# Software Model Checking and its Tools

Kapitel 24, „Softwarewerkzeuge“

Courtesy to Willem Visser. Used by permission.

Prof. U. Aßmann, TU Dresden

Available at:

http://www.visserhome.com/willem/presentations/presentations.html

# Software Model Checking

Shortened from

Willem Visser, Tutorial at ASE 2002

Research Institute for Advanced Computer Science
NASA Ames Research Center

# Overview

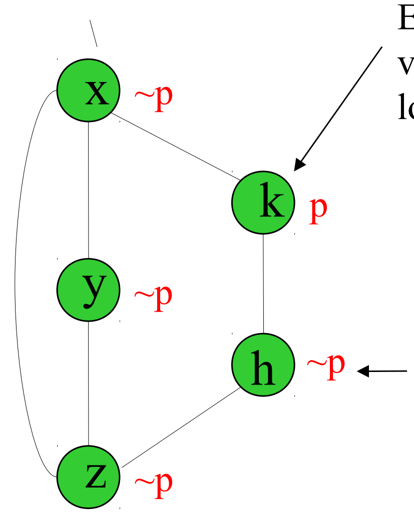
- Introduction to Model Checking
  - Hardware and Software Model Checking
- Program Model Checking
  - Major Trends
    - Abstraction
    - Improved model checking technology
  - A Brief History
    - SPIN
    - Hand-translations
    - State-less model checking
    - Semi-automated translations
    - Fully automated translations
  - Current Trends
    - Custom-made model checkers for programs
    - SLAM
    - JPF
    - Summary
- NASA Case Studies - Remote Agent, DEOS and Mars Rover
- Future of Software Model Checking

# Model Checking
## *The Intuition*

- Calculate whether a system satisfies a certain behavioral property:
  - Is the system deadlock free?
  - Whenever a packet is sent will it eventually be received?
- So it is like testing? No, major difference:
  - Look at *all* possible behaviors of a system
- Automatic, if the system is finite-state
  - Potential for being a push-button technology
  - Almost no expert knowledge required
- How do we describe the system?
- How do we express the properties?

# Kripke Structures are Labeled State Graphs plus Predicates

Each transition represents an execution step in the system



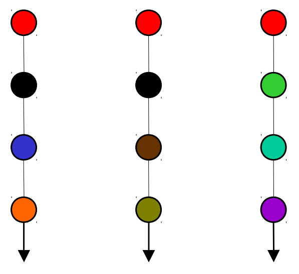
Each state represents all variable values and location counters

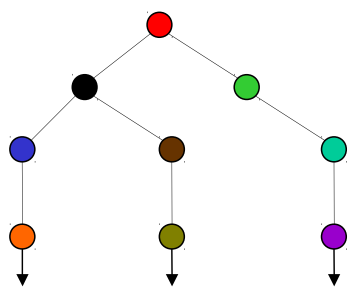
The labels represent predicates in each state e.g. (x = 5)

$$K = (\{p,\sim p\},\{x,y,z,k,h\},R,\{x\},L)$$

# Property Specifications with Temporal Logic

- Temporal Logic
  - Express properties of event orderings in time
  - e.g. “Always” when a packet is sent it will “Eventually” be received
- Linear Time
  - Every moment has a unique successor
  - Infinite sequences (words)
  - Linear Time Temporal Logic (LTL)
- Branching Time
  - Every moment has several successors
  - Infinite tree
  - Computation Tree Logic (CTL)





# Safety and Liveness

- Safety properties
  - Invariants, deadlocks, reachability, etc.
  - Can be checked on finite traces
  - “something bad never happens”
- Liveness Properties
  - Fairness, response, etc.
  - Infinite traces
  - “something good will eventually happen”

# Direction

- Model checking can be done
- Forward:
  - Searching from the initial state to reachable states, checking the condition
- Backward
  - Searching from the states in which a condition should hold backward to the initial state
  - In particular possible for reachability questions

# Mutual Exclusion Example

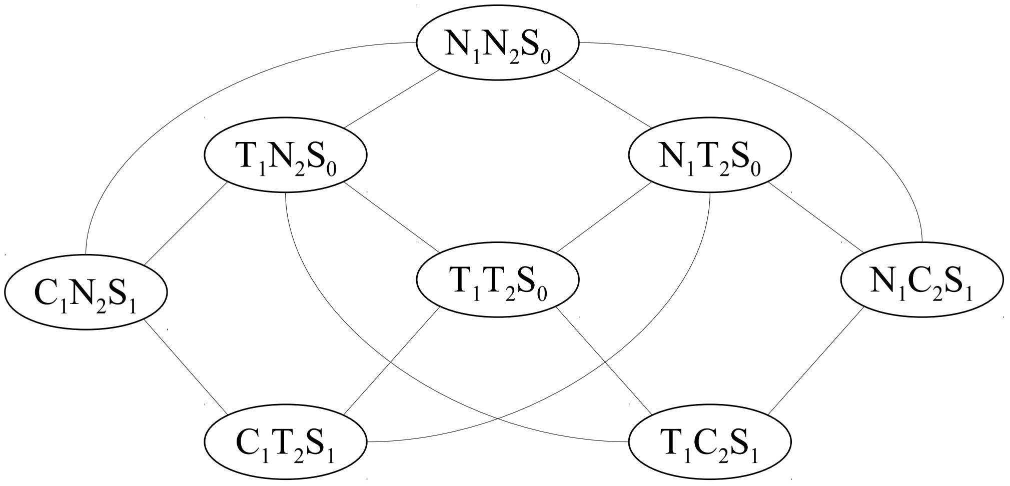
- Two process mutual exclusion with shared semaphore
- Each process has three states
  - Non-critical (N)
  - Trying (T)
  - Critical (C)
- Semaphore can be available ($S_0$) or taken ($S_1$)

•Model checkers construct a global system state space from the process

- Initially both processes are in the Non-critical state and the semaphore is available --- ($N_1\, N_2\, S_0$)

$$\begin{array}{lll} N_1 & \rightarrow & T_1 \\ T_1 \wedge S_0 & \rightarrow & C_1 \wedge S_1 \\ C_1 & \rightarrow & N_1 \wedge S_0 \end{array} \quad \Bigg\| \quad \begin{array}{lll} N_2 & \rightarrow & T_2 \\ T_2 \wedge S_0 & \rightarrow & C_2 \wedge S_1 \\ C_2 & \rightarrow & N_2 \wedge S_0 \end{array}$$

