



Open Source Software Runtime Verification

Between Testing and Model Checking

Zhe Chen

Institute of Software
Nanjing University of Aeronautics and Astronautics



Who am I?

- Assistant Professor at Institute of Software, Nanjing University of Aeronautics and Astronautics
- Ph.D. in computer science, from Institut National des Sciences Appliquées (INSA) in France (2010.07)
- Researcher in Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique (LAAS-CNRS), France (2007.09 - 2010.07)
- Researcher in Natural Language Computing Group, Microsoft Research Asia





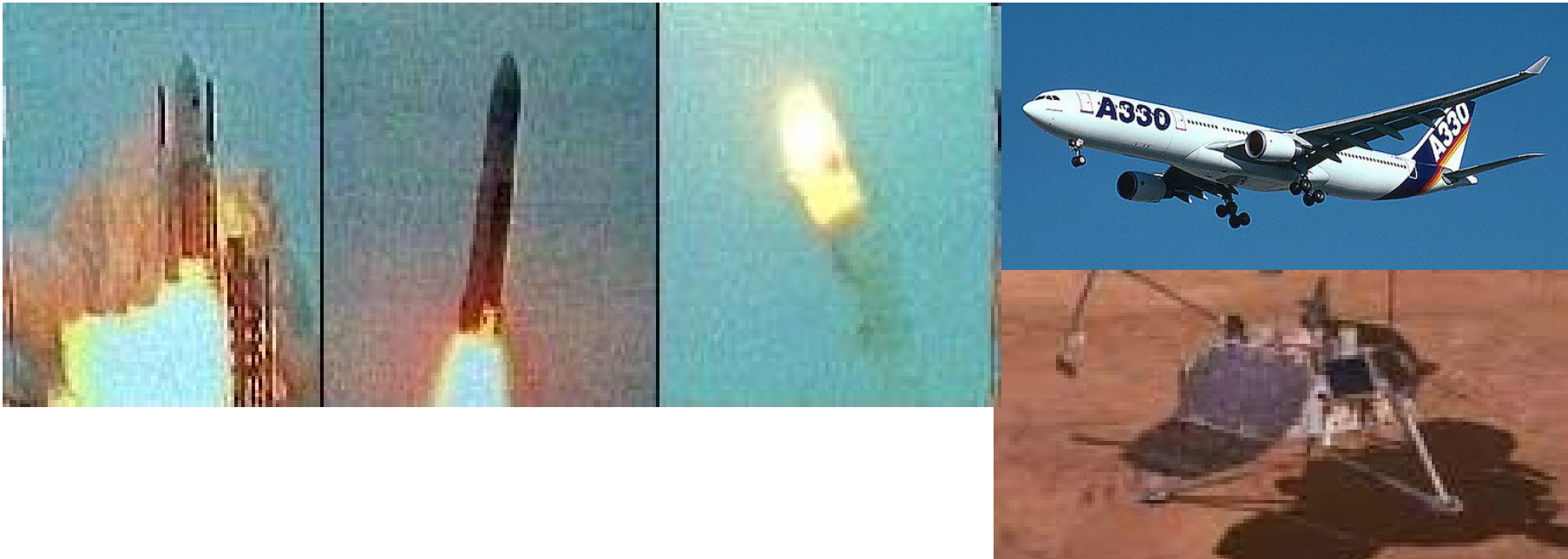
Outline

- Background
 - Software Verification
 - Testing
 - Model Checking
- Runtime Verification
 - Principle
 - Implementation
 - Code Instrumentation
 - Constructing Monitors
- Conclusion

Verification is important

■ Avionic accidents

- The Mars Polar Lander (U.S.) – crashed
- Ariane 5 (France) – self-explosion
- Airbus A330 (France) – plunged
- ...





