Model Checking with SPIN

Maurice H. ter Beek

FM&&T, ISTI-CNR

Pisa, Italy

Wednesday 20 April 2005

FoTS, University of Antwerp

Antwerp, Belgium

⇒ inspired by slides on SPIN by Theo Ruys, Gerard Holzmann, and Diego Latella

Outline

- model checking
- SPIN in brief
- Promela models
- LTL formulae
- some theory
- Xspin in brief
- a case study

Model Checking

an automatic technique to verify if a concurrent system design satisfies its specifications

very hard in standard ways (like, e.g., testing) due to non-determinism & interleaving

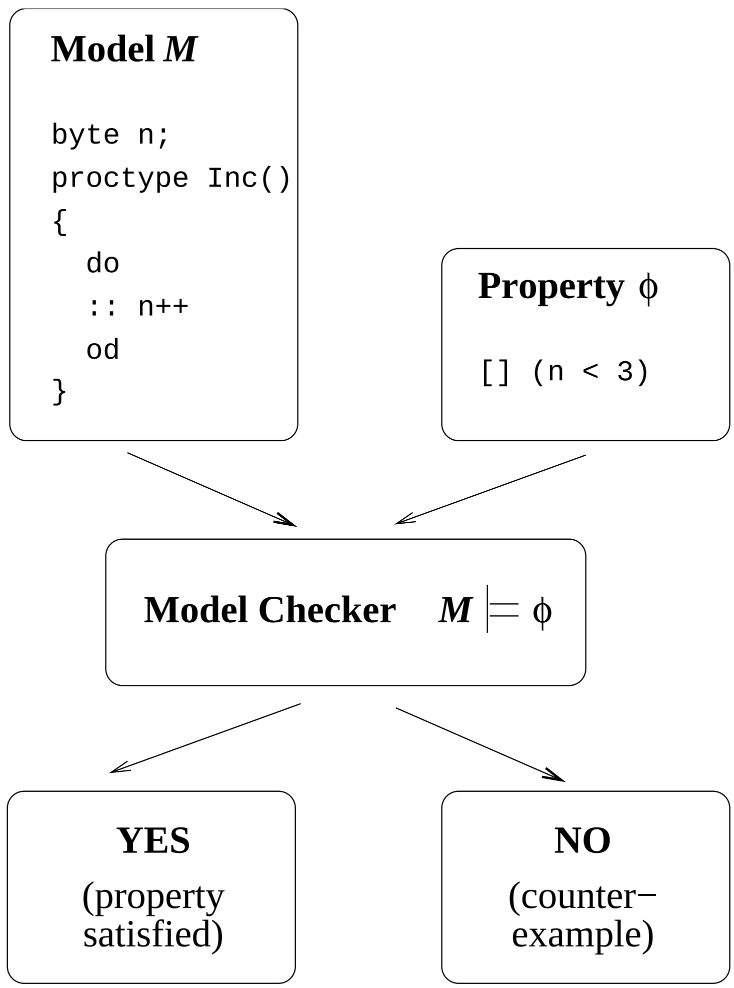
+ exhaustive verification, i.e. takes into account all possible input combinations & states

− risk of running out of memory due to a state-space explosion

⇒ a simplified model is used, still capturing the core of the system design while abstracting from unnecessary details

SPIN among the most powerful model checkers

Model Checking Approach



SPIN – Simple Promela INterpreter

a state-of-the-art & on-the-fly model checker

developed at Bell Labs by Gerard Holzmann :

Design and Validation of Computer Protocols

Prentice Hall, 1991, 500 pp.

The Spin Model Checker—Primer and Reference Manual

Addison-Wesley, 2003, 608 pp.

⇒ winner ACM System Software Award '01

SPIN in brief

formal verification of concurrent systems (esp. communication protocols) specified in Promela

verifies deadlocks / assertions / unreachable code / LTL formulae / liveness / etc.

provides counterexample if property is violated

based on over two decades of research on CAV

very well documented: `http://spinroot.com` and scientific literature

very nice user-friendly graphical interface: Xspin

Model Checking with SPIN

1. write abstract Promela model of a system

2. formalise correctness properties (e.g. LTL)

3. run the model checking algorithm of SPIN

4. interpret one of the three possible results:

   - the model satisfies the property $\surd$

   - the model can violate the property
     $\Rightarrow$ study the given counterexample

   - insufficient resources to solve problem
     $\Rightarrow$ construct a more abstract model

5. revise 1 or 2 & repeat 3–5 until satisfied

Promela – PROcess MEta LAnguage

non-deterministic C-like specification language

allows dynamic creation of concurrent processes

(very large) finite-state systems communicating through channels: synchronous (rendez-vous) or asynchronous (buffered)

borrows notation for I/O operations from CSP

loosely based on Dijkstra's guarded commands

interleaving semantics: processes interleaved, statements atomic (except for the rendez-vous communication)

Promela Model

macros `#define numUsers 3`

type declarations `mtype = {get, got, ...};`

channel declarations

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

variable declarations

```
bool waitingForCheckedOut;
```

process declarations

```
proctype User(byte id) {...}
```

initialisation process `init {...}`

Defining a Process

a process type (`proctype`) consists of name; list of parameters; local variable declarations; body

```
proctype User(byte id) {
  byte edit[numFiles], registered[numFiles];
  bool waitingForCheckedOut = false;
  do
  :: (!waitingForCheckedOut) ->
       userToCC!get,id;
       doneGet: skip;
       ccToUser[id]?got;
       registered[0] = true
  :: ...
  :: (!edit[0] && !waitingForCheckedOut) ->
       waitingForCheckedOut = true;
       userToCC!checkOut,id
  od
}
```