# System State Space (Backward Reachability Question)



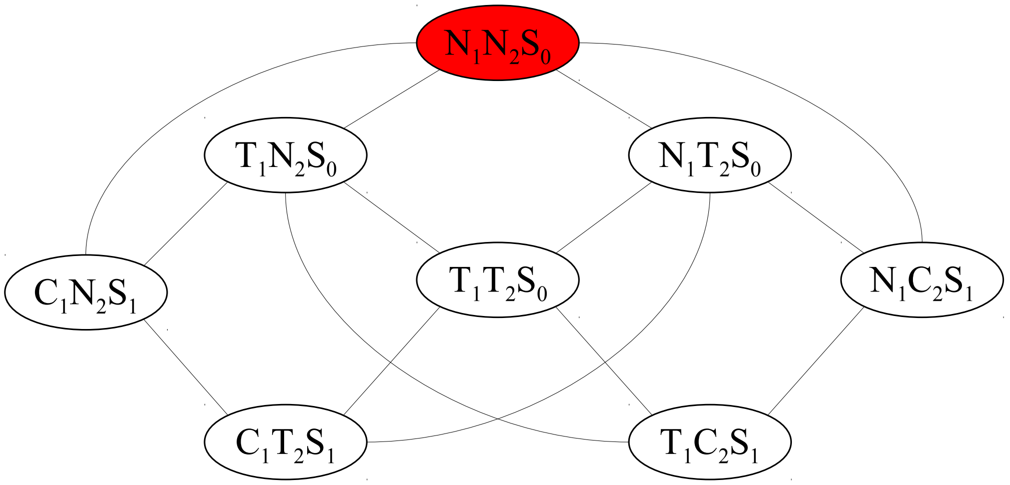
K $\models$ AG EF ($N_1$ and $N_2$ and $S_0$)

*All Globally*

*Exist Finally*

*No matter where you are there is always a way to get to the initial state*

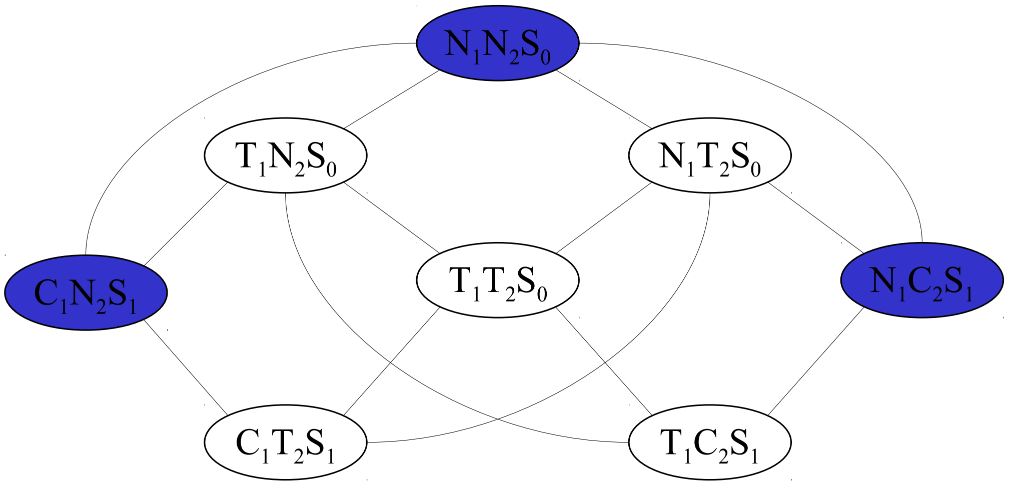
# Mutual Exclusion Example: Backward Analysis



$K \models$ AG EF ($N_1$ and $N_2$ and $S_0$)

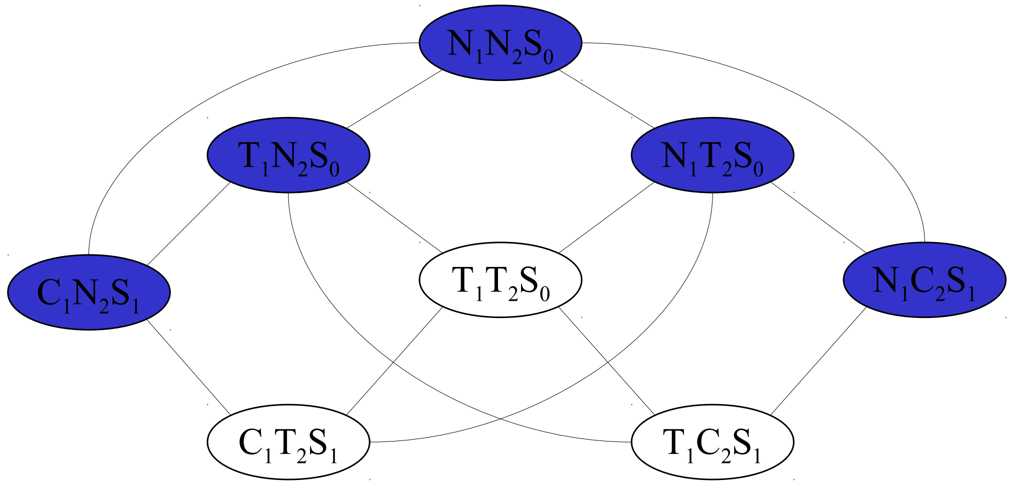
Model checkers do reachability of states: here, backward Search for paths.

# Mutual Exclusion Example



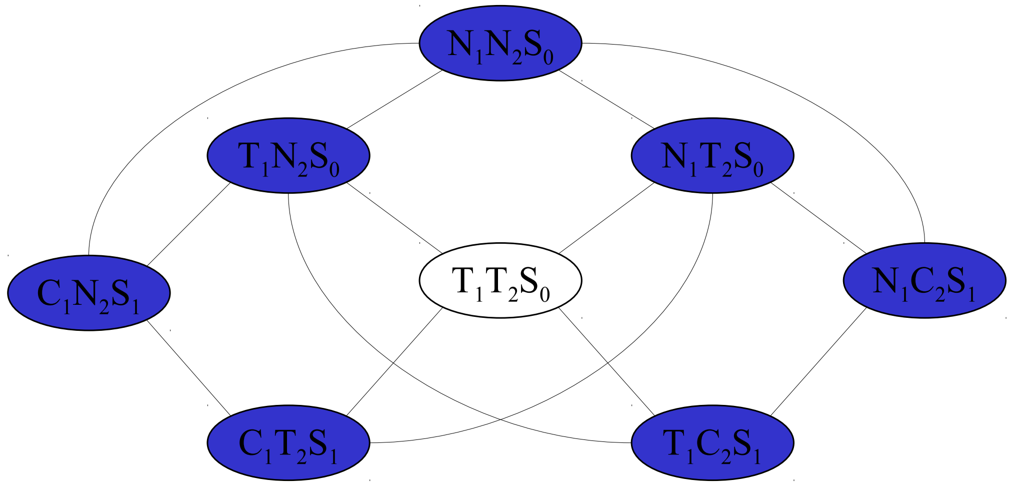
$K \models$ AG EF ($N_1$ and $N_2$ and $S_0$)

