Model checking
-
Applications to safety-critical systems

A. Fantechi
Università di Firenze

The promise of Model Checking

- Due to the possibility offered by model checking to give a definite result on the satisfaction of a property by a system, model checking has been considered as a very interesting technique in the realm of critical systems, where safety could be put at stake by software errors.

- System |= AG (~ badstate) **SAFETY PROPERTY**

- *Exhaustive technique*, opposed to testing: 100% coverage of the system states
- *Pushbutton technique:* in principle no need of effort to reason on the system

Problems

- Complexity of real systems
- Scalability of verification techniques to enormous number of states
- Definition of (safety) properties
- Relationships with accepted guidelines for the development of Safety Critical Systems
- Industrial strength model checking tools

Outline of the lecture

- Definitions about safety
- The role of Model Checking in safety critical software:
  - Software Model Checking
  - Model Based Development
  - Experiences in safety-critical domains
    - Space
    - Avionics
    - Railway Signalling
- Some bits on the role of Model Checking in safety critical systems:
  - Quantitative safety evaluation
- Model Checking and safety guidelines

Classical definitions of Safety and Liveness

- Safety properties
  - Invariants, deadlocks, reachability, etc.
  - Can be checked on finite traces
  - "something bad never happens"
  - AG ~ bad
- Liveness Properties
  - Fairness, response, etc.
  - Infinite traces
  - "something good will eventually happen"
  - EF good
  - "something good will infinitely often happen"
  - AGF good

These definitions have only in part something to do with the reality of safety-critical systems

Criticality levels - a view

- **safety-critical systems**: in which a failure can cause deaths or serious injuries, or serious environmental damage.
  - chemical plant control systems,
  - X-by-wire systems, where X = fly (avionics)
    or X ∈ {drive,brake} (automotive)
- **mission-critical systems**: in which a failure can cause aborting an activity aimed to an important objective
  - the navigation system of a space probe going to explore a far planet: if the probe is lost, the investment is completely gone.
- **business-critical systems**: in which a failure can cause enormous money loss
  - a bank's client accounts management system

Criticality levels (from the *Avionics Handbook*)

- A ***flight-critical function*** is one whose loss might result in the loss of the aircraft itself, and possibly the persons on-board as well. In the latter case, the system is termed ***safety-critical***.
- Here, a distinction can be made between a civil transport, where flight-critical implies safety-critical, and a combat aircraft. The latter admits to the possibility of the crew ejecting from an unflyable aircraft, so its system reliability requirements may be lower.
- A ***mission-critical function*** is one whose loss would result in the compromising or aborting of an associated mission. For avionics systems, a higher cost usually associates with the loss of an aircraft than with the abort of a mission (an antimissile mission to repel a nuclear weapon could be an exception). Thus, a full-time flight-critical system would normally pose much more demanding reliability requirements than a flight-phase mission-critical system.

Safety definitions

- **safety:** Freedom from unacceptable levels of risk. (CENELEC EN 50126)
- **safety:**
  - An acceptable level of freedom from risks to personnel and material at all times.
  - The inherent property of a system, subsystem or item that enables it to possess and to maintain an acceptable level of risk during all situations and activities occurring during its specified life cycle. (AOP 38)
- **safety:** Freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment. (MIL-STD-882D)
- **safety:** The state in which risk is lower than the boundary risk. The boundary risk is the upper limit of the acceptable risk. It is specific for a technical process or state. (ARP4754)

From such definitions, two views of safety: Absolute vs. Probabilistic Safety

- Absolute safety requires that all causes of threats to safety are removed
- Probabilistic safety acknowledges the existence of possible residual unsafe events, although with less than a required maximum probability of occurrence

Systematic vs. Random faults

A parallel can be done with the nature of faults:

- some are systematic, and have their root in some design error:
  - all design errors should be removed
  - it is difficult/impractical to estimate a residual probability of occurrence of a systematic error
- some are random, and a probability of occurrence can be estimated on the basis of previous failure experience

Very roughly speaking:

- *Hardware has random failures*
- *Software has systematic failures*

Hence, hardware is subject to quantitative analysis of safety, (probabilistic safety) safe software should be "just" correct (absolute safety).

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

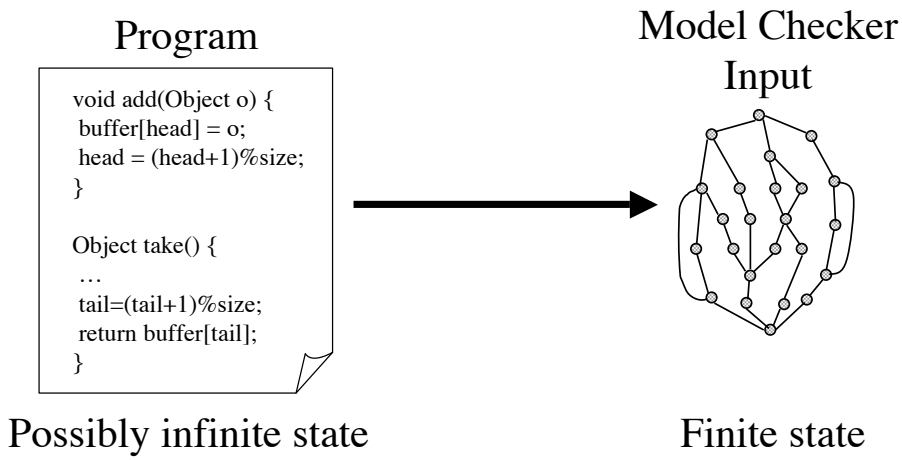
Software Model Checking

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



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

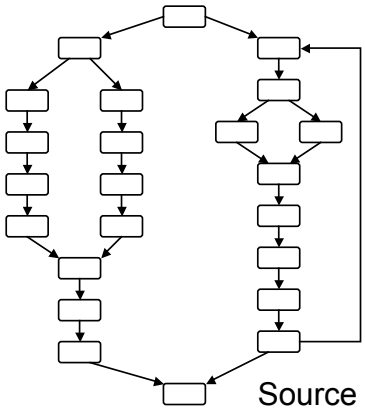
Under-Approximation
"Meat-Axe" Abstraction

- 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

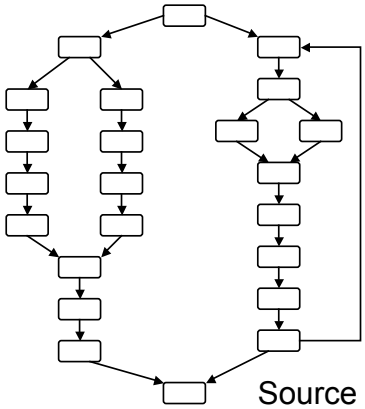


Source program

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



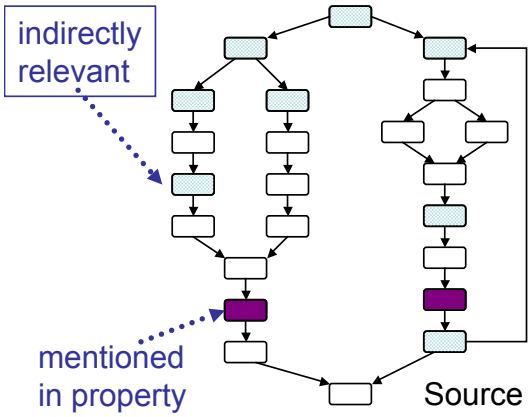
Source program

- slicing criterion generated automatically from observables mentioned in the property

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



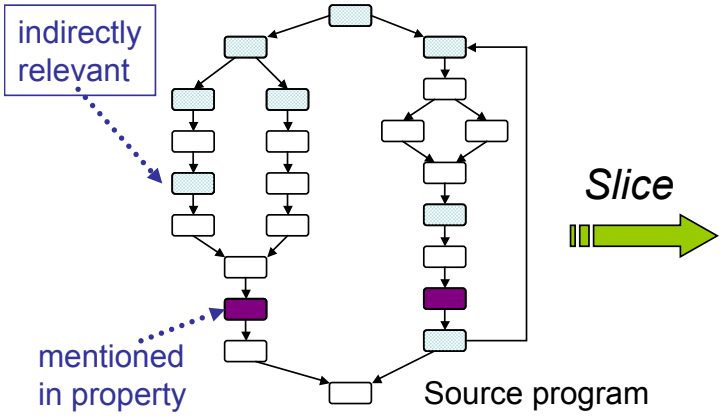
Source program

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

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



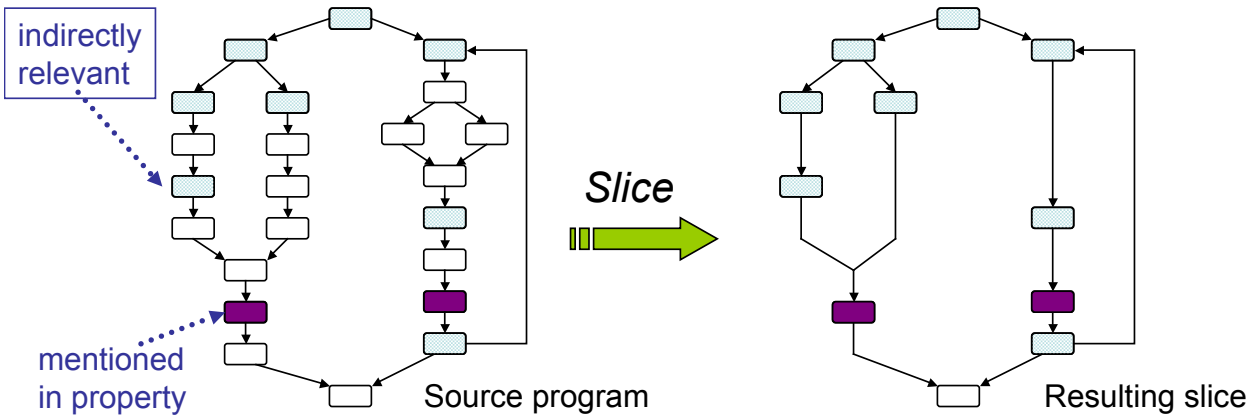
- slicing criterion generated automatically from observables mentioned in the property
- backwards slicing automatically finds all components that might influence the observables.