Functioning of a Process

a process is created using the `run` statement (which returns the process id) `run User(1);`

a process may be created at any point of the execution (within any process)

a process starts executing the moment it is `run`

there may be several processes of the same type, but each process has its own local state (`proctype` location + contents local variables)

a process communicates with other processes by using channels or global—shared—variables

Types and Variables

type conflicts are detected at runtime

basic types `bit, bool, byte, short, int`

arrays `byte edit[numFiles];`
(indicing starts at 0) `bit flags[4];`

records `typedef Record { bit r1; byte r2 };`

variables must be declared `Record rec;`

variables are initialised by argument/message passing or by assignment `rec.r1 = 2`

default initial value of basic variables is 0

variables can be used in expressions

Statements

a process body consists of a list of statements

a statement is either executable or blocked; an assignment is always executable

an expression is a statement; executable if it evaluates to non-zero `2 < 3; x < 3; 3 + x`

the `skip` statement is always executable; often used to label point in code `doneGet: skip;`

the `assert(exp)` statement is always executable; if `exp` evaluates to 0, SPIN exits with an error; often used to check whether a property is valid in a state `assert(writeLock == false);`

`if`- and `do`-statements

```
if/do
:: ...
:: guard_i -> stat_i1; stat_i2; ...
:: ...
fi/od;
```

the statement is executable if at least one guard is executable; otherwise it is blocked

by non-deterministically choosing to execute one of the executable guards, the statement is executed (for the `do`-statement the choice is repeated)

the **`else`** guard in an `if`-statement becomes executable if no other guard is executable

the **`break`** statement in a **`do`**-statement is always executable and exits a **`do`**-loop

Communication via Channels

```
chan userToCC = [0] of {mtype, byte};
```

name

dimension = 0 (rendez-vous)

type of elements transmitted

```
chan ccToUser[numUsers] = [1] of {mtype};
```

array of channels

FIFO buffer of dimension = 1

Asynchronous Communication

`!` send: put a message into a channel

```
ccToUser[id]!got;
```

`got` should be of the declared type;
executable if the channel is not full

`?` receive: get a message out of a channel

```
ccToUser[id]?got;
```

executable if the channel is not empty;
the first message is removed from the channel and stored in / if equal to `got`

(message passing / message testing)

Synchronous Communication

a.k.a. rendez-vous or handshake communication

the number of elements in the channel is zero

```
proctype User(byte id)
{
  ...
  userToCC!get,id;
  ...
}
```

```
proctype CC()
{
  byte id; ...
  userToCC?get,id
  ...
}
```

if both statements are executable and both `get` and `id` match, then the statements can be executed simultaneously and `CC`'s `id := User`'s `id`

State Vector

the info to uniquely identify a system state:

- global variables
- channel contents
- for each process:
  - local variables
  - process counter

it is important to minimise the state vector:

$$\begin{array}{r} \text{state vector: } m \text{ bytes} \\ \text{state space: } n \text{ states} \end{array} \Rightarrow \begin{array}{l} \text{storing the state space} \\ \text{may require } n \times m \text{ bytes} \end{array}$$

$\Rightarrow$ SPIN provides a number of algorithms to reduce the size of the state vector

Reducing the Size of the State Vector

partial order reduction (enabled by default)
if in global state a process can execute only local statements, let other processes wait

bitstate hashing
do not store state explicitly, use only 1 bit

hash compaction; state vector compression; slicing; minimised automaton (effective but slow)

limit the number of processes: combine some

since all data ends up in the state vector:

- limit the values a variable can be assigned
- limit the dimension of (buffered) channels
- prefer local variables over global variables

Reducing the Number of States

```
atomic{run Vault(); run CC(); run User(1);...}
```

all statements are executed in a single step; no interleaving with statements of other processes

is executable if the first statement is executable

no pure atomicity: blocks if a statement blocks

```
d_step{edit[0] = true;
                  waitingForCheckedOut = false}
```

is a more efficient version of `atomic`: no intermediate states are generated and stored

may only contain deterministic steps; causes a runtime error if a statement blocks

Properties

recall model checking tools automatically verify

$$M \models \phi$$

finite-state model of some system $\nearrow$ $\nwarrow$ property in some formal notation

safety: "nothing bad ever happens"

SPIN: find a computation leading to the 'bad' thing; if there is no such computation, the property is satisfied

liveness: "something good eventually happens"

SPIN: find a loop in which the 'good' thing does not happen; if there is no such loop, the property is satisfied

$\Rightarrow$ we need a precise way to express properties

LTL – Linear Temporal Logic

introduced by Amir Pnueli in the late 1970's

a propositional logic with temporal operators

always / eventually / until / etc.

direct link with the theory of Büchi automata

$\Rightarrow$ ideal to specify liveness properties in SPIN

Syntax of LTL in SPIN

LTL formula ::=

- `true`, `false`
- propositional symbols p, q, . . .
- (f), unary f, f binary f

unary ::=

`[]`	(always, henceforth)
`<>`	(eventually)
`!`	(logical negation)

binary ::=

`U`	(strong until)
`&&`	(logical and)
`||`	(logical or)
`->`	(logical implication)
`<->`	(logical equivalence)

