Trust Your Model - Verifying Aerospace System Models with Java™ Pathfinder

Peter C. Mehlitz
PSGS / NASA Ames Research Center
<Peter.C.Mehlitz@nasa.gov>

Guillaume Brat
RIACS / NASA Ames Research Center
<guillaume.p.brat@nasa.gov>

Outline

- Motivating Example ---------- does my diagram work?
- Verification Approach ---------- how do we check?
  - Toolchain and Steps
  - UML to Java Translation Scheme
  - UML source formats
  - Testing vs. Model Checking
  - Java Pathfinder Software Model Checker
- Properties ---------- what can we check for?
  - Generic (JPF builtin)
  - UML specific (JPF UML framework)
  - Application specific
- Model Checking Modes ---------- where to check?
  - scriptless
  - guidance scripts
- Conclusions

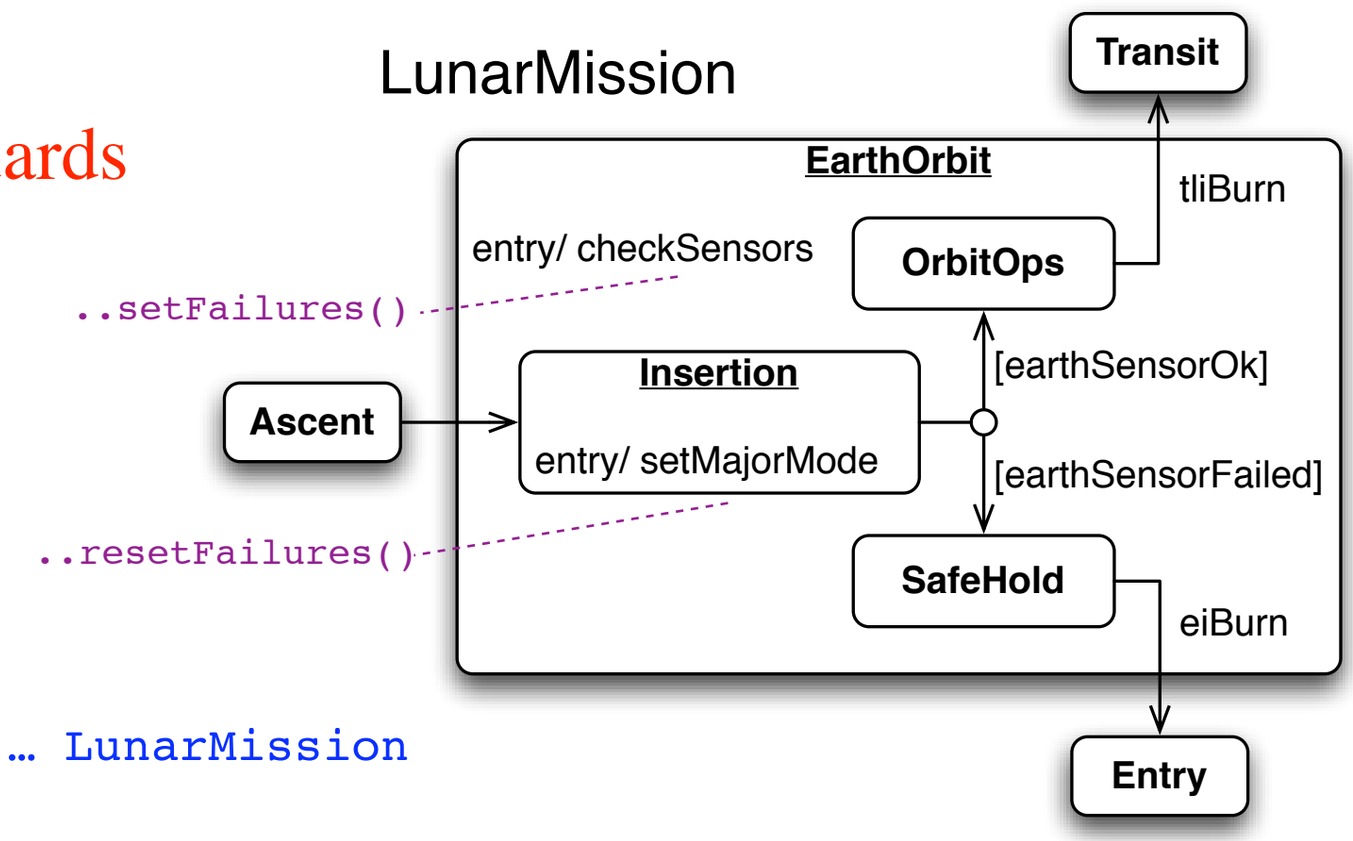
Motivating Example

- LunarMission example: all states look reachable from the diagram
- BUT diagram contains verification "loopholes": embedded code in actions and guards
- `setMajorMode()` implementation resets `earthSensor` failures → subsequent `earthSensorFailed` guard never holds

How can we make sure UML diagram with actions & guards is consistent?