Verification is important

- Avionic accidents
 - The Mars Polar Lander (U.S.) – crashed
 - Ariane 5 (France) – self-explosion
 - Airbus A330 (France) – plunged
 - ...
- 80% of the accidents are caused by software faults
- We need software engineering techniques to verify correctness



Verify Open Source Software

- Provide reliable and stable software
 - Operating Systems: Linux
 - Compilers: GCC
 - IDE: Eclipse
 - Office suites: OpenOffice





Verification Techniques

■ Review and analysis

- Qualitative assessment of correctness.
- Easy to use.
- Doesn't scale well – manual analysis.
- Expensive – cost time and human resources.

■ Simulation

- Simulate executable object.
- Checks a simulation, not an implementation.
- Hard to find concurrency and algorithm faults.
- Informal, doesn't provide guarantees.

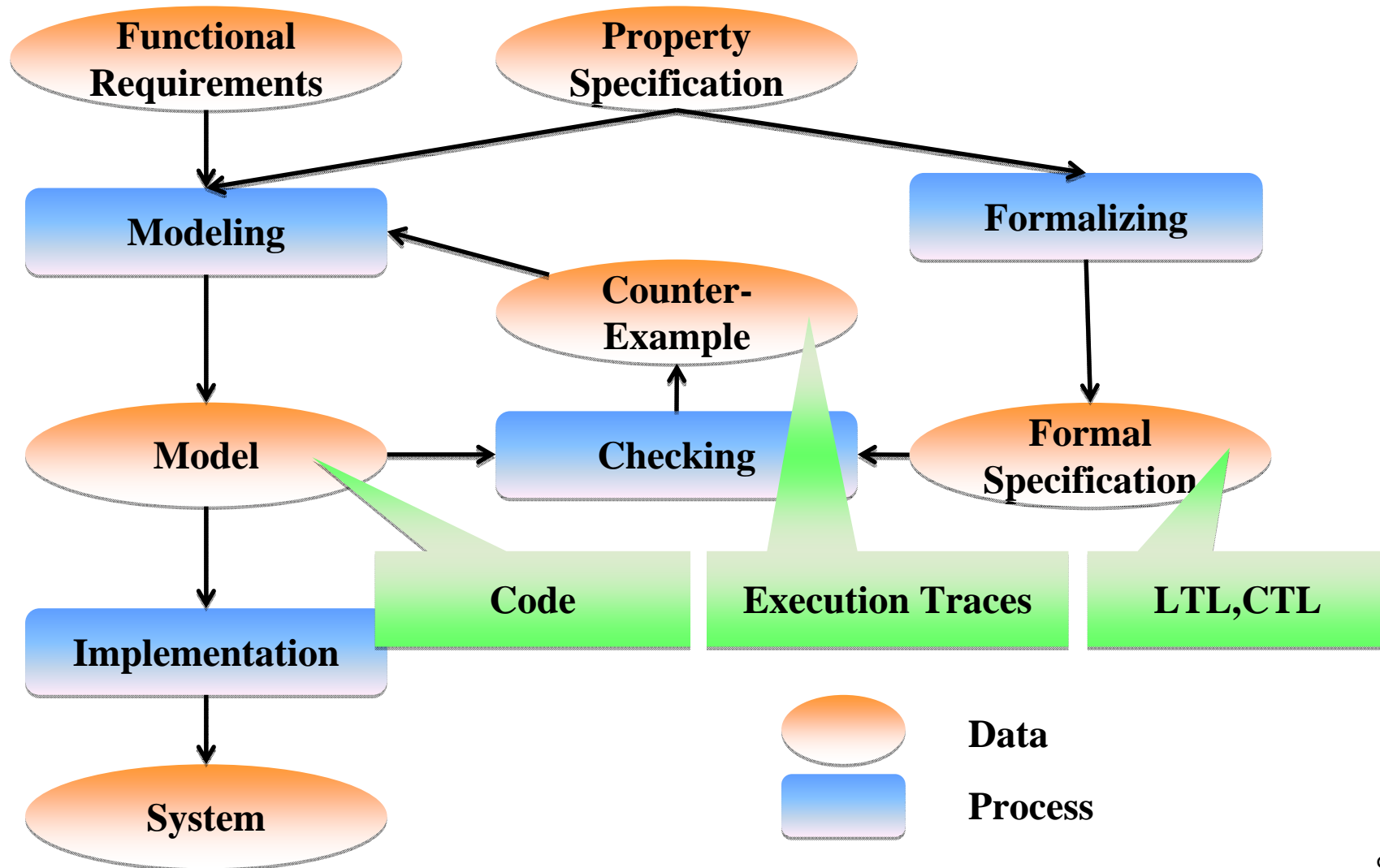


Verification Techniques

- Software testing (ad-hoc checking)
 - Most widely used technique in the industry.
 - Scales well, usually inexpensive.
 - Test an implementation directly.
 - Informal, doesn't provide guarantees.
- Model checking
 - Formal, sound, provides guarantees.
 - Doesn't scale well - state explosion problem.
 - Checks a model, not an implementation.
 - Some people fear it – too much formalism.



Principle of Model Checking





Formal Verification Tools

- Static analysis - examining the code without executing the program
 - Astree – runtime errors of C programs
 - aiT – worst case execution time
 - Stackanalyzer – upper bound of memory
- Model checking – exhaustive checking
 - SPIN – temporal property
 - SMV – temporal property
 - UPPAAL – real-time property

Automatic!!



Outline

- Background
 - Software Verification
 - Testing
 - Model Checking
- Runtime Verification
 - Principle
 - Implementation
 - Code Instrumentation
 - Constructing Monitors
- Conclusion



Runtime Verification

- Attempt to bridge the gap between formal methods and software testing.
 - A program is monitored while it is running and checked against properties of interest.
 - Properties are specified in a formal notation.
 - Dealing only with finite traces.
- Considered as a light-weight formal method technique.
 - Testing with formal “flavour”.
 - Still doesn’t provide full guarantees.