Temporal Semantics of LTL

if $\alpha = q_0 a_1 q_1 \cdots a_i q_i \cdots$ is a computation, then:

$$\alpha \models f \quad \text{iff} \quad q_0 \models f$$

with

$$
\begin{array}{lcl}
q_i \models [\,]\, f & \text{iff} & \forall\, k \geq i: \; q_k \models f \\
q_i \models <>\; f & \text{iff} & \exists\, k \geq i: \; q_k \models f \\
q_i \models e\, \mathtt{U}\, f & \text{iff} & (\exists\, k \geq i: \; q_k \models f \\
 & \text{and} & \forall\, i \leq j < k: \; q_j \models e)
\end{array}
$$

Typical LTL formulae

$[\,]\, p$	always p	invariance
$<>\ p$	eventually p	guarantee
$p \mathrel{->} (<>\ q)$	p implies eventually q	response
$p \mathrel{->} (q \,\mathtt{U}\, r)$	p implies q until r	precedence
$[\,]\ <>\ p$	always eventually p	recurrence (progress)
$<>\ [\,]\, p$	eventually always p	stability (no progress)
$<>\ p \mathrel{->} <>\ q$	eventually p implies eventually q	correlation

user can <u>always eventually</u> *get* a document

$[\,]\ <>$ User[pid]@doneGet

id User process   label

SPIN's Model Checking Algorithm

SPIN uses a depth first search algorithm to generate and explore the complete state space

simultaneous construction and error checking: SPIN is an on-the-fly model checker

Promela processes		property
$P_1, P_2, \ldots, P_n$		ϕ
$\Downarrow$ interleaving		$\Downarrow$
Büchi automaton		$\neg\phi$
$\Downarrow$		translation $\Downarrow$
language intersection	$\Leftarrow$	Büchi automaton
(should be empty)		(never claim)

Underlying Theory

$$\mathcal{L}(\text{model}) = \bigcup \mathcal{L}(\text{processes})$$

$$\mathcal{L}(\text{model}) \subseteq \mathcal{L}(\text{property})$$

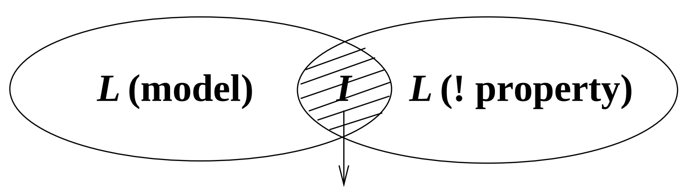
$$\Updownarrow$$

$$\mathcal{L}(\text{model}) \cap (\Sigma^\omega \setminus \mathcal{L}(\text{property})) = \varnothing$$

i.e.

$$\mathcal{L}(\text{model}) \cap \mathcal{L}(!\,\text{property}) = \varnothing$$

logical negation of the property



language intersection

$I = \varnothing$: the model satifies the property
$I \neq \varnothing$: the model can violate the property and I contains at least one counterexample

Finite Automaton

$$\mathcal{A} = (Q, \Sigma, \delta, I, F)$$

Q finite set of states

Σ finite set of letters/symbols: alphabet

$\delta \subseteq Q \times \Sigma \times Q$ transition relation $\qquad q \xrightarrow{a} q'$

$I \subseteq Q$ set of initial states

$F \subseteq Q$ set of final states

$\alpha = q_0 a_1 q_1 \cdots a_n q_n$ with $q_0 \in I$ and $(q_{i-1}, a_i, q_i) \in \delta$ for all $1 \leq i \leq n$ is a (finite) computation of $\mathcal{A}$

α is accepting if $q_n \in F$, in which case the (finite) word $a_1 a_2 \cdots a_n$ is accepted by $\mathcal{A}$

$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid w \text{ is a word accepted by } \mathcal{A}\}$ is the (finitary) language of $\mathcal{A}$

Finite Automata $\mathcal{A}$, $\mathcal{A}_1$, and $\mathcal{A}_2$

$\mathcal{L}(\mathcal{A}_1 \cup \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ polynomial

$\mathcal{L}(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ polynomial

deterministic and non-deterministic FA equally expressive, but determinisation is exponential

$\overline{\mathcal{A}}$ complement of $\mathcal{A}$
Σ alphabet of $\mathcal{A}$

$\mathcal{L}(\overline{\mathcal{A}}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$ exponential if $\mathcal{A}$ non-deterministic

$\mathcal{L}(\mathcal{A}) = \varnothing\,?$ linear

Büchi Automaton

a FA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ with Büchi acceptance:

$\alpha = q_0 a_1 q_1 \cdots$ with $q_0 \in I$ and $(q_{i-1}, a_i, q_i) \in \delta$ for all $i \geq 0$ is an infinite computation of $\mathcal{A}$

α is accepting if there exists a $q_j \in F$ that appears infinitely often in α, in which case the infinite (or ω-) word $a_1 a_2 \cdots$ is accepted by $\mathcal{A}$

the stutter extension rule is used to extend a finite computation $q_0 a_1 q_1 \cdots a_n q_n$ to the infinite computation $q_0 a_1 q_1 \cdots a_n q_n (\lambda q_n)^\omega$

$\mathcal{L}_\omega(\mathcal{A}) = \{w \in \Sigma^\omega \mid w \text{ is a word accepted by } \mathcal{A}\}$ is the infinitary or ω-language of $\mathcal{A}$

Büchi Automata $\mathcal{A}$, $\mathcal{A}_1$, and $\mathcal{A}_2$

$\mathcal{L}_\omega(\mathcal{A}_1 \cup \mathcal{A}_2) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$ polynomial

$\mathcal{L}_\omega(\mathcal{A}_1 \cap \mathcal{A}_2) = \mathcal{L}_\omega(\mathcal{A}_1) \cap \mathcal{L}_\omega(\mathcal{A}_2)$ polynomial

non-deterministic Büchi automata strictly more expressive than deterministic Büchi automata !

