Towards Model Checking Stochastic Aspects of the thinkteam User Interface*

FULL VERSION

Maurice H. ter Beek, Mieke Massink, and Diego Latella

CNR/ISTI – 'A. Faedo', Via G. Moruzzi 1, 56124 Pisa, Italy
{m.terbeek,m.massink,d.latella}@isti.cnr.it

Abstract. Stochastic model checking is a recent extension of traditional model-checking techniques for the integrated analysis of both qualitative and *quantitative* system properties. In this paper we show how stochastic model checking can be conveniently used to address a number of usability concerns that involve quantitative aspects of a user interface for the industrial groupware system thinkteam. thinkteam is a ready-to-use Product Data Management application developed by think3. It allows enterprises to capture, organise, automate, and share engineering product information and it is an example of an asynchronous and dispersed groupware system. Several aspects of the functional correctness, such as concurrency aspects and awareness aspects, of the groupware protocol underlying thinkteam and of its planned publish/subscribe notification service have been addressed in previous work by means of a traditional model-checking approach. In this paper we investigate the trade-off between two different design options for granting users access to files in the database: a retrial approach and a waiting-list approach and show how stochastic model checking can be used for such analyses.

Keywords: stochastic model checking, groupware, stochastic process algebras, stochastic temporal logic

1 Introduction

Computer Supported Cooperative Work (CSCW) is an interdisciplinary research field which deals with the understanding of how people work together, and the ways in which computer technology can assist them [19]. This technology mostly consists of multi-user computer systems called groupware (systems) [3, 16]. 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.* [16]. In this paper we deal with thinkteam, which is an asynchronous and dispersed groupware system. Other examples include electronic mail and collaborative writing systems.

In recent years there has been an increasing interest in the use of model checking for the formal verification of (properties of) groupware systems [6, 26, 28]. Among the properties of interest are those related to important groupware design issues such as user awareness and concurrency control. In [8] we have used a traditional model-checking approach, using the model checker SPIN [23], to formalise and verify a number of properties specifically of interest for the correctness of groupware protocols in general, *i.e.* not limited to the context of thinkteam.

thinkteam is think3's Product Data Management (PDM) application catering the product/document management needs of design processes in the manufacturing industry. thinkteam allows enterprises to capture, organise, automate, and share engineering product information in an efficient way. In [8] we studied the addition of a lightweight and easy-to-use publish/subscribe notification service to thinkteam. The goal of adding such a service to an application is to increase user awareness by intelligent data sharing: whenever a user publishes a document by sending it to a centralized repository, automatically all users that are subscribed to that document are asynchronously notified via a multicast communication. Due to a potentially large number of users, it is fundamental to use subscription-based multicast rather than broadcast communication.

* This work has been partially funded by the EU project AGILE (IST-2001-32747).

Many interesting properties of groupware protocols and their interfaces can be analysed *a priori*, *i.e.* before implementation, by means of a model-based approach and model-checking techniques. There are, however, also usability issues that are influenced by the *performance* of the groupware system rather than by its functional behaviour. One such issue was raised by the analysis of thinkteam with a traditional model-checking approach in [7, 8]. It was shown that it was possible for the system to exclude a user from obtaining a file simply because other users competing for the same file were more 'lucky' in their attempts to obtain the file. Such behaviour was explained by the fact that users were only provided with a file-access mechanism based on a retrial principle. While analysis with traditional model checking can be used to show that such a problem exists, it cannot be used to quantify the effect that it has on the user. In other words, it cannot be used to find out how often, in the average, a user needs to perform a retry in order to get a single file. Of course, the number of retries a user has to perform before obtaining a file is an important ingredient for measuring the usability of the system. If a high number of retries is necessary it may be more useful to have, *e.g.*, a waiting list for files to which users can subscribe.

In this paper we address the retrial problem by means of stochastic model checking. Stochastic model checking is a relatively new extension of traditional model checking that allows also for the analysis of *quantitative* properties of systems such as those related to performance and dependability issues [10, 21, 25, 29]. Other work on the use of stochastic modelling for usability analysis has been performed in [11], where it has been shown that Markov models and traditional Markov-chain analysis can be used to obtain measures that give an indication of how hard it is for a user to use certain interactive devices. In [14], stochastic modelling has been used to analyse the finger-tracking interface of a whiteboard with augmented reality. The analysis was based on stochastic process algebra models with general (*i.e.* not necessarily exponential) distributions and discrete event simulation. The main purpose of this paper is to show that stochastic model checking can be a convenient model-based analysis technique to address usability issues that involve quantitative measures. In this paper we explain *how* such analysis can be performed in the context of a simple but relevant industrial case study. In a further study we plan to collect more detailed statistical information on typical use of thinkteam in practice in order to calibrate the models for the analysis of specific situations. The current paper shows the formal specification of users and the system and their interactions, the formalisation of relevant quantitative properties and a number of numerical results.

We begin this paper with a brief description of thinkteam and of its underlying protocol, followed by a brief introduction to stochastic model checking and to the related tool PRISM. Subsequently we specify and verify a number of relevant properties related to the retrial problem for the thinkteam protocol, after which we compare this approach to the performance of a stochastic model of the thinkteam protocol based on a waiting list. Finally, we conclude with a discussion of future work.

2 thinkteam

In this section we present a brief overview of thinkteam. For more information we refer the reader to [7] and `http://www.think3.com/products/tt.htm`.

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

Controlled storage and retrieval of documents in PDM applications is traditionally called vaulting, the vault being a file-system-like repository. The two main functions of vaulting are: (1) to provide a single, secure, and controlled storage environment, where the documents controlled by the PDM application are managed, and (2) to prevent inconsistent updates or changes to the document base, while still allowing the maximal access compatible with the business rules. While the first function is subject to the implementation of the lower layers of the vaulting system, the second is implemented in thinkteam's underlying groupware protocol by a standard set of operations, *viz.*

get: extract a read-only copy of a document from the vault,
import: insert an external document into the vault,