make executable

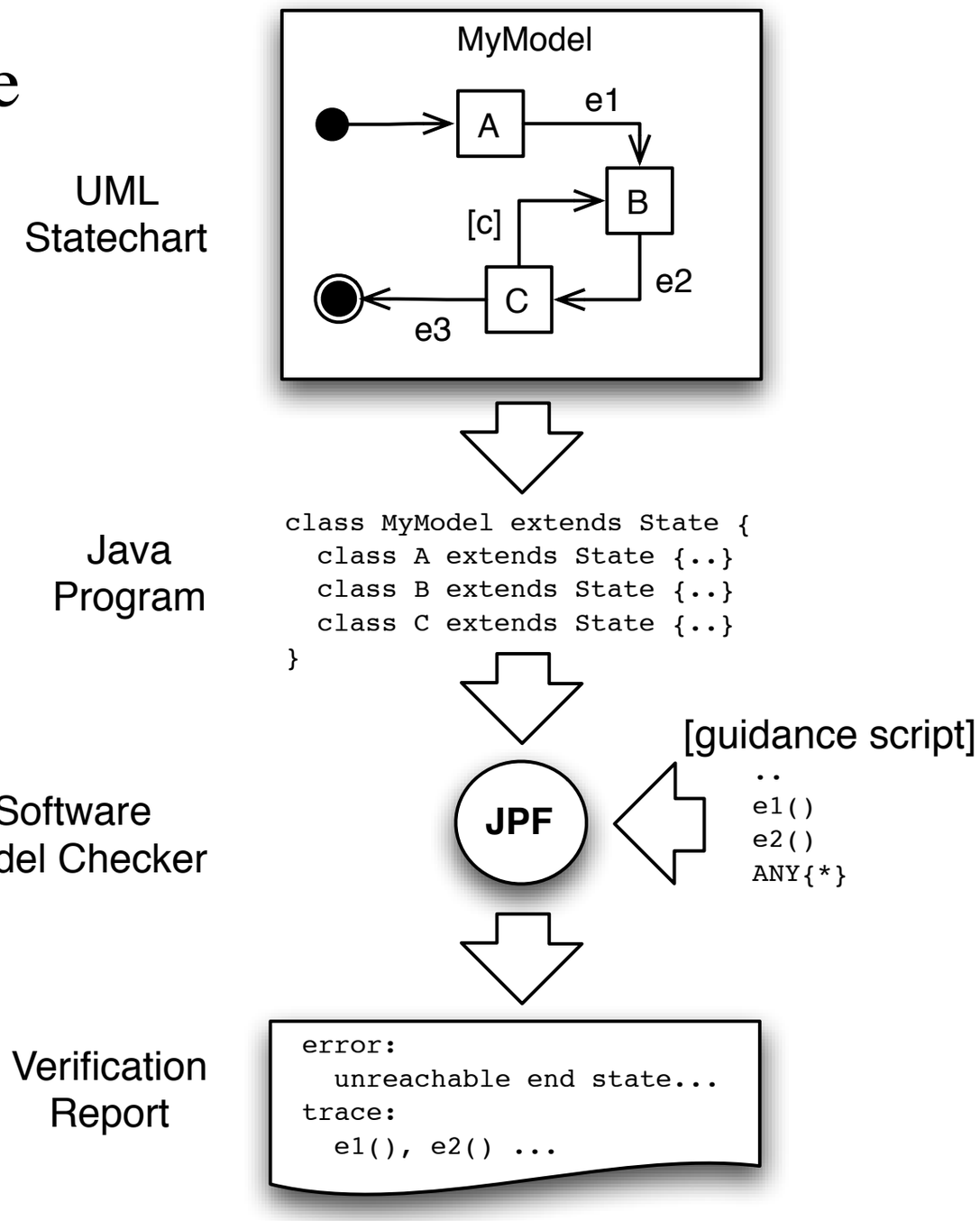
model check:




```
>jpf … LunarMission

======================== error #1
required earthOrbit.safeHold NOT COVERED
…
```

