in Fig. 17 have been obtained by verifying a formula similar to Formula P4 for Client A being in either of the states `CA_3_1`, `CA_3_2`, `CA_3_3` or `CA_5`.

Number of retries per success. The perceived usability of thinkteam also depends on how often it happens that a Client cannot obtain a file that she intends to modify. Failing to obtain a file means that the Client needs to spend time on either keep trying to obtain it at a later stage or change her work-plan and search for another file. If this situation occurs frequently, a Client might perceive this as annoying. Moreover, it may lead to the introduction of errors (the Client may forget to edit the file later, or forget what modifications to make) or to problems in the overall workflow plan, and thus result in a delay in the delivery of the final product. It would therefore be useful to be able to quantify this problem under various conditions and for different user profiles of Clients using thinkteam. For instance, the different phases of design may induce a different use of thinkteam: Initially, Clients may take more time to modify a file because of a completely new design, but closer to the deadline there might be a phase in which many Clients need to frequently make small modifications in order to fine-tune the various components.

Fig. 18 shows the results of one such an analysis. On the *x*-axis an amplification factor for both the edit and the retry rates is shown, ranging from 1 to 10. On the *y*-axis the request rate λ is shown. The *z*-axis shows the average number of retries that Client A needs before obtaining the permission to modify a file changes with the simultaneous increase of the edit and retry rates (modelling Client behaviour close to a deadline) and an increase in the frequency with which Client A needs the file time and again. The chosen edit and retry rates are 0.025 and 0.5, resp., which have been multiplied with a factor ranging from 1 to 10. The figure also shows how the number of retries per success changes with the request rate λ. Given that we consider on average three users competing for a file, the total inter-access rate is 3λ. This means that the inter-access time of files is ranging from 4 h to 1 h, modelling a usage pattern with time characteristics that is situated towards the faster end of the spectrum shown in Section 2.4.

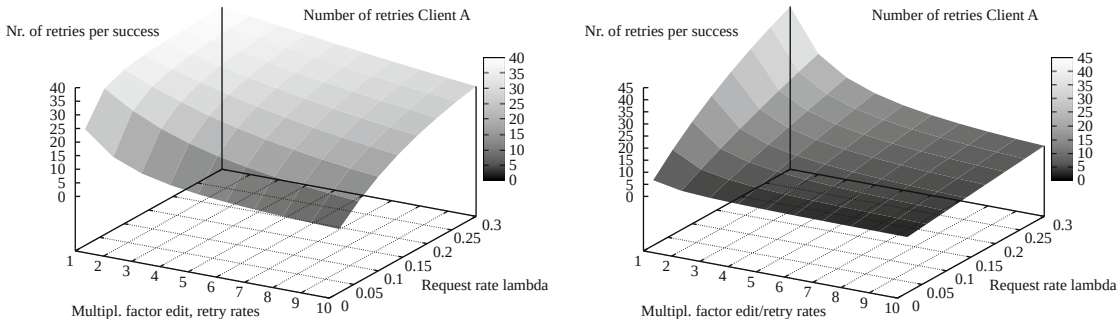


Fig. 18. (a) Average number of retries Client A needs to obtain a file. (b) Average number of retries Client A needs to obtain a file if she reorganises her work-plan.

Fig. 18a has been obtained by extending the PRISM specification of our model with a reward structure to count the number of successful requests and another one to count the number of retries. Reward structures defined over a CTMC define which actions need to be counted. Subsequently, the model can be analysed with respect to a number of reward-based properties. We use the steady-state reward formula that calculates the average reward on the long run. Its syntax is $\mathcal{R}\{\text{"label"}\}=?\,[\,S\,]$, in which "label" is a reward structure and S denotes that the steady-state reward is calculated. The number of repeated failed requests per successful request shown in Fig. 18a is thus obtained by dividing the outcome of this formula for label=NrFailedRequestsClientA by that for label=NrSuccessfulRequestsClientA. The definition of the reward structures can be found in Appendix H.

Despite the fact that there are relatively few Clients needing the same file more or less in the same period, the number of retries required to obtain the permission to edit a file is relatively high. Few Clients would be happy to have to try more than thirty times to get access to a file. If we were to adjust our model by allowing Clients to change work-plan in case a file is not available, i.e. let the Client choose another file for editing, then the situation would improve. Such an adjustment is relatively simple: It suffices to add one transition in `CA` from the retry state `CA_2` to the initial state `CA`, and similarly in `CB` and `CC` from `CB_2` and `CC_2`, resp., to their initial sates. The results for the new model of the above reward-based properties are shown in Fig. 18b. Of course this model is relatively optimistic in the sense that usually clients need to have access to a file within a given time and cannot always deliberately decide to perform other activities first.

Other solutions can be conceived which would still be based on variations of the retry policy like adding further retry states with different rates, or even replacing the simple retry state with an additional CTMC modelling different sub-phases of a retry phase based on a more sophisticated discipline. In particular, refining the retry state into a new CTMC characterising a (sequence of) Phase-Type distribution(s) [50] could be one way for approximating general (i.e. non-exponential) distributions of (subsequent phases of) the retry phase.

We leave these extensions for further study and, instead, in the present paper we showed the impact of replacing the retry policy with a reservation policy or extending thinkteam with an event notification service, in a simple version of thinkteam with one single centralised Vault.

7. Lessons learned

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 acquired a basic knowledge of SPIN and of the stochastic model checker PRISM. In fact, we have been able to use these model checkers in various ways to present the behaviour of our models of the groupware protocols underlying thinkteam and its proposed extensions. Examples include simulation, message sequence charts and counterexamples. This has helped to detect a number of ambiguities and unclear aspects of the designs that think3 had in mind regarding the proposed thinkteam extensions. Examples include the two criteria (CC-3 and DoS) that were shown to be invalid (cf. Section 4), the additional Criterion AW-2b that was formulated in the context of our co-operation and the numerous questions that were addressed during these interactive meetings. To give an idea of the type of questions, we list three of them, with their answers.

When exactly are which requests enabled? Our initial model of thinkteam prohibited a user to perform a *get* after initiating a *checkOut* for the same file. However, think3 realised that it was preferred to allow a *get* also after a *checkedOut* has been obtained. Such a *get* should be seen as a kind of 're-get'.

What are the exact semantics of requests? Our initial model of thinkteam allowed a user to (*un*)*register* for a file for which she possessed editing rights. think3 showed us that in practice such situations are undesirable. Our model does, however, allow a user to (*un*)*register* for a file as long as she is waiting for a response to the *checkOut* that she sent for this file. After all, this response may also be negative.

How are concurrent requests handled? As mentioned before, access to files through thinkteam is currently based on the 'retrial' principle: There is no queue (or reservation system) handling concurrent file requests. As a result of our co-operation, think3 is investigating whether it would be useful to equip thinkteam with a file reservation system.

Think3 furthermore has expressed the intention to use our model of thinkteam extended with a pub/sub event notification service as a basis for the planned implementation of this extension, which of course will provide increased confidence on the usefulness of the system design.

The results we have obtained with the model checker PRISM have been essential to obtain feedback on the performance aspects of the problems detected by qualitative model checking. Our relatively simple models were easy to grasp, but at the same time allowed the analysis of quite a number of different aspects, e.g. by varying the values of the model's parameters. In this respect, the analysis of the log-file that has been obtained from a real use of thinkteam turned out to be extremely useful for obtaining a realistic estimate of the parameter values. On the other hand, this log-file analysis has also showed that some data that would have been useful for a further evaluation of usability issues is not currently collected in thinkteam's log-files. Hence, our co-operation has also served to develop further ideas about which data to log in the future.

