

# Automated Design-Time Analysis for the GOES-R System

David Hall and Corina Păsăreanu  
SGT and CMU SV, NASA Ames Research Center

# GOES

- Geostationary Operational Environmental Satellites (GOES)
  - Operated by the National Oceanic and Atmospheric Administration (NOAA)
  - Provide continuous weather imagery and monitoring of meteorological and space environment data
  - To protect life and property across the US

# GOES-R

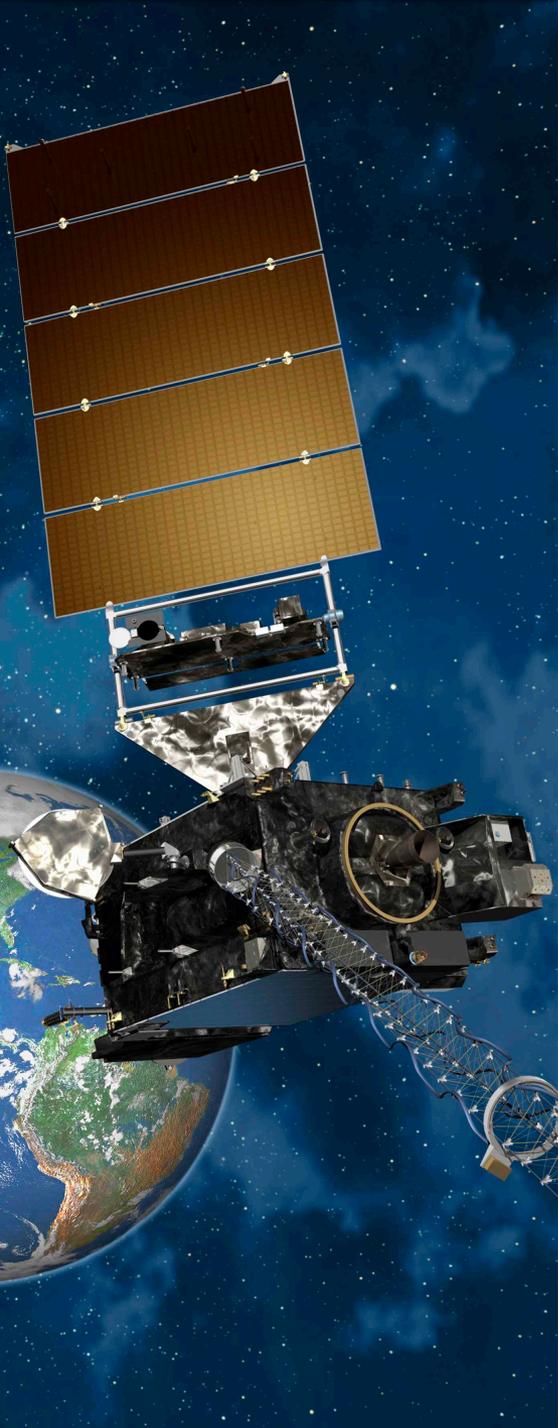
- NOAA/NASA GOES-R
  - Next generation GOES
  - Continuity of GOES
  - Improvement of remotely sensed environmental data
  - Must be extremely reliable and correct
  - Software intensive
  - Many interactive components
  - ... challenge to verification

# Testing

- typically used to ensure software reliability
- often manual and time-consuming
- used late in the software life cycle
- after the code has been written
- when it is expensive to fix discovered interaction errors

