Lightweight Verification of a Multi-Task Threaded Server

A Case Study With The Plural Tool

Ijaz Ahmed
Néstor Cataño

Carnegie Mellon University—Portugal Partnership
University of Madeira, Portugal

22 March, 2011
MTTS

Plural Specifications Approach

MTTS Verified Properties
  Processing Task
  Mutual Exclusion
  Absence of Deadlocks
  Destroying Acquired Locks
  Processing Database Queries
  Good Programming Practices

MTTS Design Properties
  Checking the Modularity

Conclusion

NOVABASE, Lisbon, Portugal
Multi-Task Threaded Server (MTTS)

- Task organization and distribution server
- Used in financial sector and is written by Novabase
- Organizes tasks through queues
- Schedules threads to execute the task queues
- Has 3 main components TaskRegistration, RemoteOperationControl and QueueManager
MTTS Implementation

- Implemented in three packages, mtts-api, il (intelligent lock) and server
- mtts-api defines tasks and queues
- il package implements a mutex algorithm to synchronize tasks
- server package distributes tasks to threads and uses the mtts-api and il packages
An Eclipse plug-in based on modular static analysis by CMU

A lightweight verification tool of annotated Java programs

Verify access permissions and typestates

Pre and post conditions are annotated as the specification
@Perm(requires=P, ensures=Q)

Specification contains state invariants unlike JML class invariants
Access Permissions and TypeStates

- **Access Permission**
  - Abstraction describing how objects are accessed
  - Can be used to achieve parallelism
  - 5 Types, Unique, Full, Pure, Share, Immutable

- **TypeStates**
  - Statistically checkable abstraction to model object behavior
  - Provides a way to safe operation over an object
  - Can be used to detect logical errors and reason implementation
Specifications Approach

- Specifications based
  - A technical document describing MTTS (we wrote it)
  - Reviewed code comments
  - Discussions with Novabases engineers

- Specifications order
  - mtts-api package
  - il package
  - server package
Specifications Goal

- Specifications goal is to verify domain specific properties
  - Tasks
  - Threads
  - Synchronization
  - Database
  - Design
  - Miscellaneous
Processing Task

- Task is a basic activity
- Define 4 typestates Created, Ready, Running and Finished
- We ensured that
  A task is in Ready state only if method setData() is called
  Only Ready tasks can be executed (pre condition)
  A Finished task cannot be executed again (pre condition)

- @Full(requires=Created, ensures=Ready)
  public void setData() throws Exception
- @Full(requires=Ready, ensures=Finished)
  public void execute () throws Exception
Mutual Exclusion

- Violation of property can lead to undesired result
- We ensured that
  - No two threads enter simultaneously in a critical section
  - By defining different pre- and post conditions for method `acquire()`
  - Only acquired lock can be released (pre condition)

- `@Full(requires=FStat, ensures=Acquired)`
  ```java
public void acquire () throws Exception
```
- `@Full(requires=Acquired, ensures=FStat)`
  ```java
public void release() throws Exception
```
Absence of Deadlocks

- Thread does not release an acquired lock can lead to deadlock
- We ensured that
  A thread must release an acquired lock by defining same pre and post conditions of the method
  Must have a matching acquire() and release() inside method code.

```java
@Perm(requires=MutexNotAcq, ensures=MutexNotAcq)
private void doErrorRecovery (){
    mutex.acquire();
    mutex.release();
}
```
Destroying Acquired locks

- Can lead system to deadlock or to an undefined state
- We ensured that
  Only non acquired lock can be removed by defining a precondition to the method that removes lock
- ```
@Perm(requires=full(m) in NotAcq)
public void destroy(IMutexImpl m) {}```
Processing Database Queries

- Stores (retrieves) tasks in (from) database
- Non-existing (closed) connection leads to runtime error
- We ensured this property
  By specifying standard Java database libraries
- @Perm(requires=OpenConnection, ensures=OpenConnection)
  public boolean commit() {} throws SQLException;
Good Programming Practices

- Not rely on by-default value of variables
  Can lead to undesired behavior of method
  An un initialized variable "terminate" in class ExecutionThread
  Plural catch this error for constructor (note state invariant)

- @State(name=\texttt{ThreadCreated}, inv=\texttt{terminate==false})
  @Perm(ensures=\texttt{Unique(this) in ThreadCreated})
  ExecutionThread (...){}

NOVABASE, Lisbon, Portugal
Checking the Modularity

- Modularity of design is an important property
- MTTS server runs as a system service
- We ensured that
  - The MTTS service is running if (module) MTTS server is running (state invariant)
  - The MTTS service is shutdown if (module) MTTS server is shutdown (state invariant)
@ClassStates({
@State(name= SrvcStart, inv= full(mtts) in SrvrStart),
@State(name=SrvcShutdown, inv=full(mtts) in SrvrShutdown)
})
public class ServerWrapper extends GenericServlet {
private MttsServer mtts;
@Full(ensures=SrvcStart)
public void init() throws ServletException {...}
@Full(requires=SrvcStart, ensures=SrvcShutdown)
public void destroy() {...}
}
Checking the Modularity

- Has 3 main modules, TaskRegistration, RemoteOperationControl and QueueManager.
- We wanted to ensure that:
  - The server is running if all of its modules are running (state invariants).
  - The server is shutdown if all of its modules are shutdown (state invariants).
- However we could not ensure:
  - About shutdown, as implementation code only shutdown QueueManager (next slide).
@ClassStates(
  @State(name=SrvrStart, inv=full(qm)in QMStart * full(oc)in OCStart * full(tr)in TRStart)
  @State(name=SrvrShutdown, inv=full(qm)in QShutdown * full(oc)in OCShutdown * full(tr)in TRShutdown)
)
public class MttsServer {
  @Full(ensures=SrvrStart)
  public void start() throws MttsException {...}
  @Full(requires=SrvrStart, ensures=SrvrShutdown)
  public void stop() qm.shutdown(); ??
}
We wanted to check the same modularity of design for smaller (inner) components. However, we noticed that implementation does not adhere to this property for small components.
Plural Limitations

- Plural limitations include
  - Does not provide reachability analysis to check unreachable states
  - Does not provide support to check invariants having integer arithmetic
  - Does not provide support to reason intermediate states between requires and ensures typestates
MTTS Metrics

- MTTS is a medium size concurrent industrial application
- Packages and Classes
  3 packages and 55 classes
- Methods and LoC (Lines of Code)
  376 methods and 14451 lines of code
- Lines of Specification
  546 lines of specifications
Conclusion

- Formal specification and verification of domain specific properties
- Properties like mutual exclusion and absence of deadlock is difficult to prove by regular test techniques
- Checked design inconsistencies
  - High(upper) level modules follow modularity principle
  - Inner level modules do not follow modularity principle
Conclusion

- The written specification can be used to enhance
  Software Requirements Specification Document
  Software Design Document
- The invariants can be used to develop test cases
- The invariants can be used to counter-check newly added code