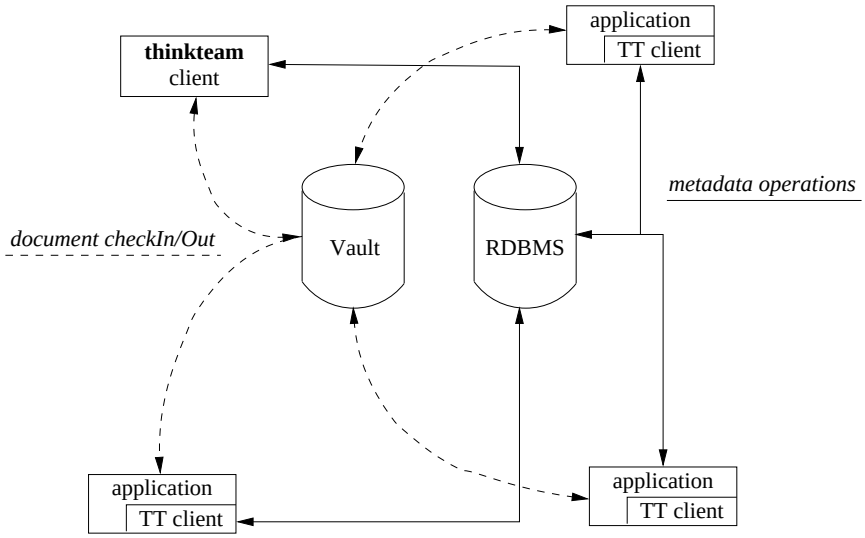

Fig. 1. The thinkteam structure.

checkOut: extract a copy of a document from the vault with the intent of modifying it (exclusive, *i.e.* only one checkout at a time is possible),
unCheckOut: cancel the effects of a previous checkout,
checkIn: replace an edited document in the vault (the document must previously have been checked out), and
checkInOut: replace an edited document in the vault, while at the same time retaining it as checked out.

It is important to note that access to documents (through the *checkOut* operation) is based on the 'retrial' principle: currently there is no queue (or reservation system) handling the requests for editing rights on a document.

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

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

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

In [7] several correctness properties of thinkteam have been verified. These properties addressed issues like concurrency control, awareness, and denial of service. Most properties were found to hold, with the exception of one concurrency-control property and a denial-of-service property. The problem with the concurrency-control property arises when a user simply 'forever' forgets to *checkIn* the file it has checked out, which means that the lock on this file is never released. In thinkteam, such a situation is resolved by the intervention of a system administrator.

The problem with the denial-of-service property, on the other hand, is that it may be the case that one of the users can 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 its desire to perform a *checkOut*. Such behaviour forms

an integral part of the thinkteam protocol. This is because, as mentioned before, access to documents is based on the 'retrial' principle: thinkteam currently has no queue (or reservation system) handling simultaneous requests for a document. Before simply extending thinkteam with a reservation system, it would be useful to know (1) how often, in the average, users have to express their requests before they are satisfied and (2) under which system conditions (number of users, file processing time, *etc.*) such a reservation system would really improve usability. We come back to this in the sequel.

3 Stochastic Analysis

Traditionally, functional analysis of systems (*i.e.* analysis concerning their functional correctness) and performance analysis have been two distinct and separate areas of research and practice. Recent developments in model checking, in particular its extension to address also performance and dependability aspects of systems, have given rise to a renewed interest in the integration of functional and quantitative analysis techniques.

A widely used model-based technique for performance analysis is based on Continuous Time Markov Chains (CTMCs). CTMCs provide a modelling framework which has proved to be extremely useful for practical analysis of quantitative aspects of system behaviour. Moreover, in recent years proper stochastic extensions of temporal logics have been proposed and efficient algorithms for checking the satisfiability of formulae of such logics on CTMCs (*i.e.* stochastic model checkers) have been implemented [10, 21, 25, 29].

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 $EXP(\lambda)$—if the probability of X being at most t, *i.e.* $\text{Prob}(X \leq t)$, 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 so called *memoryless property*, *i.e.* $\text{Prob}(X > t + t' \mid X > t) = \text{Prob}(X > t')$, for $t, t' \geq 0$.

CTMCs have been extensively studied in the literature (a comprehensive treatment can be found in [24]; we suggest [20] for a gentle introduction). For the purposes of the present paper 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 $\mathbf{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ is the *rate matrix*. The rate matrix characterises the transitions between the states of $\mathcal{M}$. If $\mathbf{R}(s, s') \neq 0$, then it is possible that a transition from state s to state s' takes place, and the probability of such a transition to take place within time t, is $1 - e^{-\mathbf{R}(s,s') \cdot t}$. If $\mathbf{R}(s, s') = 0$, then no such a transition can take place.[1] 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 standard analysis techniques (*e.g.* transient and steady-state analyses) and self-loops moreover turn out to be useful when model checking CTMCs [5]. Therefore we will allow them in this paper.

Usually CTMCs are not obtained in a direct way, due to the increasing complexity of the analysed systems, but they are instead generated in an automatic way from higher-level specifications given in languages such as *e.g.* stochastic Petri nets, stochastic process algebras or stochastic activity nets [9]. Our approach is no different. We specify our model of thinkteam's underlying groupware protocol in the stochastic process algebra PEPA (Performance Evaluation Process Algebra) [22]. This specification is consequently read into a stochastic model checker, which then automatically builds a CTMC from it.

4 Stochastic Model Checking

In the model-checking approach to performance and dependability analysis a *model* of the system under consideration is required together with a desired *property* or *performance/dependability measure*. In case of stochastic modelling, such models are typically CTMCs or high-level specification languages to generate them, while properties are usually expressed in some form of extended temporal logic. In this

[1] The reader should be warned that the above intuitive interpretation is correct only if there is only one transition originating from s. If this is not the case, then a *race condition* arises among all transitions originating from s.

paper, Continuous Stochastic Logic (CSL) [2, 4] is used, which is a stochastic variant of the well-known Computational Tree Logic (CTL) [12]. CTL allows for stating properties over *states* as well as over *paths*.

CSL extends CTL by two probabilistic operators that refer to the steady-state and transient behaviour of the system being studied. The steady-state operator refers to the probability of residing in a particular state or set of *states* (specified by a state formula) in the long run. The transient operator allows us to refer to the probability mass of the set of *paths* in the CTMC that satisfy a given (path) property. In order to express the time span of a certain path, the path operators until ($\mathcal{U}$) and next (X) are extended with a parameter that specifies a time interval. Let I be an interval on the real line, p a probability value and $\bowtie$ a comparison operator, *i.e.* $\bowtie \in \{ <, \leq, \geq, > \}$. The syntax of CSL is:

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 $\bowtie p$
$\mathcal{P}_{\bowtie p}(\varphi)$: probability that a path fulfills $\varphi \bowtie p$

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 fulfills 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 like conjunction ($\wedge$), implication ($\Rightarrow$), true (TRUE) and false (FALSE), and so forth, can be defined in the usual way. The state formula $\mathcal{S}_{\bowtie p}(\Phi)$ asserts that the steady-state probability for the set of states satisfying Φ, the Φ-states, meets the bound $\bowtie p$. $\mathcal{P}_{\bowtie p}(\varphi)$ asserts that the probability measure of the set of paths satisfying φ meets the bound $\bowtie p$. The operator $\mathcal{P}_{\bowtie p}(.)$ replaces the usual CTL path quantifiers $\exists$ and $\forall$. In CSL, the formula $\mathcal{P}_{\geq 1}(\varphi)$ holds if *almost all* paths satisfy φ. Moreover, clearly $\exists \, \varphi$ holds whenever $\mathcal{P}_{>0}(\varphi)$ holds.