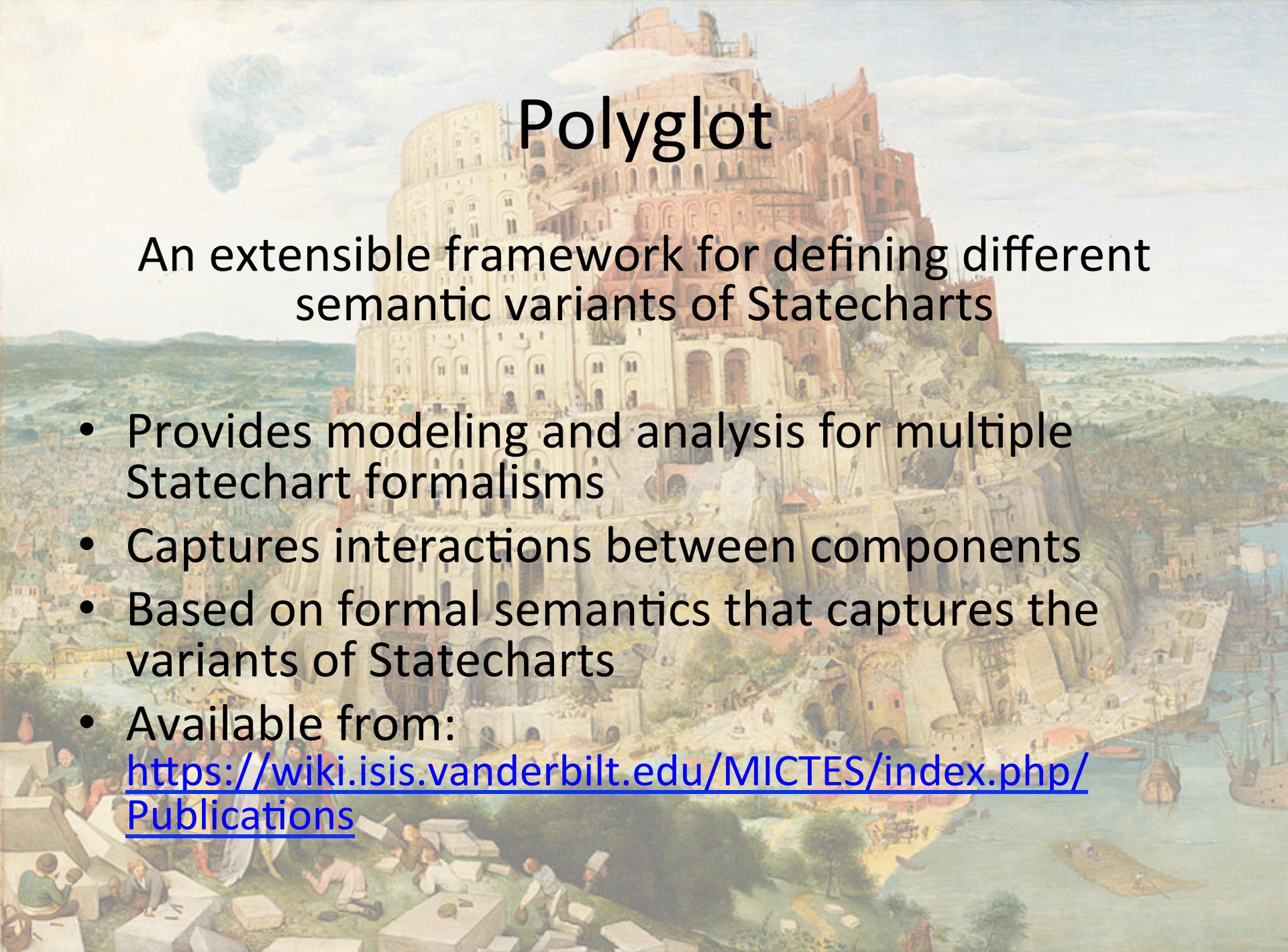
# Our Goal

- Systematic **design time** analysis
- Identifying and correcting errors at design time is easier and more cost effective
- Even if the system is already implemented
  - behavior identified from the design specifications can be used to guide testing and assess completeness of the test cases



# Analysis Approach

- The state transition behavior of the ground segment of the GOES-R
- Translated into MathWorks' Stateflow notation
  - Natural mapping between GOES-R design documents and state-charts
  - Translated using Polyglot
  - Analyzed using Ames JPF open-source verification tool-set