# Mutual Exclusion Example



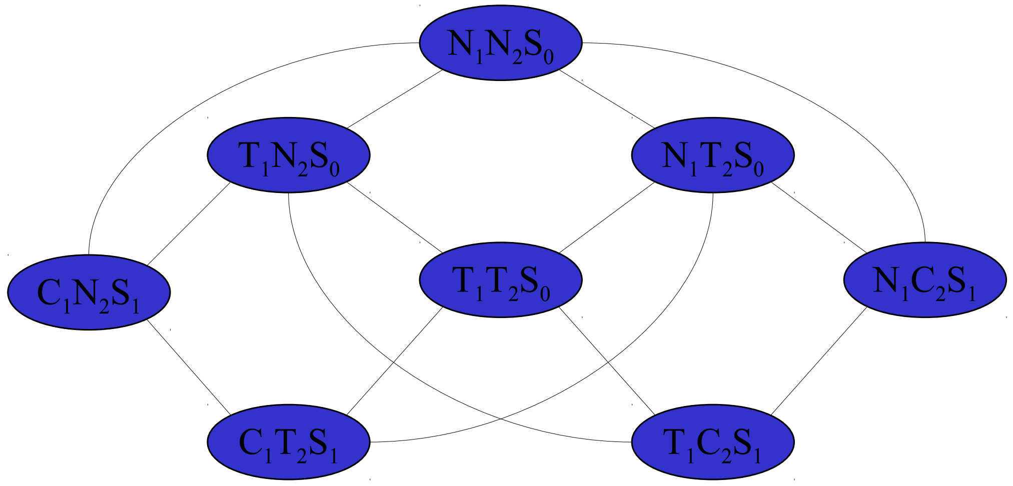
K $\models$ AG EF ($N_1$ and $N_2$ and $S_0$)

# Mutual Exclusion Example



K $\models$ AG EF ($N_1$ and $N_2$ and $S_0$)

# Mutual Exclusion Example



$K \models AG\ EF\ (N_1 \text{ and } N_2 \text{ and } S_0)$

Proven.

# Model Checking

- Given a Kripke structure *M = (S,R,L)* that represents a finite-state concurrent system and a temporal logic formula *f* expressing some desired specification, find the set of states in *S* that satisfy *f*:

$$\{ s \text{ in } S \mid M,s \models f \}$$

- Normally, some states of the concurrent system are designated as initial states. The system satisfies the specification provided all the initial states are in the set. We often write: $M \models f$

# Explicit vs. Symbolic Model Checking

- Explicit State
  - states are enumerated on-the-fly
  - *Forwards* analysis
  - Stores visited states in a hashtable
- Characteristics
  - Memory intensive
  - Good for finding concurrency errors
  - Short execution paths are better, but long execution paths can also be handled
  - Can handle dynamic creation of objects/threads
  - Mostly used in software

- Symbolic
  - **Sets of states** are manipulated at a time
  - Typically a *backwards* analysis in the automaton
  - Transition relation encoded by Binary Decision Diagrams (BDDs) or as a satisfiability problem
- Characteristics
  - Can handle very large state spaces
  - Not as good for asynchronous systems
  - Cannot deal well with long execution traces
  - Works best with a static transition relation, hence doesn't deal well with dynamic creation of objects/threads
  - Mostly used in hardware

# Overview

- Introduction to Model Checking
  - Hardware Model Checking
  - Software Model Checking
- Program Model Checking
- Case Studies
- Future of Software Model Checking

# Hardware Model Checking

- BDD-based model checking was the enabling technology
  - Hardware is typically synchronous and regular, hence the transition relation can be encoded efficiently
  - Execution paths are typically very short
- The Intel Pentium bug
  - got model checking on the map in the hardware industry
  - Intel, IBM, Motorola, etc. now employ hundreds of model checking experts

# Software Model Checking

- Until 1997 most work was on software designs
  - Since catching bugs early is more cost-effective
  - Problem is that everybody use a different design notation, and although bugs were found the field never really moved beyond some compelling case-studies
  - Reality is that people write code first, rather than design
- The field took off when the seemingly harder problem of analyzing actual source code was first attempted

# Program Model Checking

- Why is program analysis with a model checker so much more interesting?
  - Designs are hard to come by, but buggy programs are everywhere!
  - Testing is inadequate for complex software (concurrency, pointers, objects, etc.)
  - Static program analysis was already an established field, mostly in compiler optimization, but also in verification.

# The Trends in Program Model Checking

*Most model checkers cannot deal with the features of modern programming languages*

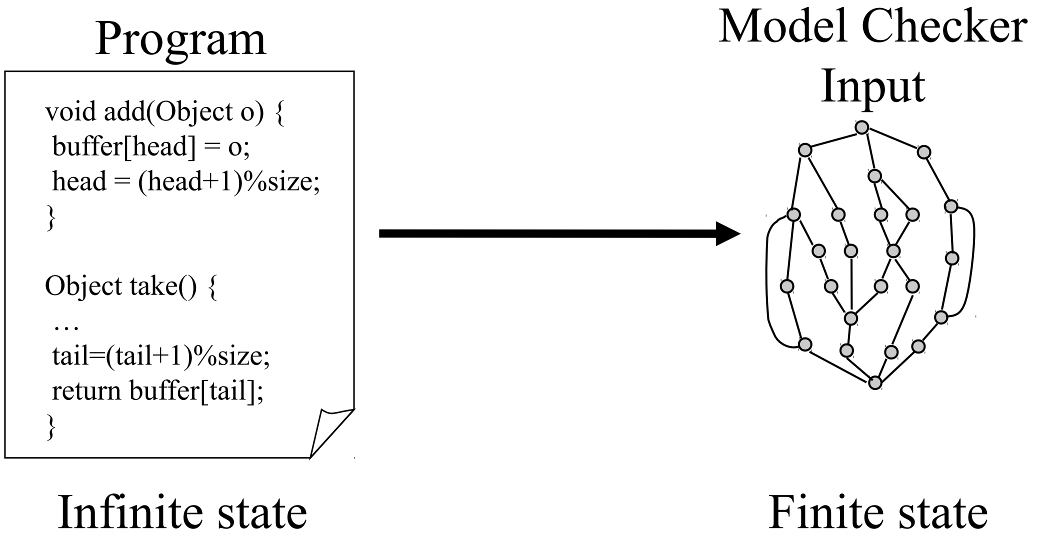
- Bringing programs to model checking
  - By abstraction (including translation)
- Bringing model checking to programs
  - Improve model checking to directly deal with programs as input

# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
    - Abstraction
    - Improved model checking technology
  - A Brief History
  - Current Trends
- Case Studies
- Future of Software Model Checking

# Program (Software) Model Checking

Abstraction of the Control-Flow Graph
from Stack Automaton to Finite Automaton



Program

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

Infinite state

Model Checker
Input

Finite state

# Abstraction

- Model checkers don’t take real “programs” as input
- Model checkers typically work on finite state systems
- Abstraction therefore solves two problems
  - It allows model checkers to analyze a notation they couldn’t deal with before, and,
  - Cuts the state space size to something manageable
- Abstraction comes in three flavors
  - **Over-approximations**, i.e. *more behaviors* are added to the abstracted system than are present in the original
  - **Under-approximations**, i.e. *less behaviors* are present in the abstracted system than are present in the original
  - **Precise abstractions**, i.e. *the same behaviors* are present in the abstracted and original program

# Under-Approximation
## *“Meat-Axe” Abstraction*

- Remove parts of the program deemed “irrelevant” to the property being checked
  - Limit input values to 0..10 rather than all integer values
  - Queue size 3 instead of unbounded, etc.
- Typically manual, with no guarantee that the right behaviors are removed
- Precise abstraction, w.r.t. the property being checked, may be obtained if the behaviors being removed are indeed not influencing the property
  - Program *slicing* is an example of an automated under-approximation that will lead to a precise abstraction w.r.t. the property being checked
  - However, can be incorrect

