Software safety - DEF-STAN 00-55

- "Where safety is dependent on the safety related software (SRS) fully meeting its requirements, demonstrating **safety** is equivalent to demonstrating **correctness** with respect to the Software Requirement".

Ultimate problems addressable by model checking

- Checking correctness of the code running on the application
  - Two main approaches:
  - Code Model Checking (*Software Model Checking*)
  - Model Based Development
- Checking safety of the system (the system never runs into an unsafe state)
  - Concentrating on safety properties on a Model of the system
  - Opening to probabilistic safety

Software Model Checking

- Although the early papers on model checking focused on software, not many applications to prove the correctness of code, until 1997
- Until 1997 most work was on software designs
  - Finding bugs early is more cost-effective
  - Reality is that people write code first, rather than design
- Only later the harder problem of analyzing actual source code was first attempted
- Pioneering work at NASA

Software Model Checking

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

- Bringing programs to model checking
  - **Translation** to a standard Model Checker
- Bringing model checking to programs
  - **Ad hoc** model checkers that directly deal with programs as input

- *In both cases, need of* ***Abstraction****.*

Abstraction

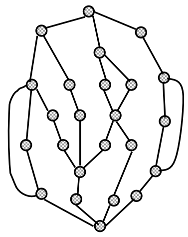
Program

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

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

Possibly 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 cuts the state space size to something manageable*
- Abstraction eliminates details irrelevant to the property
- Disadvantage: Loss of Precision: False positives/negatives

- 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

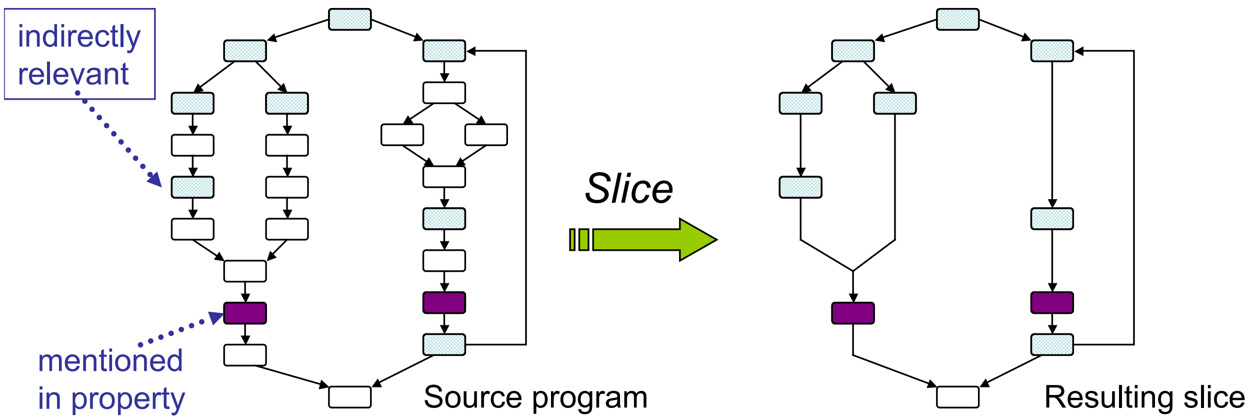
Under-Approximation
"Meat-Axe" Abstraction

- Remove parts of the program considered "irrelevant" for the property being checked, e.g.
  - Limit input values to 0..10 rather than all integer values
  - Queue size 3 instead of unbounded, etc.
- The abstraction of choice in the early applications of software model checking
- Used during the translation of code to a model checker's input language
- Typically manual, no guarantee that only the irrelevant behaviors are removed.

Precise abstraction

- 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

Property-directed Slicing



- slicing criterion generated automatically from observables mentioned in the property
- backwards slicing automatically finds all components that might influence the observables.

Over-Approximations

Abstract Interpretation

- Maps sets of states in the concrete program to one state in the abstract program
  - Reduces the number of states, but increases the number of possible transitions, and hence the number of behaviors
  - Can in rare cases lead to a precise abstraction
- Type-based abstractions (-->)
- Predicate abstraction (-->)
- Automated (conservative) abstraction
- Problem: Eliminating spurious errors
  - Abstract program has more behaviors, therefore when an error is found in the abstract program, is that also an error in the original program?

Data Type Abstraction

Abstraction homomorphism h: int --> Sign

Replace int by Sign abstraction {neg,pos,zero}

$$h(x) = \begin{cases} \text{NEG} & \text{if } x<0 \\ \text{ZERO} & \text{if } x=0 \\ \text{POS} & \text{if } x>0 \end{cases}$$