The formula $\Phi \, \mathcal{U}^I \, \Psi$ is satisfied by a path if Ψ holds at time $t \in I$ and at every preceding state on the path, if any, Φ holds. In CSL, temporal operators like $\Diamond$, $\Box$ and their real-time variants $\Diamond^I$ or $\Box^I$ can be derived, *e.g.*, $\mathcal{P}_{\bowtie p}(\Diamond^I \Phi) = \mathcal{P}_{\bowtie p}(\text{TRUE} \; \mathcal{U}^I \; \Phi)$ and $\mathcal{P}_{\geq p}(\Box^I \Phi) = \mathcal{P}_{<1-p}(\Diamond^I \neg \Phi)$. The untimed next and until operators are obtained by $X \, \Phi = X^I \, \Phi$ and $\Phi_1 \, \mathcal{U} \, \Phi_2 = \Phi_1 \; \mathcal{U}^I \; \Phi_2$ for $I = [0, \infty)$. In a variant of CSL the probability p can be replaced by a question mark, denoting that one is looking for the value of the probability rather than verifying whether the obtained probability respects certain bounds.

The model checker PRISM [25, 27] is a prototype tool that supports, among others, the verification of CSL properties over CTMCs. It checks the validity of CSL properties for given states in the 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). PRISM accepts system descriptions in different specification languages, among which the process algebra PEPA and the PRISM language—a simple state-based language based on the Reactive Modules formalism of Alur and Henzinger [1]. Alternative model checkers for CSL are ETMCC [21] and Prover [29]. The latter is based on discrete event simulation rather than numerical computation.

5 Stochastic Model of thinkteam—The Retrial Approach

In this section we describe the stochastic model of thinkteam for the retrial approach, after which we perform several analyses on this model. We use the stochastic process algebra PEPA to specify our model. 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. Before explaining our model, we briefly describe the PEPA language constructs that we will use. For a more detailed description of these constructs we refer the reader to [22].

The basic mechanism for constructing behaviour expressions is by means of prefixing. The component $(\alpha, r).P$ carries out activity (α, r), with action type α and duration Δt determined by rate r. The component subsequently behaves as component P.

The component $P + Q$ represents a system which may behave either as component P or as component Q. The continuous nature of the probability distributions ensures that the probability of P and Q both completing an activity at the same time is zero. The choice operator represents a competition (the race condition) between components.

The cooperation operator $P \bowtie_L Q$ defines the set of action types L on which the components P and Q must synchronise or *cooperate*. Both components proceed independently with any activities that do not occur in L. The expected duration of a cooperation where activities are shared (*i.e.* L is not empty) will be greater than or equal to the expected durations of the corresponding activities in the cooperating components. A special case is the situation in which one component is *passive* (*i.e.* has the special rate $-$) with respect to the other component. In this case the total rate is determined by that of the *active* component only.

We are now ready to explain our model in detail. In Fig. 2 the models of the User and CheckOut components are drawn as automata for reasons of readability.

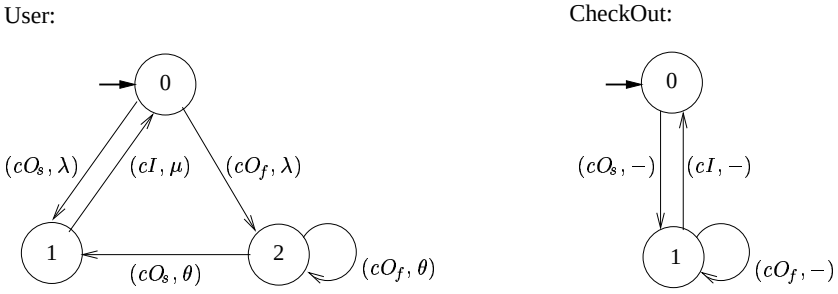

Fig. 2. The retrial approach: automata of the User and the CheckOut components.

We consider the case that there is only one file, the (exclusive) access to which is handled by the CheckOut component. A user can express interest in checking out the file by performing a *checkOut*. This operation can either result in the user being granted the access to the file or in the user being denied the access because the file is currently already checked out by another user. In Fig. 2, the *s*uccessful execution of a *checkOut* is modelled by activity (cO_s, λ), while a *f*ailed *checkOut* is modelled by (cO_f, λ). The exponential rate λ is also called the *request* rate. If the user does not obtain the file, then she may retry to check out the file, which is modelled by activity (cO_s, θ) in case of a *s*uccessful retry and by (cO_f, θ) in case the *checkOut f*ailed. The exponential rate θ is also called the *retrial* rate. The *checkIn* operation, finally, is modelled by activity (cI, μ) and the exponential rate μ is also called the *file processing* 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 $\langle 1 \rangle$) or not (state $\langle 0 \rangle$). When a User tries to obtain the file through the *checkOut* activity, then she is denied the file if it is currently checked out by another user while she might obtain the file if it is available.

The formal PEPA specifications of the User and the CheckOut component, and the composed model for three User components and the CheckOut component

$$(\text{User} \parallel \text{User} \parallel \text{User}) \bowtie_{\{cO_s, cO_f, cI\}} \text{CheckOut}$$

is given in Appendix A. These specifications are accepted as input by PRISM and consequently translated into the PRISM language. The resulting specification is given in Appendix B. From such a specification PRISM automatically generates a CTMC with 19 states and 54 transitions. Each state is represented as a quadruple (a, b, c, d), where a, b and c give the state in the User automaton of the first, the second and the third user, respectively, while d gives the state of automaton CheckOut. In Fig. 3 we have drawn this

CTMC's and we have sketched how this CTMC is lumping equivalent to the much simpler CTMC given in Fig. 4.[2]