© Willem Visser 2002

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.

Sep-30-10 SEFM School 2010 18

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;
```

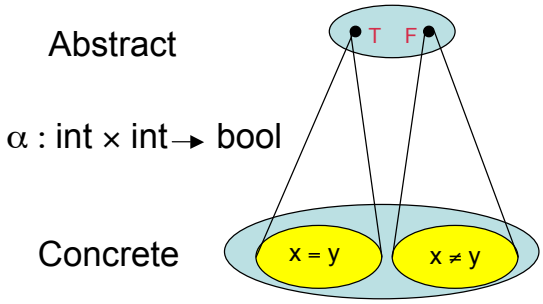
h →

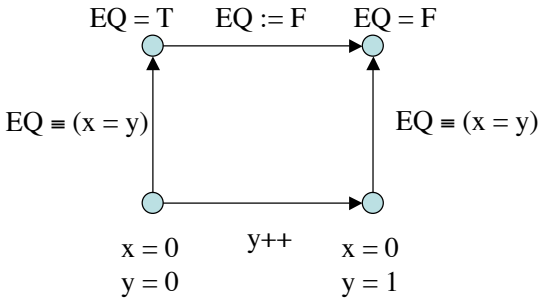
Abstract Interpretation

```
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.



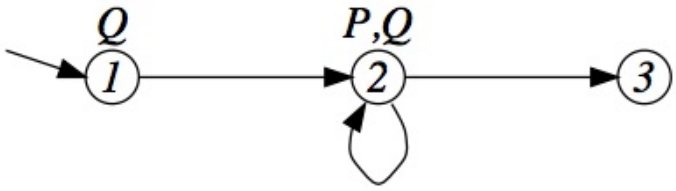


- Mapping of a concrete system to an abstract system, whose states correspond to truth values of a set of predicate
- Create abstract state-graph during model checking, or,
- Create an abstract transition system before model checking

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 ⊆ S × S' is a simulation relation between M and M' iff

(s,s') ∈ R implies

1. L(s) = L'(s')
2. for all t s.t. s → t ,
   exists t' s.t. s' →' t' and
   (t,t') ∈ R.

M' simulates M (M ~ M') iff (s_0, t_0) ∈ 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

Universal Abstraction - Preservation

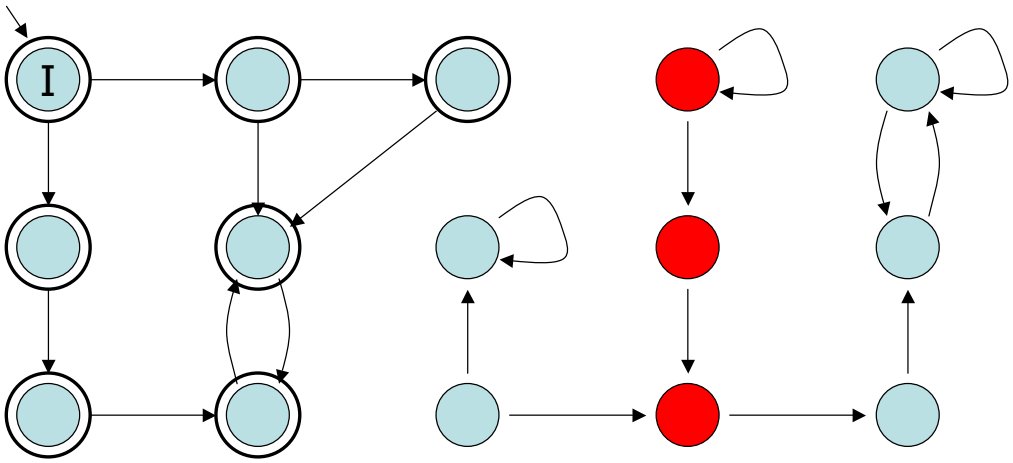
- Let ϕ be a existential-quantified property (i.e., expressed in ECTL*) and M simulates M'
- Preservation Theorem

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

- Converse does not hold

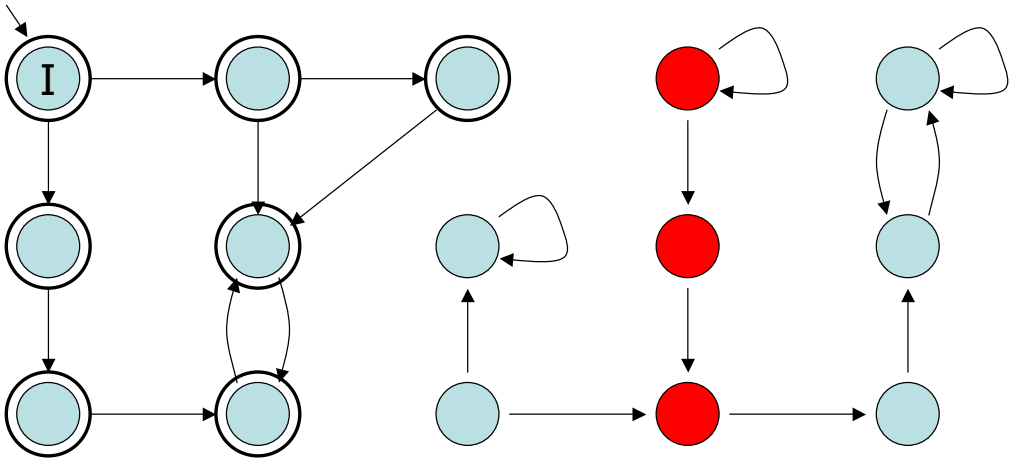
$$M \models f \not\rightarrow M' \models f$$

Model Checking (safety)



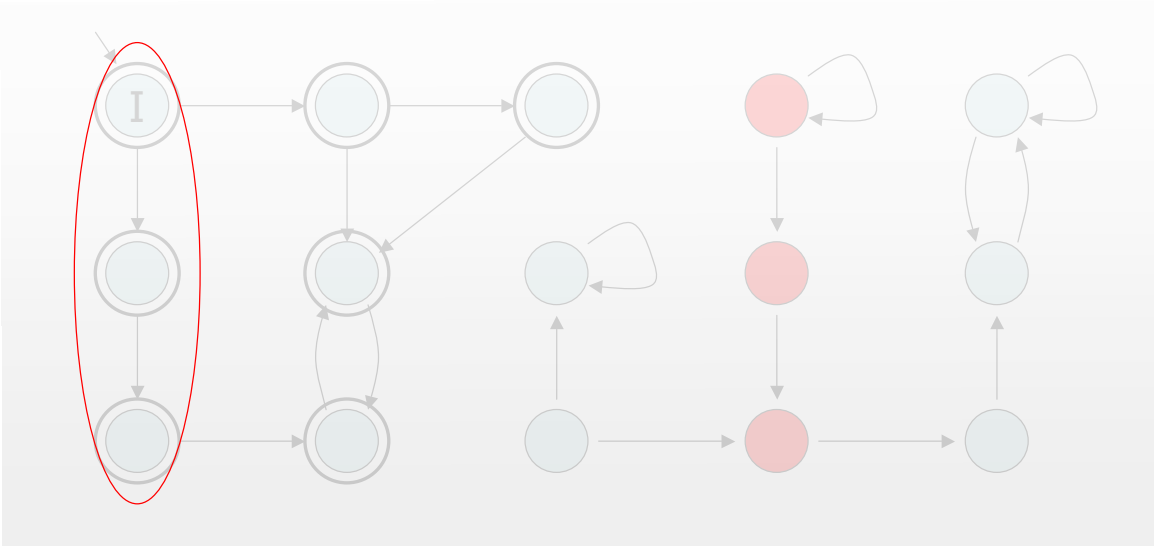
= unsafe state *AG ~ unsafe* *(true property)*

Abstraction:

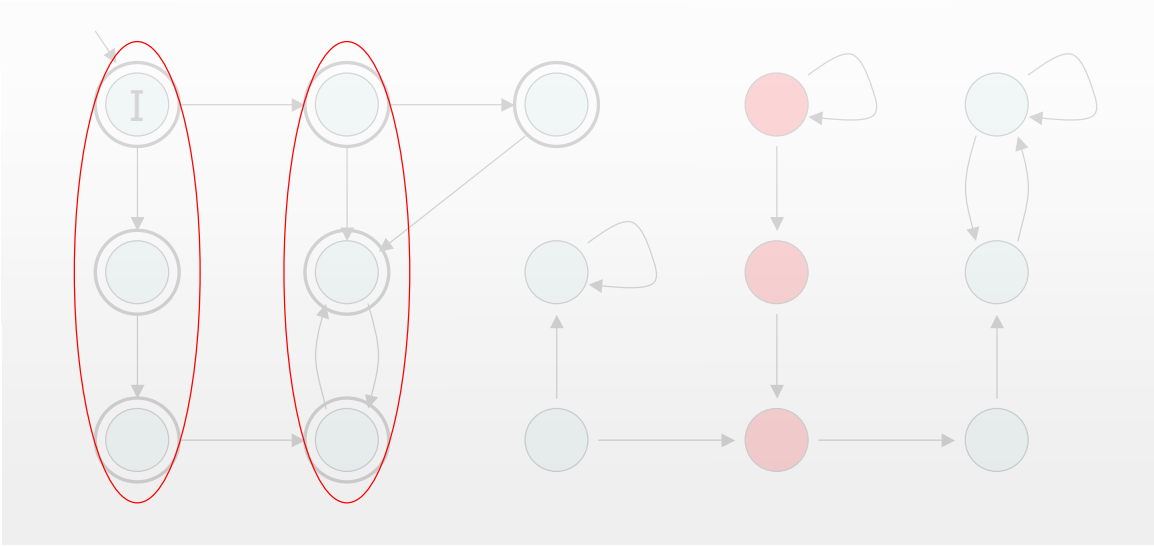


Abstraction:

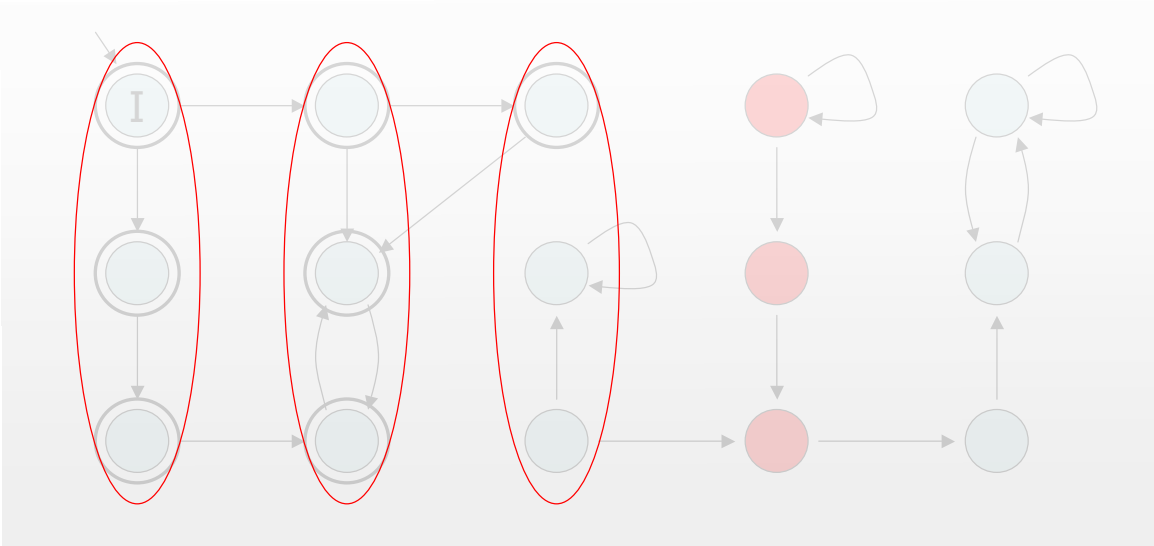
Abstraction:



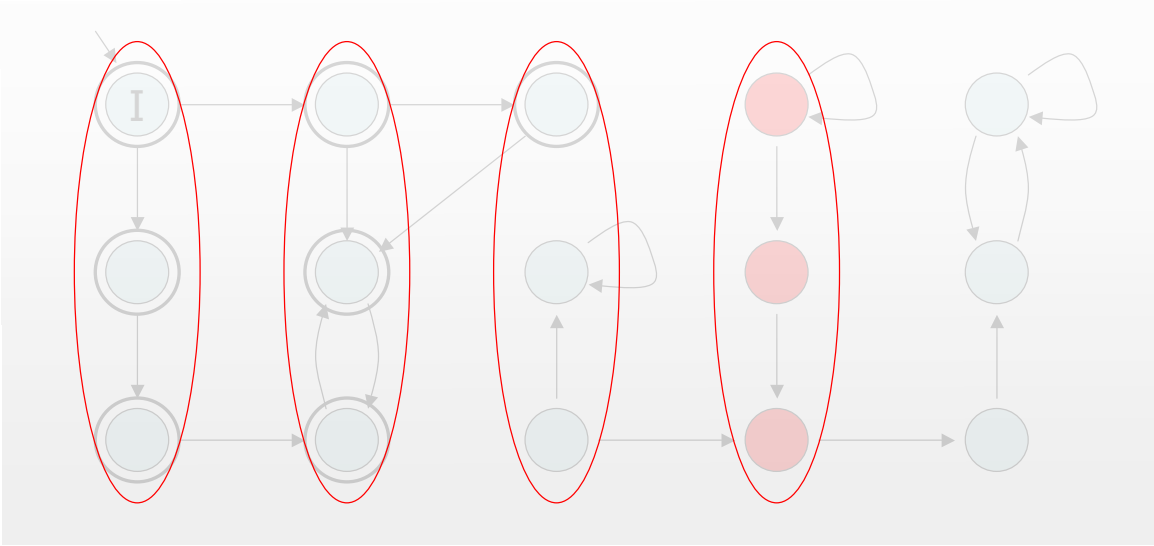
Abstraction:



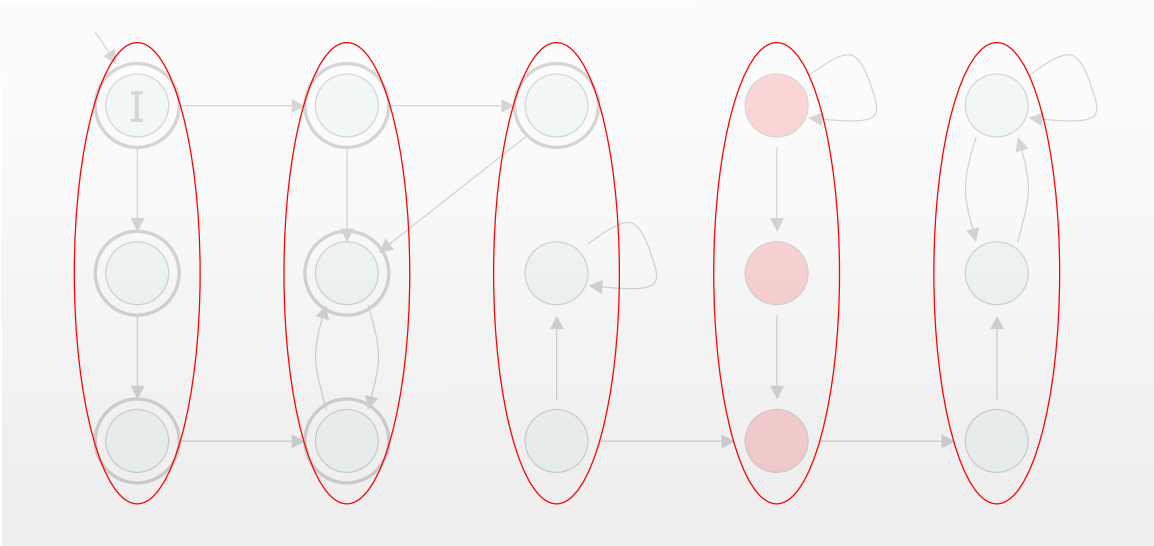
Abstraction:



Abstraction:

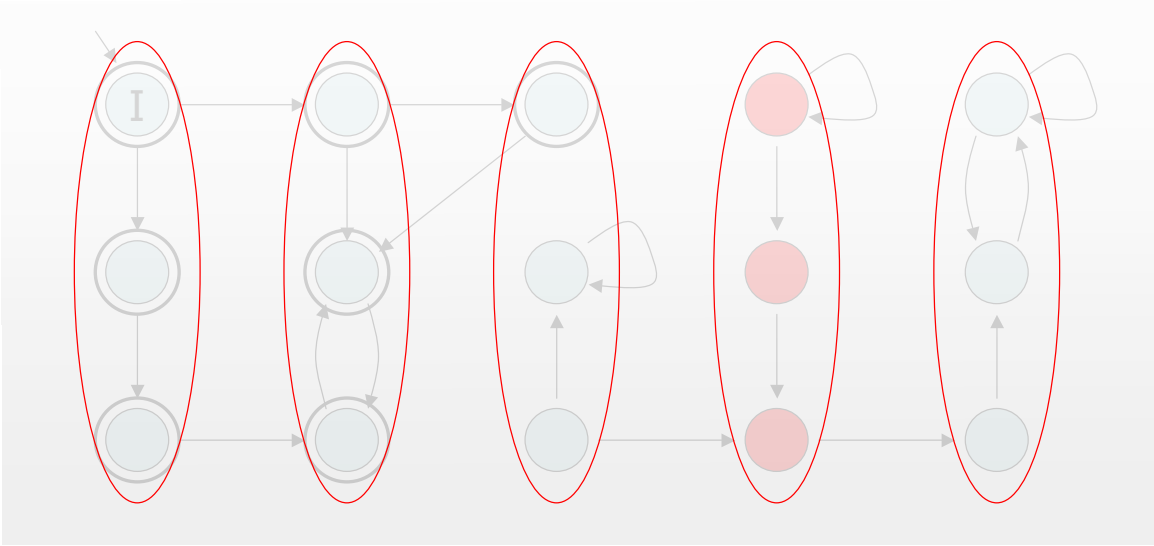


Abstraction:

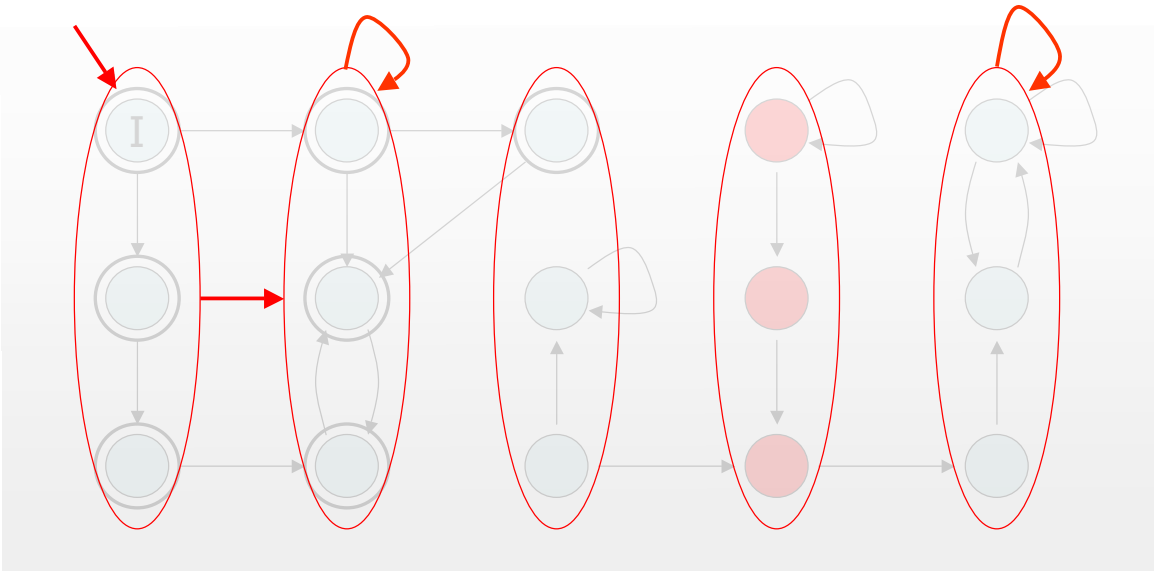


Sep-30-10 SEFM School 2010 30

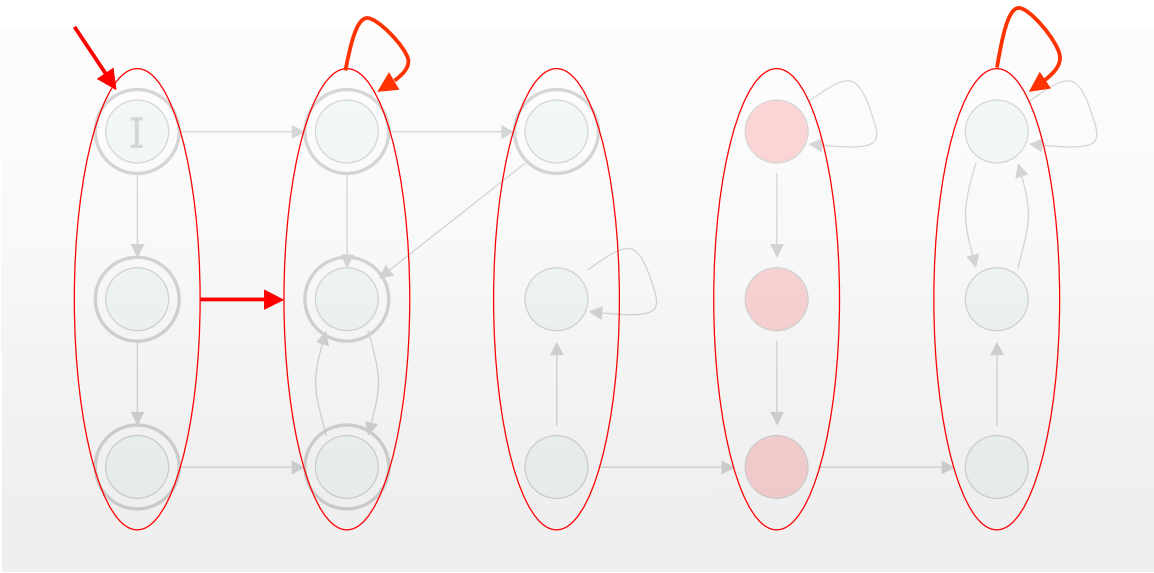
Abstraction: Under-Approximation



Abstraction: Under-Approximation

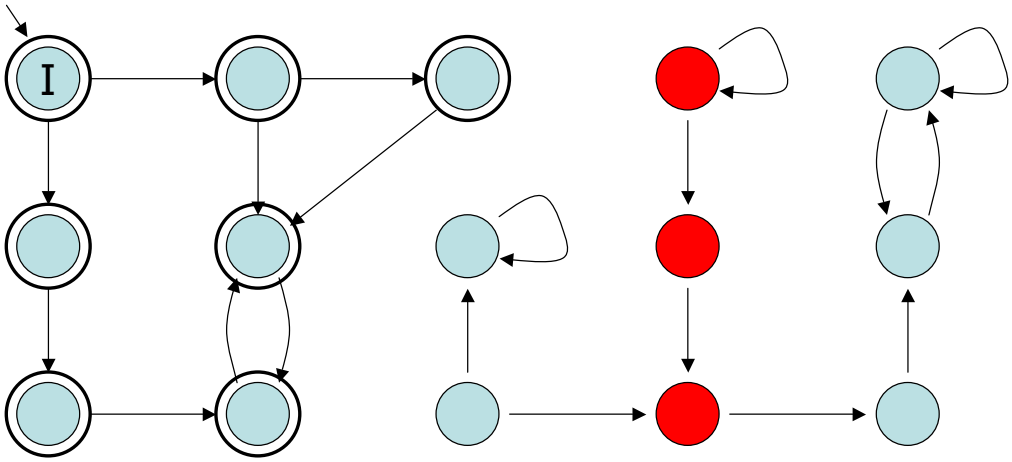


Abstraction: Under-Approximation

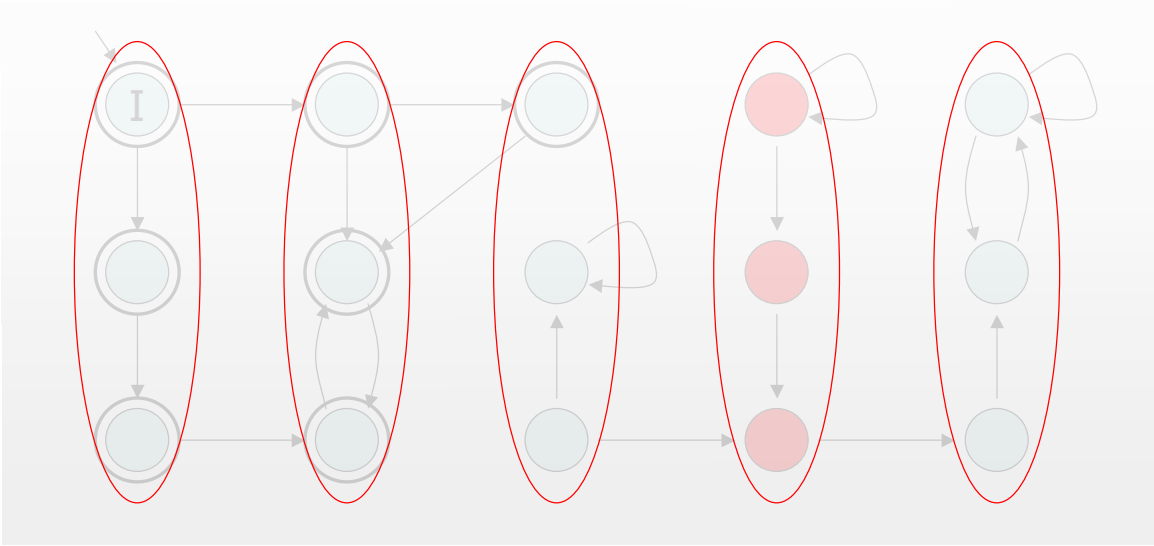


AG ~ unsafe ***true*** *(but it is not preserved)*

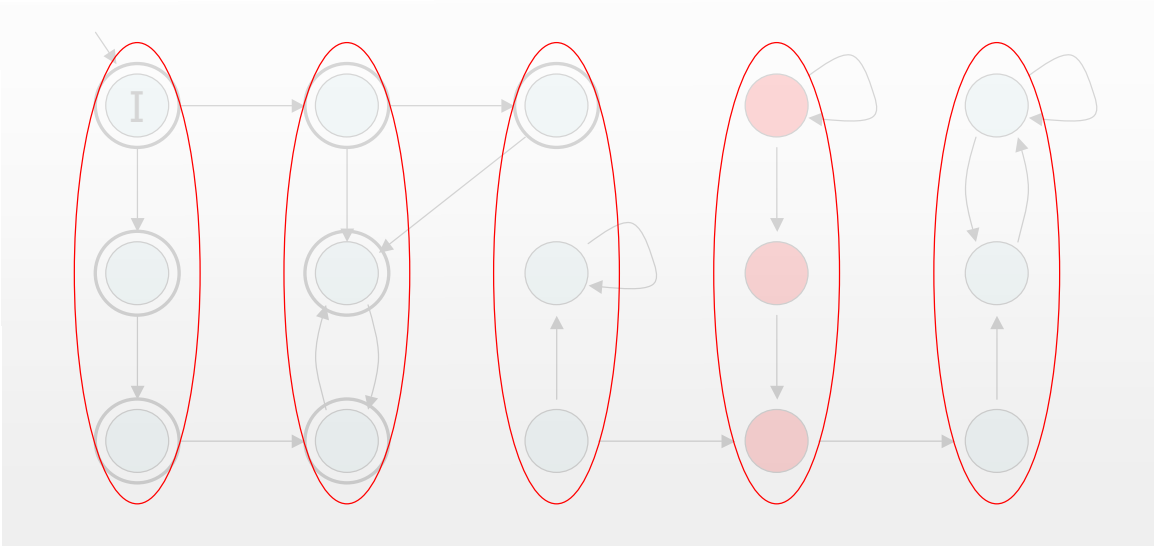
Abstraction:



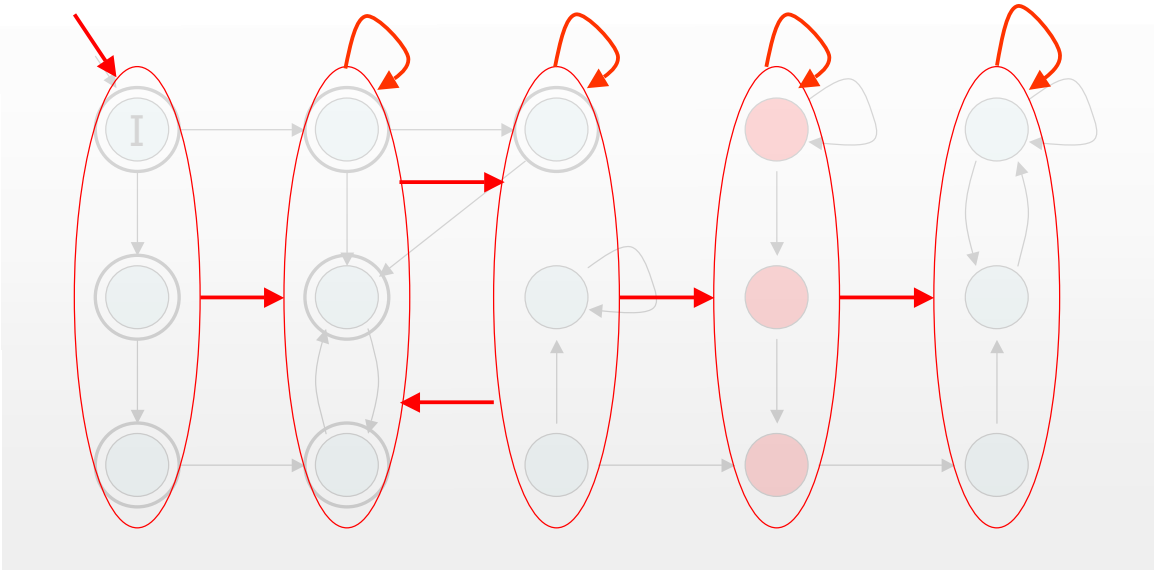
Abstraction:



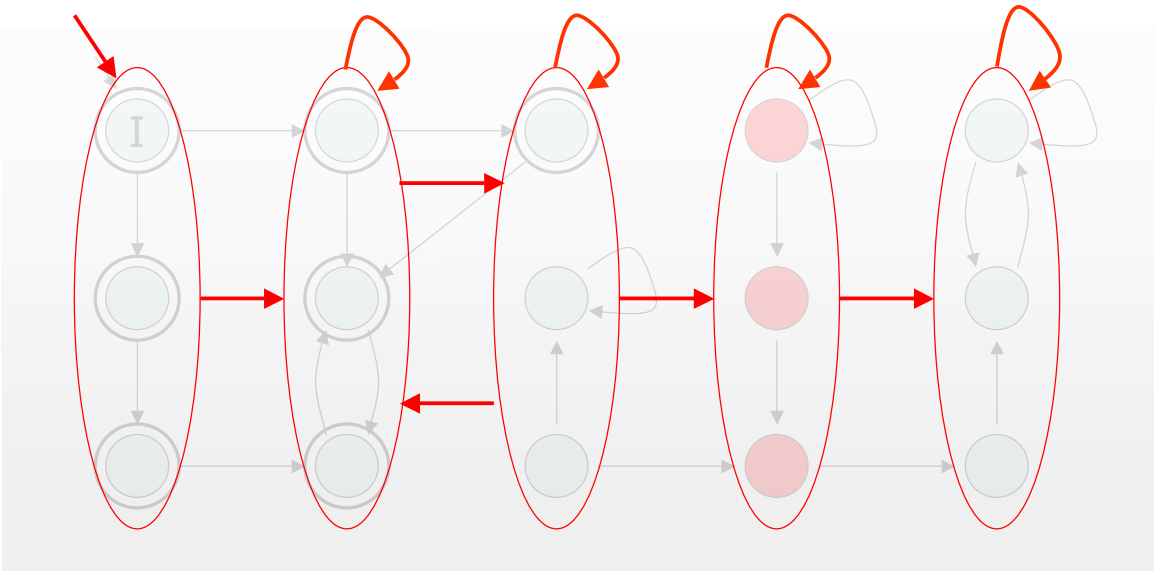
Abstraction: Over-Approximation



Abstraction: Over-Approximation

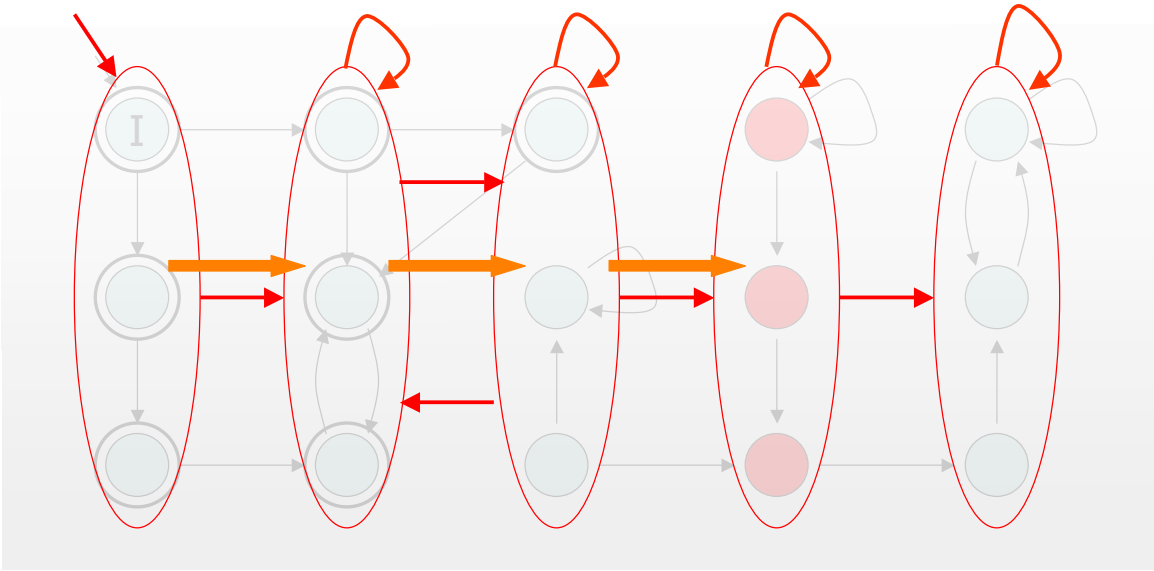


Abstraction: Over-Approximation



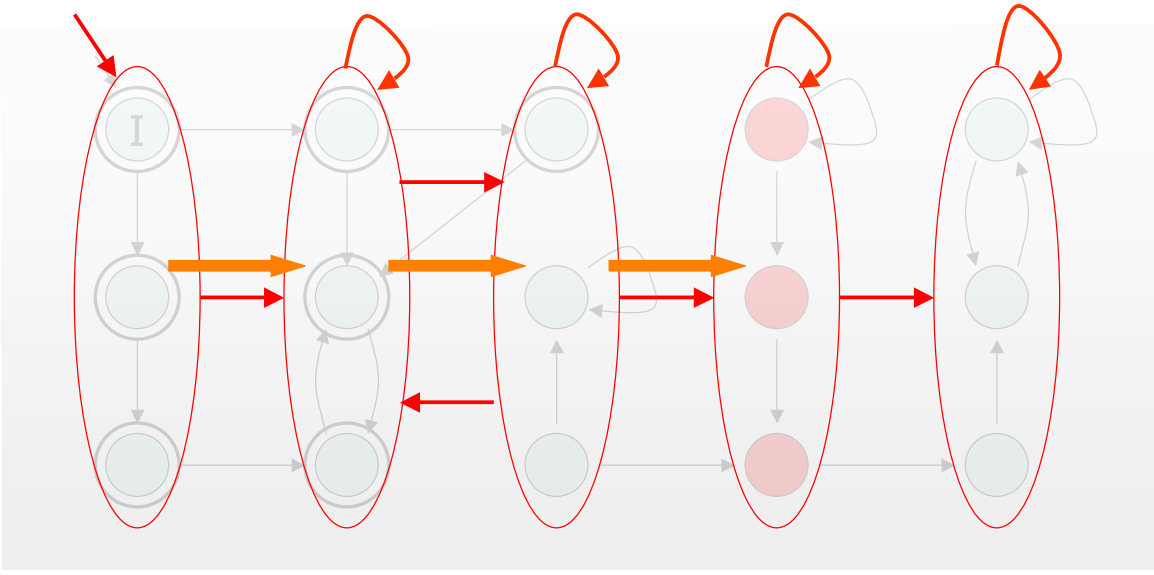
AG ~ unsafe

Abstraction: Over-Approximation



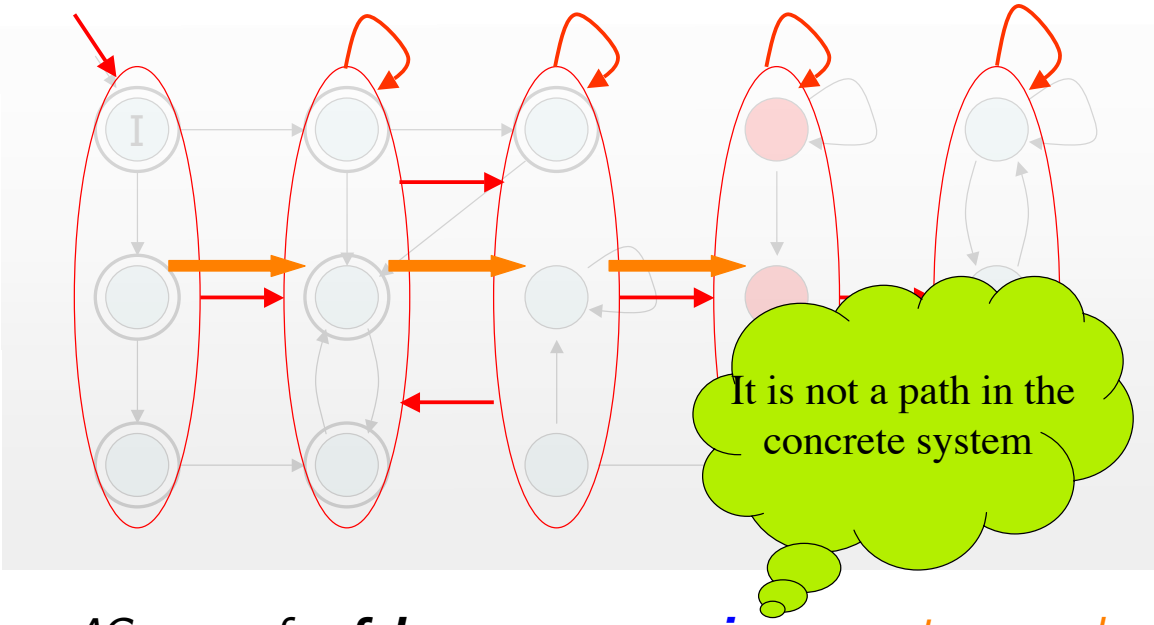
AG ~ unsafe ***false*** *counterexample*

Abstraction: Over-Approximation



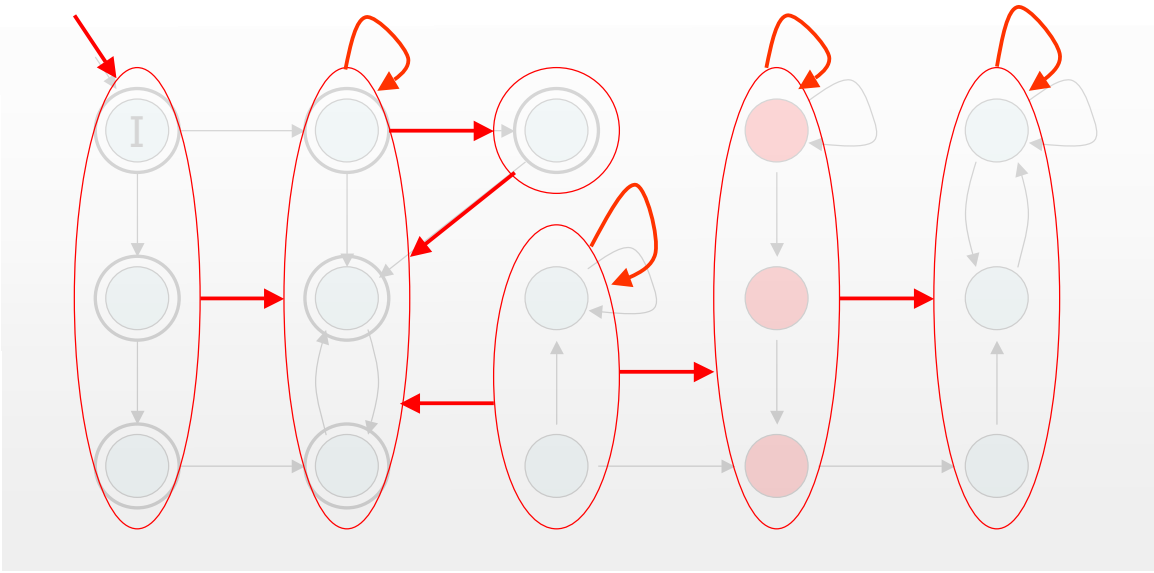
AG ~ unsafe ***false*** ***spurious*** *counterexample*

Abstraction: Over-Approximation



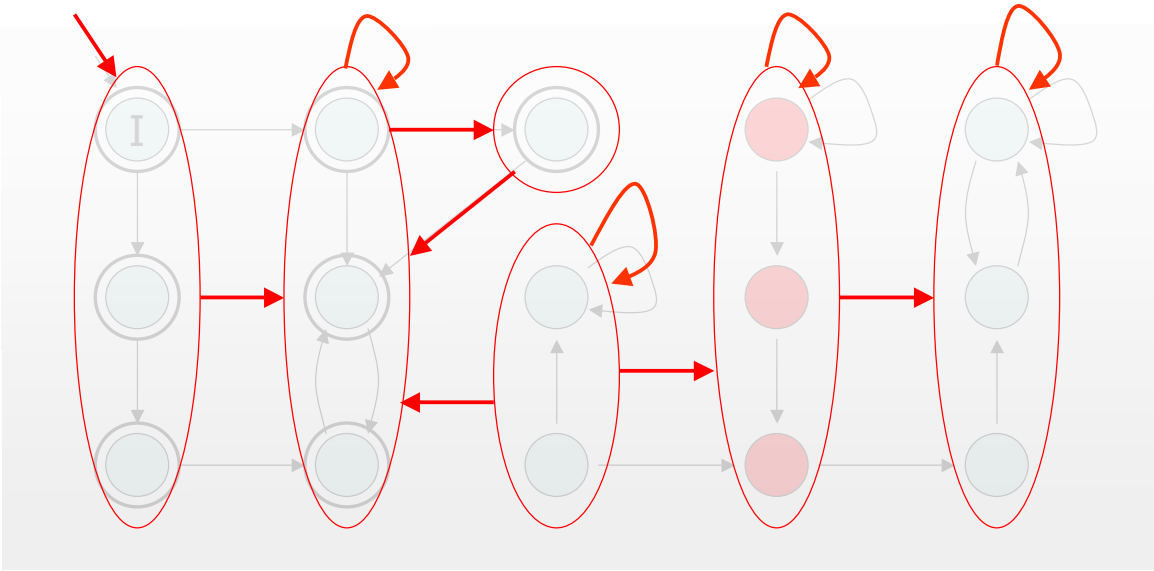