Code

```
int x = 0;
…
if (x == 0)
  x = x + 1;
```

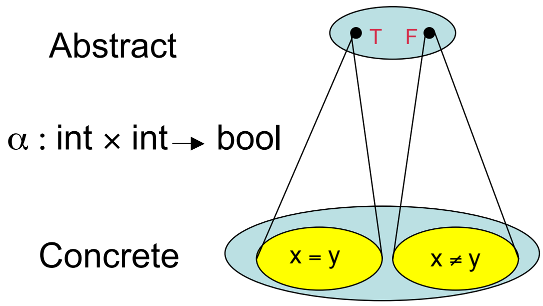
$\xrightarrow{h}$

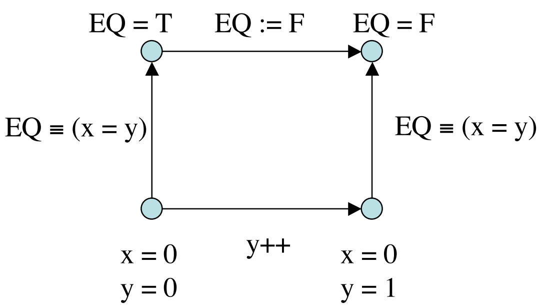
Abstract Interpretation

```
Sign  x = ZERO;
…
if (Sign.eq(x,ZERO))
  x = Sign.add(x,POS);
```

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.



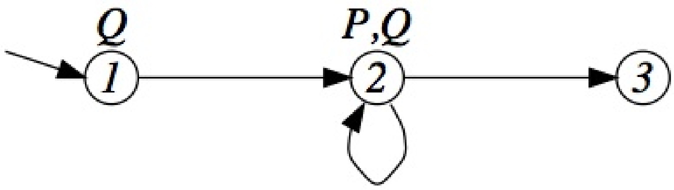


- Mapping of a concrete system to an abstract system, whose states correspond to truth values of a set of predicate
- Create abstract state-graph during model checking, or,
- Create an abstract transition system before model checking

How do we Abstract Behaviors?

- Abstract domain A
  - Abstract concrete values to those in A
- Then compute transitions in the abstract domain
  - Over-approximations: Add extra behaviors
  - Under-approximations: Remove actual behaviors

Underlying model: Kripke Structures



- M = (S,s_0,–>,L) on AP
  - S: Set of States
  - s_0: Initial State
  - ->: Transition Relation
  - L: S -> 2^{AP}, Labeling on States

Simulations on Kripke Structures

M = (S, s_0, ->, L)
M' = (S', s'_0, ->', L')
Definition: $R \subseteq S \times S'$ is a simulation relation
between *M* and *M'* iff

(s,s') $\in$ R implies

1. L(s) = L'(s')
2. for all t s.t. $s \rightarrow t$,
   exists t' s.t. $s' \rightarrow' t'$ and
   (t,t') $\in$ R.

M' simulates M (M ~ M') iff (s_0, t_0) $\in$ R

Intuitively, every transition in *M* can be matched by some transition in *M'*

Preservation of properties by the Abstraction

- M concrete model, M' abstract model
- *Strong Preservation:*
  - M' |= P iff M |= P
- *Weak Preservation:*
  - M' |= P => M |= P
- *Simulation* preserves ACTL* properties
  - If M ~ M' then M' |= *AG* p => M |= *AG* p

Abstraction Homomorphisms

- Concrete States S, Abstract states S'
- Abstraction function (Homomorphism)
  - h: S -> S'
  - Induces a partition on S equal to size of S'
- Existential Abstraction - Over-Approximation
  - Make a transition from an abstract state if **at least one** corresponding concrete state has the transition.
  - Abstract model M' simulates concrete model M
- Universal Abstraction - Under-Approximation
  - Make a transition from an abstract state if **all** the corresponding concrete states have the transition.

Existential Abstraction - Preservation

- Let ϕ be a Universally quantified formula (es, an ACTL* property)
- M' existentially abstracts M, so M ~ M'
- Preservation Theorem

$$M' \models \phi \rightarrow M \models \phi$$

- Converse does not hold

$$M' \models \phi \not\rightarrow M \models \phi$$

- $M' \not\models \phi$: counterexample may be spurious
- *NOTE: ACTL* is the universal fragment of CTL**

Universal Abstraction - Preservation

- Let ϕ be a existential-quantified property (i.e., expressed in ECTL*) and M simulates M'
- Preservation Theorem

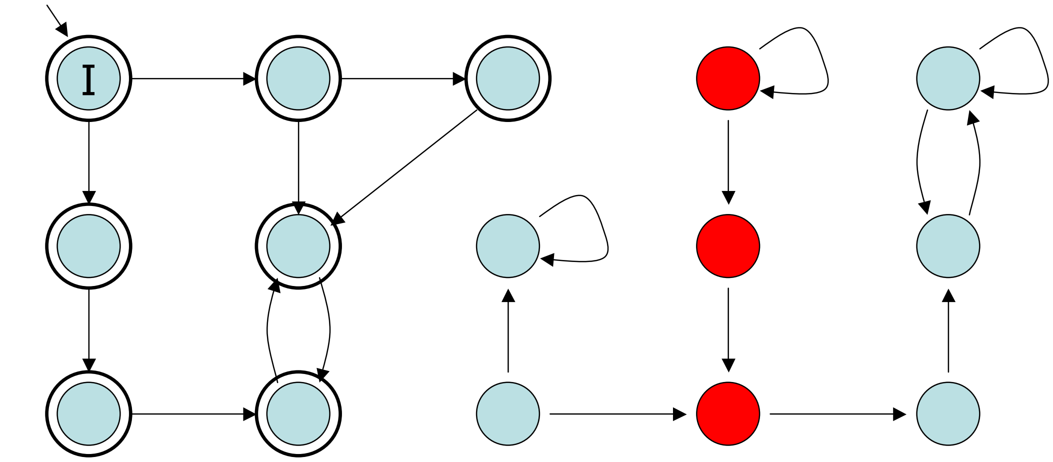
$$M' \models \phi \rightarrow M \models \phi$$