For think3, the experience with model checking specifications of concurrent systems before actually implementing them, has been a true eye opener. They have recognised the inherent concurrency aspects of groupware systems like thinkteam, as well as their intricate behaviour. The relatively simple, lightweight and abstract high-level models that we have developed during our co-operation, furthermore have 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 and PRISM in order to get more acquainted with these model checkers and—ultimately—to acquire the skills to perform automated verification of the (groupware) protocols underlying their software systems themselves. This makes our co-operation an excellent example of technology transfer to industry.

8. Conclusions and future work

The research described in this article has taken place in the context of the national Italian research project tocai.it, which aims at the application and development of knowledge-based technologies to support the aggregation of enterprises over the Internet. think3 is one of the industrial partners in this project. Groupware products like thinkteam are in continuous evolution. Moreover, thinkteam is used by many important manufacturing industries that nowadays have several dislocated design departments, each of which needs reliable and efficient software systems to co-operate in an efficient way. The many inherent concurrency aspects that think3 needs to face when producing their software, and their awareness of the difficulties this implies when assessing the quality of their products, made it easier to raise their interest for the use of model-checking techniques in the early phases of software design.

In this article, we have illustrated that with relatively simple abstract models important issues concerning asynchronous dispersed groupware systems can be addressed. Examples include awareness, concurrency control and usability-related issues. The feature-oriented approach combined with a modular, prototyping-like way of generating models and results, has shown helpful also for making think3 familiar with the possibilities and limitations of current model-checking techniques.

We have also showed the importance of a combined qualitative and quantitative analyses to be able to make informed design decisions. The development of the models in close collaboration with think3 has showed that the activity was worthwhile to obtain precise and unambiguous specifications and has helped to provide better documentation of thinkteam. On the other hand, the models and results have benefited considerably from the information that we have managed to obtain by analysing the log-file of the actual use of thinkteam. Still further analyses of such data can be of help to obtain models that can also be used to analyse thinkteam when used under different usage patterns. We believe this to be an interesting topic for further technology study. The models and results in their turn have also generated ideas for improved logging of thinkteam's user activities, in order to get more insight into the usability aspects of thinkteam at work.

Finally, the expected shift from dedicated software for collaborative systems to architectures for *global/mobile computing*, which will be *service oriented* in nature, will raise even more issues related to concurrency and performance. Issues which have to be addressed are those of *distribution awareness* and *mobility*—intended both as mobile *computing*, where mobile devices require network connections to be open and closed dynamically, and as mobile *computation*, where code mobility is required—typical of global/mobile computing as well as those of *workflow management* and *session/correlation* handling, typical of service-oriented computing. Several calculi and languages have been proposed in the literature which address the abovementioned issues (e.g. KLAIM [51], COWS [52], CaSPiS [53] and many more), together with their stochastic extensions (e.g. StoKLAIM [54,55], Stochastic COWS [56], MarCaSPiS [57]) and related logics and prototype model-checking tools (e.g. [58,59,60]). The goal to assess the adequacy of the above languages, logics and tools for the groupware application domain, or to properly customise them in order to integrate them in the software development processes of groupware industries, will be one of the future challenges.

9. Acknowledgments

The authors thank the reviewers for the valuable corrections and suggestions which greatly helped improving the paper.

Appendix

A. PROMELA and LTL Specifications of Augmented thinkteam Protocol

In this appendix the PROMELA Specification of Augmented thinkteam Protocol discussed in Section 4.2.1 is given (Appendix A.1), together with the LTL formulae in the format accepted by SPIN (Appendix A.2).

A.1. PROMELA *model*

```
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 */
  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 CC()
{
  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;
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
  :: 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) */
       ID = 0;                                       /* (1) 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) ->
                      doneIOUpdate0: skip           /* (label for verification purposes) */
                 :: (ID == 1) ->
                      doneIOUpdate1: skip           /* (label for verification purposes) */
                 :: (ID == 2) ->
                      doneIOUpdate2: skip           /* (label for verification purposes) */
                 fi
            :: else -> skip
            fi;
            ID++
       :: else -> break
       od;
       ID = 0;                                     /* (2) notifies all registered users: */
       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) ->
                      doneIONotify0: skip           /* (label for verification purposes) */
                 :: (ID == 1) ->
                      doneIONotify1: skip           /* (label for verification purposes) */
                 :: (ID == 2) ->
                      doneIONotify2: skip           /* (label for verification purposes) */
                 fi
            :: else -> skip
            fi;
            ID++
       :: else -> break
       od
 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 */
 od
}

/* Initialisation process: */

init {
```

```
  byte Users = 0;

  atomic
  {
    run Vault();                                                          /* pid = 1 */
    run CC();                                                           /* 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
  }
}
```

A.2. SPIN *LTL formulae*

```
/* LTL formula for 'No illegal notify' property */

! (! getOrRegister U notify) && [] ( unRegister -> ! (! getOrRegister U notify))

/* with the following macro definitions added in the model */

#define getOrRegister   (User[3]@doneGet || User[3]@doneRegister)
#define notify   (UserAdmin[4]@doneNotify)
#define unRegister (User[3]@doneUnRegister)

/* The formula for 'No illegal udate' similar and not reported here */

/* ================================= */
/* LTL formula for 'Notify if registered*/

[]!(registered0 && ((! unRegistered0) U (checkOut1or2 && []noNotified0)))

/* with the following macro definitions added in the model */

#define registered0   (CC[2]@doneGet0 || CC[2]@doneRegister0)
#define unRegistered0   (CC[2]@doneUnRegister0)
#define checkOut1or2   (CC[2]@doneCheckedOut1 || CC[2]@doneCheckedOut2)
#define noNotified0   (! CC[2]@doneNotify0)

/* ================================= */
/* LTL formula for 'Update if registered*/

[]!(registered0 && ((! unRegistered0) U (or && []noUpdate0)))

/* with the following macro definitions added in the model */

/* #define registered0   (CC[2]@doneGet0 || CC[2]@doneRegister0)*/
/* #define unRegistered0   (CC[2]@doneUnRegister0) */
#define or   (CC[2]@doneUnCheckedOut1 || CC[2]@doneUnCheckedOut2 || \
CC[2]@doneCheckedIn1 || CC[2]@doneCheckedIn2 || \
```

```
CC[2]@doneCheckedInOut1 || CC[2]@doneCheckedInOut2)
#define noUpdate0   (! (CC[2]@doneUpdate0 || CC[2]@doneIOUpdate0))
```

B. PEPA specification of retry-based approach with three users

```
lambda = 1.0;     % request rate (user/hour)
mu = 5.0;         % edit rate (user/hour)
theta = 5.0;      % retry rate (user/hour)

#User = (cO_s,lambda).User1 + (cO_f,lambda).User2;   % request mode
#User1 = (cI,mu).User;                              % file in possession
#User2 = (r_s,theta).User1 + (r_f,theta).User2;      % retry mode

#CheckOut = (cO_s,infty).CheckOut1;
#CheckOut1 = (cI,infty).CheckOut + (cO_f,infty).CheckOut1;

(User <> User <> User) <cO_s,cO_f,cI> CheckOut
```

C. PRISM Specification of retry-based approach with 3 users

In this appendix the PRISM Specification of thinkteam with 3 Users discussed in Section 5.1 is given (Appendix C.1), together with the CSL formulae in the format accepted by PRISM (Appendix C.2).