# Over-Approximations

## *Abstract Interpretation*

- Over-Approximation maps sets of states in the concrete program to one state in the *abstract program (Abstract Interpretation)*
  - Reduces the number of states, but increases the number of possible transitions, and hence the number of behaviors
- Type-based abstractions
  - Replace int by Signs abstraction {neg,pos,zero}
- Predicate abstraction
  - Replace predicates in the program by boolean variables, and replace each instruction that modifies the predicate with a corresponding instruction that modifies the boolean.
- Automated (conservative) abstraction: correct
- Eliminating spurious errors is the big problem
  - Abstract program has more behaviors, therefore when an error is found in the abstract program, is that also an error in the original program?
  - Most research focuses on this problem, and its counter-part the elimination of spurious errors, often called *abstraction refinement*

# Bringing Model Checking to Programs

- Allow model checkers to take modern programming languages as input
  - Major hurdle is how to encode the state of the system efficiently
  - Alternatively state-less model checking
    - No state encoding or storing
- Almost exclusively explicit-state model checking
- Abstraction can still be used as well
  - Source to source abstractions

# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
    - SPIN
    - Hand-translations
    - State-less model checking
      - Partial-order reductions
      - VeriSoft
    - Semi-automated translations
    - Fully automated translations
  - Current Trends
- Case Studies
- Future of Software Model Checking

# The Early Years

- Hand-translation with ad-hoc abstractions
  - 1980 through mid 1990s
- Semi-automated, table-driven translations
  - 1998
- Automated translations still with ad-hoc abstractions
  - 1997-1999
- State-less model checking for C
  - VeriSoft 1997

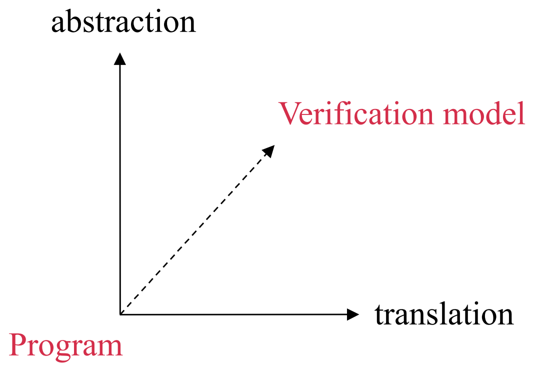
# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
    - SPIN
    - Hand-translations
    - State-less model checking
      - Partial-order reductions
      - VeriSoft
    - Semi-automated translations
    - Fully automated translations
  - Current Trends
- Case Studies
- Future of Software Model Checking

# SPIN Model Checker

- Kripke structures are described as “programs” in the PROMELA language
  - Kripke structure is generated on-the-fly during model checking
- Automata based model checker
  - Translates LTL formula to Büchi automaton
- By far the most popular model checker
  - SPIN workshop
- Relevant theoretical papers can be found here
  - http://netlib.bell-labs.com/netlib/spin/whatispin.html
- Ideal for software model checking due to expressiveness of the PROMELA language
  - Close to a real programming language
- Gerard Holzmann won the ACM software award for SPIN

# Hand-Translation



- Hand translation of program to model checker’s input notation
- “Meat-axe” approach to abstraction (under-approximation)
- Labor intensive and error-prone

# Hand-Translation Examples

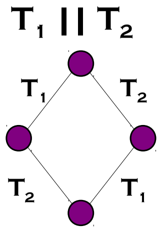
- Remote Agent – Havelund,Penix,Lowry 1997
  - http://ase.arc.nasa.gov/havelund
  - Translation from Lisp to Promela (most effort)
  - Heavy abstraction
  - 3 man months
- DEOS operating system – Penix, Visser, *et al.* 1998/1999
  - http://ase.arc.nasa.gov/visser
  - C++ to Promela (most effort in environment generation)
  - Limited abstraction - programmers produced sliced system
  - 3 man months

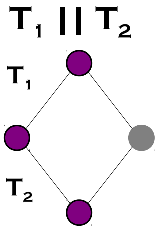
# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
    - SPIN
    - Hand-translations
    - State-less model checking
      - Partial-order reductions
      - VeriSoft
    - Semi-automated translations
    - Fully automated translations
  - Current Trends
- Case Studies
- Future of Software Model Checking

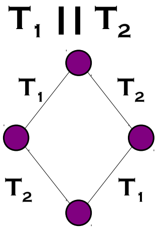
# VeriSoft

- The first model checker that could handle programs directly
  - C programs running on Unix
- Relies on partial-order reductions to limit the number of times a state is revisited
  - Persistent sets
    - Reduce states visited
  - Sleep sets
    - Reduce transitions executed
- Paths must be replayed from the initial state to try new branches
  - No check-pointing





Persistent Sets



Sleep Sets

# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
    - SPIN
    - Hand-translations
    - State-less model checking
      - Partial-order reductions
      - VeriSoft
    - Semi-automated translations
    - Fully automated translations
  - Current Trends
- Case Studies
- Future of Software Model Checking

# Semi-Automatic Translation

- Table-driven translation and abstraction
  - Feaver system by Gerard Holzmann
  - User specifies code fragments in C and how to translate them to Promela (SPIN)
  - Translation is then automatic
  - Found 75 errors in Lucent's PathStar system
  - http://cm.bell-labs.com/cm/cs/who/gerard/
- Advantages
  - Can be reused when program changes
  - Works well for programs with long development and only local changes

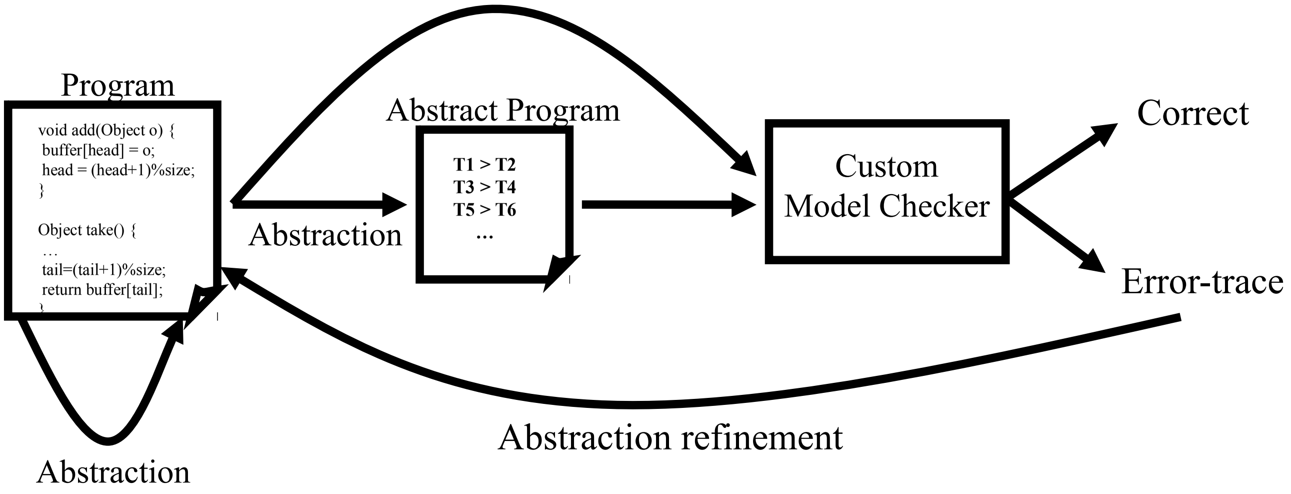
# Fully Automatic Translation

- Advantage
  - No human intervention required
- Disadvantage
  - Limited by capabilities of target system
- Examples
  - Java PathFinder 1- http://ase.arc.nasa.gov/havelund/jpf.html
    - Translates from Java to Promela (Spin)
  - JCAT - http://www.dai-arc.polito.it/dai-arc/auto/tools/tool6.shtml
    - Translates from Java to Promela (or dSpin)
  - Bandera - http://www.cis.ksu.edu/santos/bandera/
    - Translates from Java bytecode to Promela, SMV or dSpin

# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
  - Current Trends
    - Custom-made model checkers for programs
    - Abstraction
    - SLAM
    - JPF
    - Summary
    - Examples of other software analyses
- Case Studies
- Future of Software Model Checking

# Program Model Checking
## Current Trends



- Custom-made model checkers for programming languages with automatic abstraction at the source code level
- Automatic abstraction & translation based transformation to new “abstract” formalism for model checker
- Abstraction refinement mostly automated

# Custom-made Model Checkers

- Translation based
  - dSpin
    - Spin extended with dynamic constructs
    - Essentially a C model checker
    - Source-2-source abstractions can be supported
    - http://www.dai-arc.polito.it/dai-arc/auto/tools/tool7.shtml
  - SPIN Version 4
    - PROMELA language augmented with C code
    - Table-driven abstractions
  - Bandera
    - Translated Bandera Intermediate Language (BIR) to a number of back-end model checkers, but, a new BIR custom-made model checker is under development
    - Supports source-2-source abstractions as well as property-specific slicing
    - http://www.cis.ksu.edu/santos/bandera/

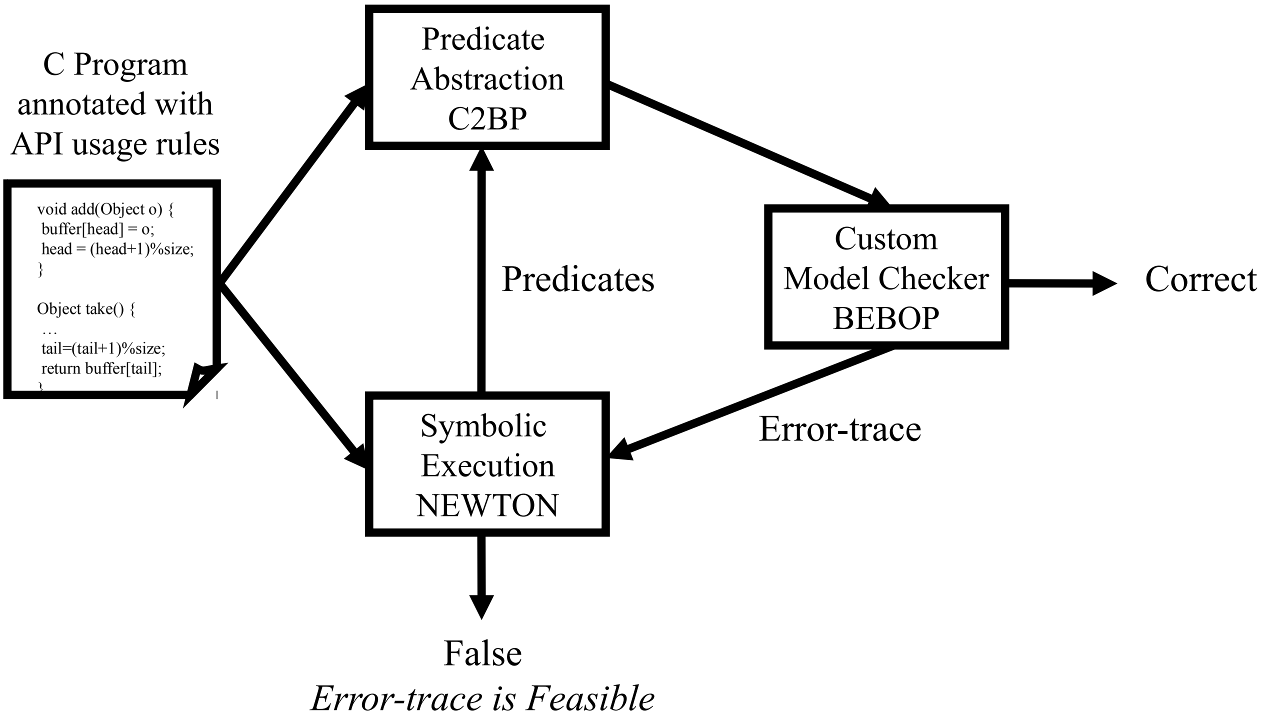
# Custom-made Model Checkers

- Abstraction based
  - SLAM
    - C programs are abstracted via predicate abstraction to boolean programs for model checking
    - http://research.microsoft.com/slam/
  - BLAST
    - Similar basic idea to SLAM, but using *lazy* abstraction, i.e. during abstraction refinement don’t abstract the whole program only certain parts
    - http://www-cad.eecs.berkeley.edu/~tah/blast/
  - 3-Valued Model Checker (3VMC) extension of TVLA for Java programs
    - http://www.cs.tau.ac.il/~yahave/3vmc.htm
    - http://www.math.tau.ac.il/~rumster/TVLA/

# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
  - Current Trends
    - Custom-made model checkers for programs
    - Abstraction
    - SLAM
      - Abstraction Refinement
    - JPF
    - Summary
    - Examples of other software analyses
- Case Studies
- Future of Software Model Checking

# SLAM
## *Simplified View*



C Program annotated with API usage rules

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

Predicate Abstraction C2BP

Predicates

Custom Model Checker BEBOP

Correct

Error-trace

Symbolic Execution NEWTON

False

*Error-trace is Feasible*

# SLAM

- Check API usage rules for sequential C programs
  - Mostly applied to device driver code
- C2BP
  - Inputs: C program and predicates
  - Output: boolean program over the predicates
- BEBOP
  - Symbolic interprocedural data flow analysis
  - Concrete CFG and BDD encoding of states
- NEWTON
  - Symbolic execution of C programs
  - Using Simplify theorem prover for checking feasibility of conditionals

# Abstraction Refinement Example

### Adapted from Ball & Rajamani POPL02

Property: *if a lock is held it must be released before reacquiring*

```
do {
     //get the write lock
   KeAcquireSpinLock(&devExt->writeListLock);

   nPacketsOld = nPackets;
   request = devExt->WLHeadVa;

   if (request){
          KeReleaseSpinLock(&devExt->writeListLock);
          ...
          nPackets++;
   }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock(&devExt->writeListLock);
```

# Initial Abstraction and Model Checking

## Boolean Program (simpler, abstracted)

```
[1] do
   //get the write lock
[2]      AcquireLock();
[3]      if (*) then
[4]        ReleaseLock();
         fi
[5] while (*);
[6] ReleaseLock();
```