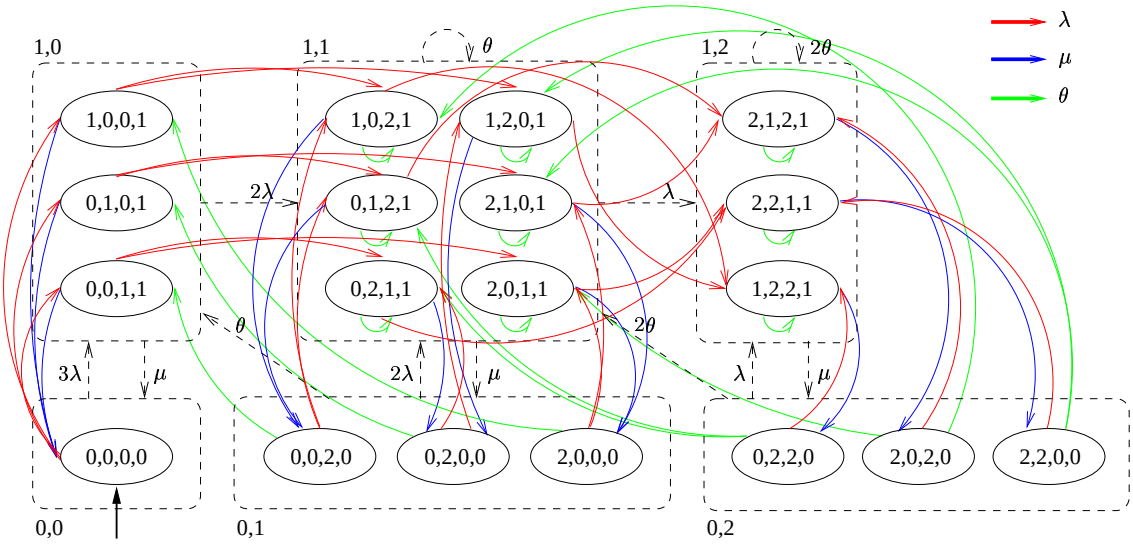

Fig. 3. The state space of the CTMC obtained for the retrial approach in case of 3 users and a sketch of its lumping equivalence to the CTMC of Fig. 4.

Intuitively, lumping equivalence is a notion of behavioural equivalence for the purpose of transient and steady-state analysis. We now define this notion formally [22]. An equivalance relation over the state space of a Markov chain induces a partition on the state space. Consequently, an *aggregated* Markov chain is obtained by constructing such a partition and by letting each class in the partition of the original Markov chain form one state of this aggregated Markov chain. If the original state space is $\{s_1, s_2, \ldots, s_n\}$, then the aggregated state space is something like $\{s_{[1]}, s_{[2]}, \ldots, s_{[m]}\}$, where $m < n$. Ideally, $m \ll n$. Now a Markov chain is *lumpable* w.r.t. a partition σ iff for any $s_{[j]}, s_{[k]} \in \sigma$ and $s, s' \in s_{[j]}$, $\mathbf{R}'(s, s_{[k]}) = \mathbf{R}'(s', s_{[k]})$, where $\mathbf{R}' : \mathcal{S} \times 2^{\mathcal{S}} \to \mathbb{R}_{\geq 0}$ is defined as $\mathbf{R}'(s, s_{[k]}) = \sum_{j \in [k]} \mathbf{R}'(s, s_j)$. Now we are ready to define lumping equivalence. Two Markov chains $\mathcal{M}$ and $\mathcal{M}'$ are *lumping equivalent* iff there is a lumpable partition σ of $\mathcal{M}$ and a lumpable partition σ' of $\mathcal{M}'$ such that there exists an injective function f that satisfies $\mathbf{R}''(s_{[j]}, s_{[k]}) = \mathbf{R}''(s'_{f([j])}, s'_{f([k])})$, where $s_{[j]}, s_{[k]} \in \sigma$, $s'_{f([j])}, s'_{f([k])} \in \sigma'$, and $\mathbf{R}'' : 2^{\mathcal{S}} \times 2^{\mathcal{S}} \to \mathbb{R}_{\geq 0}$ is defined as $\mathbf{R}''(s_{[j]}, s_{[k]}) = \sum_{i \in [j]} \pi_j(s_i) \mathbf{R}'(s_i, s_{[k]})$ in which $\pi_j(s_i)$ denotes the conditional steady-state probability of being in each state s_i of the class $[j]$ of the partition σ. Hence two Markov chains are lumping equivalent iff they have lumpable partitions with the same number of elements and there exists a one-to-one correspondence between the partitions such that the aggregated transition rates between partitions are also matched.[3] It is not difficult to verify that the CTMC of Fig. 3 is indeed lumping equivalent to the CTMC of Fig. 4.

The states of this latter CTMC are tuples $\langle x, y \rangle$ in which x denotes whether the file is checked out ($x = 1$) or not ($x = 0$) and $y \in \{0, 1, 2\}$ denotes the number of users that are currently retrying to perform a *checkOut*. Note, however, that when a user inserts the file she has checked out back into the Vault through 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 retrial rates associated to the *checkOut* activity (*cf.* states $\langle 0, 1 \rangle$ 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

[2] Moreover, the CTMC obtained by removing the self-loops from that of Fig. 4 is frequently used in the theory of *retrial queues* [17, 18, 24].

[3] Notice that there exists a close relationship between lumping equivalence and the strong equivalence as defined in [22], which helps to fill the gap between model checking theories and behavioural equivalences in the context of process algebras.

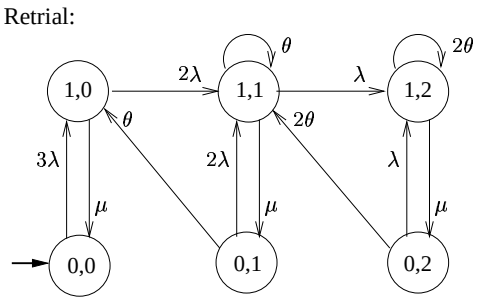

Fig. 4. The lumping-equivalent CTMC obtained for the retrial approach in case of 3 users.

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 $\theta + \lambda$.

We use the stochastic model and the stochastic logic to formalise and analyse various usability issues concerning the retrial approach 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 $60/5 = 12$ minutes on the average.

While the specifications in Appendices A-D assume three User components and one CheckOut component, our analyses below sometimes consider upto ten User components composed with one CheckOut component. Note that this leads to models that can no longer be verified by hand. For instance, the resulting composition in case of ten User components has a total of $6,143$ states and $43,500$ transitions and the lumping-equivalent CTMC has 20 states and 52 transitions.

All analyses reported in this paper have been performed by running PRISM Version 2.0 and each took only a few seconds of CPU time. The iterative numerical method used for the computation of probabilities was Gauss-Seidel and the accuracy 10^{-6}. For details about these issues, we refer the reader to [27].

5.1 Analyses of Performance Properties

We first analyse the probability that a user that has requested the file and is now in 'retry mode' (state $\langle 2 \rangle$ of the User component), obtains the requested file within the next five hours. This measure can be formalised in CSL as (in a pseudo-notation close to the PRISM notation):

$$\mathcal{P}_{=?}([\ \text{TRUE}\ \mathcal{U}^{\leq 5}\ (\text{User\_STATE} = 1)\ \{\text{User\_STATE} = 2\}\ ]),$$

which must be read as follows: "what is the probability that path formula TRUE $\mathcal{U}^{\leq 5}$ (User_STATE $= 1$) is satisfied for state User_STATE $= 2$?".

