Ames Research Center

Verifying Aerospace Software

Willem Visser

RIACS/NASA Ames
Robust Software Engineering Group

Ames Research Center

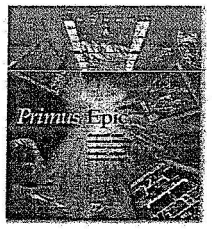
Overview

- Robust Software Engineering Group's case studies in aerospace software analysis
  - Remote Agent
  - DEOS
  - K9 Rover
- Lessons learned
- Research gaps
- Verifying autonomy software

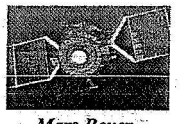
Ames Research Center

Case Studies

DEOS



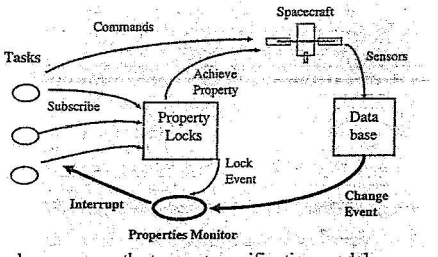
Remote Agent



Mars Rover

Ames Research Center

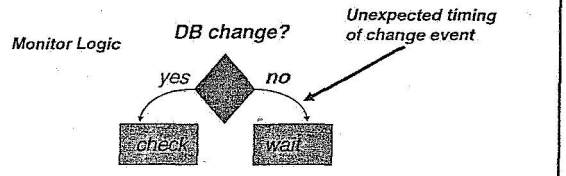
Case Study: DS-1 Remote Agent



- Several person-months to create verification model.
- One person-week to run verification studies.

Case Study: DS-1 Remote Agent

Ames Research Center



- Five difficult to find concurrency errors detected
- "[Model Checking] has had a substantial impact, helping the RA team improve the quality of the Executive well beyond what would otherwise have been produced." - RA team
- During flight RA deadlocked (in code we didn't analyze)
  - Found this deadlock with JPF

5

Remote Agent Lessons Learned

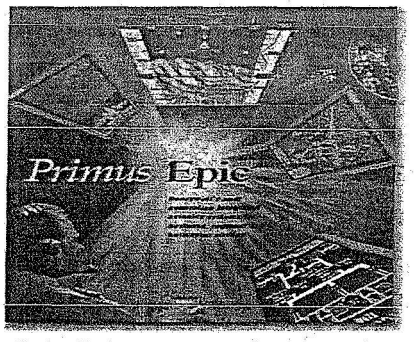
Ames Research Center

- Model checking is suitable for analysis
- Hand translation of code into notation suitable for analysis doesn't scale
  - Also error-prone
- Model checker must work on notation the program is written in

6

Honeywell Next Generation Cockpit

Ames Research Center



7

Digital Engine Operating System

Ames Research Center

- Integrated Modular Avionics (IMA)
  - DEOS Guarantee Space and Time partitioning
- FAA Certification Process
  - Requires Structural Testing Coverage (MC/DC)
  - Inadequate for finding Time Partitioning Errors
    - Timing Error not found by Testing occurred
- Behavioral Analysis of Time Partitioning
  - NASA Ames and Honeywell HTC collaboration
  - Model Check slice of DEOS containing timing error

8

Starting DEOS Analysis

Ames Research Center

- One day briefing by HTC at NASA Ames
  - System Description
  - High-level description of known Error
    - NASA team did not know what the precise error was and how to make it appear
- HTC delivered ±3000 lines of C++ code
- Model Check Actual Source Code
  - Model Extraction Requires Expert Users
  - Goal is to Improve Certification Process

9

Model Checking DEOS

Ames Research Center

- Use SPIN model checker
- Translate DEOS C++ code to PROMELA
  - Systematic Translation Process (by hand)
    - 1-to-1 Mapping: C++ to PROMELA
- Developed Nondeterministic Environment
  - Model Timer and System Ticks to Remove Real-time
    - Time Modeled by Nondeterministic Choice of Values
  - Highly Flexible Model of Threads
    - Thread creation, deletion and API calls can occur dynamically

10

Analysis Results

Ames Research Center

- Translation required 3 man-months
  - C++ translation was straight-forward
  - Environment development took most time
- Found Error by Checking Temporal Property
  - $\Box$(startPeriod $\longrightarrow$ (endPeriod $\cup$ idleRun))
    - The "idle" thread runs when nothing else can, hence time partitioning is violated if idle does not run between the start and end of a specific period
- DEOS Team Reaction
  - Surprised that error was found by directly checking code
  - They expected NASA team to ask for smaller "slice"