AG ~ unsafe ***false*** ***spurious*** *counterexample*

Refinement of the abstraction :



Separate states that are the reason of the spurious counterexample

Refinement of the abstraction :

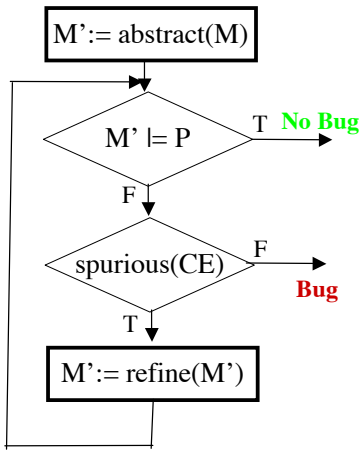


Separate states that are the reason of the spurious counterexample

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)

Java Code

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

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

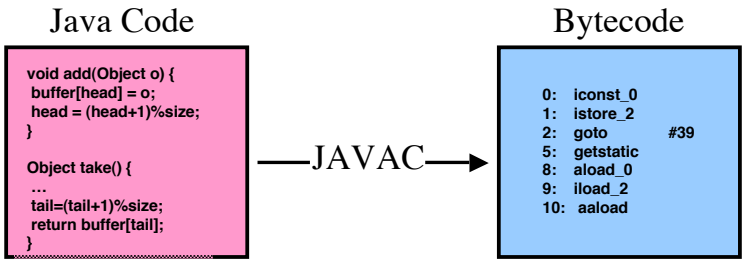

Java PathFinder (JPF)