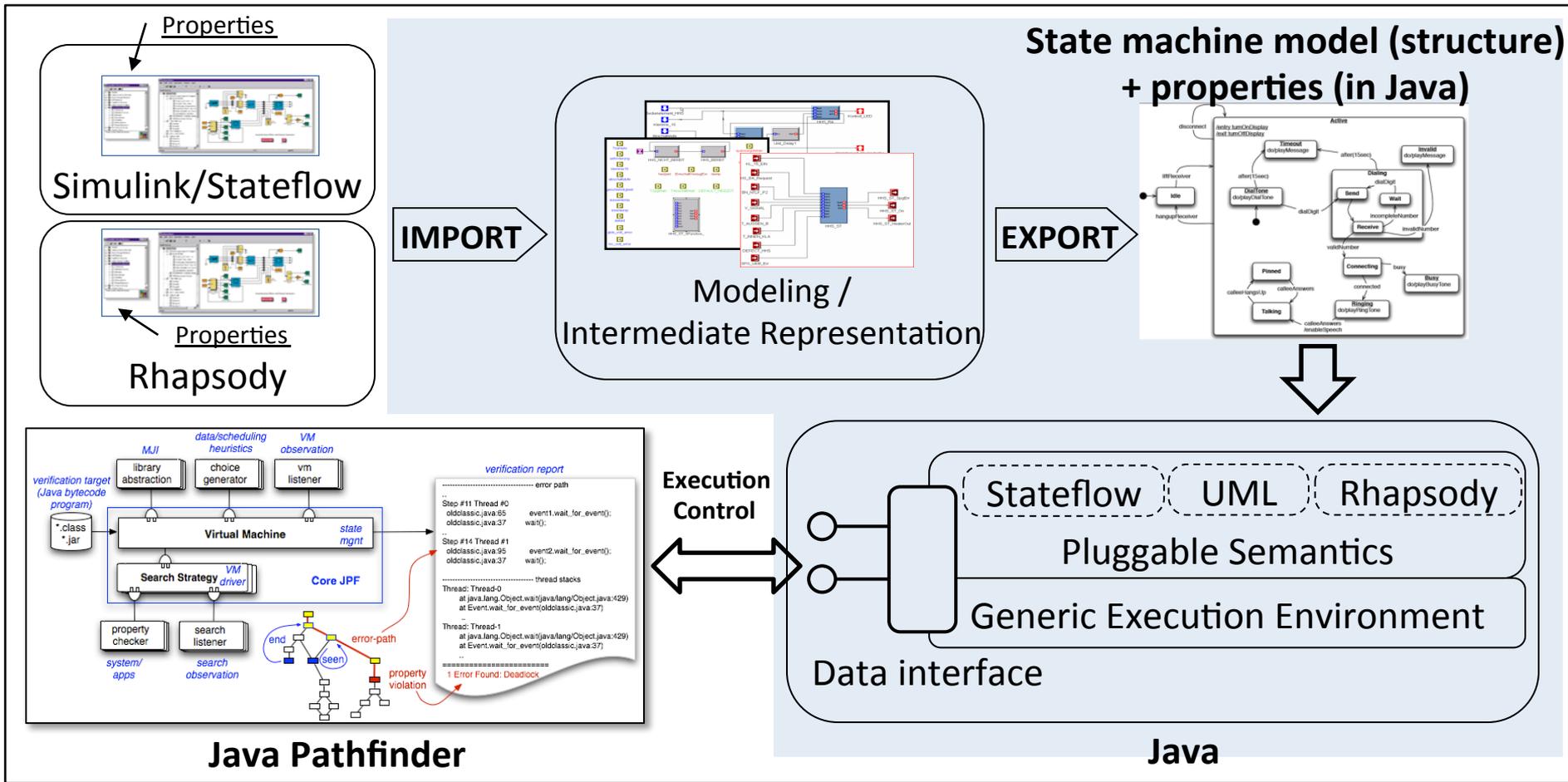


# Polyglot

An extensible framework for defining different semantic variants of Statecharts

- Provides modeling and analysis for multiple Statechart formalisms
- Captures interactions between components
- Based on formal semantics that captures the variants of Statecharts
- Available from:  
<https://wiki.isis.vanderbilt.edu/MICTES/index.php/Publications>

