Formal Methods in Practice

Marko van Eekelen

Radboud University, Nijmegen, The Netherlands

Herfstcursus Software Analysis
November 24-28, Nunspeet, NL, 2008
Laboratory for Quality Software (LaQuSo)

- **Joint** initiative of Technical University Eindhoven (Henk Schimmel, Mark van den Brand) and Radboud University Nijmegen (Marko van Eekelen)
- **Focus on Quality Assessment of Industrial Software Products**
  - This includes all kinds of intermediate Software Artefacts
  - Architecture description, Requirements specification, Technical Design, Source code, Executable
- **We are working towards**
  Standardized Software Product Certification
The recently created research group

Supervision: Marko van Eekelen,

Digital Security: Software Security and Program Correctness

- Resource Analysis
  - NWO open competition funded AHA project
  - TFP07, TLCA07, WFLP07, TFP08, ..., ..., ...
  - next step: Real-Time Java: EU incl. Luminis, NLR, TUTwente
  - Olha Shkaravska (postdoc, non-monotonic polynomial size analysis, typing, decidability)
  - Alejandro Tamalet (phd, applicability for functional and object-oriented programming)

- Software Assessment with Model Checking/Theorem Proving
  - funded by the RU and by LaQuSo projects
  - Sjaak Smetsers (lecturer, modeling, verification)
  - Leonard Lensink (phd, code generation)
  - Ken Madlener (phd, security)
  - Luc Rutten (external phd, IBM Tivoli, security)
  - Jantien Sassink (external phd, GX, web applications)
  - M.Sc/B.Sc. students, most notably: Bernard van Gastel
**Earlier Motivational Case Studies**

**FMICS2007**

  - Load Balancer for Document Production from Databases (Aia, NL)
  - Severe problems, too hard to detect by testing or code review
  - Code abstracted, **model checked**, bugs found, code improved
  - No problems since improvements were made
  - With right choice of abstraction, industrial software can be model checked close to the actual code.

  - Smart Card Personalization System (Cibernétix Group, FR)
  - Model checked heavily by others
  - **Transformed to PVS**, proven for arbitrary number of processes
  - Model checked specifications can be transformed to PVS specifications to prove general results

- Reentrant variant of classic algorithm
- Industrial Code
- First, model checked
- Then, transformed to PVS, proven for arbitrary number of processes
- Finally, derived improved code
- FMICS 2008 Best Paper Award
What is our observation?

Model checking combines well with theorem proving

- **Model checking**
  - the **modelling** process is fast
  - **right abstraction** is important
  - bugs are found, with **scenarios**
  - suffers from **state explosion**
  - proofs have assumptions: **limitations**
  - in principle code can be generated

- **Theorem proving**
  - defining a model is a lot of work
  - but, transforming a **checked model** is fast
  - proving is **tedious work**
  - model checking helps also in finding invariants
  - results have **general validity**
  - in principle code can be generated
Our proposed methodology

Software Surgery

Combining Formal Methods for Software Improvement for LaQuSo industrial software assessment and certification projects

ERCIM NEWS 75, Special on Safety-Critical Software, Oct 2008, pp. 36-37
What do we want to do next?

Towards formal methods based 'software surgery'

- study abstraction from code to model and concretization from model to code
- study structured transformations from code to model checker and to theorem prover
- study code generation and reversion of abstraction techniques
- new case studies
  - Maeslantkering Sea Barrier (Rijkswaterstaat, NL)?
  - Nuclear Reactor Software (NRG, NL)?
Reentrant Readers-Writers
– a Case Study Combining Model Checking with Theorem Proving –

Bernard van Gastel, Leonard Lensink, Sjaak Smetsers, Marko van Eekelen
Radboud University, Nijmegen, The Netherlands
IPA, 2008, Herfstcursus Software Analysis
November 24-28, Nunspeet, NL, 2008
The reentrant readers-writers problem

- **Classic**: multiple readers can read, only one of multiple writers can write, readers/writers exclude each other
- **Variant**: writers preference (possible readers starvation)
- **Reentrant**: locks can be acquired multiple times, even when the lock is not released
  - important for modular programming
  - weakly reentrant: either nested read locks or nested write locks
  - **strongly** reentrant: allow combinations such as
    - write_lock, read_lock, unlock, unlock
    - but not read_lock, write_lock, unlock, unlock to avoid deadlocks
- **C++ implementation in Trolltech’s Qt Library**
  - Qt: cross-platform application development framework
    - Nokia’s since January 2008
    - used in KDE, Opera, Google Earth, Skype, Qtopia, Photoshop Elements, VirtualBox and OPIE
The code