**Error-trace : 1,2,3,5,1,2**

# Symbolic Execution

```
[1] do {
[2]       KeAcquireSpinLock(&devExt->writeListLock);
          nPacketsOld = nPackets;
       request = devExt->WLHeadVa;
[3]       if (request){
[4]         KeReleaseSpinLock(&devExt->writeListLock);
            ...
            nPackets++;
       }
[5] } while (nPackets != nPacketsOld);
[6] KeReleaseSpinLock(&devExt->writeListLock);
```

Symbolic execution of 1,2,3,5,1,2 shows that when 5 is executed `nPackets == nPacketsOld` hence the path is infeasible. The predicate `nPackets == nPacketsOld` is then added and used during predicate abstraction

# Next Abstraction and Model Checking

**New Predicate b : (nPacketsOld == nPackets)**

```
[1] do
[2]      AcquireLock();
[3]    b = true; // nPacketsOld = nPackets
[4]      if (*) then
[5]        ReleaseLock();
[6]        b = b ? false : *; // nPackets++
       fi
[7] while (!b); //(nPacketsOld != nPackets)
[8] ReleaseLock();
```

## Now property holds

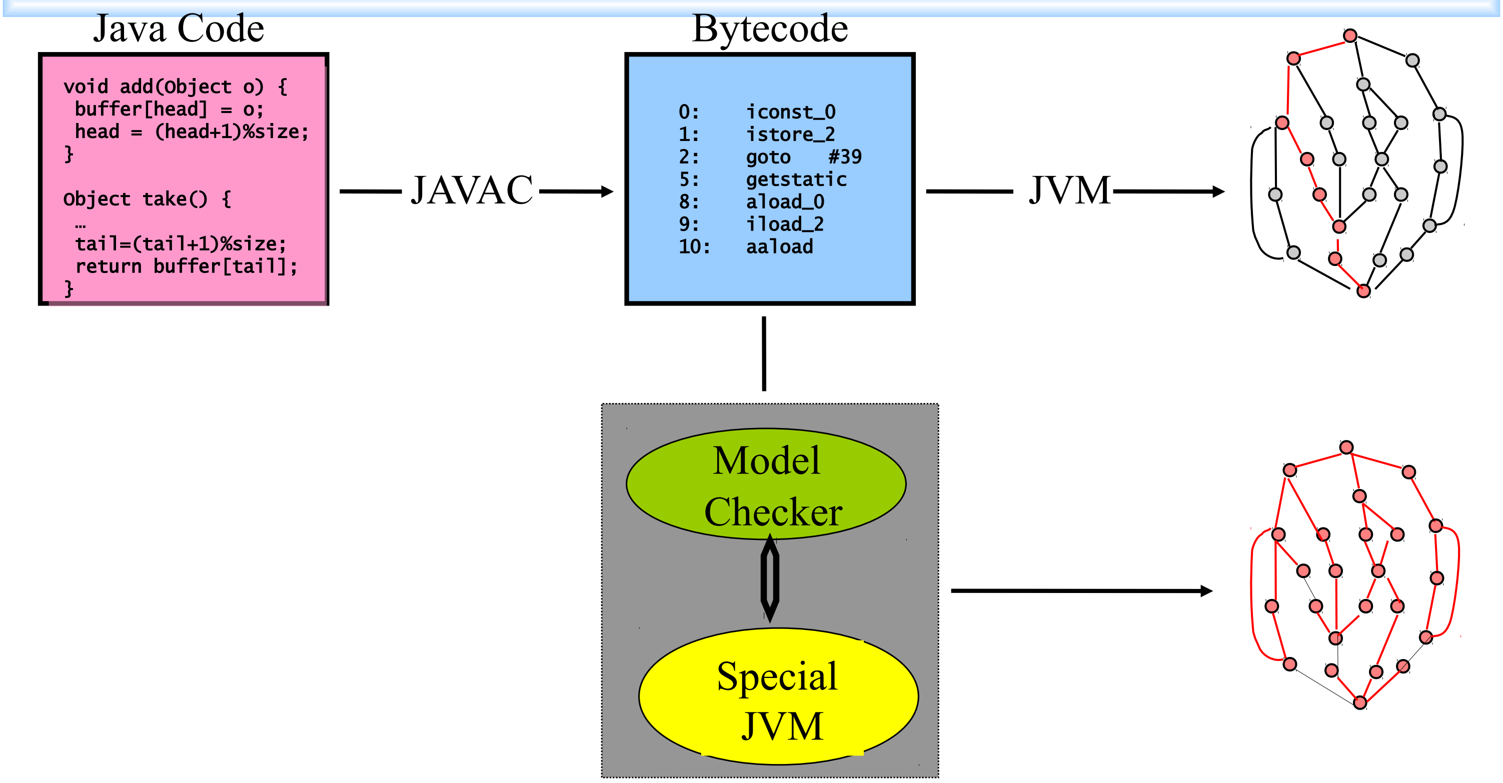
# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
  - Current Trends
    - Custom-made model checkers for programs
    - SLAM
    - JPF
      - Abstractions
      - Partial-order and symmetry reductions
      - Heuristics
      - New Stuff
    - Summary
    - Examples of other software analyses
- Case Studies
- Future of Software Model Checking

# Java PathFinder (2)
# Direct Approach

- Based on custom-made Java Virtual Machine
  - Handle all of Java, since it works with bytecodes
  - Written in Java
- Efficient encoding of states
- Modular design for easy extensions
- Supports LTL checking with properties expressed in Bandera’s BSL notation
- Incorporates a number of search strategies
  - DFS, BFS, A*, Best-first, etc.
- Supports source-2-source abstractions
- http://ase.arc.nasa.gov/jpf

