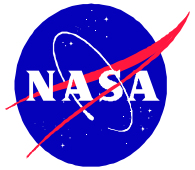
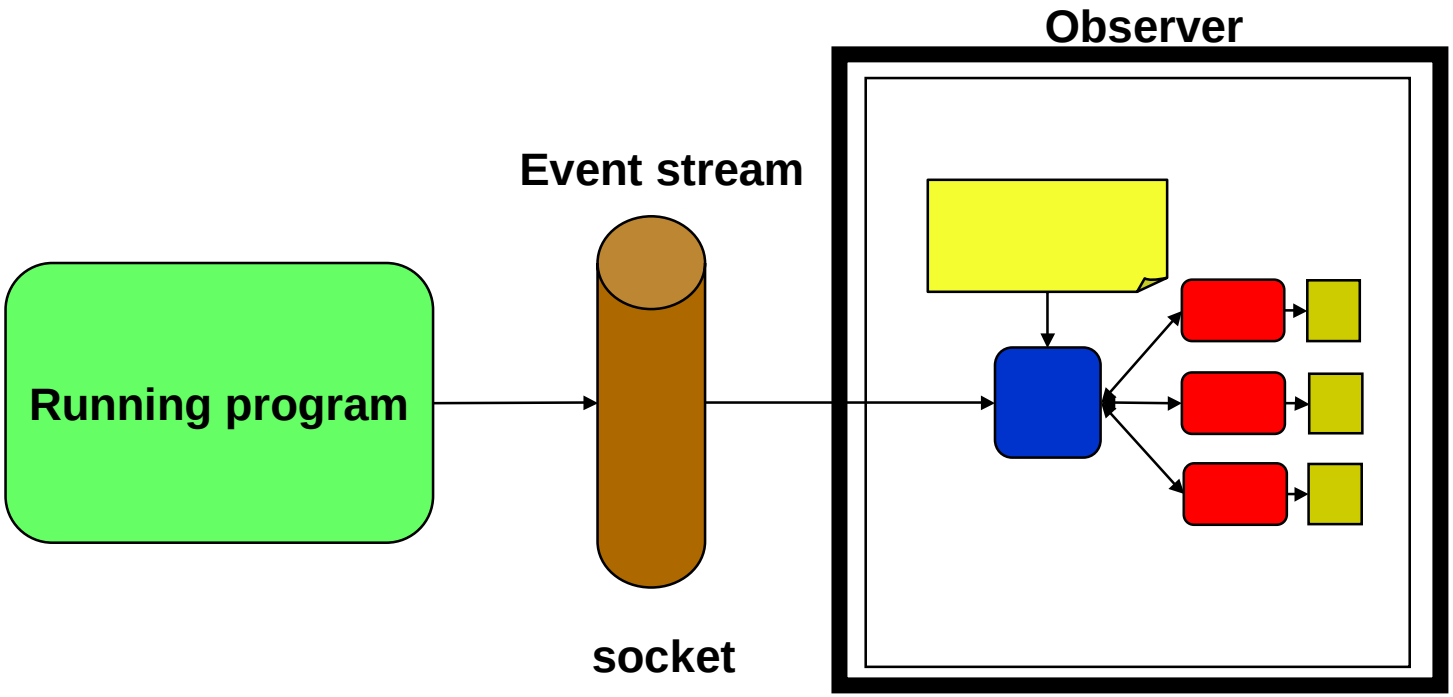
Runtime Analysis



- General Approach:
  1. Instrument program automatically to emit events
  2. Run instrumented program and extract event trace
  3. Analyze event trace with various algorithms
- Runtime Analysis Algorithms:
  - Requirements monitoring:
    - Requirements can be stated in powerful temporal logic or as state machines
  - Concurrency analysis:
    - Dataraces: low-level and high-level
    - Deadlocks: resource and communication
- Can be combined with test-case generation:
  - Purpose is to increase coverage
  - Test input generation
  - Schedule generation for multi-threaded programs