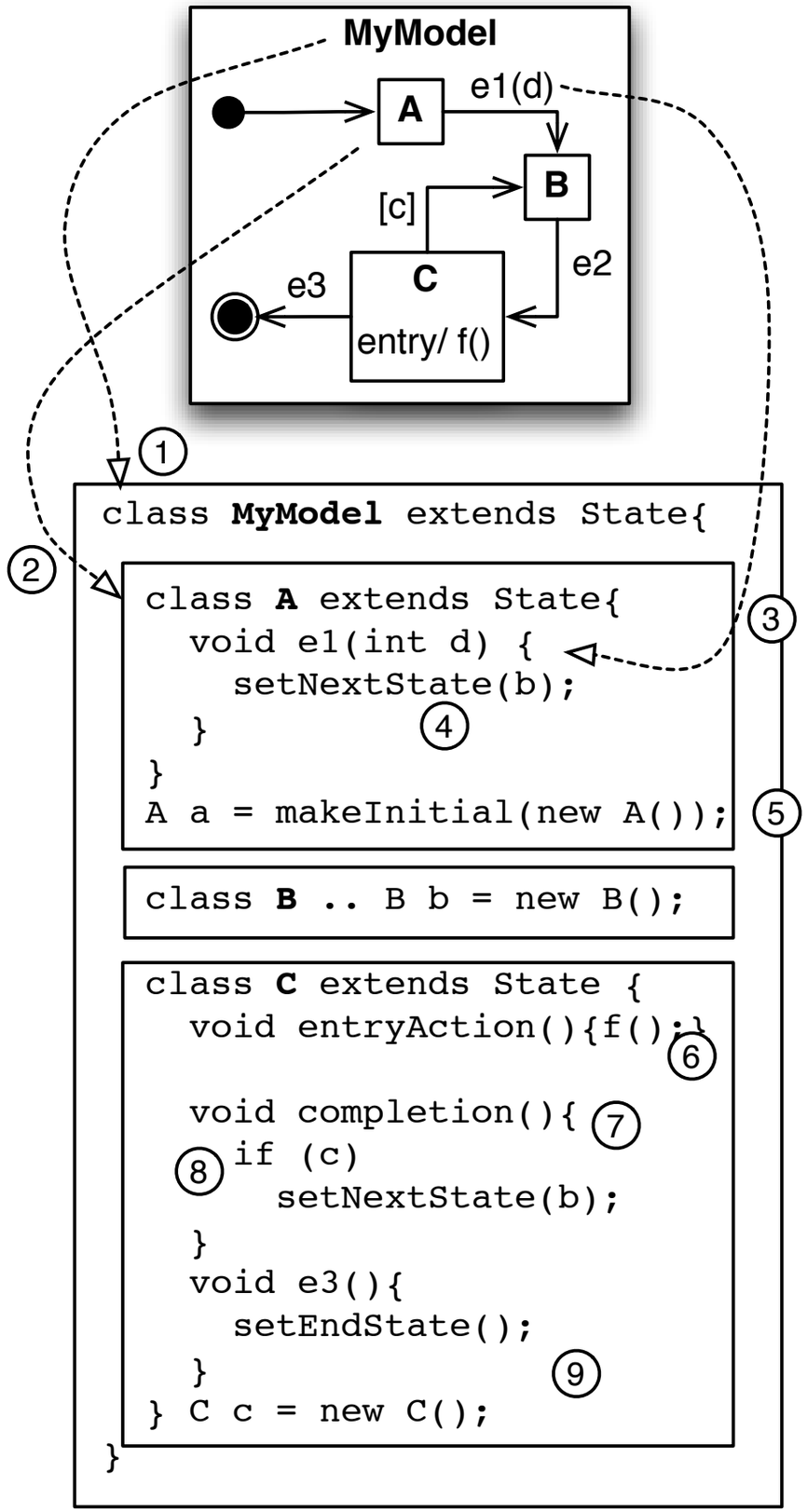
Approach (1): Toolchain

- (1) create UML statechart with diagram editor
- (2) translate diagrams into executable code (Java program)
- (3) model check Java program with Java Pathfinder (JPF)



Approach (2): UML to Java Translation

- (1) set of diagrams → one toplevel class
- (2) each substate → nested class
- (3) each trigger → Java method
- (4) transitions → `setNextState(state)`
- (5) initial states / orthogonal regions → `makeInitial(..)`
- (6) entry/ exit/ actions → corresponding `entry/exitAction()` methods
- (7) completion triggers → `completion()` method
- (8) guards → boolean java expressions
- (9) end states → `setEndState(..)` calls



Approach (3): Program Structure

- layer 1: domain model (code translated from UML diagram)
- layer 2: UML modeling library (part of JPF distribution)
- goals: model readability, UML/program state space alignment

domain model ← ***UML library***

- created from UML diagram
- structure, no policy
- no exec, no events

- execution semantics/policy
- environment
- sim & model checking

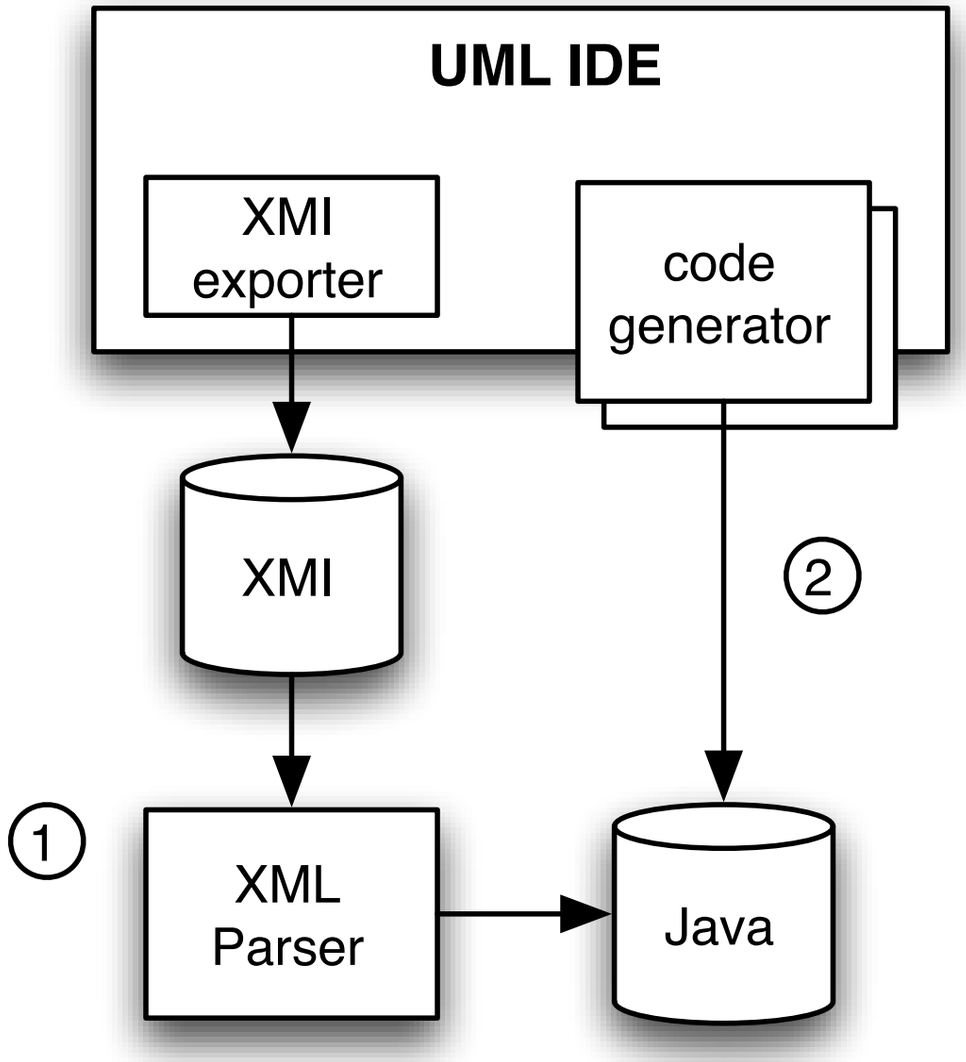
```
class MyMachine
      extends State {
  class A : State {
    void e1 () {
      if (cond)
        setNext(d)
    }
    void e2 () {..}
  } A a = new A();
  ...
}
```

```
class State {..}

class StateMachine {..}

class Event {..}

class Environment {..}

...
```