- Converse does not hold

$$M \models f \not\rightarrow M' \models f$$

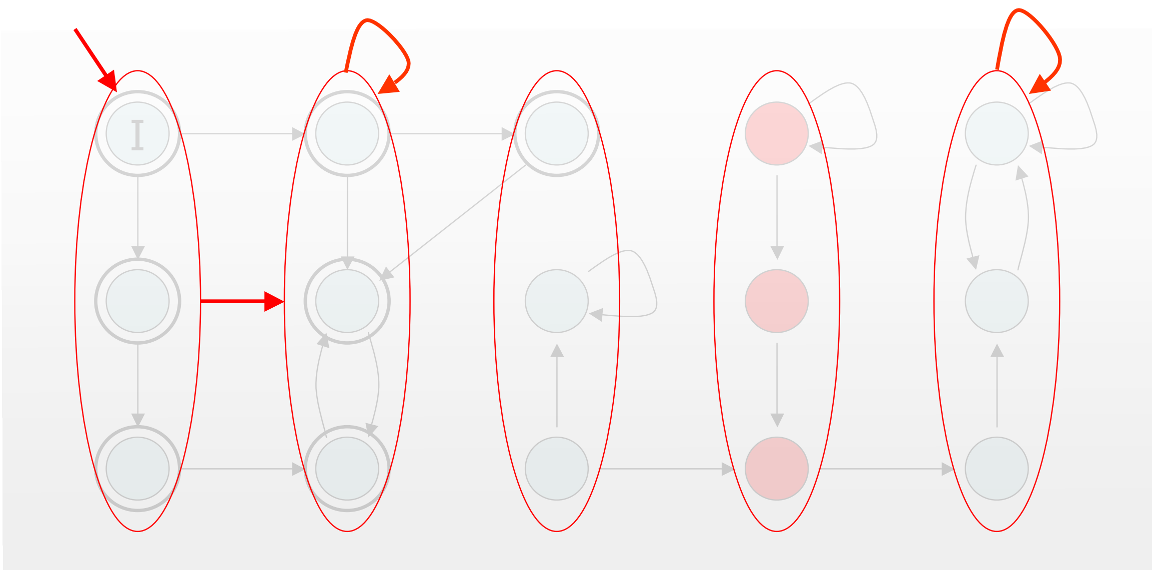
NOTE: ECTL is the universal fragment of CTL**

Model Checking (safety)



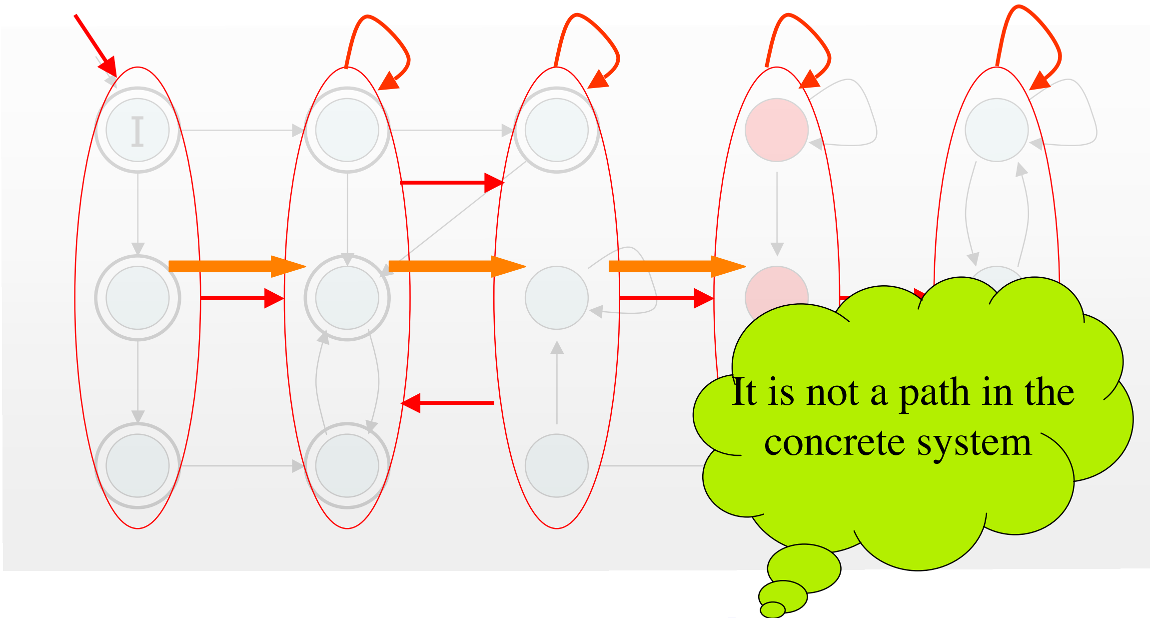
= unsafe state *AG ~ unsafe* *(true property)*