A snapshot of the actual code (1)

```c
struct QReadWriteLockPrivate
{
    QReadWriteLockPrivate()
    : accessCount(0),
      currentWriter(0),
      waitingReaders(0),
      waitingWriters(0)
    {
    }
    QMutex mutex;
    QWaitCondition readerWait,
    writerWait;
    Qt::HANDLE currentWriter;
    int accessCount,
    waitingReaders,
    waitingWriters;
};
```
void QReadLock::lockForRead()
{
    QMutexLocker lock(&d->mutex);
    while (d->accessCount < 0 ||
            d->waitingWriters)
    {
        ++d->waitingReaders;
        d->readerWait.wait(&d->mutex);
        --d->waitingReaders;
    }
    ++d->accessCount;
}

void QReadLock::lockForWrite()
{
    QMutexLocker lock(&d->mutex);
    Qt::HANDLE self =
            QThread::currentThreadId();
    while (d->accessCount != 0)
    {
        if (d->accessCount < 0 &&
            self == d->currentWriter)
        { break; // recursive write lock
        }
        ++d->waitingWriters;
        d->writerWait.wait(&d->mutex);
        --d->waitingWriters;
    }
    d->currentWriter = self;
    --d->accessCount;
}
void QReadWriteLock::unlock()
{
    QMutexLocker lock(&d->mutex);
    if ((d->accessCount > 0 &&
        --d->accessCount == 0) ||
     (d->accessCount < 0 &&
        ++d->accessCount == 0)) {
        d->currentWriter = 0;
        if (d->waitingWriters) {
            d->writerWait.wakeOne();
        } else if (d->waitingReaders) {
            d->readerWait.wakeAll();
        }
    }
}
The Conditional Mutex library was modeled in Uppaal

First, a Basic Mutex by a single separate process that create critical sections by synchronized 'lock'-'unlock' communication
**Mutex model with conditions**

- Basic Mutex extended with 2 conditions: `readerWait` and `writerWait`
- Included `wakeAll` for `readerWait` and `wakeOne` for `writerWait`
- **Local counters** counting number of sleeping processes on the locks
Model methods lockForRead, lockforWrite and unlock

One big process model with a central Start state from which the methods can be entered

An Abort state to model the assertions

lockForRead

lockForWrite

unlock
Deadlock scenario

- Process0 enters the writelock method
- Process1 enters the readlock method
- Process1 takes a read lock
- Process1 leaves the readlock method
- Process0 attempts a write lock, is suspended
- Process1 attempts a reentrant read lock, is also suspended (writers preference)
- *Deadly embrace!*
Correct Uppaal Model

- Non reentrant read lock requests still accepted only if no writers waiting for the lock
- But **reentrant read locks always succeed** (check for process number: tid)
- Limit state space by max on depth of nested calls

Uppaal model of the correct version of lockForRead
Limitations of the model checking

What has been shown?

- Two Uppaal models: a 22 state process model and a 4 state mutex model
- For 4 processes and a maximum of 5 nested reentrant calls, the model is proven to be deadlock free
- Limits are below common values in industrial practice
- State explosion can be bypassed by transforming to a theorem prover
The PVS model

**how to create a general PVS specification**

- Create an **array of threads** with arbitrary length in PVS
- **Collapse state transitions** within a critical section into single functions
- Make system **state** and thread **info** explicit

```plaintext
ThreadID : TYPE = below(NT)
ThreadLocation : TYPE = \{ START, RWAIT, RBLOCKED, WWAIT, WBLOCKED \}
ThreadInfo : TYPE = [# status : ThreadLocation, current : nat #]

System : TYPE = [# waitingWriters, waitingReaders,
numberOfThreads : nat,
currentWriter : below(NT+1),
threads : ARRAY [ThreadID \rightarrow\ ThreadInfo] #]
```
The PVS model

**how to create a general PVS specification**

- **Step function** transforming a state into the next state
- **validState?** predicate reflecting the invariant
- **state domain** is lifted to include bottoms (undefined) for not permitted states
- **explicit interleaving**

\[
\text{step}(\text{tid}: \text{ThreadID}, \ s1, s2:(\text{validState}?)):\ \text{bool} = \\
\text{writelock}(s1, \text{tid}) = \text{up}(s2) \lor \text{readlock}(s1, \text{tid}) = \text{up}(s2) \lor \\
\text{unlock}(s1, \text{tid}) = \text{up}(s2)
\]

\[
\text{interleave} \ (s1, s2:(\text{validState}?)):\ \text{bool} = \\
\exists \ (\text{tid}: \text{ThreadID}): \ \text{step}(\text{tid}, s1, s2) \land \\
\forall \ (\text{other}_\text{tid}: \text{ThreadID}): \ \text{other}_\text{tid} \neq \text{tid} \Rightarrow \\
\ s1'\text{\_threads}(\text{other}_\text{tid}) = s2'\text{\_threads}(\text{other}_\text{tid})
\]
how to create a general PVS specification

- Direct transformations of readlock, writelock, ....

readlock(s1:(validState?), tid:ThreadID) : lift[(validState?)] =

LET thread = s1' threads(tid) IN

CASES thread's status OF

START:

IF thread's current > 0

THEN up(s1 WITH [threads := s1' threads WITH

[tid := thread WITH [current := thread's current+1]]])

ELSIF s1' currentWriter ≠ NT ∨ s1' waitingWriters > 0

THEN up(s1 WITH [waitingReaders := s1' waitingReaders + 1,

threads := s1' threads WITH

[tid := thread WITH [status := RWAIT]]])

ELSE ...