C.1. PRISM *model*

```
ctmc

const double lambda = 1.0;
const double mu = 5.0;
const double theta = 5.0;
const int User = 0;
const int User1 = 1;
const int User2 = 2;
const int CheckOut = 0;
const int CheckOut1 = 1;

module User

        User_STATE : [0..2] init User;

        [cO_s] (User_STATE=User) -> lambda : (User_STATE'=User1);
        [cO_f] (User_STATE=User) -> lambda : (User_STATE'=User2);
        [cI] (User_STATE=User1) -> mu : (User_STATE'=User);
        [cO_s] (User_STATE=User2) -> theta : (User_STATE'=User1);
        [cO_f] (User_STATE=User2) -> theta : (User_STATE'=User2);

endmodule

module User_2

        User_2_STATE : [0..2] init User;

        [cO_s] (User_2_STATE=User) -> lambda : (User_2_STATE'=User1);
        [cO_f] (User_2_STATE=User) -> lambda : (User_2_STATE'=User2);
        [cI] (User_2_STATE=User1) -> mu : (User_2_STATE'=User);
        [cO_s] (User_2_STATE=User2) -> theta : (User_2_STATE'=User1);
        [cO_f] (User_2_STATE=User2) -> theta : (User_2_STATE'=User2);
```

```
endmodule

module User_3

        User_3_STATE : [0..2] init User;

        [cO_s] (User_3_STATE=User) -> lambda : (User_3_STATE'=User1);
        [cO_f] (User_3_STATE=User) -> lambda : (User_3_STATE'=User2);
        [cI] (User_3_STATE=User1) -> mu : (User_3_STATE'=User);
        [cO_s] (User_3_STATE=User2) -> theta : (User_3_STATE'=User1);
        [cO_f] (User_3_STATE=User2) -> theta : (User_3_STATE'=User2);

endmodule

module CheckOut

        CheckOut_STATE : [0..1] init CheckOut;

        [cO_s] (CheckOut_STATE=CheckOut) -> 1 : (CheckOut_STATE'=CheckOut1);
        [cI] (CheckOut_STATE=CheckOut1) -> 1 : (CheckOut_STATE'=CheckOut);
        [cO_f] (CheckOut_STATE=CheckOut1) -> 1 : (CheckOut_STATE'=CheckOut1);

endmodule

system
       ((User ||| (User_2 ||| User_3)) |[cO_s,cO_f,cI]| CheckOut)
endsystem
```

C.2. PRISM *CSL formulae*

```
%Formula P0
P=?([true U<=5 User_STATE=User1 {User_STATE =User2}])

%Formula for steady-state probability of the user being in retry mode
S=?([User_STATE=User2])

%Formula for steady-state probability of the user being in request mode
S=?([User_STATE=User])
```

D. PEPA specification of waiting-list approach with three Users

```
lambda = 1.0;    % request rate
mu = 5.0;        % edit rate

#User_0 = (cO_0,lambda).User_0a;
#User_0a = (cI_0,mu).User_0;
#User_1 = (cO_1,lambda).User_1a;
#User_1a = (cI_1,mu).User_1;
#User_2 = (cO_2,lambda).User_2a;
#User_2a = (cI_2,mu).User_2;

#FIFO = (cO_0,infty).F_0 + (cO_1,infty).F_1 + (cO_2,infty).F_2;
#F_0 = (cI_0,infty).FIFO + (cO_1,infty).F_01 + (cO_2,infty).F_02;
#F_1 = (cI_1,infty).FIFO + (cO_0,infty).F_10 + (cO_2,infty).F_12;
#F_2 = (cI_2,infty).FIFO + (cO_0,infty).F_20 + (cO_1,infty).F_21;
#F_01 = (cI_0,infty).F_1 + (cO_2,infty).F_012;
```

```
#F_02 = (cI_0,infty).F_2 + (cO_1,infty).F_021;
#F_10 = (cI_1,infty).F_0 + (cO_2,infty).F_102;
#F_12 = (cI_1,infty).F_2 + (cO_0,infty).F_120;
#F_20 = (cI_2,infty).F_0 + (cO_1,infty).F_201;
#F_21 = (cI_2,infty).F_1 + (cO_0,infty).F_210;
#F_012 = (cI_0,infty).F_12;
#F_021 = (cI_0,infty).F_21;
#F_102 = (cI_1,infty).F_02;
#F_120 = (cI_1,infty).F_20;
#F_201 = (cI_2,infty).F_01;
#F_210 = (cI_2,infty).F_10;

(User_0 <> User_1 <> User_2) <cO_0,cO_1,cO_2,cI_0,cI_1,cI_2> FIFO
```