Abstraction: Under-Approximation



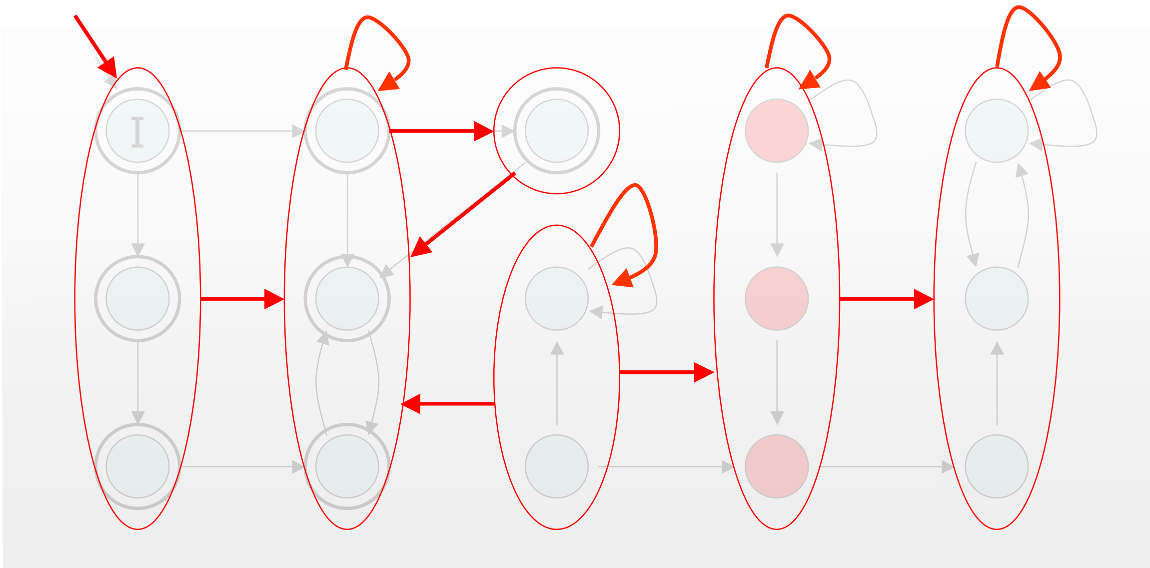
AG ~ unsafe ***true*** *(but it is not preserved)*

Abstraction: Over-Approximation



AG ~ unsafe ***false*** ***spurious*** *counterexample*

Refinement of the abstraction :

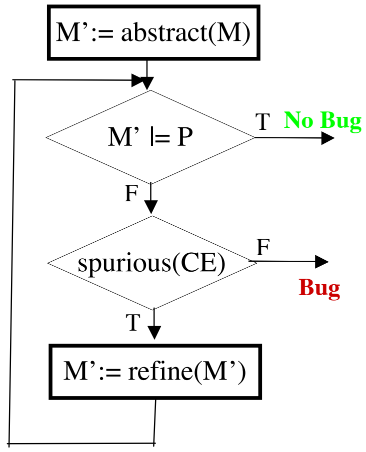


Separate states that are the reason of the spurious counterexample

AG ~ unsafe ***true***

Automated Abstraction/Refinement

- Counterexample-Guided AR (CEGAR)
  - Build an abstract model M'
  - Model check property P, M' |= P?
  - If M' |= P, then M |= P by Preservation Theorem
  - Otherwise, check if Counterexample (CE) is spurious
  - Refine abstract state space using CE analysis results
  - Repeat



Hand-Translation
Early applications at NASA

- 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 – 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

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

Bringing Model Checking to Programs

- Allow model checkers to take programming languages as input, (or notations of similar expressive power)
- Major problem: how to encode the state of the system efficiently
- Alternatively state-less model checking
  - No state encoding or storing
  - *On the fly* model checking
- Almost exclusively explicit-state model checking
- Abstraction can still be used as well
  - Source to source abstractions