# Polyglot -- Tool Overview

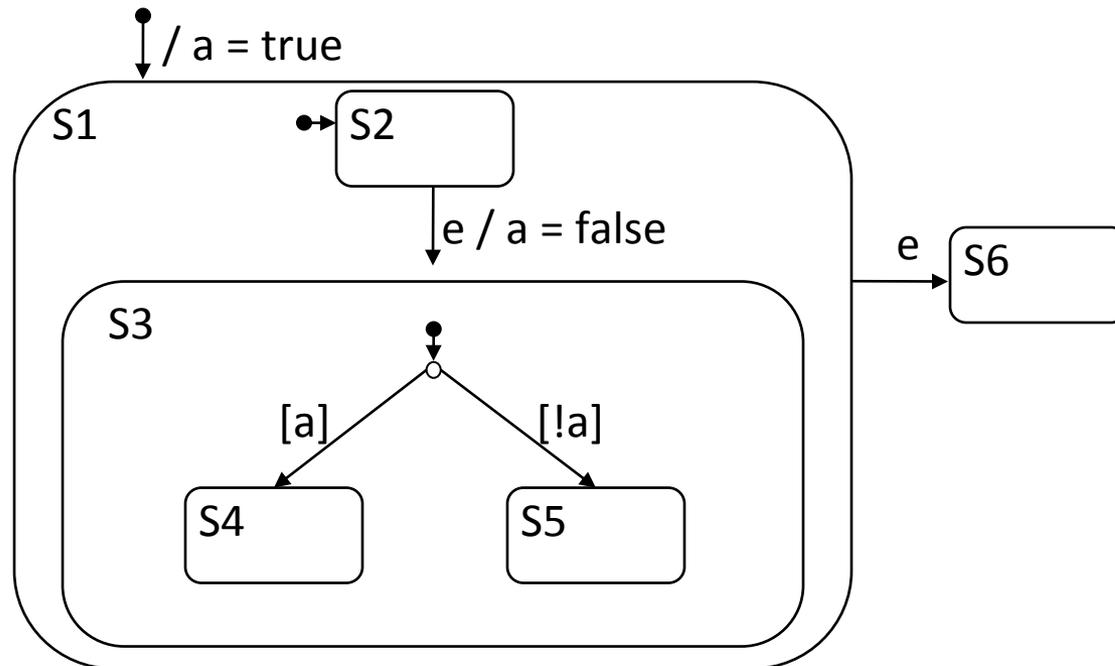


# Usage

1. User defines only the **structure** of the Statechart
2. The “pluggable” semantics are selected from a defined set
3. Temporal properties specified with pattern-based system
4. Analysis performed with:
  - Simulation (run the Java program)
  - Java Pathfinder: explicit state model checking
  - Symbolic Pathfinder: automated test-sequence generation  
available from: <http://babelfish.arc.nasa.gov/trac/jpf/wiki>

# Statechart review

- Consider this Statechart\* :



- Event “e” leads to S4 (UML), S5 (Rhapsody), or (S6) Stateflow
- UML semantics evaluate transition actions at the end of a transition path, and Rhapsody semantics perform transition actions when they are encountered.

# Design Choice

Java as a common language representation

- *Executable* representation for the models, for quick validation and debugging
- Enables modular and extensible design for Polygot
- Leverage JPF and SPF for model analysis and test-case generation
- Large action languages can be mapped to Java

# Property Specification

- Earlier study by Dwyer et al. found 92% of real world specifications fall into a small category
- Property consists of 2 pieces:
  1. Scope: when should property hold
  2. Pattern: the conditions that should be satisfied
- Implemented as graphical extension to Simulink/Stateflow