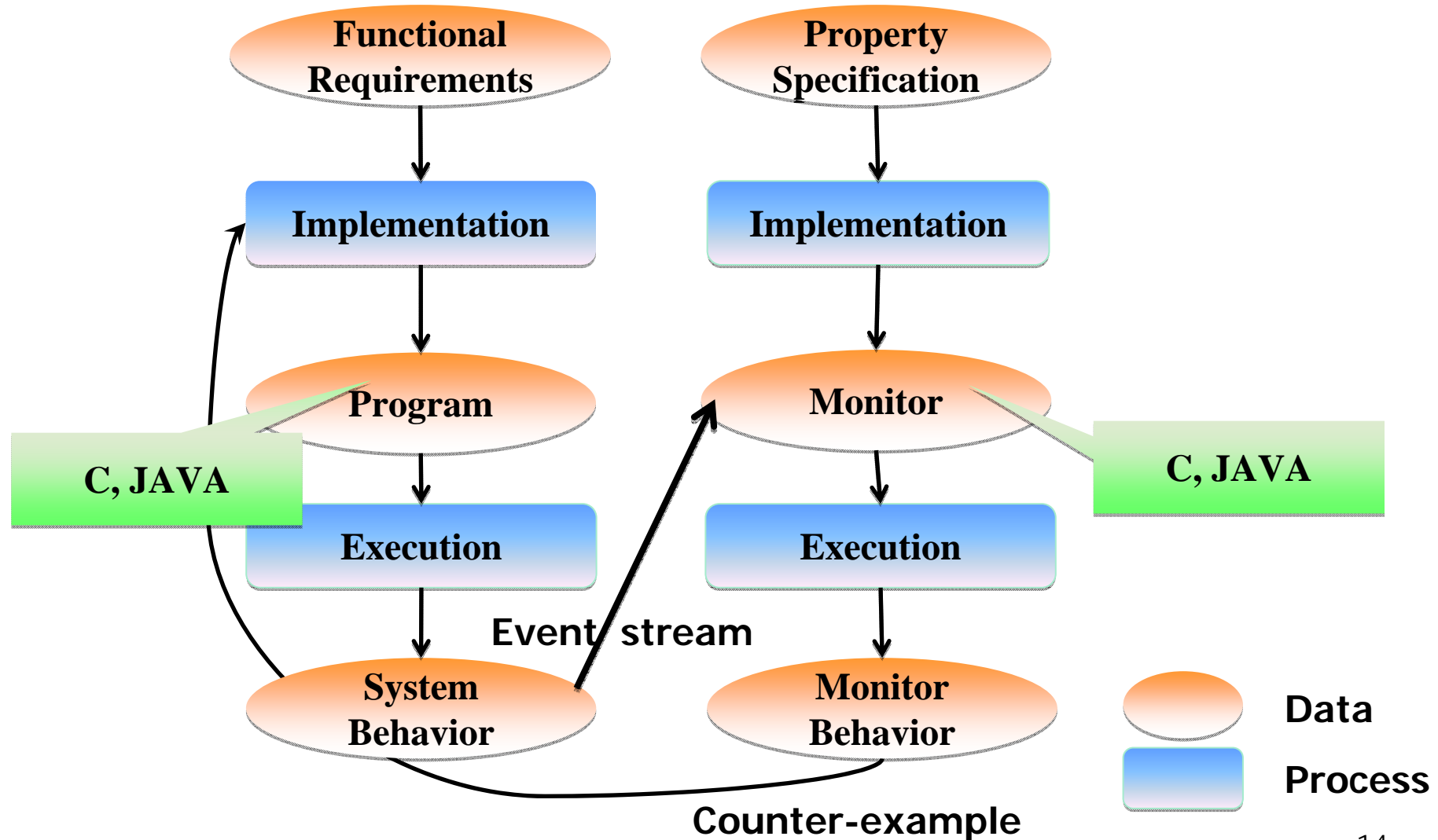


Runtime Verification

- **Observe** a run of the system
 - By code instrumentation
- **Check** it against desired properties
 - Explicit or implicit properties
- **React / report** if needed
 - Error messages, throw exceptions, recover