E. PRISM Specification of Waiting-list Approach with 3 Users

```
ctmc

const double lambda = 1.0;
const double mu = 5.0;
const int User_0 = 0;
const int User_0a = 1;
const int User_1 = 0;
const int User_1a = 1;
const int User_2 = 0;
const int User_2a = 1;
const int FIFO_empty = 0;
const int F_0 = 1;
const int F_01 = 2;
const int F_012 = 3;
const int F_02 = 4;
const int F_021 = 5;
const int F_1 = 6;
const int F_10 = 7;
const int F_102 = 8;
const int F_12 = 9;
const int F_120 = 10;
const int F_2 = 11;
const int F_20 = 12;
const int F_201 = 13;
const int F_21 = 14;
const int F_210 = 15;

module User_0

        User_0_STATE : [0..1] init User_0;

        [cO_0] (User_0_STATE=User_0) -> lambda : (User_0_STATE'=User_0a);
        [cI_0] (User_0_STATE=User_0a) -> mu : (User_0_STATE'=User_0);

endmodule

module User_1

        User_1_STATE : [0..1] init User_1;

        [cO_1] (User_1_STATE=User_1) -> lambda : (User_1_STATE'=User_1a);
        [cI_1] (User_1_STATE=User_1a) -> mu : (User_1_STATE'=User_1);
```

```
endmodule

module User_2

        User_2_STATE : [0..1] init User_2;

        [c0_2] (User_2_STATE=User_2) -> lambda : (User_2_STATE'=User_2a);
        [cI_2] (User_2_STATE=User_2a) -> mu : (User_2_STATE'=User_2);

endmodule

module FIFO_empty

        FIFO_empty_STATE : [0..15] init FIFO_empty;

        [c0_0] (FIFO_empty_STATE=FIFO_empty) -> 1 : (FIFO_empty_STATE'=F_0);
        [c0_1] (FIFO_empty_STATE=FIFO_empty) -> 1 : (FIFO_empty_STATE'=F_1);
        [c0_2] (FIFO_empty_STATE=FIFO_empty) -> 1 : (FIFO_empty_STATE'=F_2);
        [cI_0] (FIFO_empty_STATE=F_0) -> 1 : (FIFO_empty_STATE'=FIFO_empty);
        [c0_1] (FIFO_empty_STATE=F_0) -> 1 : (FIFO_empty_STATE'=F_01);
        [c0_2] (FIFO_empty_STATE=F_0) -> 1 : (FIFO_empty_STATE'=F_02);
        [cI_0] (FIFO_empty_STATE=F_01) -> 1 : (FIFO_empty_STATE'=F_1);
        [c0_2] (FIFO_empty_STATE=F_01) -> 1 : (FIFO_empty_STATE'=F_012);
        [cI_0] (FIFO_empty_STATE=F_012) -> 1 : (FIFO_empty_STATE'=F_12);
        [cI_0] (FIFO_empty_STATE=F_02) -> 1 : (FIFO_empty_STATE'=F_2);
        [c0_1] (FIFO_empty_STATE=F_02) -> 1 : (FIFO_empty_STATE'=F_021);
        [cI_0] (FIFO_empty_STATE=F_021) -> 1 : (FIFO_empty_STATE'=F_21);
        [cI_1] (FIFO_empty_STATE=F_1) -> 1 : (FIFO_empty_STATE'=FIFO_empty);
        [c0_0] (FIFO_empty_STATE=F_1) -> 1 : (FIFO_empty_STATE'=F_10);
        [c0_2] (FIFO_empty_STATE=F_1) -> 1 : (FIFO_empty_STATE'=F_12);
        [cI_1] (FIFO_empty_STATE=F_10) -> 1 : (FIFO_empty_STATE'=F_0);
        [c0_2] (FIFO_empty_STATE=F_10) -> 1 : (FIFO_empty_STATE'=F_102);
        [cI_1] (FIFO_empty_STATE=F_102) -> 1 : (FIFO_empty_STATE'=F_02);
        [cI_1] (FIFO_empty_STATE=F_12) -> 1 : (FIFO_empty_STATE'=F_2);
        [c0_0] (FIFO_empty_STATE=F_12) -> 1 : (FIFO_empty_STATE'=F_120);
        [cI_1] (FIFO_empty_STATE=F_120) -> 1 : (FIFO_empty_STATE'=F_20);
        [cI_2] (FIFO_empty_STATE=F_2) -> 1 : (FIFO_empty_STATE'=FIFO_empty);
        [c0_0] (FIFO_empty_STATE=F_2) -> 1 : (FIFO_empty_STATE'=F_20);
        [c0_1] (FIFO_empty_STATE=F_2) -> 1 : (FIFO_empty_STATE'=F_21);
        [cI_2] (FIFO_empty_STATE=F_20) -> 1 : (FIFO_empty_STATE'=F_0);
        [c0_1] (FIFO_empty_STATE=F_20) -> 1 : (FIFO_empty_STATE'=F_201);
        [cI_2] (FIFO_empty_STATE=F_201) -> 1 : (FIFO_empty_STATE'=F_01);
        [cI_2] (FIFO_empty_STATE=F_21) -> 1 : (FIFO_empty_STATE'=F_1);
        [c0_0] (FIFO_empty_STATE=F_21) -> 1 : (FIFO_empty_STATE'=F_210);
        [cI_2] (FIFO_empty_STATE=F_210) -> 1 : (FIFO_empty_STATE'=F_10);

endmodule

system
       ((User_0 ||| (User_1 ||| User_2)) |[c0_0,c0_1,c0_2,cI_0,cI_1,cI_2]| FIFO_empty)
endsystem
```

F. PEPA specification of thinkteam with three users and three vaults

```
lambda = 0.1;         % request rate (client/hour)
gamma_a_A = 60.0;     % download rate (file/hour) between Vault A & Client A
gamma_a_B = 20.0;     % download rate (file/hour) between Vault A & Client B
gamma_a_C = 40.0;     % download rate (file/hour) between Vault A & Client C
gamma_b_A = 40.0;     % download rate (file/hour) between Vault B & Client A
```

```
gamma_b_B = 60.0;     % download rate (file/hour) between Vault B & Client B
gamma_b_C = 20.0;     % download rate (file/hour) between Vault B & Client C
gamma_c_A = 20.0;     % download rate (file/hour) between Vault C & Client A
gamma_c_B = 40.0;     % download rate (file/hour) between Vault C & Client B
gamma_c_C = 60.0;     % download rate (file/hour) between Vault C & Client C
mu_a_A = 60.0;        % upload rate (file/hour) between Vault A & Client A
mu_a_B = 20.0;        % upload rate (file/hour) between Vault A & Client B
mu_a_C = 40.0;        % upload rate (file/hour) between Vault A & Client C
mu_b_A = 40.0;        % upload rate (file/hour) between Vault B & Client A
mu_b_B = 60.0;        % upload rate (file/hour) between Vault B & Client B
mu_b_C = 20.0;        % upload rate (file/hour) between Vault B & Client C
mu_c_A = 20.0;        % upload rate (file/hour) between Vault C & Client A
mu_c_B = 40.0;        % upload rate (file/hour) between Vault C & Client B
mu_c_C = 60.0;        % upload rate (file/hour) between Vault C & Client C
nu_A = 0.3;           % edit rate (file/hour) of Client A
nu_B = 0.3;           % edit rate (file/hour) of Client B
nu_C = 0.3;           % edit rate (file/hour) of Client C
theta = 6.0;          % retry rate (client/hour)
delta = 100;          % signal rate (message/hour)
epsilon = 100;        % signal rate (message/hour)
epsilon2 = 200;       % (a trick to raise the probability of
epsilon3 = 300;       %  specific "nondeterministic" choices)

% Client A (Client B and Client C are analogous):
#CA = (CA_TT_cO_s,lambda).CA_1        % successful request for checkOut
    + (CA_TT_cO_f,lambda).CA_2;       % failed request for checkOut
#CA_1 = (TT_CA_Va,infty).CA_3_1       % TT assigns Vault A for checkOut
      + (TT_CA_Vb,infty).CA_3_2       % TT assigns Vault B for checkOut
      + (TT_CA_Vc,infty).CA_3_3;      % TT assigns Vault C for checkOut
#CA_3_1 = (Va_CA,infty).CA_4;         % checkOut file from Vault A
#CA_3_2 = (Vb_CA,infty).CA_4;         % checkOut file from Vault B
#CA_3_3 = (Vc_CA,infty).CA_4;         % checkOut file from Vault C
#CA_4 = (CA_CA,nu_A).CA_5;            % edit file
#CA_5 = (CA_Va_cI,infty).CA           % checkIn file in Vault A
      + (CA_Vb_cI,infty).CA           % checkIn file in Vault B
      + (CA_Vc_cI,infty).CA;          % checkIn file in Vault C
#CA_2 = (CA_TT_cO_s,theta).CA_1       % successful retry of checkOut
      + (CA_TT_cO_f,theta).CA_2;      % failed retry of checkOut

% Vault A (Vault B and Vault C are analogous):
#Va = (Va_CA,gamma_a_A).Va            % file checkOut by Client A
    + (Va_CB,gamma_a_B).Va            % file checkOut by Client B
    + (Va_CC,gamma_a_C).Va            % file checkOut by Client C
    + (CA_Va_cI,mu_a_A).Va_1          % file checkIn by Client A
    + (CB_Va_cI,mu_a_B).Va_1          % file checkIn by Client B
    + (CC_Va_cI,mu_a_C).Va_1;         % file checkIn by Client C
#Va_1 = (Va_TT,delta).Va;             % inform TT of file checkIn

% thinkteam:
#TT = (CA_TT_cO_s,infty).TT_1_1       % grant checkOut to Client A
    + (CB_TT_cO_s,infty).TT_1_2       % grant checkOut to Client B
    + (CC_TT_cO_s,infty).TT_1_3;      % grant checkOut to Client C
#TT_1_1 = (TT_CA_Va,epsilon3).TT_2    % assign Vault A to Client A
        + (TT_CA_Vb,epsilon2).TT_2    % assign Vault B to Client A
        + (TT_CA_Vc,epsilon).TT_2;    % assign Vault C to Client A
#TT_1_2 = (TT_CB_Va,epsilon).TT_2     % assign Vault A to Client B
        + (TT_CB_Vb,epsilon3).TT_2    % assign Vault B to Client B
        + (TT_CB_Vc,epsilon2).TT_2;   % assign Vault C to Client B
#TT_1_3 = (TT_CC_Va,epsilon2).TT_2    % assign Vault A to Client C
        + (TT_CC_Vb,epsilon).TT_2     % assign Vault B to Client C
```

```
        + (TT_CC_Vc,epsilon3).TT_2;   % assign Vault C to Client C
#TT_2 = (Va_TT,infty).TT              % Vault A signals file checkIn
      + (Vb_TT,infty).TT              % Vault B signals file checkIn
      + (Vc_TT,infty).TT              % Vault C signals file checkIn
      + (CA_TT_cO_f,infty).TT_2       % deny checkOut to Client A
      + (CB_TT_cO_f,infty).TT_2       % deny checkOut to Client B
      + (CC_TT_cO_f,infty).TT_2;      % deny checkOut to Client C

% full specification:
(CA <> CB <> CC) <CA_TT_cO_s,CB_TT_cO_s,CC_TT_cO_s,CA_TT_cO_f,CB_TT_cO_f,CC_TT_cO_f,
TT_CA_Va,TT_CA_Vb,TT_CA_Vc,TT_CB_Va,TT_CB_Vb,TT_CB_Vc,TT_CC_Va,TT_CC_Vb,TT_CC_Vc,
CA_Va_cI,CA_Vb_cI,CA_Vc_cI,CB_Va_cI,CB_Vb_cI,CB_Vc_cI,CC_Va_cI,CC_Vb_cI,CC_Vc_cI,
Va_CA,Va_CB,Va_CC,Vb_CA,Vb_CB,Vb_CC,Vc_CA,Vc_CB,Vc_CC> (TT <Va_TT,Vb_TT,Vc_TT> (Va <> Vb <> Vc))
```