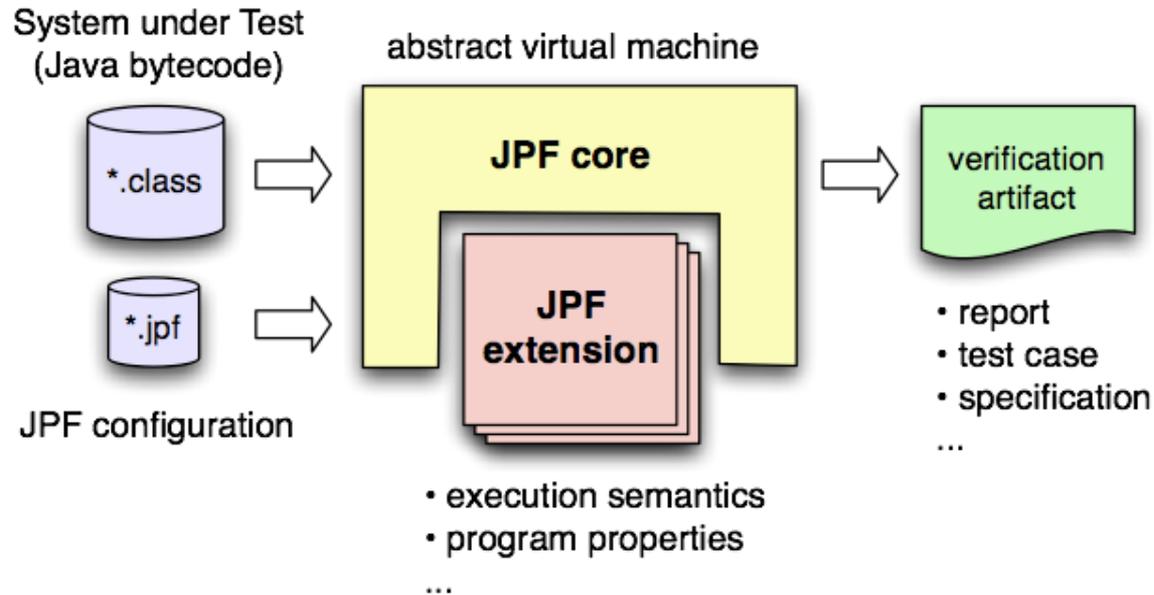
# Patterns

- 5 total patterns in 2 groups
- Occurrence
  - Absence: never true
  - Universality: always true
  - Existence: True at least once
- Order
  - Precedence: a state must precede another
  - Response: a state must follow another

# Analysis with JPF / SPF

- Analysis and test-case generation is performed with the Symbolic Pathfinder (SPF), the symbolic execution module of Java Pathfinder
- Tests reachability, generates test-vectors of input sequences
  - Can be fed back to the original modeling tool or to Polyglot

# Java Pathfinder (JPF)

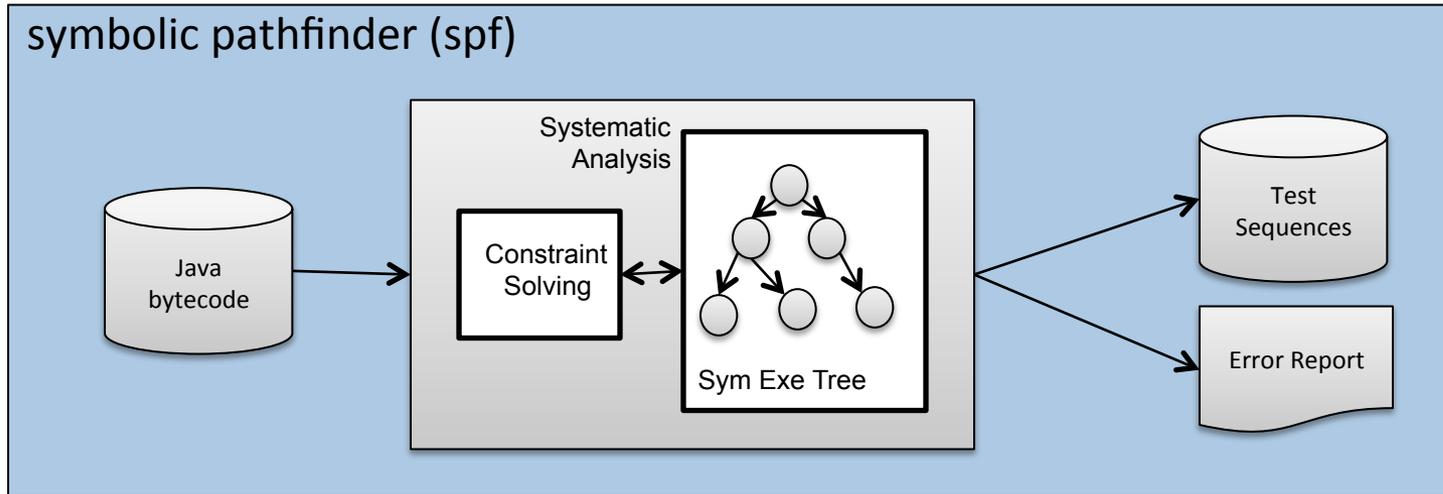


extensible virtual machine framework for java bytecode verification  
workbench to implement all kinds of verification tools

typical use cases:

- software model checking (detection of deadlocks, races, assert errors)
- test case generation (symbolic execution) ... and many more