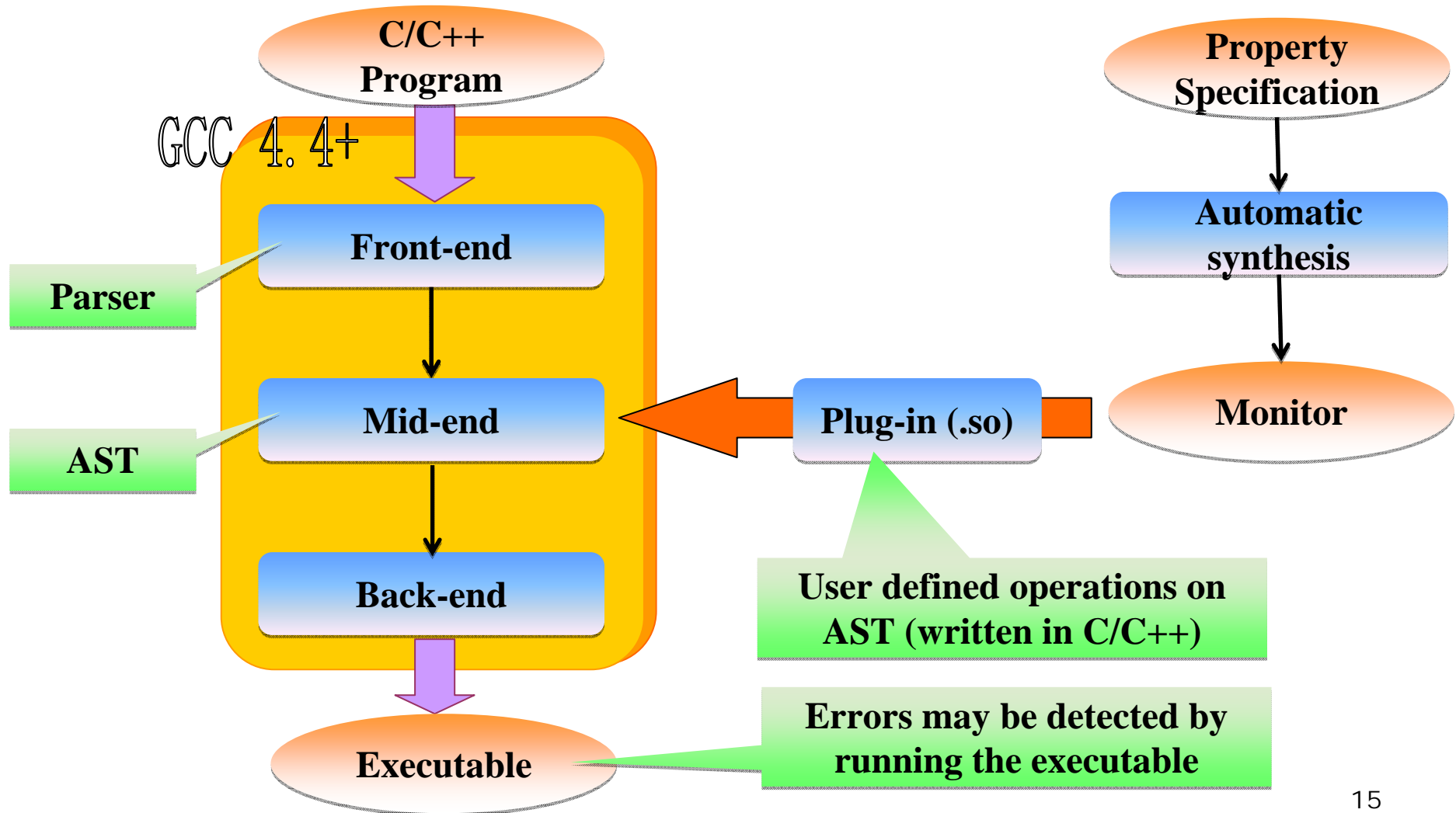


Principle of Runtime Verification





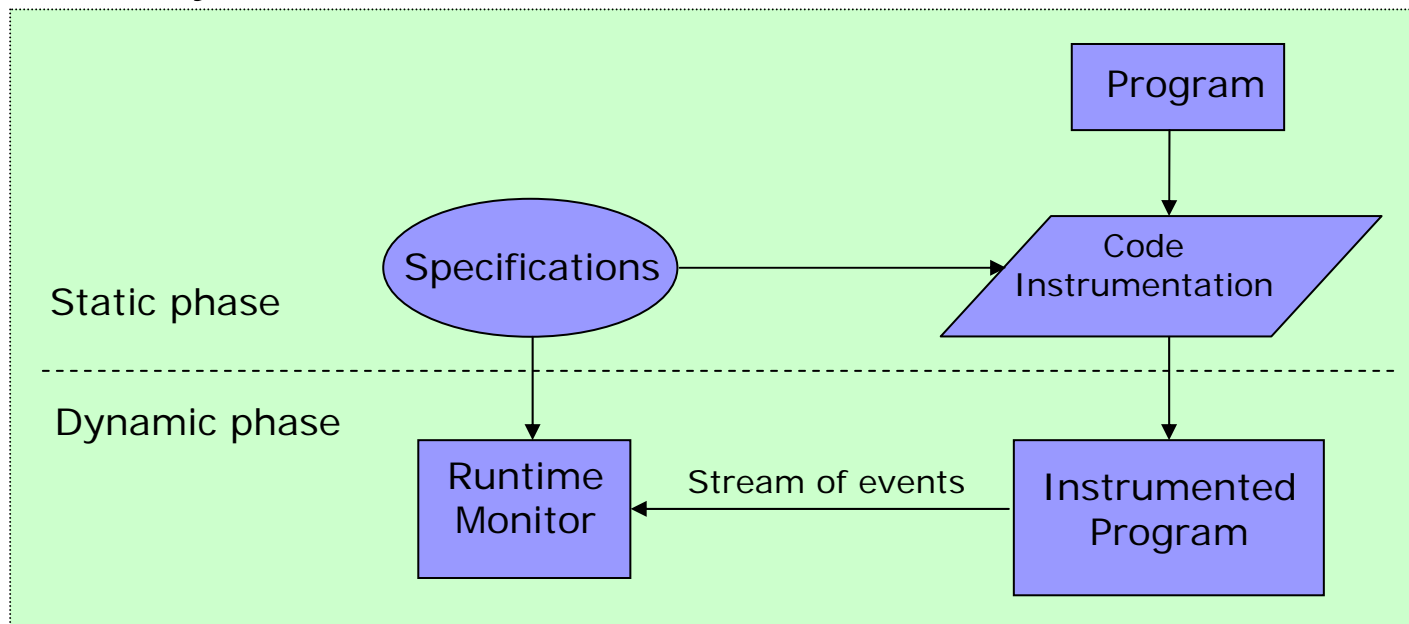
Implementation





Implementation

- Two issues for monitoring programs
 - Code instrumentation: extract events from the program while it is running.
 - Constructing Monitor: receive and then analysis the stream of events.





An (informal) example

- Property $[\](x > 0)$, i.e., $x > 0$ holds forever.
- Search code for assignments to x .
- Insert code stubs that communicate with the monitor.
- Three means of communication:
 - Networking.
 - Shared memory.
 - File system.

```
x = 5
// code stub here
// reports to monitor
...
...
x = -1
// code stub here
// reports to monitor.
// monitor detects
// violation
```



Finite traces

- Trace semantics are different for finite traces.
 - Can fake infinite trace by stuttering.
 - Repeat the last state, or
 - Repeat a “dummy” state that doesn’t fulfill any proposition.
 - Problem: meaning of [] operator changes.
 - A more accepted alternative: define different acceptance condition.



Finite traces, cont'd

- Acceptance condition: A trace is acceptable if all its eventualities have been fulfilled.
 - Example: $\square(x \rightarrow \langle \rangle y)$.
 - What if every occurrence of x is followed by a y later on except for the last one?