$\overline{\mathcal{A}}$ complement of $\mathcal{A}$
Σ alphabet of $\mathcal{A}$

$\mathcal{L}_\omega(\overline{\mathcal{A}}) = \Sigma^\omega \setminus \mathcal{L}_\omega(\mathcal{A})$ exponential, $\overline{\mathcal{A}}$ may be non-deterministic

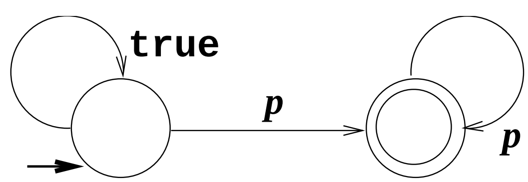
$\mathcal{L}_\omega(\mathcal{A}) = \varnothing$? linear

From LTL to Büchi Automata

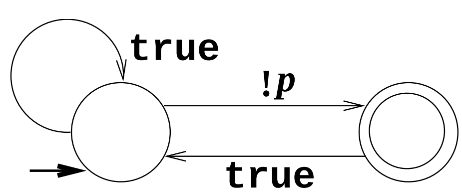
<u>theorem</u>: for any LTL formula f there exists a Büchi automaton that accepts precisely those infinite computations for which f is satisfied

Pierre Wolper, Moshe Vardi, Aravinda Sistla'83

$<>[]\,p$ corresponds to the Büchi automaton :



to turn a property into a <u>never claim</u> it suffices to negate it: $!<>[]\,p \;\equiv\; []\,!\,[]\,p \;\equiv\; []\,<>!\,p$



Xspin in brief

a nice graphical interface to help the user to :

- edit and syntax check Promela models
- simulate Promela models
  - random
  - interactive
  - guided
- verify Promela models
  - exhaustive
  - bitstate hashing
  - many options and directives to fine tune
- use additional features
  - draw automata for each process
  - easy-to-use LTL property manager
  - help (simulation & verification guidelines)

Case Study: thinkteam (TT)

think3's Product Data Management application

a dispersed & asynchronous groupware system

provides PDM needs of design processes in the manufacturing industry

strengths: rapid deployment & startup cycle, flexible, smooth integration with thinkdesign (think3's CAD solution) & 3rd party products

helps to capture, organise, automate & share engineering product information efficiently

⇒ joint work with Mieke Massink, Diego Latella & Stefania Gnesi from FM&&T; Alessandro Forghieri & Maurizio Sebastianis from think3

Vaulting

(controlled storage and retrieval of documents)

TT's vaulting subsystem:

(1) provides a single, secure & controlled storage environment, where the documents controlled by the PDM application are managed

(2) prevents inconsistent updates or changes to documents, while still allowing the maximal access compatible with the business rules

whose implementation is:

(1) subject of vaulting subsystem's lower layers

(2) in TT's underlying groupware protocol by a standard set of operations on TT's vault, a file-system-like repository

Operations on TT's Vault

get: extract a <u>read-only</u> copy of a document

import: insert an external document

checkOut: extract an <u>exclusive</u> copy of a document (with the intent to modify it)

checkIn: replace an edited (& hence previously checked out) document

checkInOut: replace an edited document (while retaining it as checked out)

unCheckOut: cancel the effects of a *checkOut*

Adding Publish/Subscribe Notification

raise user awareness by intelligent data sharing:

"whenever a user publishes a document by sending it to the vault, automatically all users that are subscribed to that document are notified via an asynchronous multicast communication"

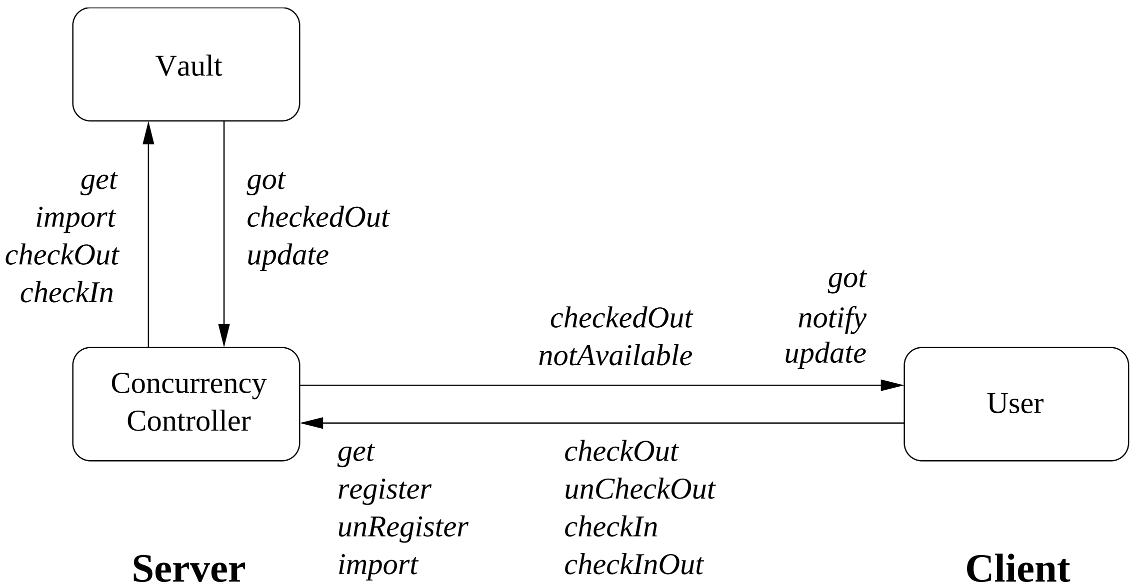
notion recently much studied in the literature:

+ "full decoupling of the communicating participants in time, space & flow" [EFGK03]

− generally difficult to verify [GKK03,ZGB03]

Aim: formally model & verify the addition of a publish/subscribe notification service to TT

The TT Protocol



every user can (un)subscribe to a document by an explicit *(un)Register* or by an implicit *get*

every user subscribed to a document receives:

- a *notify* the moment in which that document is checked out by another user

- an *update* the moment in which another user has returned that document to the vault via an *unCheckOut*, a *checkIn*, or a *checkInOut*

Most Important Assumptions

very low probability of competing user requests

$\Rightarrow$ most communication by handshake channels

there is only one document (file 0) in the vault

$\Rightarrow$ users currently cannot *import* any document

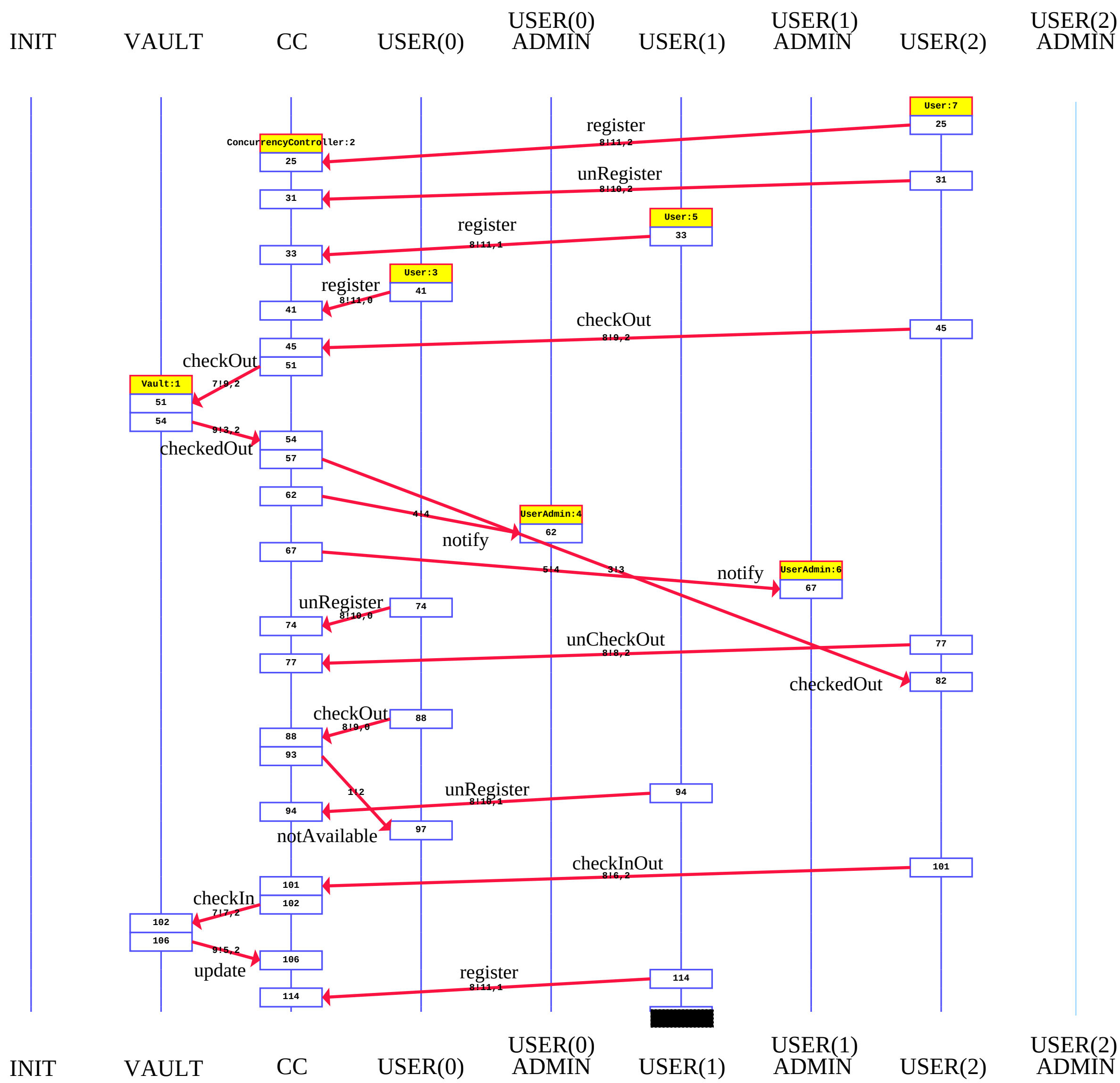
the *notify* & *update* action are always enabled