G. PRISM **Specification of thinkteam with 3 Users and 3 Vaults**

In this appendix the PRISM Specification of thinkteam with 3 Users and 3 Vaults discussed in Section 6.2 is given (Appendix G.1), together with the CSL formulae in the format accepted by PRISM (Appendix G.2).

G.1. PRISM *model*

```
ctmc

const double lambda = 0.1;
const double gamma_a_A = 60.0;
const double gamma_a_B = 20.0;
const double gamma_a_C = 40.0;
const double gamma_b_A = 40.0;
const double gamma_b_B = 60.0;
const double gamma_b_C = 20.0;
const double gamma_c_A = 20.0;
const double gamma_c_B = 40.0;
const double gamma_c_C = 60.0;
const double mu_a_A = 60.0;
const double mu_a_B = 20.0;
const double mu_a_C = 40.0;
const double mu_b_A = 40.0;
const double mu_b_B = 60.0;
const double mu_b_C = 20.0;
const double mu_c_A = 20.0;
const double mu_c_B = 40.0;
const double mu_c_C = 60.0;
const double nu_A = 0.3;
const double nu_B = 0.3;
const double nu_C = 0.3;
const double theta = 6.0;
const double delta = 100;
const double epsilon = 100;
const double epsilon2 = 200;
const double epsilon3 = 300;
const int CA = 0;
const int CA_1 = 1;
const int CA_2 = 2;
const int CA_3_1 = 3;
const int CA_3_2 = 4;
const int CA_3_3 = 5;
const int CA_4 = 6;
const int CA_5 = 7;
const int CB = 0;
```

```
const int CB_1 = 1;
const int CB_2 = 2;
const int CB_3_1 = 3;
const int CB_3_2 = 4;
const int CB_3_3 = 5;
const int CB_4 = 6;
const int CB_5 = 7;
const int CC = 0;
const int CC_1 = 1;
const int CC_2 = 2;
const int CC_3_1 = 3;
const int CC_3_2 = 4;
const int CC_3_3 = 5;
const int CC_4 = 6;
const int CC_5 = 7;
const int TT = 0;
const int TT_1_1 = 1;
const int TT_1_2 = 2;
const int TT_1_3 = 3;
const int TT_2 = 4;
const int Va = 0;
const int Va_1 = 1;
const int Vb = 0;
const int Vb_1 = 1;
const int Vc = 0;
const int Vc_1 = 1;

module CA

        CA_STATE : [0..7] init CA;

        [CA_TT_c0_s] (CA_STATE=CA) -> lambda : (CA_STATE'=CA_1);
        [CA_TT_c0_f] (CA_STATE=CA) -> lambda : (CA_STATE'=CA_2);
        [TT_CA_Va] (CA_STATE=CA_1) -> 1 : (CA_STATE'=CA_3_1);
        [TT_CA_Vb] (CA_STATE=CA_1) -> 1 : (CA_STATE'=CA_3_2);
        [TT_CA_Vc] (CA_STATE=CA_1) -> 1 : (CA_STATE'=CA_3_3);
        [CA_TT_c0_s] (CA_STATE=CA_2) -> theta : (CA_STATE'=CA_1);
        [CA_TT_c0_f] (CA_STATE=CA_2) -> theta : (CA_STATE'=CA_2);
        [Va_CA] (CA_STATE=CA_3_1) -> 1 : (CA_STATE'=CA_4);
        [Vb_CA] (CA_STATE=CA_3_2) -> 1 : (CA_STATE'=CA_4);
        [Vc_CA] (CA_STATE=CA_3_3) -> 1 : (CA_STATE'=CA_4);
        [CA_CA] (CA_STATE=CA_4) -> nu_A : (CA_STATE'=CA_5);
        [CA_Va_cI] (CA_STATE=CA_5) -> 1 : (CA_STATE'=CA);
        [CA_Vb_cI] (CA_STATE=CA_5) -> 1 : (CA_STATE'=CA);
        [CA_Vc_cI] (CA_STATE=CA_5) -> 1 : (CA_STATE'=CA);

endmodule

module CB

        CB_STATE : [0..7] init CB;

        [CB_TT_c0_s] (CB_STATE=CB) -> lambda : (CB_STATE'=CB_1);
        [CB_TT_c0_f] (CB_STATE=CB) -> lambda : (CB_STATE'=CB_2);
        [TT_CB_Va] (CB_STATE=CB_1) -> 1 : (CB_STATE'=CB_3_1);
        [TT_CB_Vb] (CB_STATE=CB_1) -> 1 : (CB_STATE'=CB_3_2);
        [TT_CB_Vc] (CB_STATE=CB_1) -> 1 : (CB_STATE'=CB_3_3);
        [CB_TT_c0_s] (CB_STATE=CB_2) -> theta : (CB_STATE'=CB_1);
        [CB_TT_c0_f] (CB_STATE=CB_2) -> theta : (CB_STATE'=CB_2);
        [Va_CB] (CB_STATE=CB_3_1) -> 1 : (CB_STATE'=CB_4);
        [Vb_CB] (CB_STATE=CB_3_2) -> 1 : (CB_STATE'=CB_4);
```

```
        [Vc_CB] (CB_STATE=CB_3_3) -> 1 : (CB_STATE'=CB_4);
        [CB_CB] (CB_STATE=CB_4) -> nu_B : (CB_STATE'=CB_5);
        [CB_Va_cI] (CB_STATE=CB_5) -> 1 : (CB_STATE'=CB);
        [CB_Vb_cI] (CB_STATE=CB_5) -> 1 : (CB_STATE'=CB);
        [CB_Vc_cI] (CB_STATE=CB_5) -> 1 : (CB_STATE'=CB);

endmodule

module CC

        CC_STATE : [0..7] init CC;

        [CC_TT_c0_s] (CC_STATE=CC) -> lambda : (CC_STATE'=CC_1);
        [CC_TT_c0_f] (CC_STATE=CC) -> lambda : (CC_STATE'=CC_2);