The results for this measure are presented in Fig. 5 for request rate $\lambda = 1$ (*i.e.* a user requests the file once an hour on average), retrial rate θ taking values 1, 5 and 10 (*i.e.* in one hour a user averagely retries one, five or ten times to obtain the file), file processing rate $\mu = 1$ (*i.e.* a user in the average keeps the file checked out for one hour) and for different numbers of users ranging from 1 to 10.

Clearly, with an increasing number of users the probability that a user gets her file within the time interval is decreasing. On the other hand, with an increasing retrial rate and an invaried file processing 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) a retrial approach would give satisfactory results from a usability point of view.

A somewhat more complicated measure can be obtained with some additional calculations involving steady-state analysis (by means of model checking). Fig. 6 shows the average number of retrials per file request for request rate $\lambda = 1$, retrial rate θ taking values 5 and 10, file processing 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 divided by the average number of requests during T.

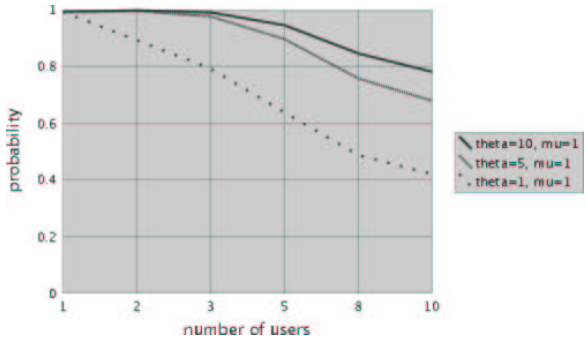

Fig. 5. Probability for users in 'retry mode' to *checkOut* requested file within the next 5 hours.

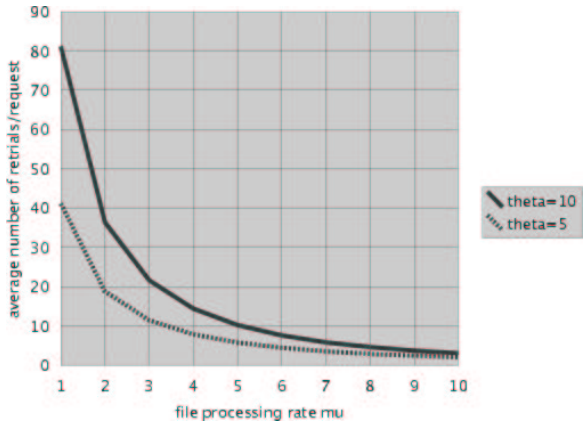

Fig. 6. Average number of retrials per file request in case of 10 users.

To compute the average number of retries (requests, resp.) we proceeded as follows. We first compute the steady-state probability p (q, resp.) of the user being in 'retry mode' ('request mode', resp.), *i.e.* $p \stackrel{\text{def}}{=} \mathcal{S}_{=?}(\text{User\_STATE} = 2)$ ($q \stackrel{\text{def}}{=} \mathcal{S}_{=?}(\text{User\_STATE} = 0)$, resp.). The fraction of time the user is in 'retry mode' ('request mode', resp.) is then given by $T \times p$ ($T \times q$, resp.). Consequently the average number of retries (requests, resp.) is $\theta \times T \times p$ ($\lambda \times T \times q$, resp.). Hence the measure of interest is $(\theta \times p)/(\lambda \times q)$.

It is easy to observe in Fig. 6 that the number of retrials decreases considerably when the file processing rate is increased (*i.e.* when the users restitute, by means of a *checkIn*, the file they had checked out after a shorter time). We also note that a relatively high file processing rate is needed for obtaining an acceptably low number of retrials in the case of 10 users that regularly compete for the same file.

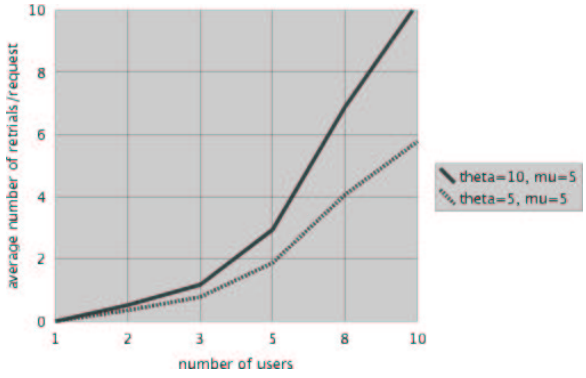

Fig. 7. Average number of retrials per file request in case of a varying number of users.

The effect on the average number of retries is even better illustrated in Fig. 7, where with a similar approach as outlined above the average number of retrials per file request is presented for request rate $\lambda = 1$, retrial rate θ taking values 5 and 10, fixed file processing rate $\mu = 5$ and various numbers of users. Clearly, the average number of retrials per file request increases sharply when the number of users increases.

6 Stochastic Model of thinkteam—The Waiting-List Approach

In this section we compare the stochastic model of the previous section to a stochastic model for a waiting-list approach, which we describe next.

We again use PEPA to specify our model and we draw the models as automata for readability. In contrast with our model for the retrial approach from the previous section, we now assume that a user that attempts to *checkOut* the file when it has already been checked out by another user is put in a FIFO queue. The moment in which the file then becomes available, the user that is first in this FIFO queue obtains the file. This implies the following changes w.r.t. the model from the previous section. Since a user no longer retries to obtain the file after her initial unsuccessful attempt to *checkOut* the file, the new User component has two states only, *viz.* state $\langle 2 \rangle$ is removed from the User component as given in Fig. 2. Moreover, since the CheckOut component now implements a FIFO policy, the new CheckOut component needs to keep track of the number of users in the FIFO queue and their relative position. This also means that the new CheckOut component needs to know exactly which of the users performs a *checkOut* or a *checkIn*. In Fig. 8 the model of the new Users and CheckOut—now called FIFO—components are drawn.

The formal PEPA specifications of these new Users and CheckOut components, and the composed model for three User components and the CheckOut component

$$(\text{User\_0} \parallel \text{User\_1} \parallel \text{User\_2}) \bowtie_{\{cO_0, cO_1, cO_2, cI_0, cI_1, cI_2\}} \text{FIFO}$$

is given in Appendix C. The translations of these specifications into the PRISM language are given in Appendix D. From these specifications PRISM automatically generates a CTMC with 16 states and 30 transitions. In Fig. 9 we have drawn this CTMC and we have sketched how this CTMC is lumping equivalent to the much simpler CTMC given in Fig. 10.

It is not difficult to verify that the CTMC of Fig 9 is indeed lumping equivalent to the CTMC of Fig. 10.

