



Using SPIN Model Checking for Verification of Flight Software

Peter R. Glück

Autonomy and Control Section
NASA's Jet Propulsion Laboratory
Pasadena, California
peter.r.gluck@jpl.nasa.gov

Dr. Gerard J. Holzmann

Computing Science Research
Bell Labs
Murray Hill, New Jersey
gerard@research.bell-labs.com

11 March 2002

IEEE Aerospace Conference

Big Sky, Montana



Objective

- Investigate the feasibility of using model-based verification to improve the quality and robustness of JPL flight software
- Prototype the techniques and guidelines for applying such techniques to JPL flight software
- Document results



Modern Flight Software

- Multiple threads (concurrent)
- Interaction between threads
- Geometric growth of thread interactions
- Lacks controllability for testing
- Limited observability



Model Checking

- Analyzes concurrent systems behavior
- Permits control of system interleavings
- Allows more thorough analysis
- Disadvantage: requires a model



SPIN (1 of 2)

- Based upon linear temporal logic (LTL)
- Three basic components
 - Asynchronous processes (threads)
 - Message channels (interprocess communication)
 - Data objects (variables)
- Requires a "closed" system
 - Test drivers (models of interaction with the environment)



SPIN (2 of 2)

- State-based model checker
 - Establishes an initial state
 - Explores possible future states (interleavings)
 - Guided by “correctness properties”
 - Provides counter-examples
 - Supports verification of both “safety” and “liveness” properties



Correctness Properties

- Rules that the system must follow
- “X shall exist before state Y is achieved”
- “The engine shall be enabled before a maneuver can be performed”
- Violation
 - Defective system
 - Improperly specified property
 - Incorrect model



Model Extraction

- Manual modelling is cumbersome for complex applications
 - Construction
 - Validation
 - Maintenance
- FEAVER is a 'C' code model extractor for SPIN
 - Automatically generates models from code
 - Rapid
 - Repeatable
 - Synchronized with source



Examples

- SPIN Sample
- DS1 Downlink Handshake
- DS1 Sequence Controller



SPIN Sample (1 of 2)

```
int shared = 0;
int *ptr;

void
thread1(void)
{
    int tmp;

    ptr = &shared;
    tmp = shared;
    tmp++;
    shared = tmp;
}
```

```
void
thread2(void)
{
    int tmp;

    if (ptr)
    {
        tmp = shared;
        tmp++;
        shared = tmp;
        assert(shared == 1);
    }
}
```



SPIN Sample (2 of 2)

```
1: thread1 [ ptr=&(shared); ]
2:         thread2:[ ( ptr )]
3: thread1:[ tmp = shared; ]
4: thread1:[ tmp++; ]
5: thread1:[ shared = tmp; ]
6:         thread2:[ tmp=shared; ]
7:         thread2:[ tmp++; ]
8:         thread2:[ shared=tmp; ]
pan: precondition false: (shared==1)
```



DS1 Downlink Handshake (1 of 6)

- Application to real flight code
- One Downlink process and one DownFifo process
- Try to detect a known defect
- Understand work effort involved
- Prototype the model checking process



DS1 Downlink Handshake (2 of 6)

- Issue: Real-Time Operating System
 - SPIN runs on a workstation
 - Flight code runs on VxWorks
 - Solution: Stub out O/S calls
 - Emit possible responses where useful
 - Reusable once built
 - 352 lines of functional 'C' code