        [TT_CC_Va] (CC_STATE=CC_1) -> 1 : (CC_STATE'=CC_3_1);
        [TT_CC_Vb] (CC_STATE=CC_1) -> 1 : (CC_STATE'=CC_3_2);
        [TT_CC_Vc] (CC_STATE=CC_1) -> 1 : (CC_STATE'=CC_3_3);
        [CC_TT_c0_s] (CC_STATE=CC_2) -> theta : (CC_STATE'=CC_1);
        [CC_TT_c0_f] (CC_STATE=CC_2) -> theta : (CC_STATE'=CC_2);
        [Va_CC] (CC_STATE=CC_3_1) -> 1 : (CC_STATE'=CC_4);
        [Vb_CC] (CC_STATE=CC_3_2) -> 1 : (CC_STATE'=CC_4);
        [Vc_CC] (CC_STATE=CC_3_3) -> 1 : (CC_STATE'=CC_4);
        [CC_CC] (CC_STATE=CC_4) -> nu_C : (CC_STATE'=CC_5);
        [CC_Va_cI] (CC_STATE=CC_5) -> 1 : (CC_STATE'=CC);
        [CC_Vb_cI] (CC_STATE=CC_5) -> 1 : (CC_STATE'=CC);
        [CC_Vc_cI] (CC_STATE=CC_5) -> 1 : (CC_STATE'=CC);

endmodule

module TT

        TT_STATE : [0..4] init TT;

        [CA_TT_c0_s] (TT_STATE=TT) -> 1 : (TT_STATE'=TT_1_1);
        [CB_TT_c0_s] (TT_STATE=TT) -> 1 : (TT_STATE'=TT_1_2);
        [CC_TT_c0_s] (TT_STATE=TT) -> 1 : (TT_STATE'=TT_1_3);
        [TT_CA_Va] (TT_STATE=TT_1_1) -> epsilon3 : (TT_STATE'=TT_2);
        [TT_CA_Vb] (TT_STATE=TT_1_1) -> epsilon2 : (TT_STATE'=TT_2);
        [TT_CA_Vc] (TT_STATE=TT_1_1) -> epsilon : (TT_STATE'=TT_2);
        [TT_CB_Va] (TT_STATE=TT_1_2) -> epsilon : (TT_STATE'=TT_2);
        [TT_CB_Vb] (TT_STATE=TT_1_2) -> epsilon3 : (TT_STATE'=TT_2);
        [TT_CB_Vc] (TT_STATE=TT_1_2) -> epsilon2 : (TT_STATE'=TT_2);
        [TT_CC_Va] (TT_STATE=TT_1_3) -> epsilon2 : (TT_STATE'=TT_2);
        [TT_CC_Vb] (TT_STATE=TT_1_3) -> epsilon : (TT_STATE'=TT_2);
        [TT_CC_Vc] (TT_STATE=TT_1_3) -> epsilon3 : (TT_STATE'=TT_2);
        [Va_TT] (TT_STATE=TT_2) -> 1 : (TT_STATE'=TT);
        [Vb_TT] (TT_STATE=TT_2) -> 1 : (TT_STATE'=TT);
        [Vc_TT] (TT_STATE=TT_2) -> 1 : (TT_STATE'=TT);
        [CA_TT_c0_f] (TT_STATE=TT_2) -> 1 : (TT_STATE'=TT_2);
        [CB_TT_c0_f] (TT_STATE=TT_2) -> 1 : (TT_STATE'=TT_2);
        [CC_TT_c0_f] (TT_STATE=TT_2) -> 1 : (TT_STATE'=TT_2);

endmodule

module Va

        Va_STATE : [0..1] init Va;

        [Va_CA] (Va_STATE=Va) -> gamma_a_A : (Va_STATE'=Va);
```

```
        [Va_CB] (Va_STATE=Va) -> gamma_a_B : (Va_STATE'=Va);
        [Va_CC] (Va_STATE=Va) -> gamma_a_C : (Va_STATE'=Va);
        [CA_Va_cI] (Va_STATE=Va) -> mu_a_A : (Va_STATE'=Va_1);
        [CB_Va_cI] (Va_STATE=Va) -> mu_a_B : (Va_STATE'=Va_1);
        [CC_Va_cI] (Va_STATE=Va) -> mu_a_C : (Va_STATE'=Va_1);
        [Va_TT] (Va_STATE=Va_1) -> delta : (Va_STATE'=Va);

endmodule

module Vb

        Vb_STATE : [0..1] init Vb;

        [Vb_CA] (Vb_STATE=Vb) -> gamma_b_A : (Vb_STATE'=Vb);
        [Vb_CB] (Vb_STATE=Vb) -> gamma_b_B : (Vb_STATE'=Vb);
        [Vb_CC] (Vb_STATE=Vb) -> gamma_b_C : (Vb_STATE'=Vb);
        [CA_Vb_cI] (Vb_STATE=Vb) -> mu_b_A : (Vb_STATE'=Vb_1);
        [CB_Vb_cI] (Vb_STATE=Vb) -> mu_b_B : (Vb_STATE'=Vb_1);
        [CC_Vb_cI] (Vb_STATE=Vb) -> mu_b_C : (Vb_STATE'=Vb_1);
        [Vb_TT] (Vb_STATE=Vb_1) -> delta : (Vb_STATE'=Vb);

endmodule

module Vc

        Vc_STATE : [0..1] init Vc;

        [Vc_CA] (Vc_STATE=Vc) -> gamma_c_A : (Vc_STATE'=Vc);
        [Vc_CB] (Vc_STATE=Vc) -> gamma_c_B : (Vc_STATE'=Vc);
        [Vc_CC] (Vc_STATE=Vc) -> gamma_c_C : (Vc_STATE'=Vc);
        [CA_Vc_cI] (Vc_STATE=Vc) -> mu_c_A : (Vc_STATE'=Vc_1);
        [CB_Vc_cI] (Vc_STATE=Vc) -> mu_c_B : (Vc_STATE'=Vc_1);
        [CC_Vc_cI] (Vc_STATE=Vc) -> mu_c_C : (Vc_STATE'=Vc_1);
        [Vc_TT] (Vc_STATE=Vc_1) -> delta : (Vc_STATE'=Vc);

endmodule
```

G.2. PRISM *CSL formulae*

```
%Formula P1
P>=1([true U CA_STATE=CA_5 {CA_STATE=CA_3_1}])

%Formula P2
P<=0([true U ((CA_STATE=CA_1) & (CB_STATE=CB_1)) |
                            ((CA_STATE=CA_1) & (CC_STATE=CC_1)) |
                            ((CB_STATE=CB_1) & (CC_STATE=CC_1))
          ])

%Formula P3
P=?([true U<=5 CA_STATE=CA_5 {CA_STATE=CA_3_1}])

%Formula P4
S=?([CA_STATE=CA_2 ])
```