The state tuples of this CTMC have the same meaning as before, but 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 an exponential request rate λ, an exponential file processing rate μ, one file, and three users means that we are dealing with a M|M|1|3 *queueing system* [20, 24]. The CTMC of Fig. 10 is then its underlying CTMC and this type of CTMC is also called a *birth-death process* [20, 24].

We now compare the two models with respect to the steady-state probability that there are users waiting to obtain the file after a *checkOut* request. To measure this 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 retrial approach this concerns the situation in which any of the User components of Fig. 2 is in state $\langle 2 \rangle$, whereas in the waiting-list approach this concerns the situation in which two or three of the new User components of Fig. 8 are in state $\langle ia \rangle$, $i \in \{0, 1, 2\}$. Hence this can be expressed as CSL steady-state formulae (again in a pseudo-notation close to that of PRISM)

$$p \stackrel{\text{def}}{=} \mathcal{S}_{=?}([\ (\text{User\_STATE} = \langle 2 \rangle) \mid (\text{User\_2\_STATE} = \langle 2 \rangle) \mid (\text{User\_3\_STATE} = \langle 2 \rangle)\ ])$$

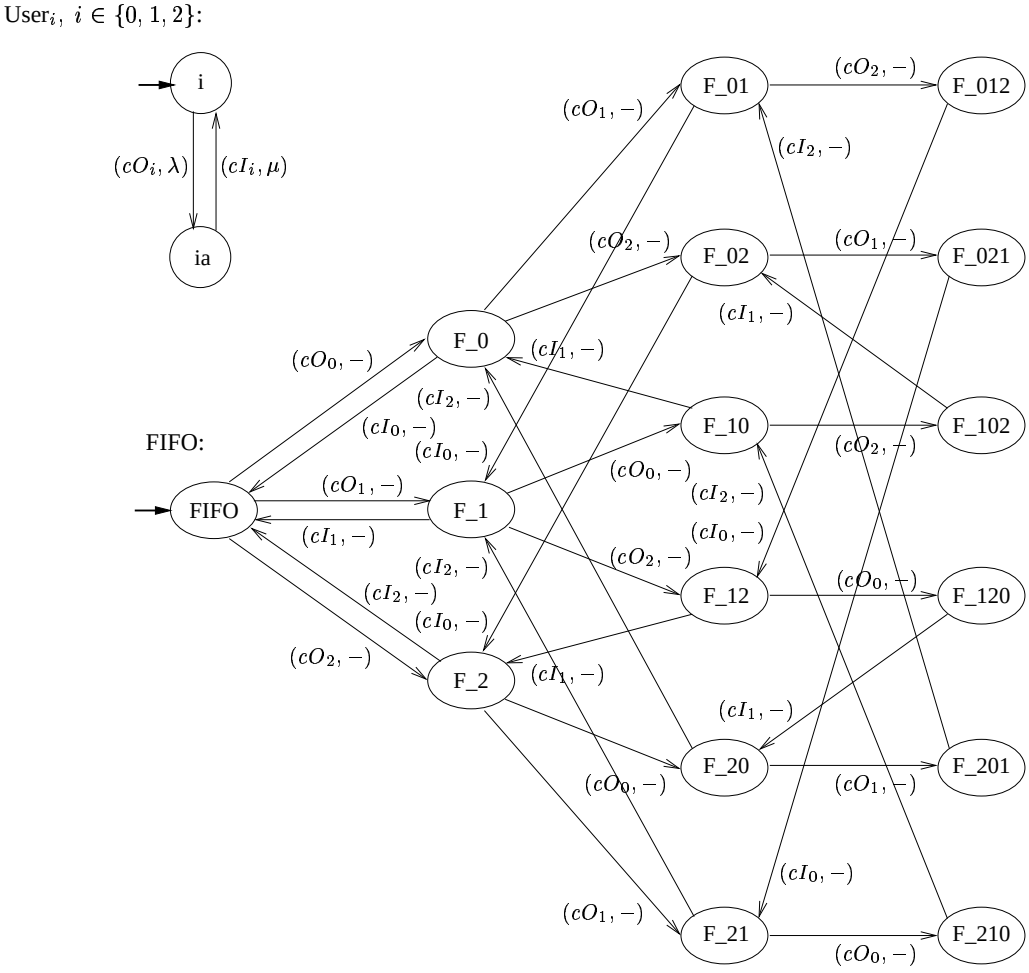

Fig. 8. The waiting-list approach: automata of the Users and the FIFO components.

in the retrial approach while[4]

$$p \stackrel{\text{def}}{=} \mathcal{S}_{=?}([\ (\text{User\_0\_STATE} = \langle 0a \rangle)\ \&\ (\text{User\_1\_STATE} = \langle 1a \rangle)\ \mid (\text{User\_1\_STATE} = \langle 1a \rangle)\ \&\ (\text{User\_2\_STATE} = \langle 2a \rangle)\ \mid (\text{User\_2\_STATE} = \langle 2a \rangle)\ \&\ (\text{User\_0\_STATE} = \langle 0a \rangle)\ ])$$

in the waiting-list approach.

The results of our comparison are presented in Fig. 11 for request rate $\lambda = 1$, retrial rate θ (only in the retrial approach of course) ranging from 1 to 10, file processing rate μ taking values 5 and 10 and 3 users.

It is easy to see that, as expected, the waiting-list approach outperforms the retrial approach in all the cases we considered: The probability to be in a situation in which two or three of the new User components of the waiting-list approach are in state $\langle ia \rangle$, $i \in \{0,1,2\}$, is always lower than the probability to be in a

[4] The formula corresponding to the statement of interest is (User_0_STATE $= \langle 0a \rangle$) & (User_1_STATE $= \langle 1a \rangle$) | (User_0_STATE $= \langle 0a \rangle$) & (User_1_STATE $= \langle 1a \rangle$) & (User_2_STATE $= \langle 2a \rangle$) | (User_1_STATE $= \langle 1a \rangle$) & (User_2_STATE $= \langle 2a \rangle$) | (User_1_STATE $= \langle 1a \rangle$) & (User_2_STATE $= \langle 2a \rangle$) & (User_0_STATE $= \langle 0a \rangle$) | (User_2_STATE $= \langle 2a \rangle$) & (User_0_STATE $= \langle 0a \rangle$) | (User_2_STATE $= \langle 2a \rangle$) & (User_0_STATE $= \langle 0a \rangle$) & (User_1_STATE $= \langle 1a \rangle$), which is logically equivalent to (User_0_STATE $= \langle 0a \rangle$) & (User_1_STATE $= \langle 1a \rangle$) | (User_1_STATE $= \langle 1a \rangle$) & (User_2_STATE $= \langle 2a \rangle$) | (User_2_STATE $= \langle 2a \rangle$) & (User_0_STATE $= \langle 0a \rangle$).