ENDIF,

RBLOCKED:

IF s1' currentWriter ≠ NT ∨ s1' waitingWriters > 0

...

ELSE:

up(s1)

ENDCASES
how to prove the PVS specification

- design sanity and safety invariants in PVS
- check them using the Uppaal model
- collect them in validState?
- prove the invariant via case distinction (400 proof commands)
The proof

the invariant

Sanity 1. \( \text{waitingReaders} = \# \text{ processes with status RWAIT or RBLOCKED} \)

Sanity 2. \( \text{waitingWriters} = \# \text{ processes with status WWAIT or WBLOCKED} \)

Sanity 3. \( \text{numberOfThreads} = \# \text{ processes with lock count } \geq 1 \)

Safety 1. any waiting process has zero current readlocks

Safety 2. if a process has obtained a write lock, then only that process can be in its critical section
The proof

the invariant

\[ s: \textit{VAR} \ \text{System} \]

\[ \text{waitingReadersInv}(s): \ \text{bool} = s`\text{waitingReaders} = \text{waitingReaders}(s) \]
\[ \text{waitingWritersInv}(s): \ \text{bool} = s`\text{waitingWriters} = \text{waitingWriters}(s) \]
\[ \text{countInv}(s): \ \text{bool} = s`\text{numberOfThreads} = \text{count}(s`\text{threads}) \]

\[ \text{statusInv}(s): \ \text{bool} = \forall(tid:\text{ThreadId}): \]
\[ \mathrm{LET} \ thr = s`\text{threads}(tid) \ \mathrm{IN} \]
\[ \quad thr`\text{status} = \text{WWAIT} \lor thr`\text{status} = \text{WBLOCKED} \lor \]
\[ \quad thr`\text{status} = \text{RWAIT} \lor thr`\text{status} = \text{RBLOCKED} \Rightarrow thr`\text{current} = 0 \]

\[ \text{writeLockedByInv}(s) : \ \text{bool} = \mathrm{LET} \ twlb = s`\text{currentWriter} \ \mathrm{IN} \]
\[ \quad twlb \neq \text{NT} \Rightarrow s`\text{numberOfThreads} = 1 \land \]
\[ \quad s`\text{threads}(twlb)`\text{status} = \text{START} \land s`\text{threads}(twlb)`\text{current} > 0 \land \]
\[ \quad \forall(tid:\text{ThreadId}): tid \neq twlb \Rightarrow s`\text{threads}(tid)`\text{current} = 0) \]

\[ \text{validState?}(s) : \ \text{bool} = \text{countInv}(s) \land \text{waitingWritersInv}(s) \land \]
\[ \quad \text{statusInv}(s) \land \text{writeLockedByInv}(s) \land \text{waitingReadersInv}(s) \]
The proof

**how to prove the PVS specification**

- **define** well founded **ordering** on the system state based on the step function
- **prove** **noDeadlock via induction** on the ordering

\[ t : \text{VAR ThreadInfo} \]
\[ \text{starting?} : \text{PRED[ThreadInfo]} = \{ t \mid t{'}\text{status} = \text{START} \land t{'}\text{current} = 0 \} \]

\[ \text{startingState}(s : (\text{validState?})): \text{bool} = \]
\[ \forall (\text{tid}: \text{ThreadID}): \text{starting?}(s{'}\text{threads}(\text{tid})) \]

\[ \text{smallerState}(s_2, s_1 : (\text{validState?})): \text{bool} = \]
\[ \text{numberWaiting}(s_2) < \text{numberWaiting}(s_1) \lor \]
\[ \text{numberWaiting}(s_2) = \text{numberWaiting}(s_1) \land \]
\[ \text{totalCount}(s_2) < \text{totalCount}(s_1) \]

**noDeadlock**: **THEOREM**
\[ \forall (s_1 : (\text{validState?})): \neg \text{startingState}(s_1) \Rightarrow \]
\[ \exists (s_2 : (\text{validState?})): \text{interleave}(s_1, s_2) \land \text{smallerState}(s_2, s_1) \]
What do we want you to remember?

- **Strongly reentrant** readers-writers is **proven** to be correct for an **arbitrary number** of processes.
- We found a bug in Trolltech’s reentrant solution in the Qt library.
- The Trolltech’s Qt library has been adapted (but it is still only weakly reentrant).
Thanks for your attention!

Questions?