# Java PathFinder (JPF)

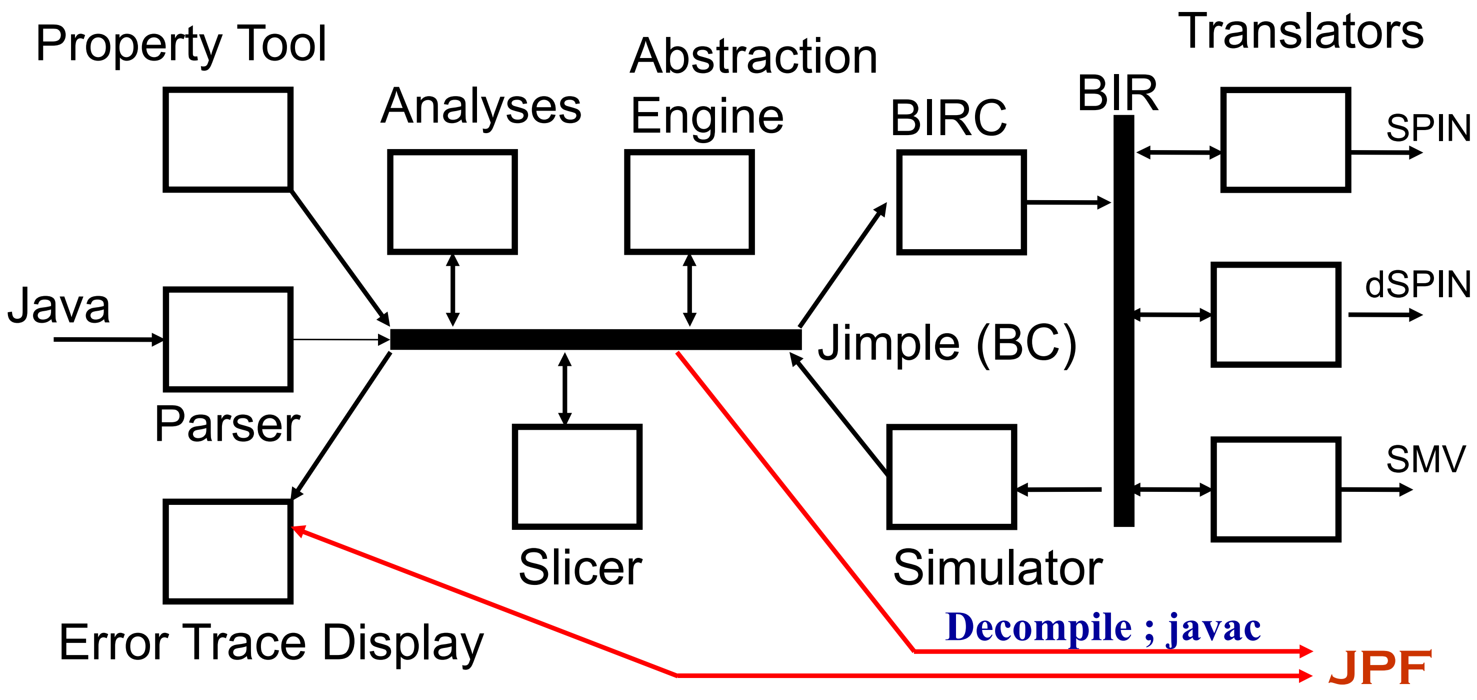


```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