# Symbolic Pathfinder (SPF)



combines symbolic execution, model checking and constraint solving  
applies to executable models and code

handles dynamic data structures, loops, multi-threading, strings

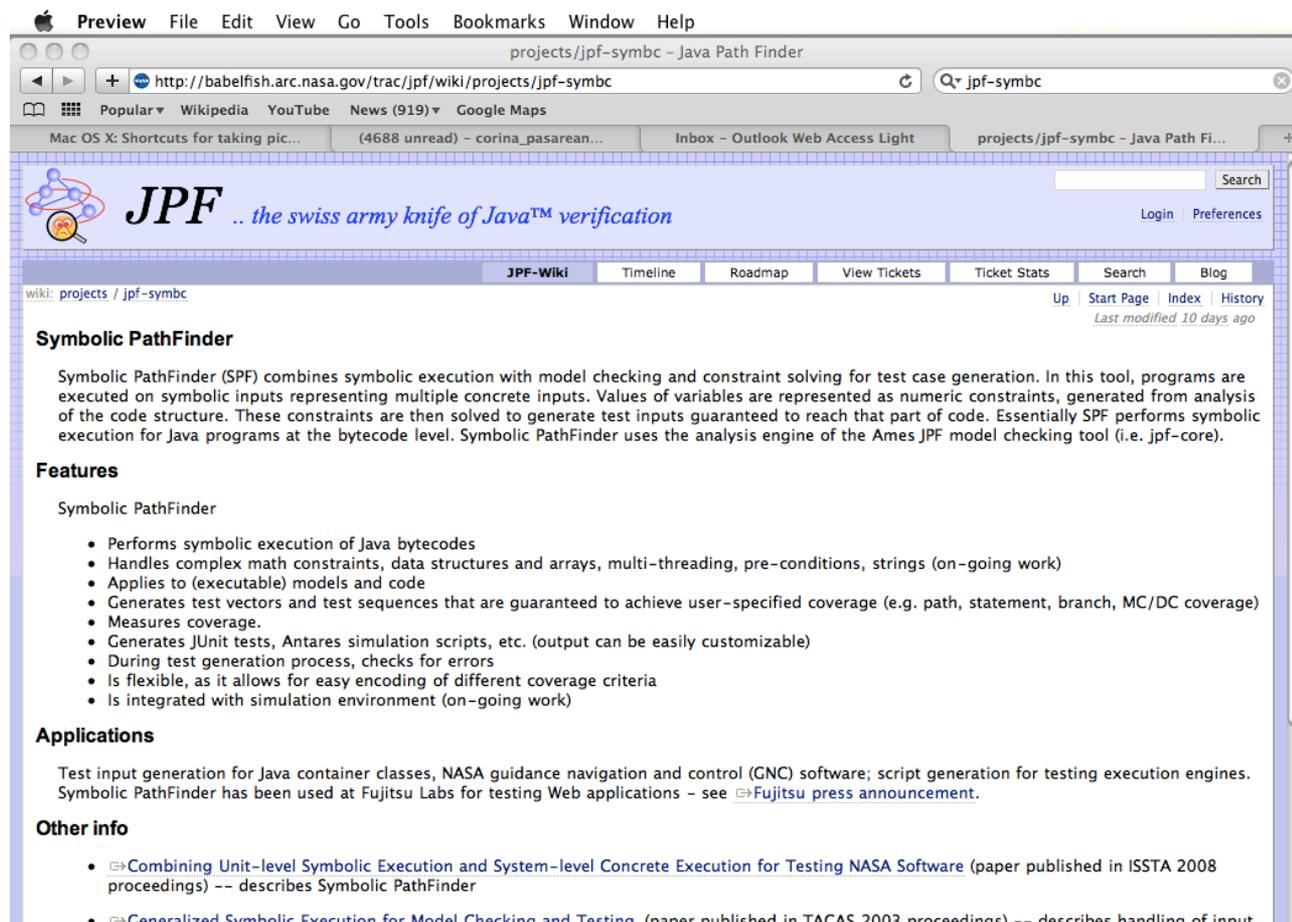
java pathfinder extension project jpf-symbc [TACAS'03, ISSTA'08, ASE'10]

generates automatically **input sequences** to drive statecharts on different paths

# Symbolic Pathfinder

available from jpf distribution

<http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc>



Preview File Edit View Go Tools Bookmarks Window Help

projects/jpf-symbc - Java Path Finder

http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

Popular Wikipedia YouTube News (919) Google Maps

Mac OS X: Shortcuts for taking pic... (4688 unread) - corina\_pasarean... Inbox - Outlook Web Access Light projects/jpf-symbc - Java Path Fi...

 **JPF** .. the swiss army knife of Java™ verification

Search Login Preferences

JPF-Wiki Timeline Roadmap View Tickets Ticket Stats Search Blog

wiki: projects / jpf-symbc Up Start Page Index History Last modified 10 days ago

## Symbolic PathFinder

Symbolic PathFinder (SPF) combines symbolic execution with model checking and constraint solving for test case generation. In this tool, programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as numeric constraints, generated from analysis of the code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code. Essentially SPF performs symbolic execution for Java programs at the bytecode level. Symbolic PathFinder uses the analysis engine of the Ames JPF model checking tool (i.e. jpf-core).