UML Java Program

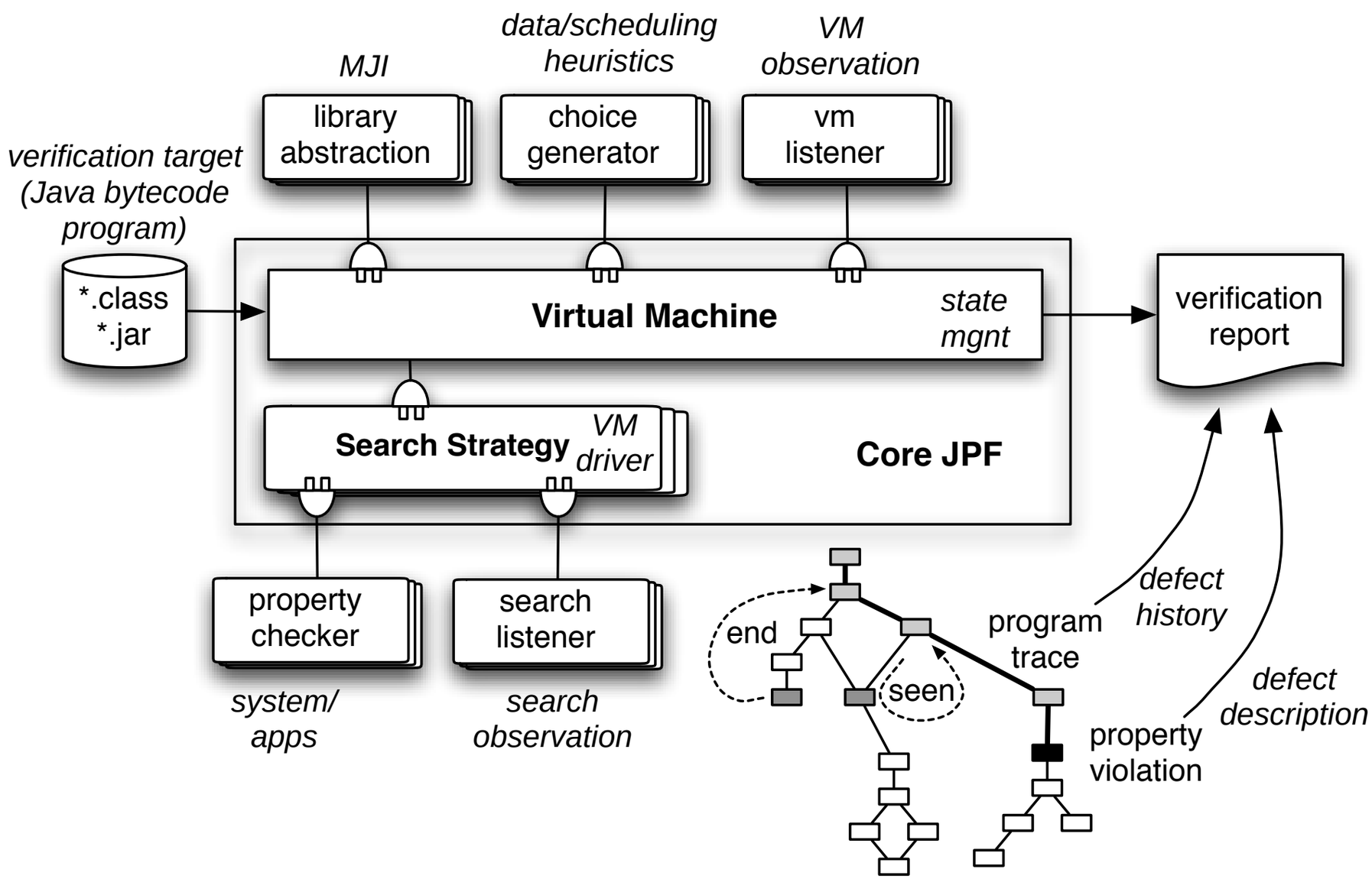
Approach (4): Code Generation

- (0) manual (possible because of low overhead of target format, can be even faster than diagrams)
- (1) via XMI (tool independent UML storage format as XML)
- (2) via IDE specific code generators