11

DEOS Lessons

Ames Research Center

- Model checking at source code level is feasible
- Environment creation is hard
  - To this day it is *THE* problem in model checking
- Research follow-up study
  - Translated C++ to Java and used JavaPathFinder (JPF) model checker directly
  - Showed filter-based environment generation has potential

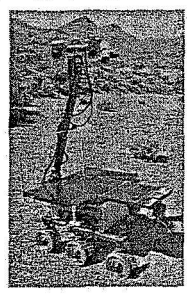
12

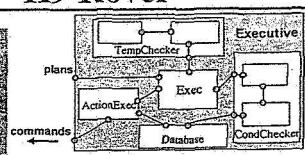
V&V Experiment

- Benchmark advanced V&V tools on autonomy software
  - Model Checking: Java Pathfinder
  - Run Time Analysis: JPaX and Temporal Rover
  - Static Analysis: PolySpace
- Objectives
  - Assess maturity / usability of each technology
  - Compare each technology with traditional testing
  - Examine whether data indicate potential synergies
  - Identify gaps with respect to autonomy V&V

13

Target system K9 Rover

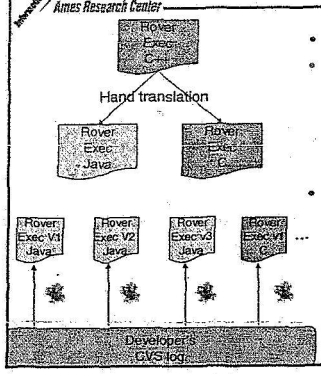




- Executes flexible plans for autonomy
  - branching on state / temporal conditions
- Multi-threaded system
  - communication through shared variables
  - synchronization through mutexes and condition variables
- Main functionality: 8KLOC, C++

14

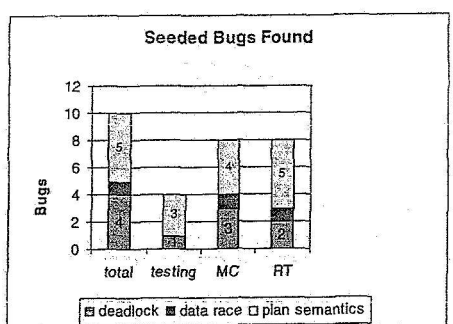
Code preparation



- Code translation for tool usage
- Seeded with 12 bugs extracted from developer's CVS log
  - Bugs are distributed over three versions of the software.
  - Some bugs appear in multiple versions.
- Bug classification
  - concurrency bugs: deadlock and data races (7/12)
  - plan bugs: plan semantics violated (5/12)

15

Bug-finding Results



16

Model Checking Setup

Ames Research Center

- "Atomic" statements added
  - Although JPF support partial-order reduction, we don't have a good static analysis tool to calculate independent transitions
  - We do now have a version of JPF that groups all transitions between synchronization statements into an atomic block
    - Do automatically what we did manually
    - Scott Stoller
- Implemented a "Factory" based infrastructure for adding abstractions
  - Abstractions play such a key role in model checking that we didn't want them to struggle with engineering issues instead of creating new abstractions
- We gave them the "point" abstraction of time
  - All time-based decisions became nondeterministic
  - It is typical to start with the most over-approximated system and use refinement as necessary
- We gave them the "Universal" planner that can create *all* plans up to a specific size nondeterministically
  - Again, common in model-checking to use a universal environment

17

Model Checking Observations

Ames Research Center

- Asked never to "run" the code, only model check it
  - Keep the results clean from any testing influence
- Performed much better than testing, and, as well as runtime-analysis
  - *Missed one concurrency error (nobody found this one) and one plan error*
- Interesting observations
  - Partially abandoned the time abstraction within the first hour for one that is closer to real-time, but might miss errors
    - It was too hard for them to determine if errors were spurious not knowing the code well enough
  - Didn't use the Universal planner as much as we anticipated
    - Rather change the training plans we gave them, probably to be more in control
  - *Lots of time spent with the heuristic options*
    - The state space is very large and heuristics were required to look at different parts
  - Found a number of bugs in the first version, had a slow 2nd version, and then found all the remaining bugs in the first part of the 3rd version
    - Took them some time to get their framework setup, but once done, they were flying
  - Found a nasty bug in floating-point arithmetic that slowed them down at the end
    - We anticipated a lot more tool errors than actually happened

18

Static Analysis Setup

Ames Research Center

- The experimental conditions for static analysis were different from those for the other tools
- PolySpace Verifier looks for run-time errors, e.g.,
  - un-initialized variables/pointers
  - out-of-bound array accesses
  - overflow/underflow
- The original C++ code was translated into C code instead of Java
- The tool had to be run overnight in a batch mode because of its slow performances
  - 4 to 8 hours for the code used in the experiment

19

Static Analysis Observations

Ames Research Center

- A priori static analysis seems easy to use:
  - You feed the program to the analyzer, and out comes a list of errors and warnings you can easily sort through
- The experiment shows that it is not that easy:
  - *Participants didn't understand how to deal with* warnings - there are many more warnings than errors
  - It is difficult to understand how approximations in the analysis algorithms impact warnings, unless one has a good understanding of the algorithms

20

Static Analysis Observations cont.