H. Additional Reward Structures of PRISM Specification

```
rewards "NrSuccessfulRequestsClientA"

        [CA_TT_cO_s] true : 1;
```

```
endrewards

rewards "NrFailedRequestsClientA"

        [CA_TT_cO_f] true : 1;

endrewards
```

References

[1] J. Stark, Product Lifecycle Management—21st Century Paradigm for Product Realisation, Springer-Verlag, Berlin, 2005.
[2] J. Grudin, CSCW—history and focus, IEEE Comput. 27 (5) (1994) 19–26.
[3] C.A. Ellis, S.J. Gibbs, G.L. Rein, Groupware—some issues and experiences, Commun. ACM 34 (1) (1991) 38–58.
[4] P. Dourish, V. Bellotti, Awareness and coordination in shared workspaces, in: J. Turner, R. Kraut (Eds.), Proceedings of the 3rd ACM Conference on Computer Supported Cooperative Work (CSCW'92), Toronto, Canada, ACM Press, New York, 1992, pp. 107–114.
[5] C. Gutwin, M. Roseman, S. Greenberg, Supporting awareness of others in groupware, in: M.J. Tauber (Ed.), Companion Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Common Ground (CHI'96), Vancouver, BC, Canada, ACM Press, New York, 1996, pp. 205–215.
[6] M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, Model checking groupware protocols, in: F. Darses, R. Dieng, C. Simone, M. Zacklad (Eds.), Cooperative Systems Design—Scenario-Based Design of Collaborative Systems, Frontiers in Artificial Intelligence and Applications, vol. 107, IOS Press, Amsterdam, 2004, pp. 179–194.
[7] C. Papadopoulos, An extended temporal logic for CSCW, Comput. J. 45 (4) (2002) 453–472.
[8] T. Urnes, Efficiently implementing synchronous groupware, Ph.D. Thesis, Department of Computer Science, York University, Toronto, 1998.
[9] M. Caporuscio, P. Inverardi, P. Pelliccione, Formal analysis of clients mobility in the Siena publish/subscribe middleware, Tech. rep., Department of Computer Science, University of L'Aquila, 2002.
[10] X. Deng, M.B. Dwyer, J. Hatcliff, G. Jung, Robby, G. Singh, Model-checking middleware-based event-driven real-time embedded software, in: F.S. de Boer, M.M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Revised Lectures of the 1st International Symposium on Formal Methods for Components and Objects (FMCO'02), Leiden, The Netherlands, Lecture Notes in Computer Science, vol. 2852, Springer-Verlag, Berlin, 2003, pp. 154–181.
[11] D. Garlan, S. Khersonsky, J.S. Kim, Model checking publish-subscribe systems, in: T. Ball, S.K. Rajamani (Eds.), Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN'03), Portland, OR, USA, Lecture Notes in Computer Science, vol. 2648, Springer-Verlag, Berlin, 2003, pp. 166–180.
[12] S. Tripakis, S. Yovine, Timing analysis and code generation of vehicle control software using taxys, in: K. Havelund, G. Rosu (Eds.), Proceedings of the 1st Workshop on Runtime Verification (RV'01), Paris, France, Electronic Notes in Theoretical Computer Science, vol. 55(2), Elsevier Science Publishers, Amsterdam, 2001, pp. 174–183.
[13] L. Zanolin, C. Ghezzi, L. Baresi, An approach to model and validate publish/subscribe architectures, in: M. Barnett, S.H. Edwards, D. Giannakopoulou, G.T. Leavens (Eds.), Proceedings of the Workshop on Specification and Verification of Component-Based Systems (SAVCBS'03), Helsinki, Finland, Tech. Rep. #03-11, Department of Computer Science, Iowa State University, 2003, pp. 35–41.
[14] D. Wells, Extreme programming: a gentle introduction, 2006. online: <http://www.extremeprogramming.org>.
[15] T.C.N. Graham, T. Urnes, R. Nejabi, Efficient distributed implementation of semi-replicated synchronous groupware, in: M. Brown, R. Rao (Eds.), Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'96), Seattle, WA, ACM Press, New York, 1996, pp. 1–10.
[16] T. Urnes, T.C.N. Graham, Flexibly mapping synchronous groupware architectures to distributed implementations, in: D.J. Duke, A.R. Puerta (Eds.), Proceedings of the 6th International Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSVIS'99), Braga, Portugal, Springer-Verlag, Berlin, 1999, pp. 133–148.
[17] G.J. Holzmann, The SPIN Model Checker—Primer and Reference Manual, Addison Wesley, Reading, 2003.
[18] M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, M. Sebastianis, Model checking publish/subscribe notification for `thinkteam`, in: A. Arenas, J. Bicarregui, A. Butterfield (Eds.), Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04), Linz, Austria, Electronic Notes in Theoretical Computer Science, vol. 133, Elsevier Science Publishers, Amsterdam, 2005, pp. 275–294.
[19] M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, M. Sebastianis, A case study on the automated verification of groupware protocols, in: C. Heitmeyer, K. Pohl (Eds.), Proceedings of the 27th International Conference on Software Engineering (ICSE'05)—Experience Reports Track, St. Louis, MO, USA, ACM Press, New York, 2005, pp. 596–603.
[20] P.T. Eugster, P. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surveys 35 (2) (2003) 114–131.
[21] M. Tivoli, P. Inverardi, V. Presutti, A. Forghieri, M. Sebastianis, Correct components assembly for a product data management cooperative system, in: I. Crnkovic, J.A. Stafford, H.W. Schmidt, K.C. Wallnau (Eds.), Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE'04), Edinburgh, UK, Lecture Notes in Computer Science, vol. 3054, Springer-Verlag, Berlin, 2004, pp. 84–99.
[22] M.H. ter Beek, M. Massink, D. Latella, Towards model checking stochastic aspects of the `thinkteam` user interface, in: S.W. Gilroy, M.D. Harrison (Eds.), Interactive Systems: Design, Specification, and Verification—Revised papers of the 12th International Workshop on Design Specification and Verification of Interactive Systems (DSVIS'05), Newcastle upon Tyne, UK, Lecture Notes in Computer Science, vol. 3941, Springer-Verlag, Berlin, 2006, pp. 39–50.
[23] P. Buchholz, J.-P. Katoen, P. Kemper, C. Tepper, Model-checking large structured Markov chains, J. Log. Algebr. Program. 56 (2003) 69–96.
[24] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, M. Siegle, A tool for model-checking Markov chains, Internat. J. Software Tools Technol. Transfer 4 (2) (2003) 153–172.
[25] M. Kwiatkowska, G. Norman, D. Parker, Probabilistic symbolic model checking with PRISM: a hybrid approach, in: J.-P. Katoen, P. Stevens (Eds.), Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Grenoble, France, Lecture Notes in Computer Science, vol. 2280, Springer-Verlag, Berlin, 2002, pp. 52–66.
[26] H.L.S. Younes, R.G. Simmons, Probabilistic verification of discrete event systems using acceptance sampling, in: E. Brinksma, K.G. Larsen (Eds.), Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02), Copenhagen, Denmark, Lecture Notes in Computer Science, vol. 2404, Springer-Verlag, Berlin, 2002, pp. 223–235.
[27] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, W. Wilner, The rendezvous architecture and language for constructing multi-user applications, ACM Trans. Comput.–Human Interact. 1 (2) (1994) 81–125.
[28] M. Roseman, S. Greenberg, Building real time groupware with groupKit, a groupware toolkit, ACM Trans. Comput.–Human Interact. 3 (1) (1996) 66–106.
[29] G.E. Krasner, S.T. Pope, A cookbook for using the model-view-controller user interface paradigm in smalltalk-80, J. Object-Oriented Program. 1 (3) (1988) 26–49.
[30] R. Levesque, SPSS Inc., SPSS Programming and Data Management: A Guide for SPSS and SAS Users (fourth ed.), SPSS Inc., Chicago, 2007. online: <http://www.spss.com>.