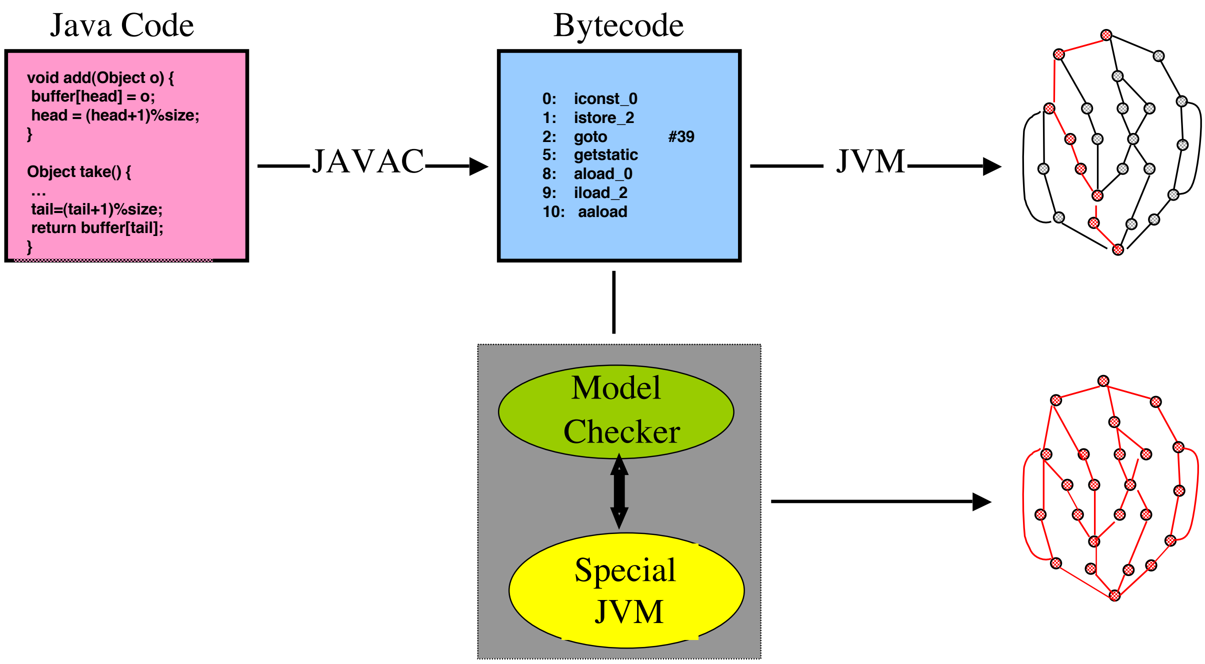
Custom-made Model Checkers

- 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/

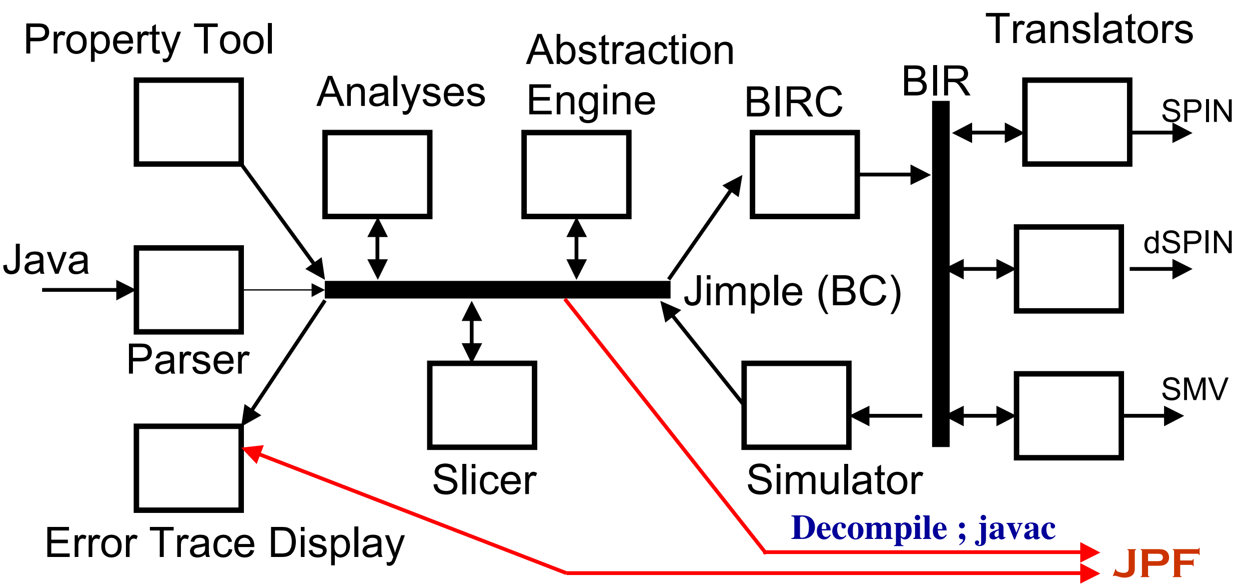
Custom-made Model Checkers

- 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/

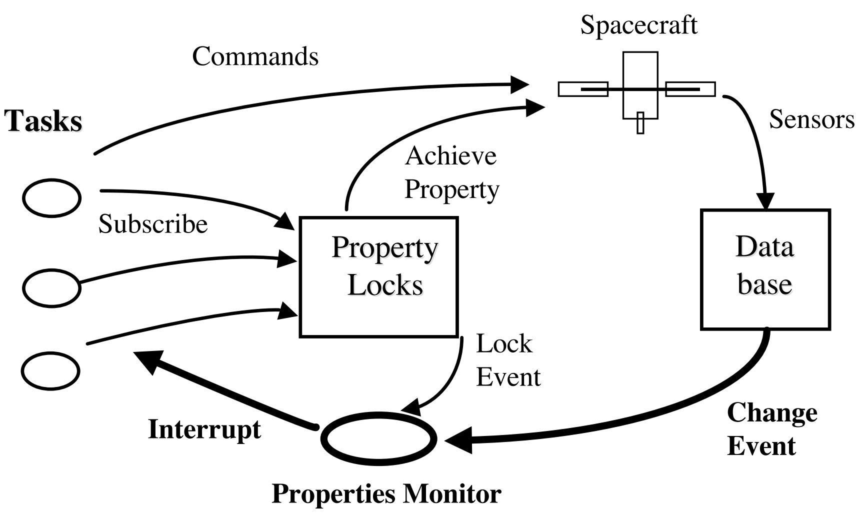
Java PathFinder (JPF)