PathExplorer

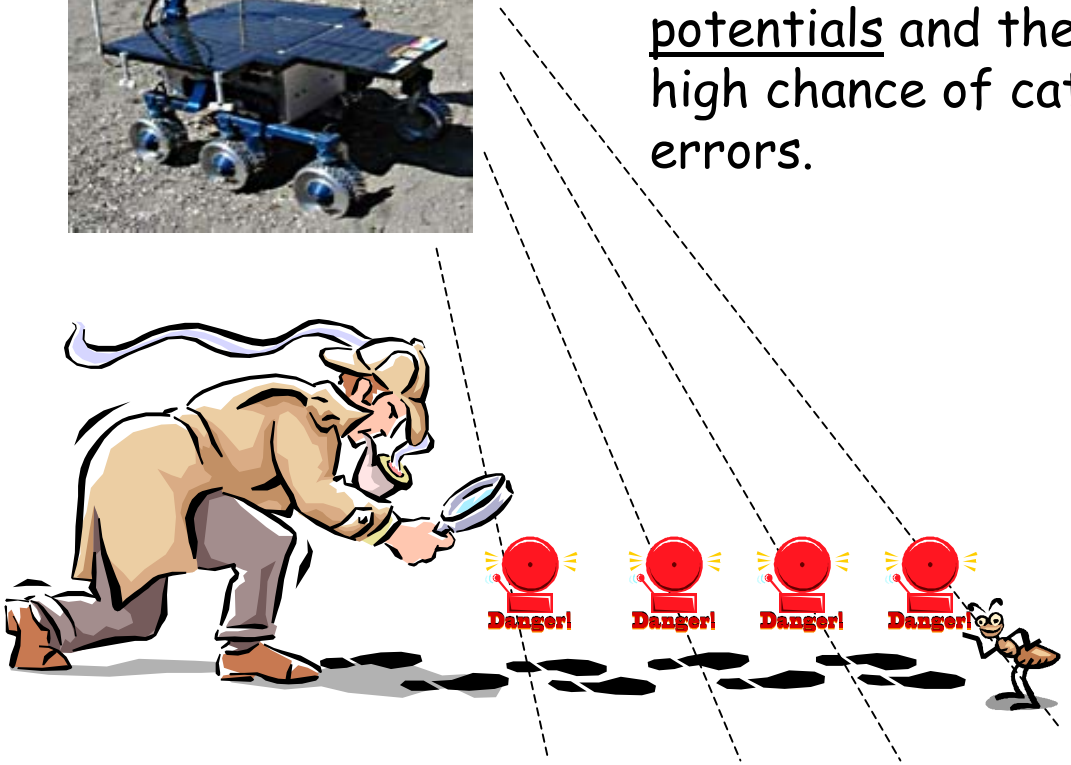


Looking for the Foot Prints Instead of for the Bug Itself

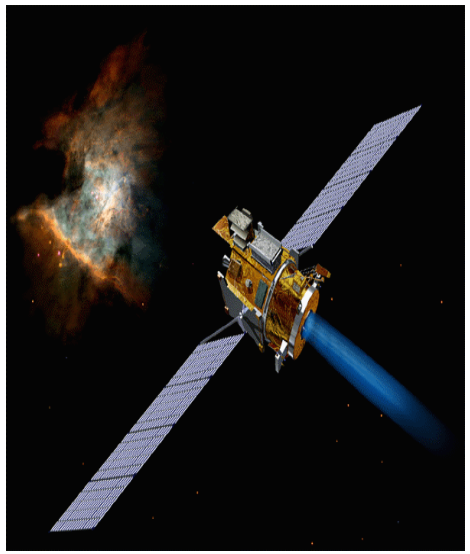
Concurrency Analysis:

- Data races
- Deadlocks

Algorithms look for error <u>potentials</u> and therefore have high chance of catching errors.



Requirements Monitoring



always(A -> **not** C **until** B)