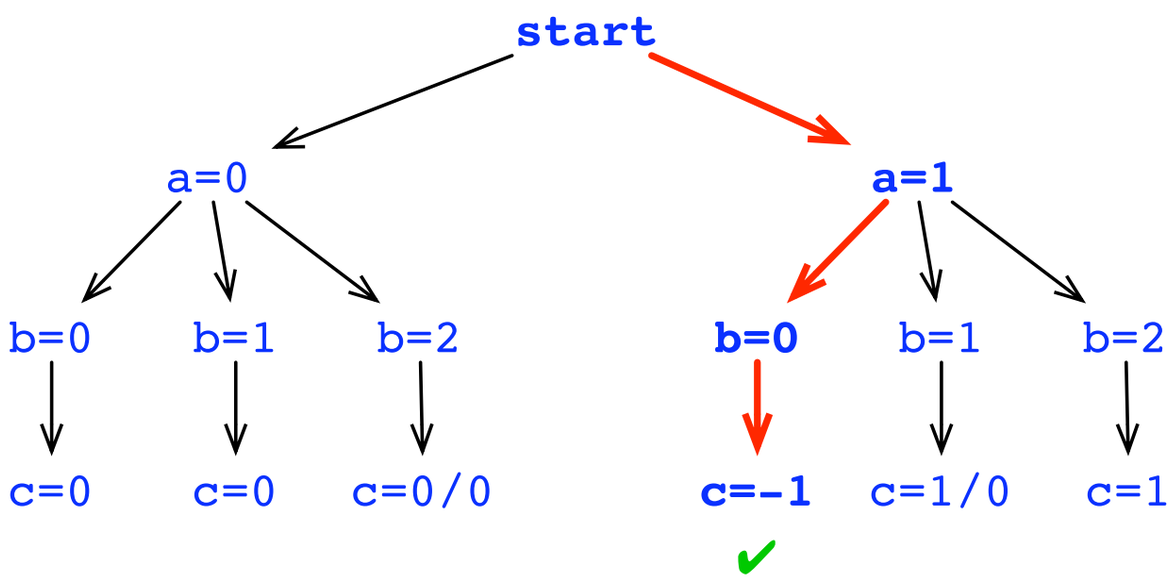
Approach (5): Java Pathfinder (JPF)

- highly configurable/extensible Virtual Machine for verification of Java bytecode
- drop in replacement for `java` command
- open sourced in 2005: http://javapathfinder.sourceforge.net



Approach (6): Testing vs. Model Checking

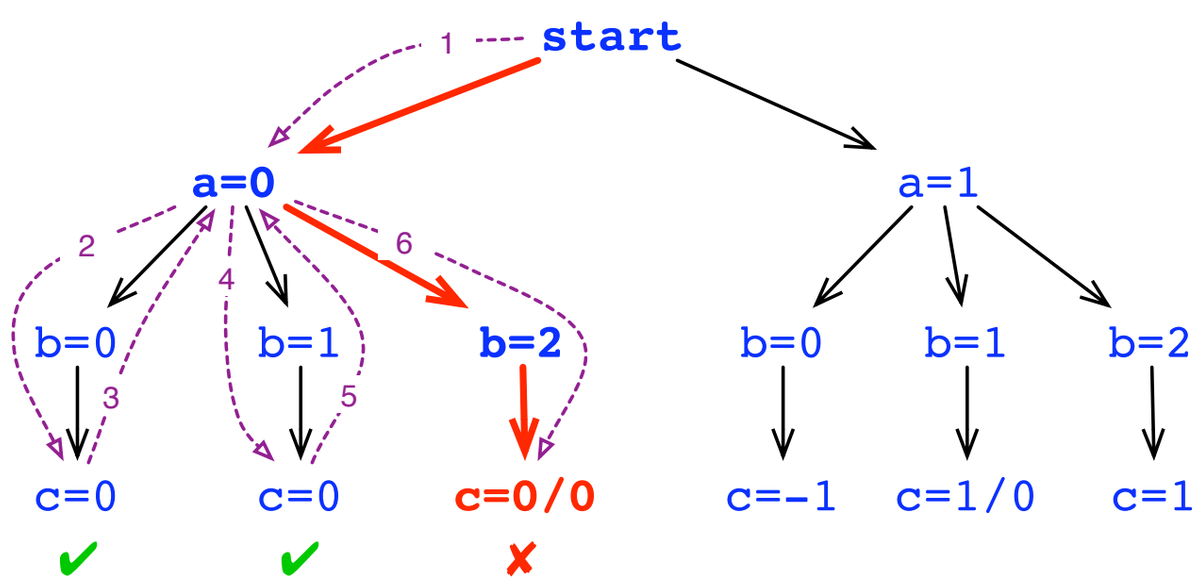
- Testing only covers one execution




```
① Random random = new Random()

② int a = random.nextInt(2)

③ int b = random.nextInt(3)

④ int c = a/(b+a -2)
```

```
>java Rand
  a=1
    b=0
      c=-1
```

- Model checking (theoretically) covers all executions




```
>jpf +cg.enumerate_random=true
        Rand
  a=0
    b=0
      c=0
    b=1
      c=0
    b=2
ERROR: ArithmeticException
```

Approach (7): Statechart Execution Semantics

- loop as long as *active* state set is not empty
- get enabling events
- loop over *active* state set
- try event on active state by executing trigger method
- if trigger fires, add target state to *next* set, otherwise add the currently processed state again
- *next* set becomes new *active* set
- continue with next step