Bandera & JPF Architecture

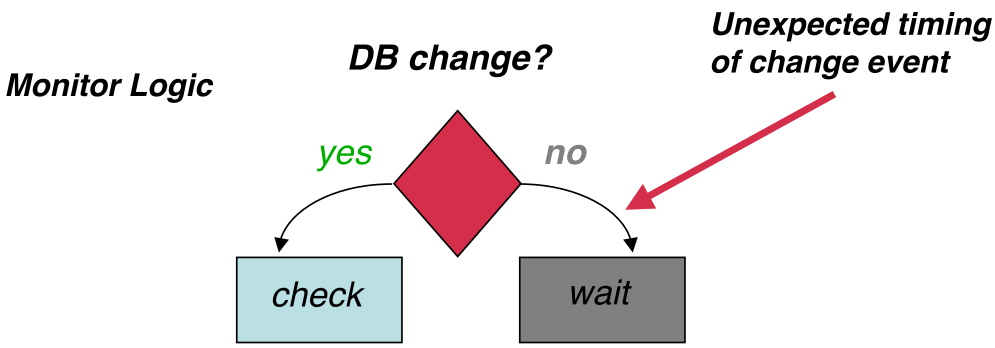


One Case Study at NASA: DS-1 Remote Agent



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

One Case Study at NASA: DS-1 Remote Agent



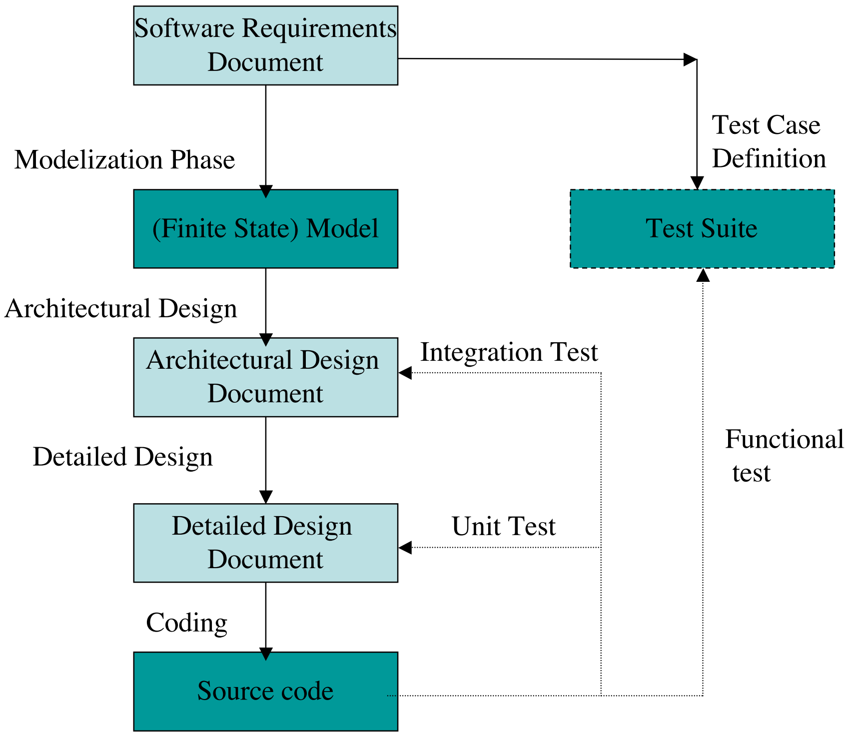
- Five difficult to find concurrency errors detected
- "[Model Checking] has had a substantial impact, helping the RA team improve the quality of the Executive well beyond what would otherwise have been produced." - RA team
- During flight RA deadlocked (in code we didn't analyze)
  - Found this deadlock with JPF

Model Based Development

- Pioneering work at NASA has concentrated on Software Model Checking, that is, work on software as it is, maybe provided by a third party.
- In a large part of the safety-critical systems industry, the Model Based Design approach has emerged as the main paradigm for the development of software.