[31] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, Massachusetts, 1999.
[32] E.A. Emerson, Temporal and modal logics, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Formal Models and Semantics, vol. B, Elsevier Science Publishers, Amsterdam, 1990, pp. 995–1072.
[33] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems—Specification, Springer-Verlag, Berlin, 1992.
[34] T. Han, J.-P. Katoen, Providing evidence of likely being on time: counterexample generation for CTMC model checking, in: K.S. Namjoshi, T. Yoneda, T. Higashino, Y. Okamura (Eds.), Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07), Tokyo, Japan, Lecture Notes in Computer Science, vol. 4762, Springer-Verlag, Berlin, 2007, pp. 331–346.
[35] D. Parker, G. Norman, M. Kwiatkowska, PRISM 2.0—Users' Guide, February 2004. online: <http://www.cs.bham.ac.uk/dxp/prism>.
[36] V. Kulkarni, Modeling and Analysis of Stochastic Systems, Chapman & Hall, London, 1995.
[37] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, Cambridge, 1996.
[38] R. Alur, T. Henzinger, Reactive modules, Formal Methods in System Design 15 (1) (1999) 7–48.
[39] A. Aziz, K. Sanwal, V. Singhal, R. Brayton, Model checking continuous time Markov chains, ACM Trans. Comput. Log. 1 (1) (2000) 162–170.
[40] C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, Model-checking algorithms for continuous-time Markov chains, IEEE Trans. Software Eng. 29 (6) (2003) 524–541.
[41] E.M. Clarke, E.A. Emerson, A. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (1986) 244–263.
[42] C. Baier, M. Kwiatkowska, On the verification of qualitative properties of probabilistic processes under fairness constraints, Inform. Process. Lett. 66 (2) (1998) 71–79.
[43] B. Haverkort, Markovian models for performance and dependability evaluation, in: E. Brinksma, H. Hermanns, J.-P. Katoen (Eds.), Lectures on Formal Methods and Performance Analysis, Lecture Notes in Computer Science, vol. 2090, Springer-Verlag, Berlin, 2001, pp. 38–83.
[44] C. Baier, B.R. Haverkort, H. Hermanns, J.-P. Katoen, Automated performance and dependability evaluation using model checking, in: M.C. Calzarossa, S. Tucci (Eds.), Performance Evaluation of Complex Systems: Techniques and Tools—Tutorial Lectures of the International Symposium on Computer Modeling, Measurement, and Evaluation (Performance'02), Rome, Italy, Lecture Notes in Computer Science, vol. 2459, Springer-Verlag, Berlin, 2002, pp. 261–289.
[45] M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri, M. Sebastianis, Model Checking Publish/Subscribe Notification for `thinkteam`, Tech. Rep. 2004-TR-20, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 2004. online: <http://fmt.isti.cnr.it/WEBPAPER/TRTT.ps>.
[46] M.H. ter Beek, M. Massink, D. Latella, Towards model checking stochastic aspects of the `thinkteam` user interface, Tech. Rep. 2005-TR-18, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 2005. online: <http://fmt.isti.cnr.it/WEBPAPER/TRdsvis.pdf>.
[47] C. Baier, J. Katoen, H. Hermanns, V. Wolf, Comparative branching-time semantics for Markov chains, Inform. Comput. (200) (2005) 149–214.
[48] G.I. Falin, A survey of retrial queues, Queueing Syst. 7 (1990) 127–168.
[49] G.I. Falin, J.G.C. Templeton, Monographs on Statistics and Applied Probability, Chapman & Hall, London, 1997.
[50] M. Neuts, Matrix-geometric Solutions in Stochastic Models—An Algorithmic Approach, The Johns Hopkins University Press, Baltimore, 1981.
[51] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: A Kernel Language for Agents Interaction and Mobility, IEEE Trans. Software Eng. 24 (5) (1998) 315–329.
[52] A. Lapadula, R. Pugliese, F. Tiezzi, Calculus for orchestration of web services, in: R. De Nicola (Ed.), Proceedings of the 16th European Symposium on Programming (ESOP'07), Lecture Notes in Computer Science, vol. 4421, Springer-Verlag, Berlin, 2007, pp. 33–37.
[53] M. Boreale, R. Bruni, R. De Nicola, M. Loreti, Sessions and pipelines for structured service programming, in: G. Barthe, F. de Boer (Eds.), Proceedings of FMOODS'08, Lecture Notes in Computer Science, vol. 5051, Springer-Verlag, Berlin, 2008, pp. 19–38.
[54] R. De Nicola, D. Latella, M. Massink, Formal modeling and quantitative analysis of KLAIM-based mobile systems, in: H. Haddad, L. Liebrock, A. Omicini, R. Wainwright, M. Palakal, M. Wilds, H. Clausen (Eds.), Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC 2005), Santa Fe, New Mexico, ACM Press, New York, 2005, pp. 428–435.
[55] R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, M. Massink, Klaim and its stochastic semantics, Tech. Rep. 6, Dipartimento di Sistemi e Informatica, Universitá di Firenze, 2006. online: <http://rap.dsi.unifi.it/~loreti/papers/TR062006.pdf>.
[56] D. Prandi, P. Quaglia, Stochastic COWS, in: B. Kramer, K.-J. Lin, P. Narasimhan (Eds.), Proceedings of the ICSOC 2007, Lecture Notes in Computer Science, vol. 4749, Springer-Verlag, Berlin, 2007, pp. 245–257.
[57] R. De Nicola, D. Latella, M. Loreti, M. Massink, MarCaSPiS: a Markovian extension of a calculus for services, in: M. Hennessy, B. Klin (Eds.), Proceedings of the 5th Workshop on Structural Operational Semantics (SOS 2008), Reykjavik, Iceland, July 6, 2008, pp. 6–20, preliminary Proceedings. Final Proceedings to appear as ENTCS by Elsevier.
[58] R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, M. Massink, Model checking mobile stochastic logic, Theoret. Comput. Sci. 382 (1) (2007) 42–70, http://dx.doi.org/10.1016/j.tcs.2007.05.008
[59] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, F. Tiezzi, A model checking approach for verifying COWS specifications, in: J.L. Fiadeiro, P. Inverardi (Eds.), Proceedings of the Conference on Fundamental Approaches to Software Engineering (FASE'08), Lecture Notes in Computer Science, vol. 4961, Springer-Verlag, Berlin, 2008, pp. 230–245.
[60] M.H. ter Beek, A. Bucchiarone, S. Gnesi, Formal methods for service composition, Ann. Math. Comput. Teleinform. 1 (5) (2007) 1–10.