Java Code

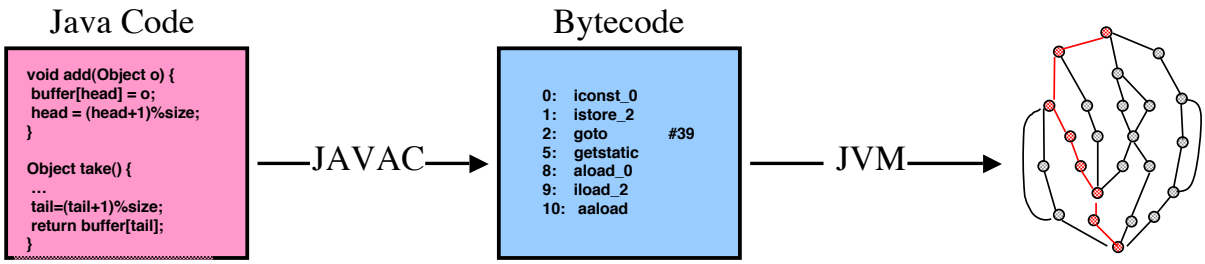
```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

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

—JAVAC→

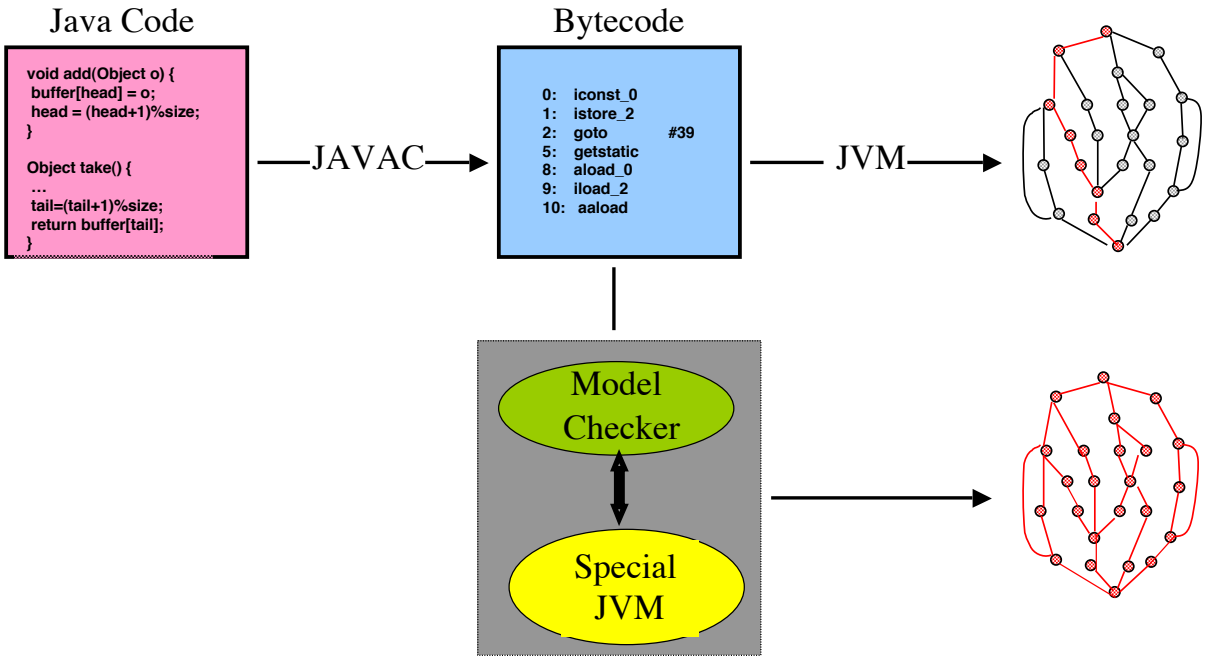
Java PathFinder (JPF)



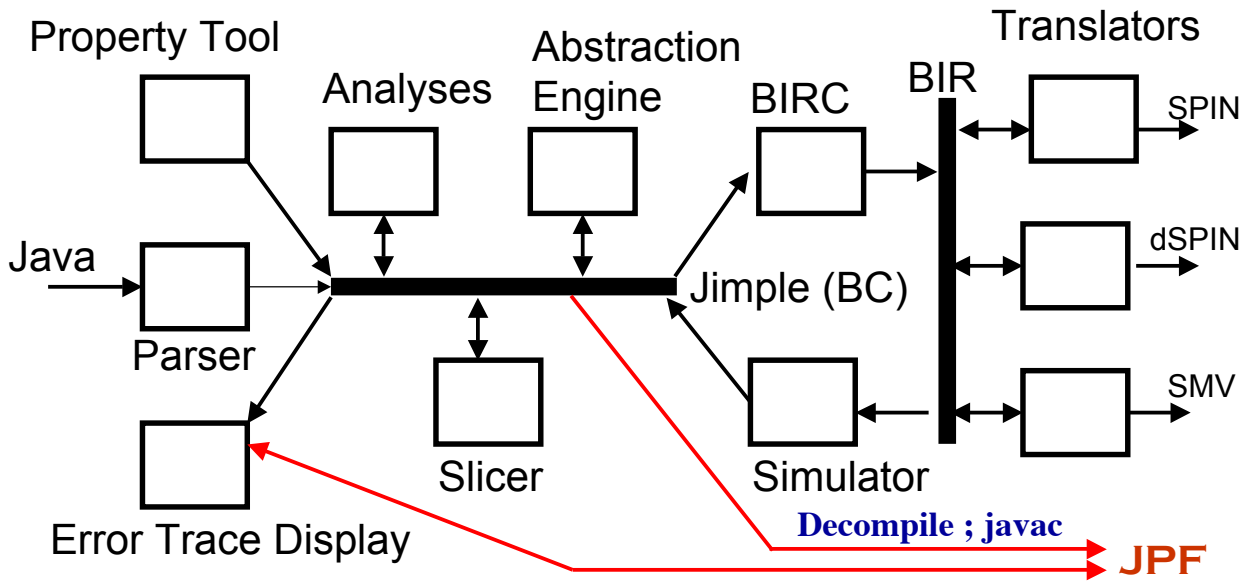
Java PathFinder (JPF)

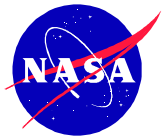


Java PathFinder (JPF)

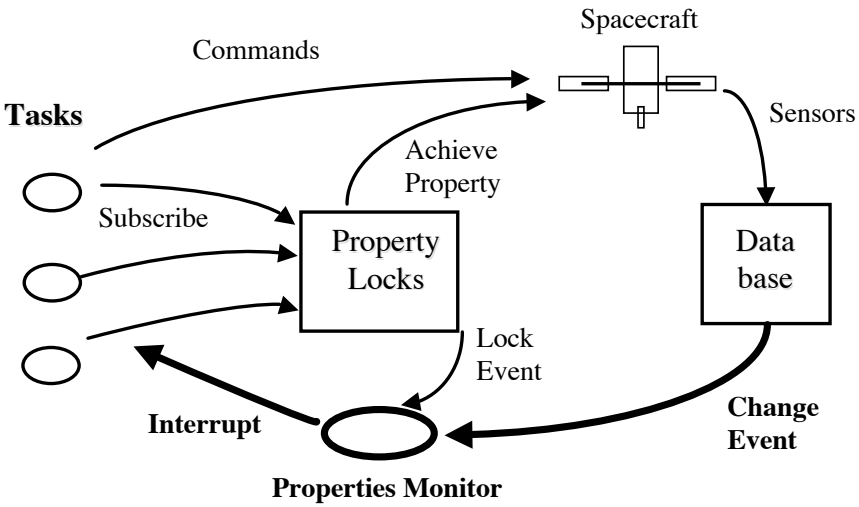


Bandera & JPF Architecture

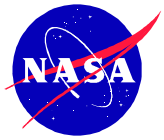




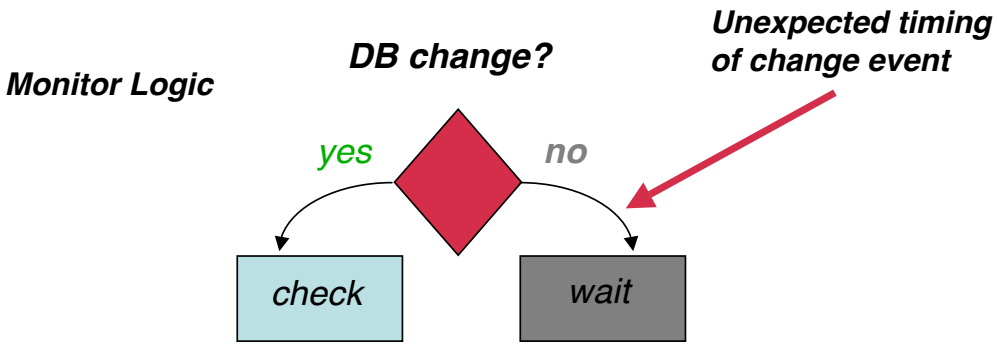
One Case Study at NASA: DS-1 Remote Agent



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



One Case Study at NASA: DS-1 Remote Agent



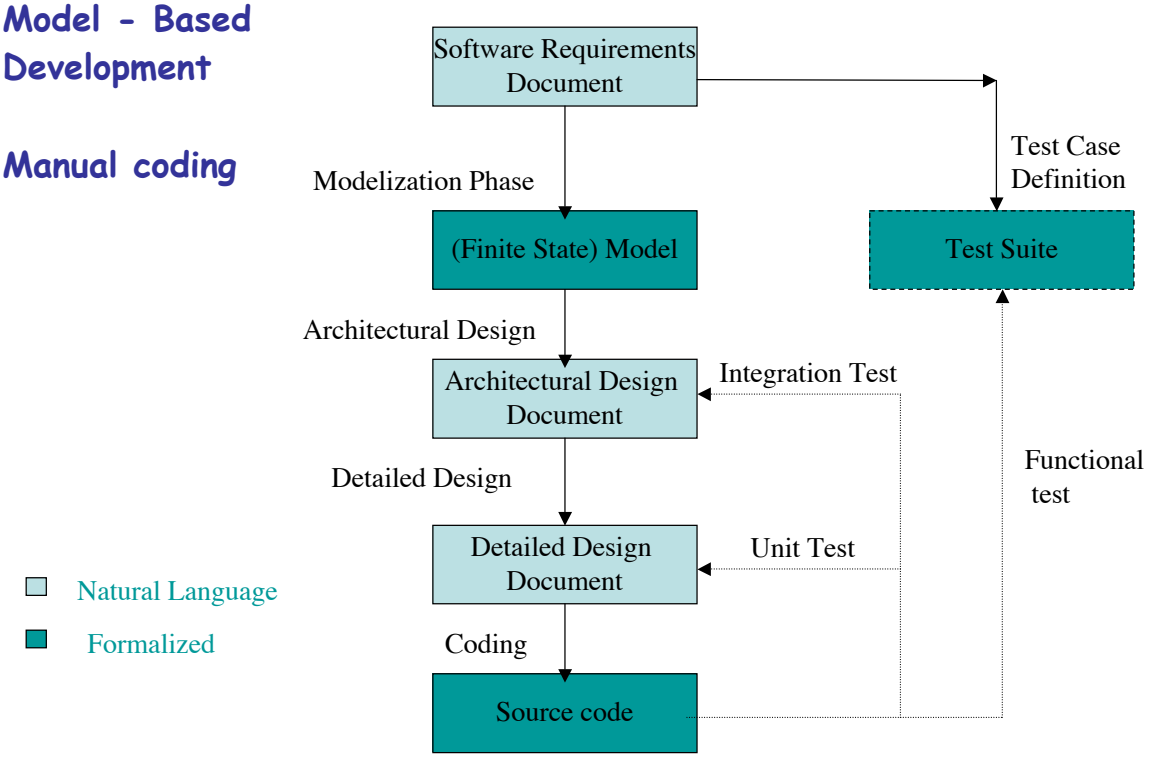