$\Rightarrow$ a UserAdmin process deals only with those

no message is ever lost

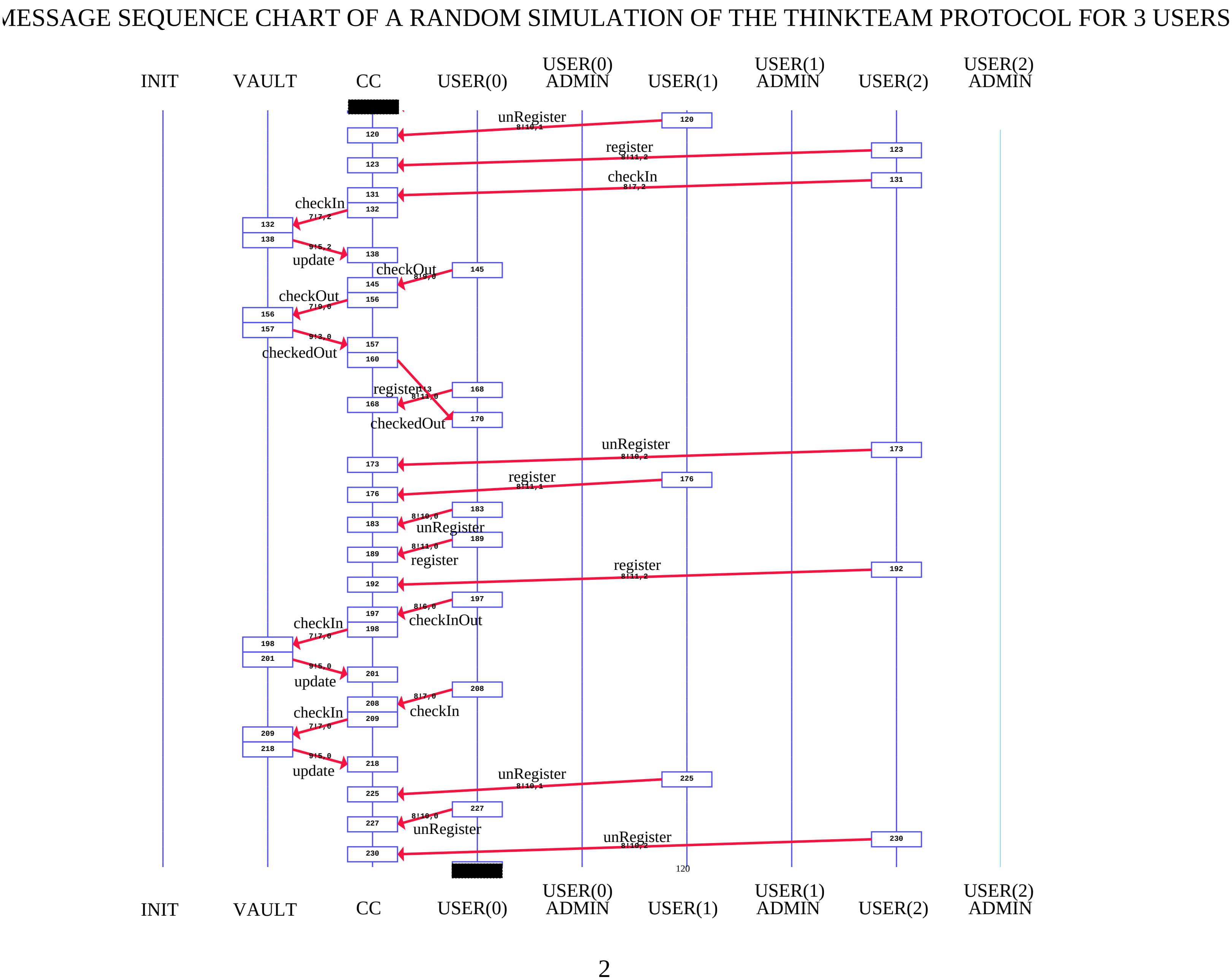
Simulation with SPIN for 3 users (1)