- The domain of applicability of each operation flagged as an orange (warning) should be checked in every possible execution context
  - There are too many warnings to do this rigorously
  - Participants didn't understand how, and where, to use assertions and stubs to eliminate oranges
- The participants chose to increase the number of execution paths that could be analyzed instead of analyzing the given program
  - They tried to make dead code reachable

Runtime Analysis Setup

- Java PathExplorer
  - Required no setup. Instrumentation is automated. No specification or program manipulation is required.
- DBRover
  - Rover code was pre-instrumented to emit events of the form (for actions 'a' and time points 't'):
    - start(a,t), success(a,t) and fail(a,t).
  - Users had to write a set of temporal formulae for each plan. This was time consuming.

Java PathExplorer
Deadlock and Data race Detection

- Students quickly learned to use tool. Interpretation of results required some training.
- Users found it very easy to apply tool. They applied it instantly when they got a new version of the code, and then with regular intervals.
- Tool found all seeded resource deadlocks and the seeded data race, and quickly. Tool is not designed to find communication deadlocks. Did therefore not find any.
- No false positives or false negatives.
- Extension of tool to handle some communication deadlocks is under way.

DBRover
Temporal Logic Monitoring

- Students quickly learned to use DBRover. Perhaps because temporal properties followed templates.
- Users found it slightly inconvenient to write specs for new plans. DBRover was therefore only used sporadically.
- Most plan errors (excluding deadlocks and data races) were found by examining printed information. Some were found due to violated temporal properties.
- Automatic generation of specs from plans would have made DBRover a clear success (in the users own words). In particular combined with a universal planner. This work is now being done.

Testing Observations

- Black box approach. Test cases constructed from *the plan specification.*
- Maintained a test suite and performed regression testing on each version.
- Looked for concurrency errors and the results of jitter by setting task durations and deadlines to nearly equal values
- Ran software on multiple platforms and modified the task priorities.

25

Static Analysis Successes

- Successfully applied Polyspace and CGS to MER rover code
  - Expert users in both cases
- JPL study found Coverity to be very good
  - Unlike Polyspace and CGS, Coverity is unsound (can miss errors)
  - Very good at ranking errors and report few false positives/warnings
- Path sensitive and unsound seem to be better than abstract interpretation based path insensitive sound analyses

26

Model Checking Research Gap

- Good at control analysis, but doesn't scale to data
- Need to keep control concrete and reason symbolically about data
- Our attempt at addressing this issue:
  - JPF now supports symbolic execution of structures, integers and strings

27

Static Analysis Research Gaps

- Suffers from too many false positives
- Path sensitivity is good, but these analyses are often unsound (due to scaling issues)
- Even the path sensitive analyses don't produce concrete paths to the errors
  - We hope to address this soon in a specialization of the JPF model checker to do path sensitive analysis for finding runtime errors in Java

28

General Problem

Ames Research Center

- Model checking and static analysis do best with mechanical (non-functional) properties
  - Model checking: concurrency errors, such as deadlock and data races
  - Static analysis: runtime errors, such as, null pointer dereferences, array out of bounds, etc.
- These are the "simple" bugs, but the real problems will come from the functional defects
- Suggestion
  - Use model checking and static analysis to derive "good" test inputs and then use advance runtime monitoring during testing

29

Autonomy V&V

Ames Research Center

- On-board autonomy
  - Remote Agent experiment a success
  - Mission managers are still skeptical
- Planning & Scheduling on Earth used to schedule Mars Exploration Rovers daily activities
- New projects
  - "A Model of Cost and Risk for Autonomy"
  - "Verifying Autonomy Software"

30

Autonomy Risk Model

Ames Research Center

- V&V risks and mitigations
- Project Plan
  - V&V survey to find what the autonomy experts think are the risks and current mitigations
    - Model verification, large environments, etc.
  - Case studies on autonomy software
    - Real autonomy code seeded with typical bugs
    - State of the art V&V tools
  - Use results to populate risk models
    - AUTONOMO (based on COCOMO & COQUALMO)

31