 - One can gather statistics about how a trace “performs” w.r.t the formula.
 - Even if a trace is acceptable, it might have been rejected if we waited long enough.



Finite traces, cont'd

- Safety properties are usually in the form of “something bad never happens”.
 - Acceptance condition is trivial: if the “bad” thing doesn’t happen along the trace, the trace is accepted.
 - Still no guarantees that “bad” thing won’t happen elsewhere. (coverage problem)
- However, if trace is rejected, clearly there is something wrong in the program.



Constructing efficient monitors

- An efficient monitor should have the following properties:
 - Forward design.
 - No backtracking.
 - Memory-less: doesn't store the trace.
 - Space efficiency.
 - Runtime efficiency.
 - A monitor that runs in time exponential in the size of the trace is unacceptable.
 - A monitor that runs in time exponential in the size of the formula is usable, but should be avoided.



Monitors: automata approach

■ Automata approach:

- Generate an automaton for the LTL property.
 - Alternating automaton is preferred: transitions are divided into existential and universal transitions.
 - Its size is linear to the size of LTL formula.
- Use a modified version of alternating automata, suitable for finite traces.
 - Each node is labelled as “acc” or “rej”, and possibly “fin”.
 - If eventualities are not fulfilled, the automaton will finish the trace at a rejecting state.

■ How to traverse the automaton:

- Depth first?
- Breadth first?



Monitors: automata approach

■ Depth first algorithm

- Problem: need to backtrack the trace.
- Example: $[(x \rightarrow \langle \rangle y)]$, where y only holds in the last state, and x holds everywhere else.
 - For every state in the trace, the algorithm needs to visit the rest of the trace looking for y .

■ Breadth first algorithm

- Memory-less. Each state is visited only once.
- Maintains all nodes that are currently “running”.
- Problem: the list of nodes can grow exponentially.



Monitors: symbolic approach

■ Formula manipulation

□ Rewriting logic

- Given a formula and a trace, the formula “consumes” the first state in the trace, and produces a new formula.

□ For example, the formula $\square(x \rightarrow \langle \rangle y)$ and given that the first state in the trace is x .

- $\square(x \rightarrow \langle \rangle y)\{x\} = \square(x \rightarrow \langle \rangle y) \wedge \langle \rangle y$
- Intuition: In order for $\square(x \rightarrow \langle \rangle y)$ to hold when the first state is x , the formula $\square(x \rightarrow \langle \rangle y) \wedge \langle \rangle y$ needs to hold at the rest of the trace after the first state.