Model - Based Development

Manual coding

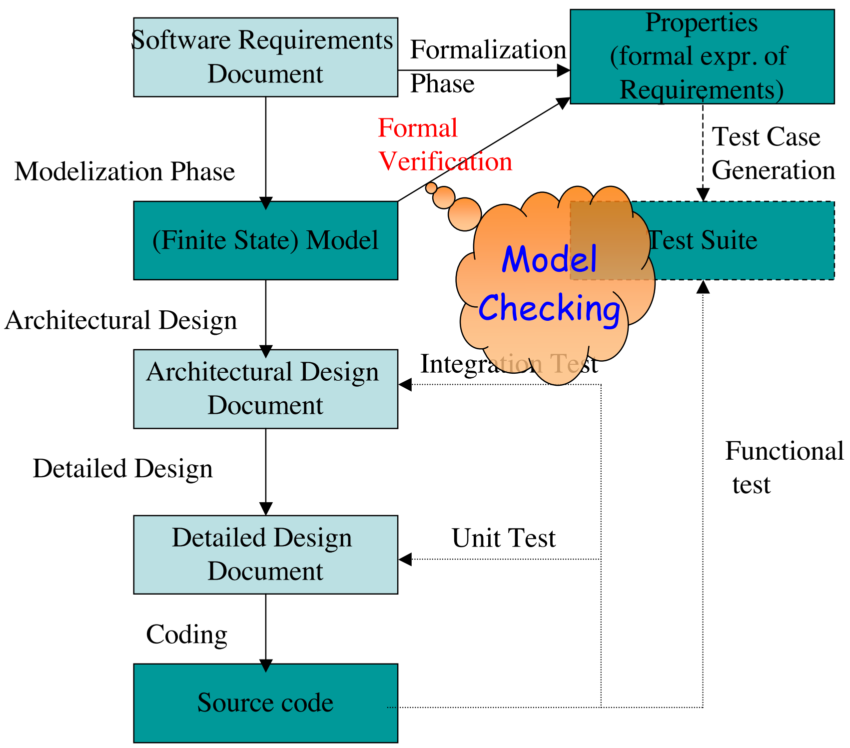


Natural Language

Formalized

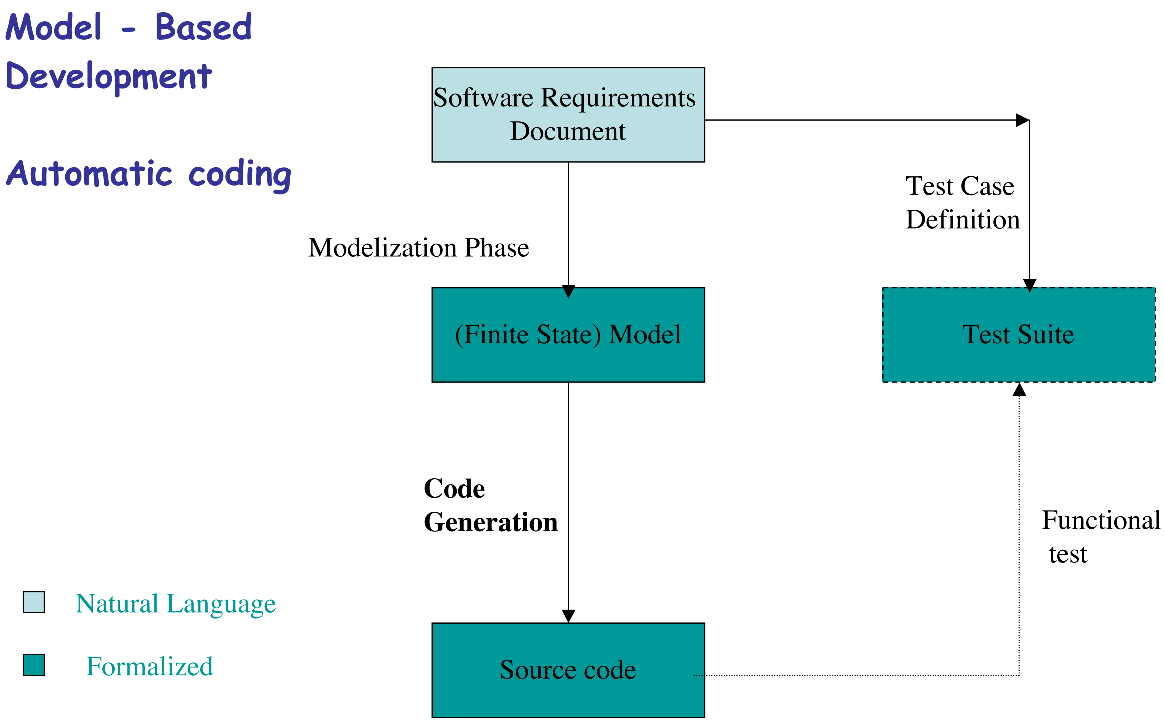
Model - Based Development

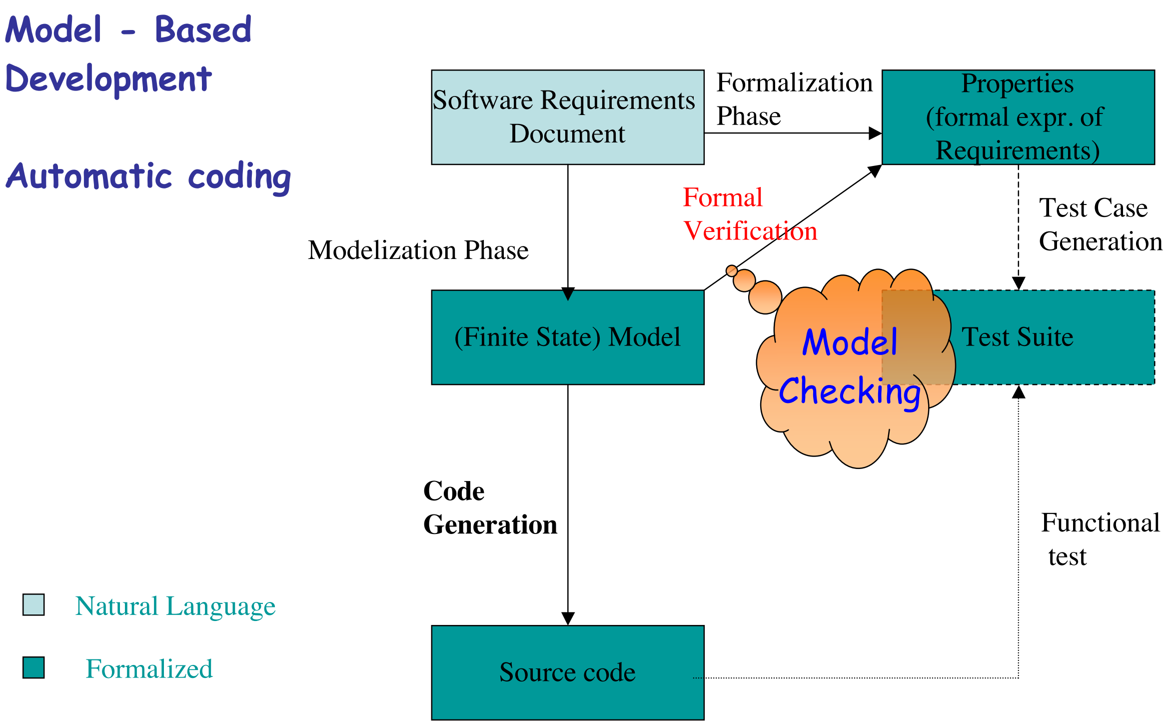
Manual coding



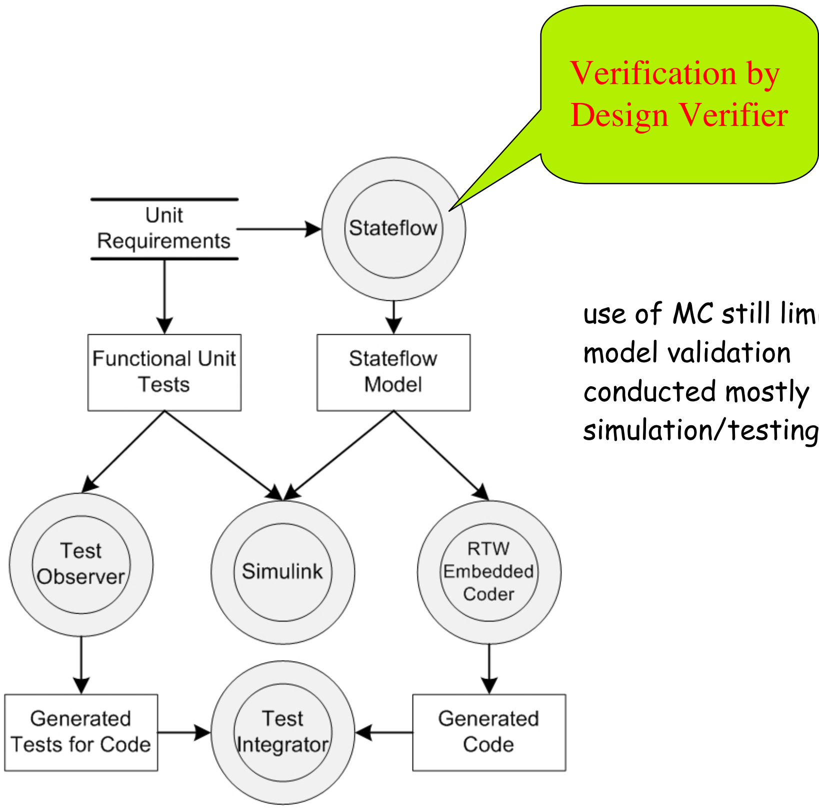
Natural Language

Formalized


Model - Based Development
Automatic coding
Software Requirements Document
Test Case Definition
Modelization Phase
(Finite State) Model
Test Suite
Code Generation
Functional test
Natural Language
Formalized
Source code


Model - Based Development
Automatic coding
Software Requirements Document
Formalization Phase
Properties (formal expr. of Requirements)
Formal Verification
Test Case Generation
Modelization Phase
(Finite State) Model
Model Checking
Test Suite
Code Generation
Functional test
Natural Language
Formalized
Source code

GETS model based development cycle



use of MC still limited: model validation conducted mostly by simulation/testing

Credits

- Willem Visser. ASE 2002 Tutorial on Software Model Checking
- Nishant Sinha. Lectures on Abstraction in Model Checking (ppt), 15817, Mar 2005.