### Features

Symbolic PathFinder

- Performs symbolic execution of Java bytecodes
- Handles complex math constraints, data structures and arrays, multi-threading, pre-conditions, strings (on-going work)
- Applies to (executable) models and code
- Generates test vectors and test sequences that are guaranteed to achieve user-specified coverage (e.g. path, statement, branch, MC/DC coverage)
- Measures coverage.
- Generates JUnit tests, Antares simulation scripts, etc. (output can be easily customizable)
- During test generation process, checks for errors
- Is flexible, as it allows for easy encoding of different coverage criteria
- Is integrated with simulation environment (on-going work)

### Applications

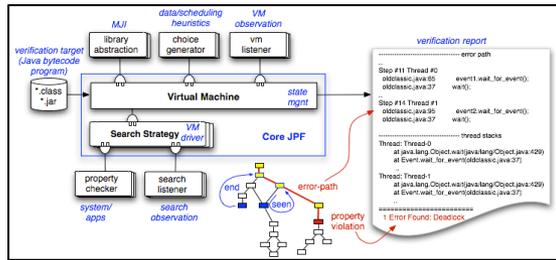
Test input generation for Java container classes, NASA guidance navigation and control (GNC) software; script generation for testing execution engines. Symbolic PathFinder has been used at Fujitsu Labs for testing Web applications - see [Fujitsu press announcement](#).

### Other info

- [Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software](#) (paper published in ISSTA 2008 proceedings) -- describes Symbolic PathFinder
- [Generalized Symbolic Execution for Model Checking and Testing](#) (paper published in TACAS 2003 proceedings) -- describes handling of input