Formalized temporal requirement

Translates into observer

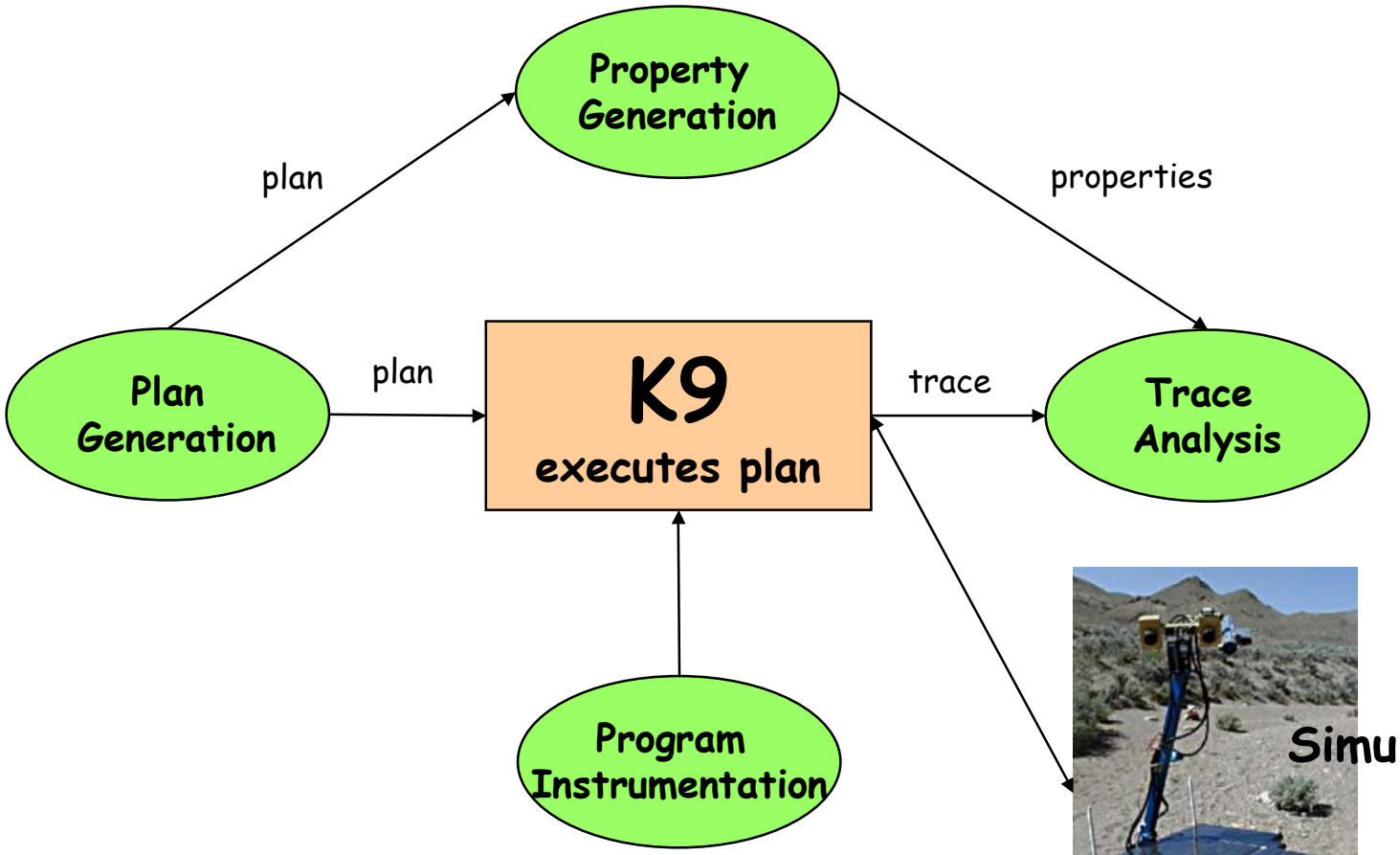
observer

events

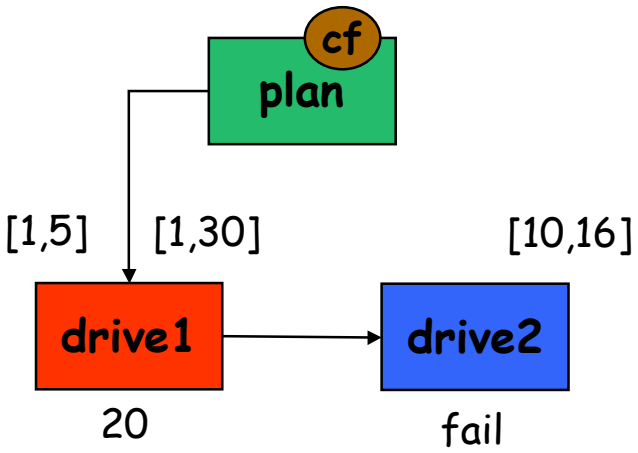
A B C A B A C

Temporal logic facilitates expression of requirements that relate a vehicle's states at different time points.

Automated Test Environment for the Planetary Rover K9



Example of Plan




```
(block :id plan
        :continue-on-failure
        :node-list (
         (task :id drive1
                :start-condition (time +1 +5)
                :end-condition (time +1 +30)
                :action BaseMove1
                :duration 20
         )
         (task :id drive2
                :end-condition (time +10 +16)
                :action BaseMove2
                :fail
         )
        ) )
```

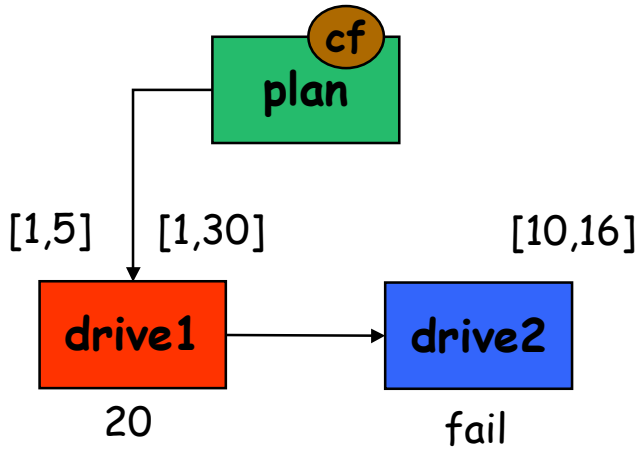
Plan Properties

◊ start(plan)

- □ (start(plan) → $\Diamond_{1,5}$ start(drive1))
- □ (start(drive1)→($\Diamond_{1,30}$success(drive1) ∨ ◊fail(drive1)))
- □ (success(drive1) → ◊ start(drive2))
- □ (end(drive2) → ◊ success(plan))

◊ success(drive1)

◊ fail(drive2)



"Demo" of K9-Explorer