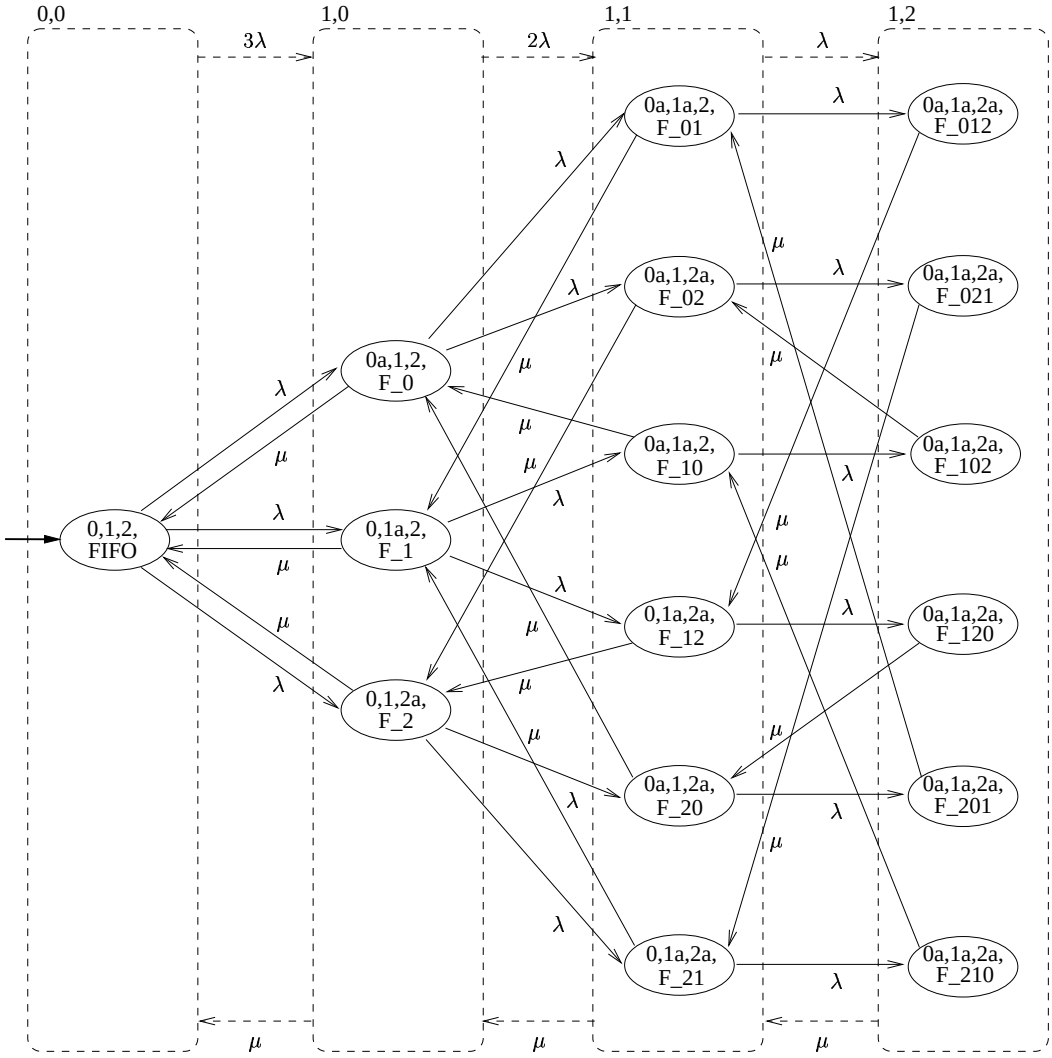

Fig. 9. The CTMC obtained for the waiting-list approach in case of 3 users and a sketch of its lumping equivalence to the CTMC of Fig. 10.

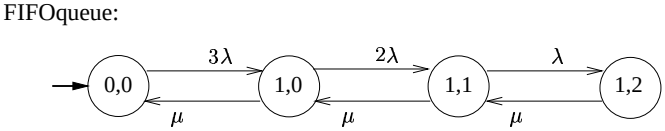

Fig. 10. The lumping-equivalent CTMC obtained for the waiting-list approach in case of 3 users.

situation in which any of the User components of the retrial approach is in state $\langle 2 \rangle$. Note that for large θ the probability in case of the retrial approach is asymptotically approaching that of the waiting-list approach, 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 thus 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 in the waiting-list approach than in the retrial approach. Furthermore, while increasing the retrial rate (*i.e.* reducing the time inbetween retries) does bring the results for the retrial approach close to those for the waiting-list approach, it takes highly unrealistic retrial rates to reach a difference between the two approaches that is insignificantly small.

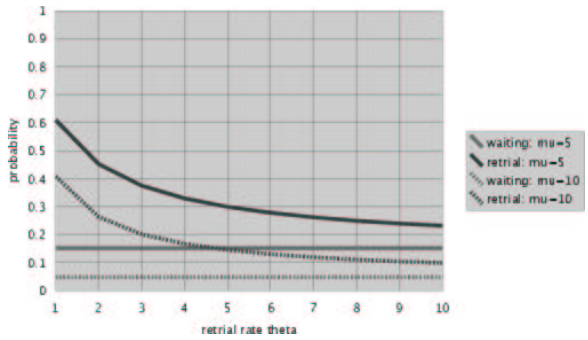

Fig. 11. The retrial approach *vs.* the waiting-list approach.

7 Conclusions and Further Work

In this paper we have addressed the formal analysis of a number of quantitative usability measures of a small, but relevant industrial case study concerning an existing groupware system for PDM, thinkteam. We have shown how the quantitative aspects of the interaction between users and the system can be modelled in a process-algebraic way by means of the stochastic process algebra PEPA. The model has been used to obtain quantitative information on two different approaches for users to obtain files from a central database; one based on retrial and one based on a reservation system implemented via a FIFO waiting list. The quantitative measures addressed the expected time that users are required to wait before they obtain a requested file and the average number of retries per file, for various assumptions on the parameters of the model. These measures have been formalised as formulae of the stochastic temporal logic CSL and analysed by means of the stochastic model checker PRISM.

The results show that, from a user perspective, the retrial approach is less convenient than the waiting-list approach. Moreover, it can be shown that the situation for the retrial approach rapidly deteriorates with an increasing number of users that compete for the same file. The use of stochastic model checking allowed for a convenient modular and compositional specification of the system and the automatic generation of the underlying CTMCs on which performance analysis is based. Furthermore, the combination of compositional specification of models and logic characterisation of measures of interest provides a promising technique for *a priori formal* model-based usability analysis where quantitative aspects as well as qualitative ones are involved.