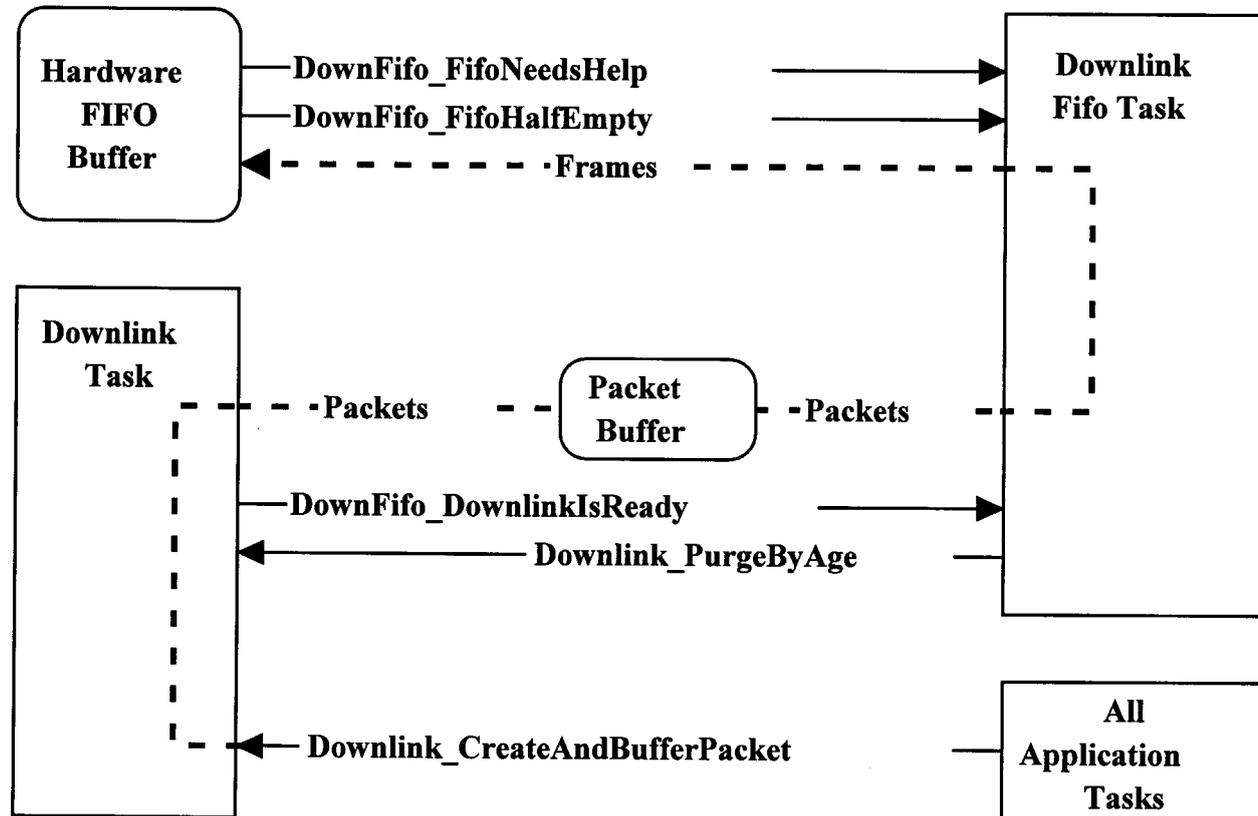


DS1 Downlink Handshake (3 of 6)

- Issue: Process and communication control
 - SPIN needs to take over process control and communication
 - Flight O/S provides this service
 - Solution: Substitute SPIN code for flight process and communication code
- Test Harness
 - 146 lines of text
 - Defines data objects and threads
 - Emits a random stream of valid input commands
 - Explores the state space



DS1 Downlink Handshake (4 of 6)





DS1 Downlink Handshake (5 of 6)

- Correctness Property
 - "It is always the case that whenever the value of **Downlink_waitingToPurge** becomes greater than zero, eventually its value must return to zero at last once."
 - In other words, **Downlink_waitingToPurge** > 0 at t_0 implies **Downlink_waitingToPurge** $= 0$ at some future t_n



DS1 Downlink Handshake (6 of 6)

- SPIN detected the known downlink handshake error
 - When the DownFifo task fails to receive the message due to queue overflow
- SPIN also discovered another scenario that violates the property
 - When a persistent stream of higher-priority messages to the Downlink task prevents processing of the purge function
 - This could occur if the processor were overutilized, or if a client task flooded the downlink with packets



DS1 Sequence Controller (1 of 3)

- Another application to real flight code
- No known defects
- One controller process and eight "engine" (execution) processes
- Apply the process discovered in the Downlink analysis
 - Reused much of the stub library
 - Test harness of similar size (141 lines)



DS1 Sequence Controller (2 of 3)

- Correctness Property
 - A sequence must become active within a finite amount of time after an activation command
- SPIN discovered a violation of this property
 - It is possible for a command to deactivate a sequence to be pending but not yet executed when an activation command for the same sequence is received.
 - The activation command will be rejected because the sequence is already active, since the deactivation has not yet occurred.



DS1 Sequence Controller (3 of 3)

- It is conceivable, though unlikely, that this could happen, because on-board autonomous fault protection uses sequences:
 - A fault occurs that causes fault protection to activate a sequence
 - While that sequence is executing, a second, more critical fault occurs, for which recovery requires the same sequence
 - When running a recovery, fault protection generally deactivates ALL sequences
 - Suppose fault protection attempted to activate the sequence again before the sequence had deactivated?
 - This possibility is not precluded



Conclusion (1 of 2)

- Model checkers can be applied to flight software
 - Detected a known error in the launch version of the DS1 flight software
 - Discovered a second scenario under which that error can occur
 - Discovered a third case in the DS1 sequencing module where a rare race condition could cause a sequence activation failure



Conclusion (2 of 2)

- Model checking verification of spacecraft flight software should include:
 - Defining and describing correctness properties
 - Constructing a test harness
 - Analyzing and interpreting results
- With the advent of model extraction, model checking requires no more effort than traditional test planning