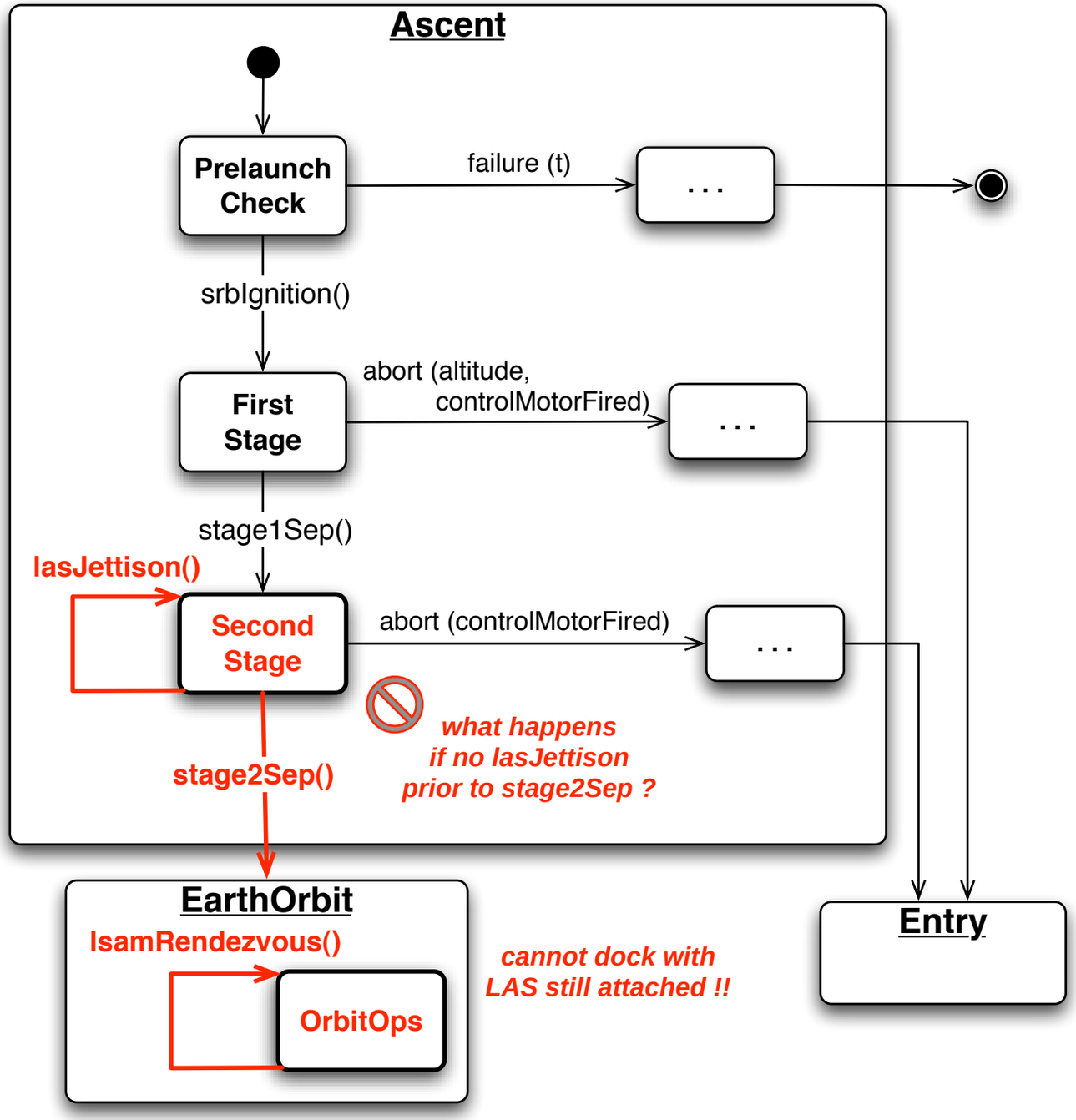
```
instantiate state machine
compute S_active (start states)

while S_active ≠ ∅
    get E_enable (S_active)
    if (E_enable ≠ ∅)
                                        model checking loop
        foreach event ∈ E_enable
            S_next = ∅
            foreach state ∈ S_active
                reset nextState
                get triggerMethod (event)
                if triggerMethod found
                    call triggerMethod

                if nextState set
                    add nextState to S_next
                else
                    add state to S_next
    S_active = S_next
```

Properties (1): Generic Safety Properties

- unhandled exception, deadlocks etc.
- do not require JPF extensions
- typically used in application as assertions:

```
class OrbitOps {..
  void lsamRendezvous(){..
    assert !spacecraft contains(LAS):
    "lsamRendezvous with LAS attached"
      ..
  } ..
}
…
```

```
============ error #1
NoUncaughtExceptionsProperty
AssertionError:
   lsamRendezvous with LAS attached
  at …
============ choice trace #1
srbIgnition()
stage1Separation()
stage2Separation()
lsamRendezvous()}
```