MESSAGE SEQUENCE CHART OF A RANDOM SIMULATION OF THE THINKTEAM PROTOCOL FOR 3 USERS



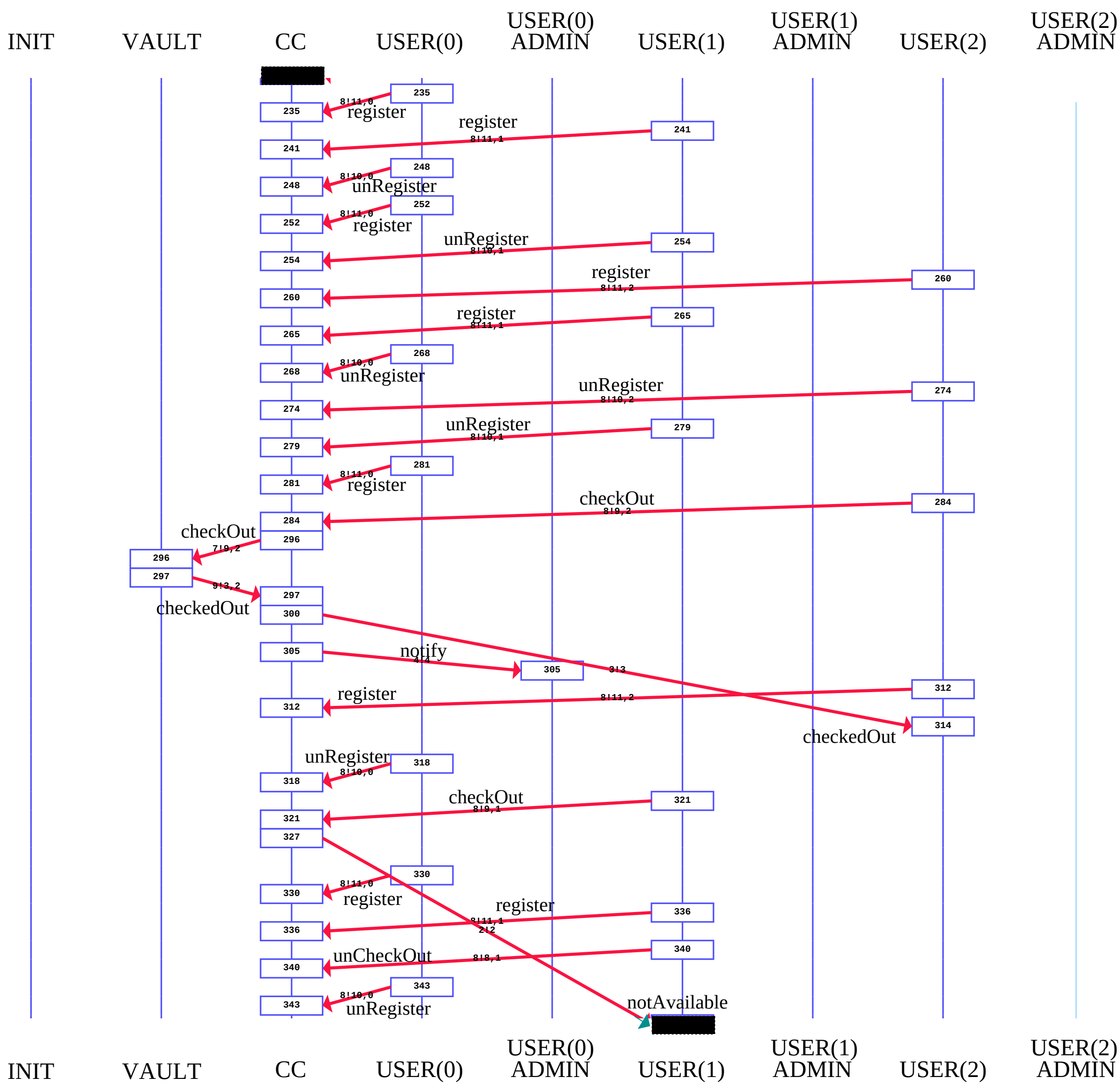
1

Simulation with SPIN for 3 users (2)



Simulation with SPIN for 3 users (3)

MESSAGE SEQUENCE CHART OF A RANDOM SIMULATION OF THE THINKTEAM PROTOCOL FOR 3 USERS



3

Validation with SPIN for 3 users

invalid endstate: SPIN's formalisation of deadlock state

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

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

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

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

[...]

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

Validation with SPIN

all verifications were performed by running SPIN Version 4.1.3 on a SUN Netra X1 workstation with 1000 Mbytes of available physical memory

full state-space searches for deadlock states:

users	state vector	depth reached	errors
2	84 byte	4423	0
3	108 byte	434033	0
4	132 byte	10484899	0

users	memory used	runtime	flags
2	37.574 Mbytes	0:0:01.3	
3	114.783 Mbytes	0:03:06.5	
4	916.095 Mbytes	8:18:36.5	-DMA = 28

(the runtime is given as hours:minutes:seconds)

Correctness Criteria of TT Protocol

Concurrency Control

1) every lock request is eventually answered
2) only one user at a time may possess a lock on a file
3) every file lock is eventually released
4) a file lock is not released after a *checkInOut*

Awareness

1) no user receives (a) a *notify* or (b) an *update* if not registered
2) every *checkOut* or *checkInOut* eventually leads to a *notify* to all (and only those) registered users
3) every *unCheckOut*, *checkIn*, or *checkInOut* eventually leads to an *update* to all (and only those) registered users

Denial of Service

1) no user is forever denied a service

⇒ formalise in LTL & verify with SPIN ! (3 users)

CC-1: Respond to Lock

"every lock request is eventually answered"

```
[] (CC[2]@doneCheckOut ->
      <> (CC[2]@doneCheckedOut ||
            CC[2]@doneNotAvailable))
```

checkOut *checkedOut* $\vee$ *notAvailable*

$\forall$comp. $\forall$ state: _____x____________x_____ _ _

SPIN: valid ! ($\pm$ 15 min.)

```
! ( <> CC[2]@doneCheckOut)
```

SPIN: <u>not</u> valid, i.e. counterexample ! ($<$ 1 sec.)

CC-3: Release File+Lock

"every file lock is eventually released"

[] (CC[2]@doneCheckedOut ->

<> (CC[2]@doneCheckIn ||

CC[2]@doneUnCheckOut))

checkIn $\vee$

checkedOut *unCheckOut*

$\forall$comp. $\forall$ state: _____x____________x_____ _ _

SPIN: <u>not</u> valid, i.e. counterexample ! (< 1 sec.)

"a user can endlessly perform *checkInOut*"

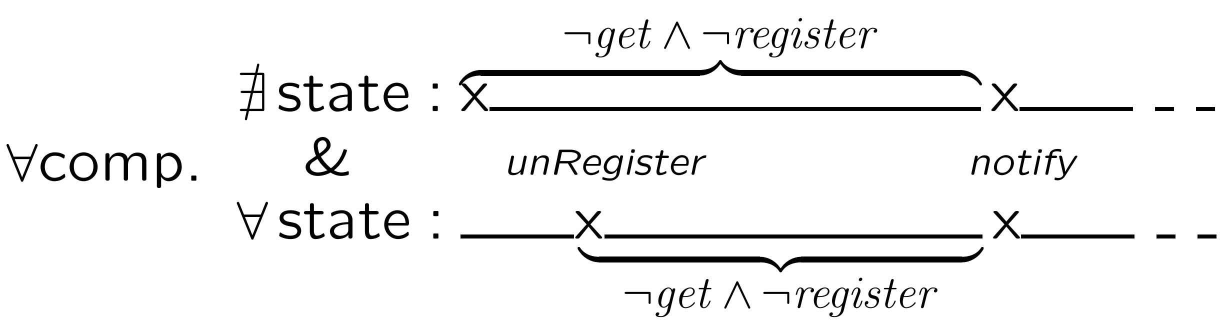
$\Rightarrow$ unavoidable property of TT protocol, which in TT practice is resolved by a 'superuser' !