Monitors: symbolic approach

- Formula manipulation is applied iteratively while consuming states from the trace.
 - The formula grows in size. At each iteration more rewriting rules are used to simplify the formula.
 - At the end of the of the trace, the formula is reduced to either *true* or *false*.
 - If eventualities are not fulfilled, formula will be false.



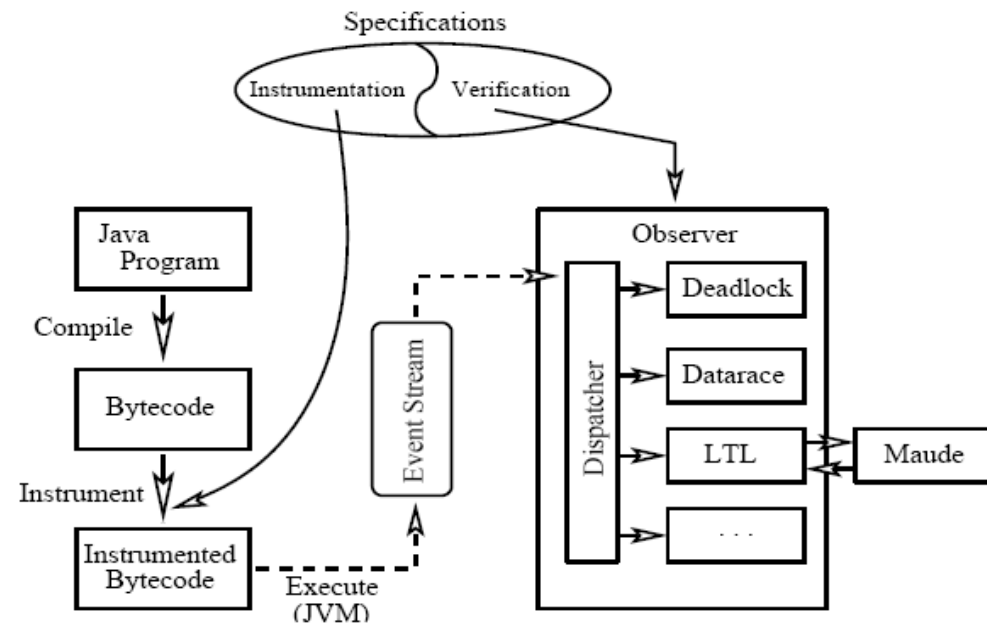
Tools

- \leq 2001
 - **MAC** (UPenn), **PAX** (NASA), **TimeRover** (commercial)
- 2002-2004
 - **HAWK/Eagle** (NASA), **MOP** (UIUC), **POTA** (UTA)
- \geq 2005:
 - **PQL** (Stanford)
 - **Tracematches** (Oxford)
 - **PTQL** (Berkeley/Stanford/Novell)
 - **Pal** (UPenn)
 - **RuleR** (Manchester)
 - ... many others



Java PathExplorer (JPaX)

- Developed at NASA Ames Research Center, by Grigore Rosu and Klaus Havelund.
- Instruments bytecode directly, not source code.
- Can use different logics simultaneously.
- Uses Maude as its logic rewriting engine.
- The definition of LTL (Future and Past) and rewriting rules is only 130 lines.





Outline

- Background
 - Software Verification
 - Testing
 - Model Checking
- Runtime Verification
 - Principle
 - Implementation
 - Code Instrumentation
 - Constructing Monitors
- Conclusion



Conclusion

- Runtime verification is a technique between testing and model checking.
 - Scales well, usually inexpensive.
 - Verifies an implementation directly.
 - Partially formal, sound and automatic.
- Limitations
 - Good at verifying safety property. But some properties are not suitable for runtime verification, e.g. liveness.
 - Need test-case generation techniques.
- Some already published case-studies show the practical use of runtime verification in the industry (MOP, Java PathExplorer, Java-MaC).



Thank you!