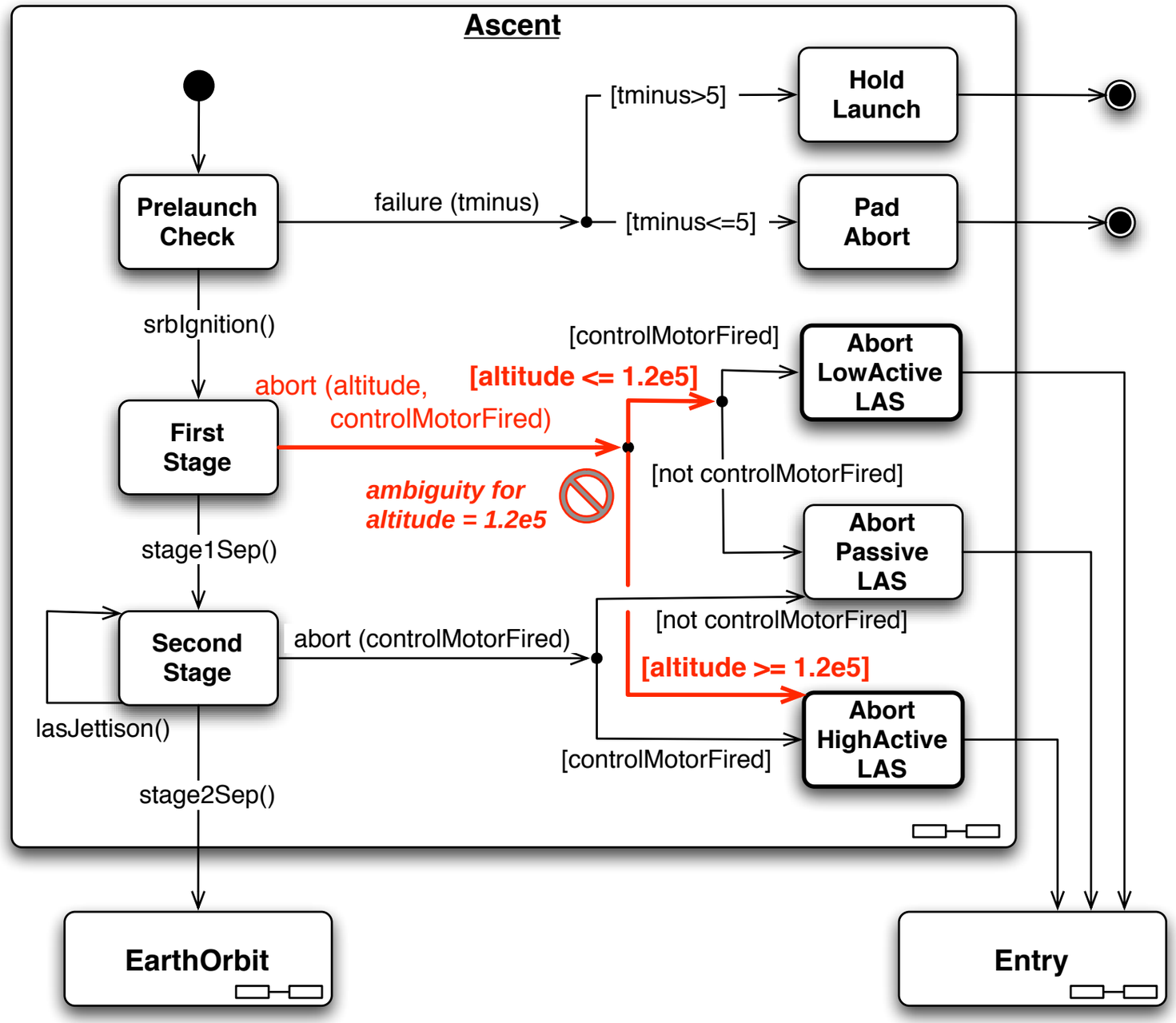


Properties (2): UML specific

- still does not require JPF extensions, checks are implemented in UML library
- e.g. "*no ambiguous transitions*"
- example: overlapping guard conditions

```
void abort(altitude,..){..
 if (altitude <= 1.2e5)
   setNextState(abortLowActive)
 if (altitude >= 1.2e5)
   setNextState(abortHighActive)
..
```

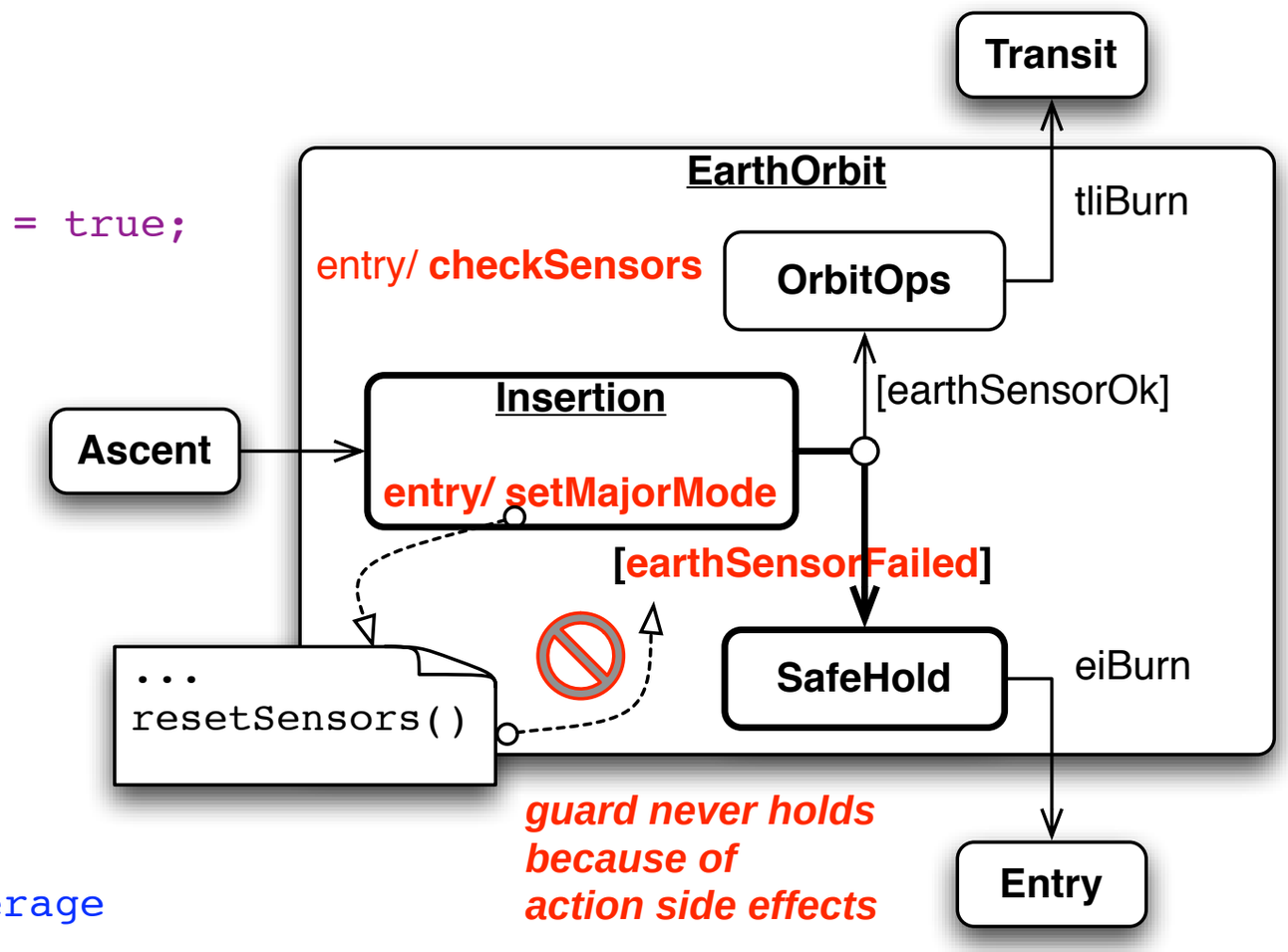
```
============ error #1 ...
AssertionError:
ambiguous transitions in:
                ascent.firstStage
processing event: abort(120000,true)
target-state 1: ascent.abortHighActiveLAS
target-state 2: ascent.abortLowActiveLAS
   at …
=========== choice trace #1
srbIgnition()
abort(120000,true)
```