We plan to apply the developed models and measures for the analysis of different user profiles of the actual thinkteam system in collaboration with think3. The user profiles will be useful to obtain realistic values for the parameters of the model and for further validation of the model. In a further extension of the model we also plan to deal with the quantitative effects of the addition of a publish/subscribe notification service to thinkteam.

References

1. R. Alur and T. Henzinger, Reactive modules. *Formal Methods in System Design* 15, 1(1999), 7–48.
2. A. Aziz, K. Sanwal, V. Singhal and R. Brayton, Model checking continuous time Markov chains. *ACM Transactions on Computational Logic* 1, 1 (2000), 162–170.
3. R.M. Baecker (ed.), *Readings in Groupware and Computer Supported Cooperation Work*. Morgan Kaufmann, 1992.
4. C. Baier, J.-P. Katoen and H. Hermanns, Approximate symbolic model checking of continuous-time Markov chains. In *Concurrency Theory, LNCS* 1664, Springer-Verlag, 1999, 146–162.
5. C. Baier, B.R. Haverkort, H. Hermanns and J.-P. Katoen, Automated performance and dependability evaluation using model checking. In *Computer Performance Evaluation*, Springer-Verlag, 2002, 261–289.
6. M.H. ter Beek, M. Massink, D. Latella and S. Gnesi, Model checking groupware protocols. In *Cooperative Systems Design*, IOS Press, 2004, 179–194.

7. M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri and M. Sebastianis, Model checking publish/subscribe notification for thinkteam, Technical Report 2004-TR-20, CNR/ISTI, 2004. URL: `http://fmt.isti.cnr.it/WEBPAPER/TRTT.ps`.
8. M.H. ter Beek, M. Massink, D. Latella, S. Gnesi, A. Forghieri and M. Sebastianis, Model checking publish/subscribe notification for thinkteam. In *Proceedings 9th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'04), Linz, Austria, 2004*, 35–52. To appear in Electronic Notes in Theoretical Computer Science.
9. E. Brinksma, H. Hermanns and J.-P. Katoen (Eds.), *Lectures on Formal Methods and Performance Analysis, LNCS* 2090. Springer-Verlag, 2001.
10. P. Buchholz, J.-P. Katoen, P. Kemper and C. Tepper, Model-checking large structured Markov chains. *Journal of Logic and Algebraic Programming* 56 (2003), 69–96.
11. P. Cairns, M. Jones and H. Thimbleby, Reusable usability analysis with Markov models. *ACM Transactions on Computer Human Interaction* 8, 2 (2001), 99–132.
12. E. Clarke, E. Emerson and A. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8 (1986), 244–263.
13. E.M. Clarke Jr., O. Grumberg and D.A. Peled, *Model Checking*. MIT Press, 1999.
14. G. Doherty, M. Massink and G.P. Faconti, Reasoning about interactive systems with stochastic models. In *Interactive Systems: Design, Specification, and Verification, Revised Papers of the 8th International Workshop (DSV-IS'01), Glasgow, Scotland, LNCS* 2220, Springer-Verlag, 2001, 144–163.
15. P. Dourish and V. Bellotti, Awareness and coordination in shared workspaces. In *Proceedings 4th Conference on Computer Supported Cooperative Work (CSCW'92), Toronto, Canada*, ACM Press, 1992, 107–114.
16. C.A. Ellis, S.J. Gibbs and G.L. Rein, Groupware—Some issues and experiences. *Communications of the ACM* 34, 1 (1991), 38–58.
17. G.I. Falin, A survey of retrial queues. *Queueing Systems* 7 (1990), 127–168.
18. G.I. Falin and J.G.C. Templeton, *Retrial Queues*. Chapman & Hall, 1997.
19. J. Grudin, CSCW—History and focus, *IEEE Computer* 27, 5 (1994), 19–26.
20. B.R. Haverkort, Markovian models for performance and dependability evaluation. In [9], 38–83.
21. H. Hermanns, J-P. Katoen, J. Meyer-Kayser and M. Siegle, A tool for model-checking Markov chains. *International Journal on Software Tools for Technology Transfer* 4, 2 (2003), 153–172.
22. J. Hillston, *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
23. G.J. Holzmann, *The SPIN Model Checker—Primer and Reference Manual*. Addison Wesley, 2003.
24. V. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, 1995.
25. M. Kwiatkowska, G. Norman and D. Parker, Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proceedings 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Grenoble, France, LNCS* 2280, Spinger-Verlag, 2002, 52–66.
26. C. Papadopoulos, An extended temporal logic for CSCW. *The Computer Journal* 45, 4 (2002), 453–472.
27. D. Parker, G. Norman and M. Kwiatkowska, *PRISM 2.0—Users' Guide*, February 2004. URL: `http://www.cs.bham.ac.uk/~dxp/prism`.
28. T. Urnes, *Efficiently Implementing Synchronous Groupware*. Ph.D. thesis, Department of Computer Science, York University, Toronto, 1998.
29. H.L.S. Younes and R.G. Simmons, Probabilistic verification of discrete event systems using acceptance sampling. In *Proceedings 14th International Conference on Computer Aided Verification (CAV'02), Copenhagen, Denmark, LNCS* 2404, Springer-Verlag, 2002, 223–235.

A PEPA Specification of Retrial Approach With 3 Users

```
lambda = 1.0;     % request rate (user/hour)
mu = 5.0;         % file processing rate (user/hour)
theta = 5.0;      % retrial 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
```

B PRISM Specification of Retrial Approach With 3 Users

```
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 PEPA Specification of Waiting-List Approach With 3 Users

```
lambda = 1.0;    % request rate
mu = 5.0;        % file processing 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
```

D 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;

[cO_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;

[cO_0] (FIFO_empty_STATE=FIFO_empty) -> 1 : (FIFO_empty_STATE'=F_0);
[cO_1] (FIFO_empty_STATE=FIFO_empty) -> 1 : (FIFO_empty_STATE'=F_1);
[cO_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);
[cO_1] (FIFO_empty_STATE=F_0) -> 1 : (FIFO_empty_STATE’=F_01);
[cO_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);
[cO_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);
[cO_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);
[cO_0] (FIFO_empty_STATE=F_1) -> 1 : (FIFO_empty_STATE’=F_10);
[cO_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);
[cO_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);
[cO_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);
[cO_0] (FIFO_empty_STATE=F_2) -> 1 : (FIFO_empty_STATE’=F_20);
[cO_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);
[cO_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);
[cO_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)) |[cO_0,cO_1,cO_2,cI_0,cI_1,cI_2]| FIFO_empty) endsystem
```