# Execution with SPF

## Java Pathfinder / Symbolic Pathfinder



4. Generates

Test vectors

Property reports

1. Input data

## Execution engine

Stateflow UML Rhapsody

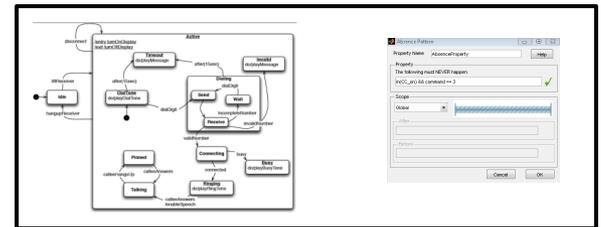
Pluggable Semantics

Generic Execution Engine

2. Inspect

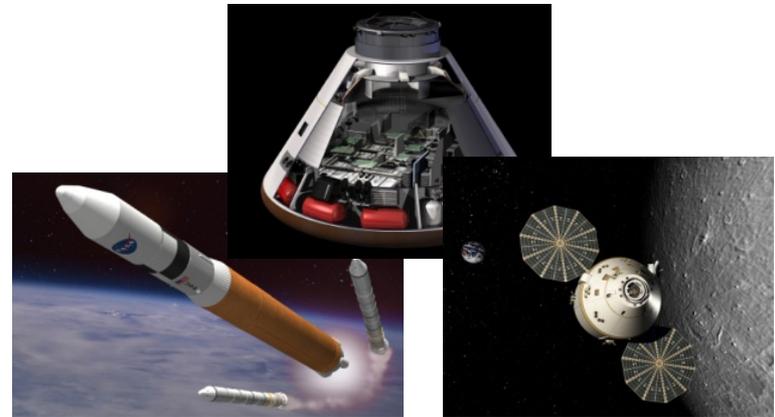
3. Set

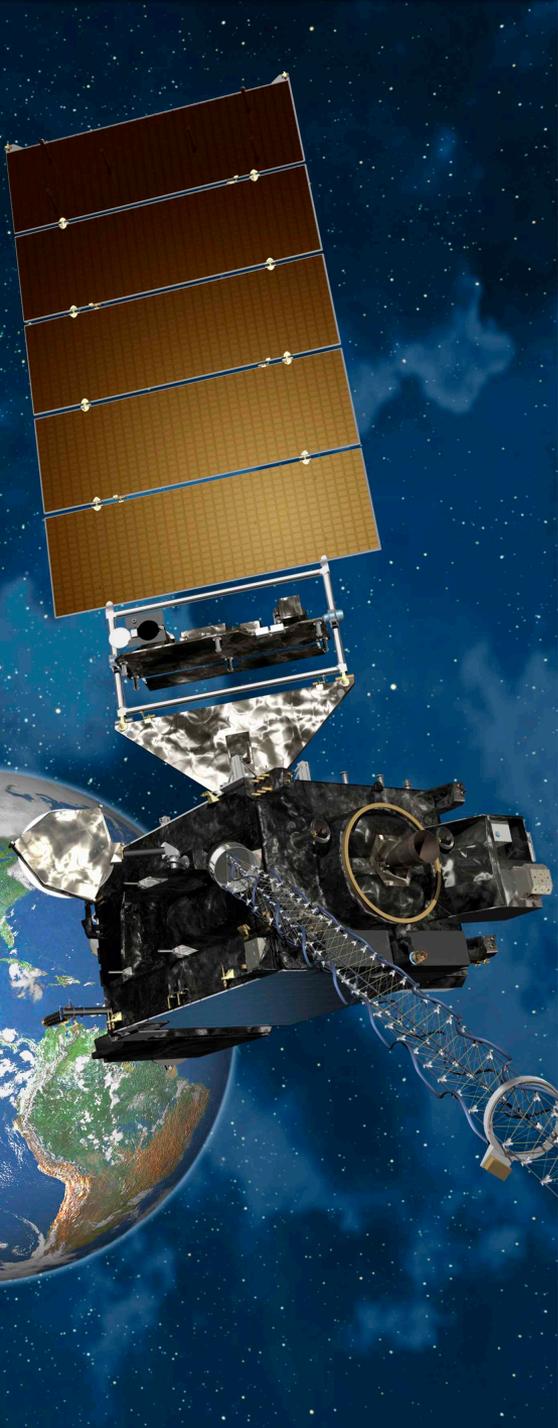
## State machine + property



# Applications

- Analysis of arbiter module for the Mars Exploration Rover [ISSTA 2011]
- Interaction between Ares launch vehicle and Orion Crew Exploration Vehicle [NFM 2012]





# Results for GOES-R

## MathWorks/Stateflow:

- GSSWRS
  - Statechart with 8 states and 35 transitions
  - Polyglot: 759 + 36 Java LOC
- GSTier:
  - Statechart with 2 parallel state machines (6 states, 24 transitions + 3 states, 6 transitions)
  - Polyglot: 701 + 36 Java LOC

# Example Simplified Model

The screenshot displays the Stateflow interface for a model named 'goesrsimple'. The main workspace shows a state machine diagram with two primary sections: 'Service' and 'Status'. The 'Service' section includes states 'Offline', 'Maintenance', and 'Online'. The 'Status' section includes states 'Operational', 'Unavailable', 'Failed', and 'Degraded'. Transitions are labeled with conditions involving 'cmd' and 'status' variables. A legend on the right defines these variables: 'cmd' (0-online, 1-offline, 2-maintain) and 'status' (0-operational, 1-unavailable, 2-failed, 3-degraded). An 'Absence Pattern' dialog box is open in the foreground, showing the property name 'AbsenceProperty' and the condition 'in(Status.Degraded) && cmd == 0' with a green checkmark. The dialog also shows a scope of 'Global' and 'After'/'Before' sections. The bottom status bar indicates 'Ready'.

Stateflow (chart) goesrexsimple/goesrsimple

File Edit View Simulation Debug Tools Format Add Patterns Help

Service

Offline

Maintenance

Online

Status

Operational

Unavailable

Failed

Degraded

Fig.422.1.2.43

cmd:  
0-online  
1-offline  
2-maintain

status:  
0-operational  
1-unavailable  
2-failed  
3-degraded

Absence Pattern

Property Name: AbsenceProperty

Property

The following must NEVER happen:

in(Status.Degraded) && cmd == 0

Scope: Global

After:

Before:

Cancel OK

Ready

# Analysis with SPF

- Ran up to depth 40
- Generates 1482 test cases
- Properties:
  - “absence”
  - Model should never be in state “Degraded” if command is “online”

# Future Work

- Expand and robustify the tool
- Compositional verification from JPF [FASE 2009] for increased scalability
- Identify and analyze more properties of ground system in the GOES-R project
  - Check conformance at different levels of abstraction
  - Measure coverage