Properties (3): JPF Listeners

- most powerful, can check for almost everything (temporal properties, complex domain properties with state..)
- requires JPF knowledge (uses JPF Listener extension mechanism)
- example: "*can all states of the diagram be reached?*"

```
class EarthOrbit {..
  void checkSensors() {..
    if (!checkEarthSensor()) earthSensorFailed = true;

  class Insertion {..
    void setMajorMode() {..resetSensors();..}
    void resetSensors() {..
      earthSensorFailed = false; }
    void completion() {
      if (earthSensorFailed)
        setNextState(safeHold);
    ..
  }
}
```




```
>jpf gov.nasa.jpf.StateMachine
   +jpf.listener=.tools.sc.Coverage
   +sc.required=earthOrbit…

======================== error #1
required earthOrbit.safeHold NOT COVERED
…
```

Properties (4): Listener Implementation

- Listener = callback mechanism based on *Observer* design pattern
- only presented to show required knowledge / effort
- JPFListener interface provides hooks to effectively monitor all operations of search & Virtual Machine
- need to know JPF internals (JVM, ThreadInfo, MethodInfo, ..) to get instructions, set field values etc.
- very powerful
- no need to change JPF core
- BUT: can be steep learning curve (esp. if property state is involved that has to be backtracked)

```
public class Coverage {
  …
  public void executeInstruction(JVM jvm) {
    Instruction insn =
           jvm.getLastInstruction();
    …
    if (insn instanceof RETURN) {
      MethodInfo mi = insn.getMethodInfo();
      if (mi==visitedMth) {
        ThreadInfo ti
               =jvm.getLastThreadInfo();
        int stateRef = ti.getThis();
        MJIEnv env = ti.getEnv();
        int id = env.getIntField(stateRef,
                                 "id");
        int nVisits =
               env.getIntField(stateRef,
                               "visited");
        int mRef =
             env.getReferenceField(stateRef,
                                   "machine");
        int mId =
          env.getIntField(mRef, "id");

        allCoverage[mId].addCoverage(id,
                                     nVisits);
      }
    }
  }
```

Modes (1): Model Checking Modes

- **scriptless** - automatic, (usually) exhausting model checking
  - continues until no more active UML states, or no new program states
  - in each execution step tries all events that are handled by current set of active UML states
  - most easy to use, (theoretically) finds all defects (once previous one got corrected)
  - BUT: hard to control what defects are found first (e.g. to bypass "uninteresting" ones)

- **guided** - use guidance scripts to constrain model checker choices
  - drive model checker into interesting parts of the diagram
  - good for incomplete models
  - good for working around "uninteresting" defects
  - can be used to scale model checking (restrict state space)
  - BUT: can miss defects due to "wrong" scripts

Modes (2): Guidance Scripts

- simple event sequence (no search)
  - good for testing nominal sequences

```
srbIgnition
stage1Separation
...
```

- choices
  - explicit list of event/parameter combinations
  - lexical patterns
  - '*' choices: all handled events

```
ANY { abort(100), abort(120) }
```

or

```
ANY { abort(1[024]0) }
```

or

```
ANY { * }
```

- iterations
  - bounded and unbounded (can cause infinite loops if statespace is not closed)

```
REPEAT 5 { fireThruster }
```

- sections
  - event equences to be processed when a UML state becomes active
  - hierarchical (lookup upwards from concrete active state and all it's parent states until matching section is found)

```
SECTION ascent {
  srbIgnition
  stage1Separation
  lasJettison
  stage2Separation
}
SECTION earthOrbit {
  ANY {*}
}
SECTION earthOrbit.orbitOps {
  lsamRendezvous
  tliBurn
}
```

Conclusions

- UML statecharts powerful tool to create complex models
- can include textual specifications for guards and actions
- → simple diagram walkthrough not sufficient for model verification

- our approach:
  - translate model into executable code
    (low overhead = 1:1 capture of invariant diagram structure)
  - use the JPF model checker to automatically find defects in code
  - report defect type & event sequences (= execution history)
  - verification process can be controlled with scripts

- enhances UML statecharts with well defined execution semantics and automated verification of the whole model (incl. textual specs)

Thank You!