AW-1a: No Illegal Notify

"no user receives a *notify* w.r.t. a file
if not registered for it"

$$\left.\begin{array}{l} !\,(\,!\,(\text{User[3]@doneGet}\,||\\[1em] \qquad \text{User[3]@doneRegister})\,\mathtt{U}\\[1em] \qquad\qquad \text{UserAdmin[4]@doneNotify})\end{array}\right\}\varphi$$

$$\&\&\ \ [\,]\ \ (\text{User[3]@doneUnRegister} -\!> \varphi)$$



SPIN: valid ! ($\pm$ 20 min.)

(User[3] & UserAdmin[4] refer to user 0, but
analogous formulae hold for users 1 & 2)

AW-2: Notify if Registered

"every *checkOut* eventually leads to
a *notify* to all (and only those) users
registered for checked out file"

$$[\,]\;!\,((\text{CC[2]@doneGet0}\,||\,\text{CC[2]@doneRegister0})$$

$$\&\&\,(<>\varphi)\,\&\&\,(\,!\,\text{CC[2]@doneUnRegister0}\ \mathtt{U}$$

$$(\varphi\ \&\&\ [\,]\;!\,\text{CC[2]@doneNotify0}))),$$

where

$$\varphi = \text{CC[2]@doneCheckedOut1}\,||$$

$$\text{CC[2]@doneCheckedOut2}$$

$$\nexists\text{comp.}\,\nexists\text{state}: \qquad \underbrace{\overset{0:\ \textit{get} \vee \textit{register}}{\times}\underline{\hspace{10em}}}_{0:\,\neg unRegister}\ \underbrace{\overset{1 \vee 2:\ \textit{checkedOut}}{\times}\underline{\hspace{4em}}\ \_\ \_}_{0:\,\neg notify}$$

SPIN: valid! ($\pm$ 40 min.)

(analogous formulae — in which users 0, 1 & 2
change roles — also hold)

DoS: Denial of Service

"no user is forever denied a service"

[] <> User[pid]@doneGet

where pid is 3 (user 0), 5 (user 1), or 7 (user 2)

$\forall$comp. $\forall$ state: $\underline{\qquad\quad}\overset{\mathit{get}}{\times}\underline{\qquad\qquad\qquad\qquad\qquad}$ - -

SPIN: <u>not</u> valid, i.e. counterexamples ! ($<$1 sec.)

"one user can endlessly keep the CC busy"

$\Rightarrow$ unavoidable property of TT protocol, due to document access based on <u>retrial</u> principle— i.e. no queue or file reservation system in TT

$\Rightarrow$ think3 interested in this for a future release !

Results of Validation with SPIN

property	depth	errors	memory used	runtime
CC-1	2703909	0	597.471 Mb	14:57.0
CC-2	434033	0	114.783 Mb	3:06.0
CC-3	310	1	353.759 Mb	0:0.7
CC-4	434033	0	114.783 Mb	3:06.0
AW-1a	3071518	0	539.769 Mb	21:22.1
AW-1b	3057025	0	558.508 Mb	22:45.4
AW-2	3338868	0	967.955 Mb	39:22.2
AW-3	4183223	0	925.049 Mb	38:57.6
DoS	123	1	33.759 Mb	0:0.1

(now the runtime is given as minutes:seconds)

+ concurrency control & awareness aspects of the TT protocol augmented with a publish/ subscribe notification service well designed !

− 'superuser' required to force a user to ever return a checked out file to the vault !

Publications

ter Beek-Massink-Latella-Gnesi-Forghieri-Sebastianis :

"A Case Study on the Automated Verification of Groupware Protocols"

(accepted for the Experience Reports Track of ICSE'05—the 27th International Conference on Software Engineering, ACM)

"Model Checking Publish/Subscribe Notification for **thinkteam**"

(FMICS'04—9th International Workshop on Formal Methods for Industrial Critical Systems, ENTCS, Elsevier)

"Automated Verification of Groupware Protocols"

(ERCIM News Special: Automated Software Engineering 58, 2004)

used experience from ter Beek-Massink-Latella-Gnesi :

"Model Checking Groupware Protocols"

(COOP'04—6th International Conference on the Design of Cooperative Systems, IOS Press)

Conclusions

- case study on formalisation & verification of concurrency control & distributed notification aspects of groupware protocol underlying TT

- show feasibility & usefulness of model checking when verifying groupware protocols in general

- among first successful applications of exhaustive model checking to verification of publish/ subscribe notification in a groupware setting

- think3 intends to use specification as basis for planned implementation of such services in TT

- think3 expressed interest in acquiring the skills to apply automated verification to the (groupware) protocols that underlie their software

Future Work

- `numFiles` > 1 in specification of TT protocol

- abandon file access based on retrial principle (i.e. handshake instead of buffered channels)

$\Rightarrow$ initial verifications show feasibility !

- extend publish/subscribe notification service so that user who checks out a file is informed automatically of existing outstanding copies

- abandon assumption "no message is ever lost" (e.g. tag messages or send redundant copies)

- perform qualitative & quantitative verification (e.g. stochastic process algebras & automata)

$\Rightarrow$ first attempt (submitted) shows feasibility !

- apply this acquired knowledge & experience to other (groupware) protocols !