```
0:    iconst_0
1:    istore_2
2:    goto    #39
5:    getstatic
8:    aload_0
9:    iload_2
10:   aaload
```

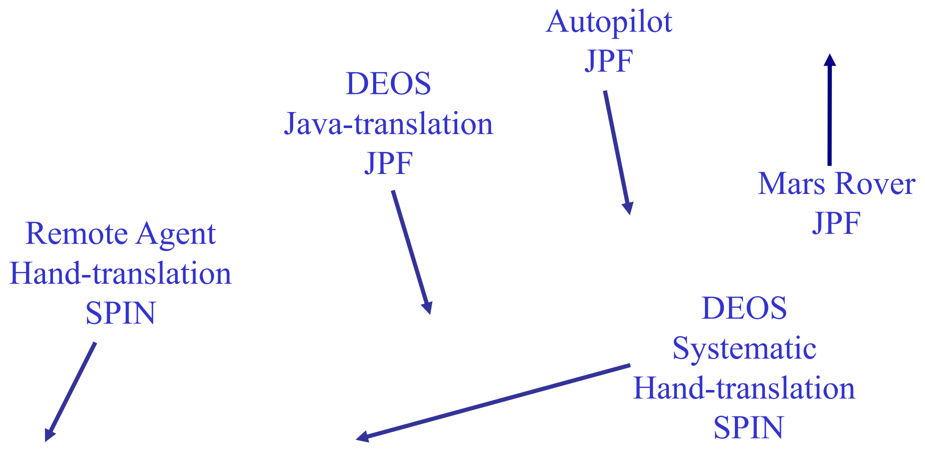
# Bandera & JPF Architecture



# Key Points

- Models can be infinite state
  - Unbounded objects, threads,…
  - Depth-first state generation (explicit-state)
  - Verification requires abstraction
- Handle full Java language
  - but only for closed systems
  - Cannot handle native code
    - no Input/output through GUIs, files, Networks, …
    - Must be modeled by java code instead
- Allows Nondeterministic Environments
  - JPF traps special nondeterministic methods
- Checks for User-defined assertions, deadlock and LTL properties

# Scaling Program Model Checking
# Error-Detection



# Overview

- Introduction to Model Checking
- Program Model Checking
  - Major Trends
  - A Brief History
  - Current Trends
    - Custom-made model checkers for programs
    - SLAM
    - JPF
    - Summary
    - Examples of other software analyses
- Case Studies
- Future of Software Model Checking

# Software Model Checking Executive summary

- Model checking by itself cannot deal with the complexity of software
- Techniques from static analysis are required
  - Abstract interpretation, slicing, alias&shape analysis, symbolic execution
- Even then, we need to borrow some more!
  - Heuristic search, constraint solving, etc.
- Abandon soundness
  - Aggressive heuristics
  - Runtime analysis and runtime monitoring

# More Software Analysis Techniques

## *A small sample*

2002

- Program Verification
  - For example, ESC/Java from Compaq
    - http://research.compaq.com/SRC/esc/
- Static analysis for runtime errors
  - For example, PolySpace for C, Ada and Java
    - http://www.polyspace.com/
- Requirements and Design Analysis
  - Analysis for SCR, RSML, Statecharts, etc.
- Runtime analysis
  - See Runtime Verification Workshops
    - http://ase.arc.nasa.gov/rv2002/
- Analysis Toolsets
  - IF (Verimag), SAL (SRI), etc.

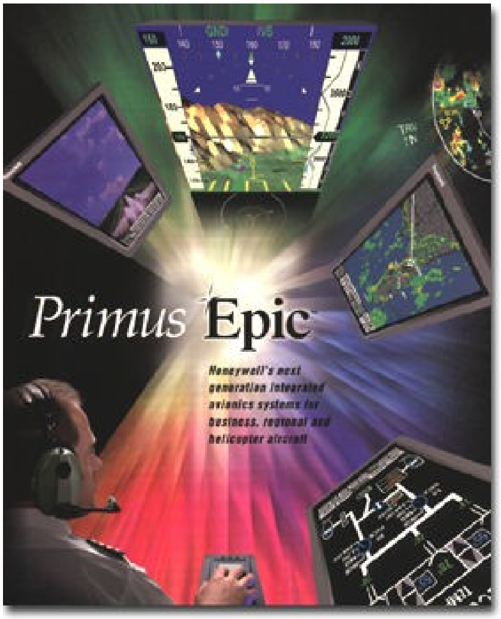
# Overview

2002

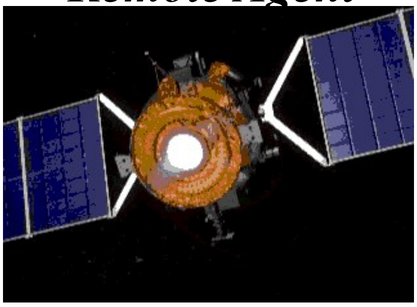
- Introduction to Model Checking
- Program Model Checking
- Case Studies
  - Remote Agent
  - DEOS
  - Mars Rover
- Future of Software Model Checking

# Case Studies of JPF

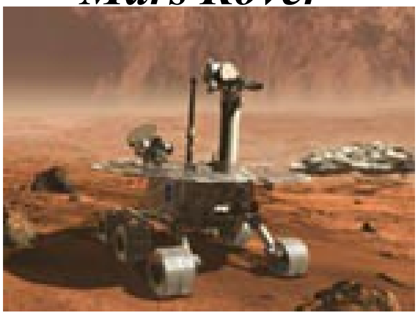
## *DEOS*


Primus Epic
Honeywell's next generation integrated avionics systems for business, regional and helicopter aircraft

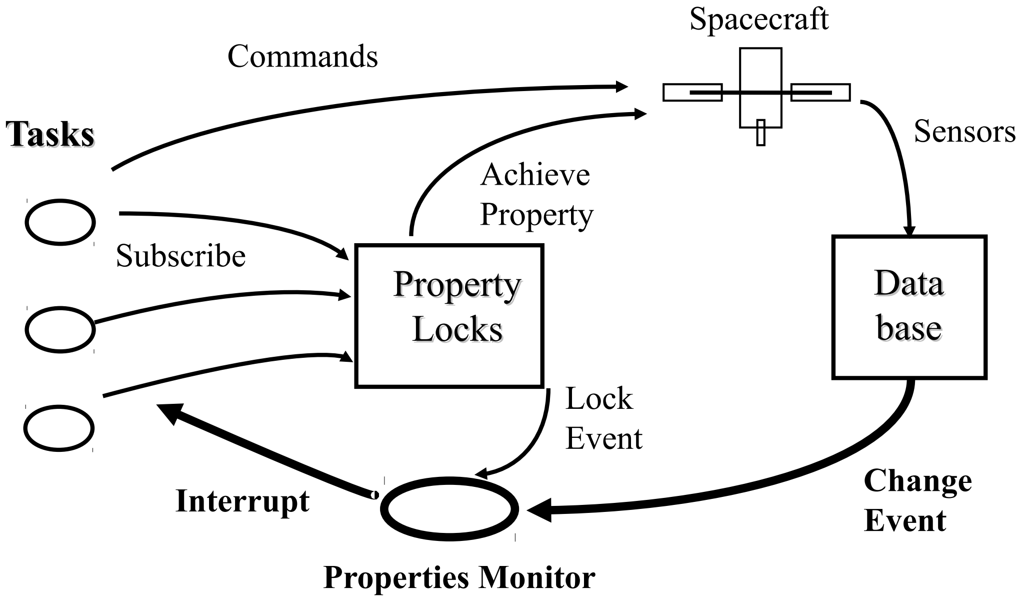
## *Remote Agent*



## *Mars Rover*

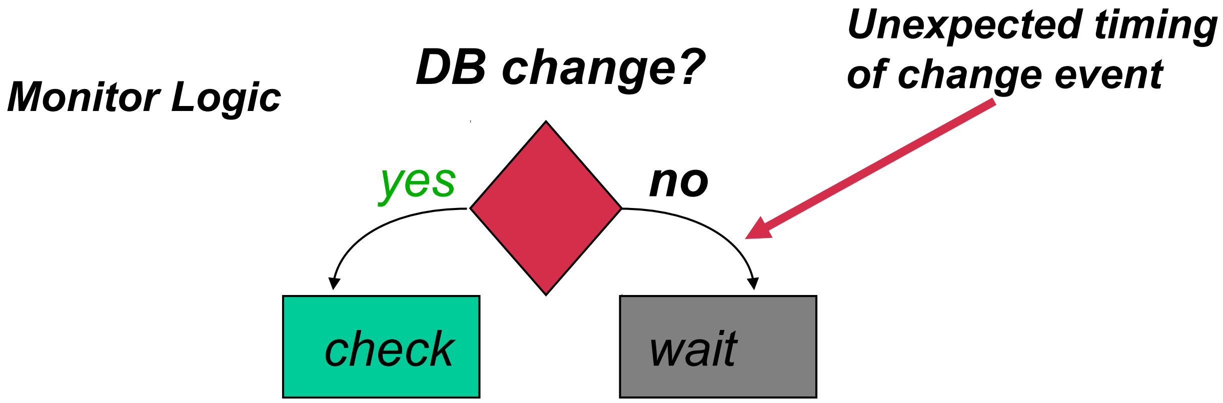


# Case Study: DS-1 Remote Agent



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

# Case Study: DS-1 Remote Agent



- 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

# DEOS Operating System

- 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

# DEOS Analysis

- Translated C++ 1-to-1 to PROMELA/SPIN (1500 lines of C++ code)
  - Found the time-partitioning error without any prior knowledge, what the error was, where it was or what made it show up.
  - Required very limited abstraction
- DEOS Team Reaction
  - Surprised that error was found by directly checking code
  - They expected NASA team to ask for smaller "slice"
  - They now have their own model checking group building on our work
- Then translated DEOS to Java and applied JPF
  - Backwards dependency analysis from the time partitioning assertion being checked revealed candidate variables to abstract
  - Applied "range" abstraction {0,1,many} to a specific integer variable
  - Too much of an over-approximation that led to many spurious errors
  - However with the choose-free heuristic *the* non-spurious error was found

# Analysis of the K9 Mars Rover

## *"The Experiment"*

- Rover is 8000 lines of code with 6 threads
  - heavy use of synchronization between the threads
  - Complex queue manipulation
- Purpose
  - Benchmark current state of the art in model checking, static analysis for runtime error detection and runtime analysis
  - Use traditional testing as baseline
  - Original code was in C++ that was translated to Java
    - About half the code was translated to C for the static analysis that used PolySpace
- Method
  - Controlled experiment: 4 groups of 2 people, each group uses one technology on the Mars rover code to find seeded bugs
  - 3 versions created and each group gets 2 days/version
  - Some bugs are removed/introduced between versions
  - Any new bugs discovered are not fixed, only known ones

# Analysis of the K9 Mars Rover
## How did Model Checking do?

- Methodology for model checking
  - Asked never to “run” the code, only model check it
    - Keep the results clean from any testing influence
  - Code is heavily dependent on time
    - Given a gross over-approximation of time, where all time-related decisions became nondeterministic
- Found all, but one, of the known concurrency errors and some new ones
  - Better than any of the other teams
  - Only team that could always produce not just the error but how to get to it!
  - Also found all the non-concurrency errors
- Interesting observations
  - 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
  - Found a number of bugs in the first version, had a slow 2nd version, and then found all the remaining bugs in the 1st hour of the 3rd version
    - Took them some time to get their framework setup, but once done, they were flying

# Overview

- Introduction to Model Checking
- Program Model Checking
- Case Studies
- Future of Software Model Checking

# The Future of Software Model Checking

- Abstraction based approaches
  - Combine object abstractions (e.g. shape analysis) with predicate abstraction
  - Automation is crucial
- Symbolic Execution
  - Solving structural (object) and numerical constraints
  - Acceleration techniques (e.g. widening)
- Model checking as a companion to testing
  - Test-case generation by model checking
  - Runtime monitoring and model checking
- Modular model checking for software
  - Exploiting the interface between components
  - Interface automata (de Alfaro & Henzinger )
- Environment generation
  - How to derive a “test-harness” for a system to be model checked
- Result representation
  - Much overlooked, but without this we are nowhere!
  - “Analysis is necessary, but not sufficient” – Jon Pincus