- Five difficult to find concurrency errors detected
- "[Model Checking] has had a substantial impact, helping the RA team improve the quality of the Executive well beyond what would otherwise have been produced." - RA team
- During flight RA deadlocked (in code we didn't analyze)
  - Found this deadlock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

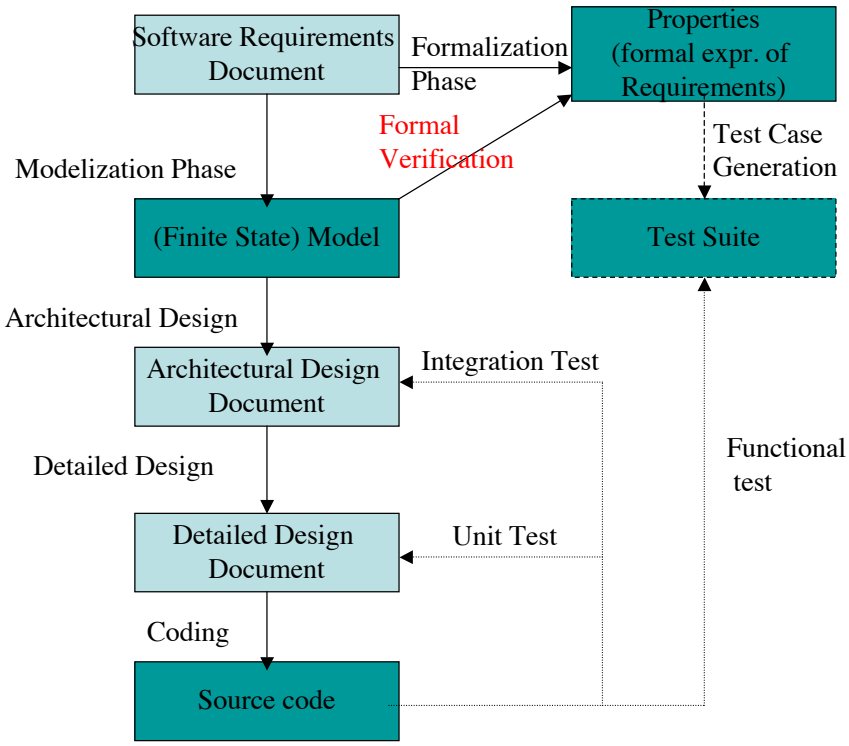
Model - Based Development

Manual coding



Model - Based Development

Manual coding

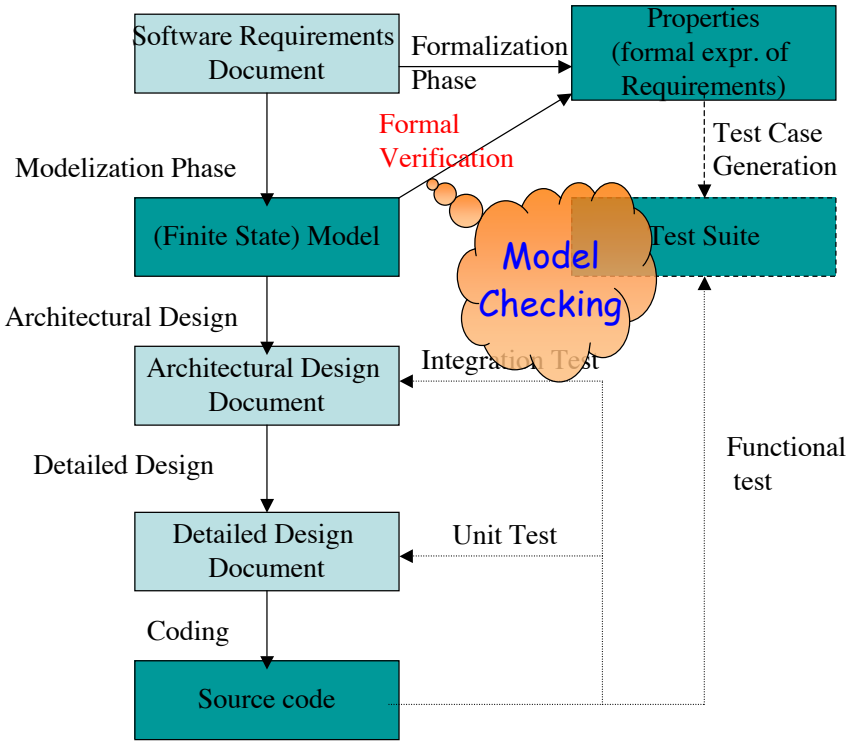


Natural Language

Formalized

Model - Based Development

Manual coding

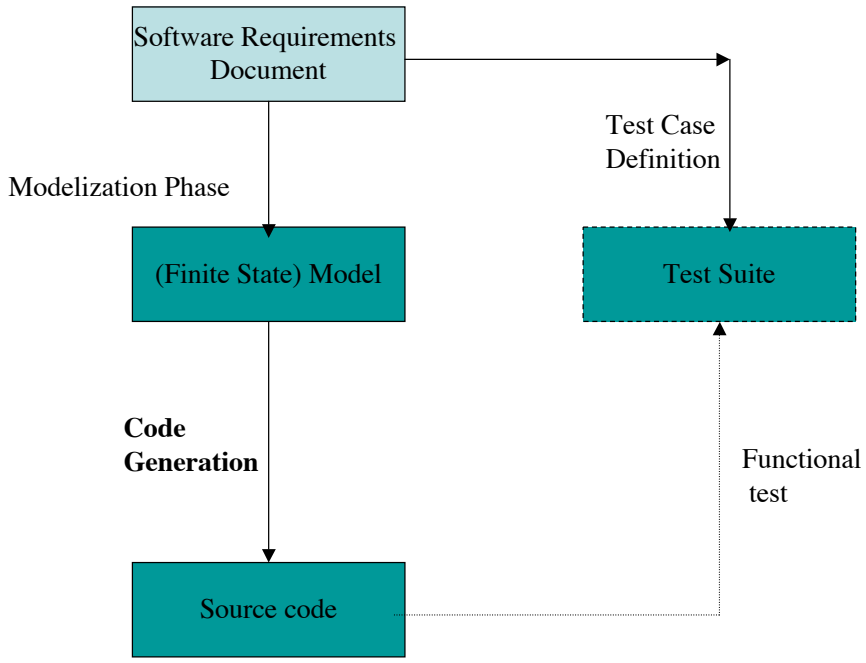


Natural Language

Formalized

Model - Based Development

Automatic coding

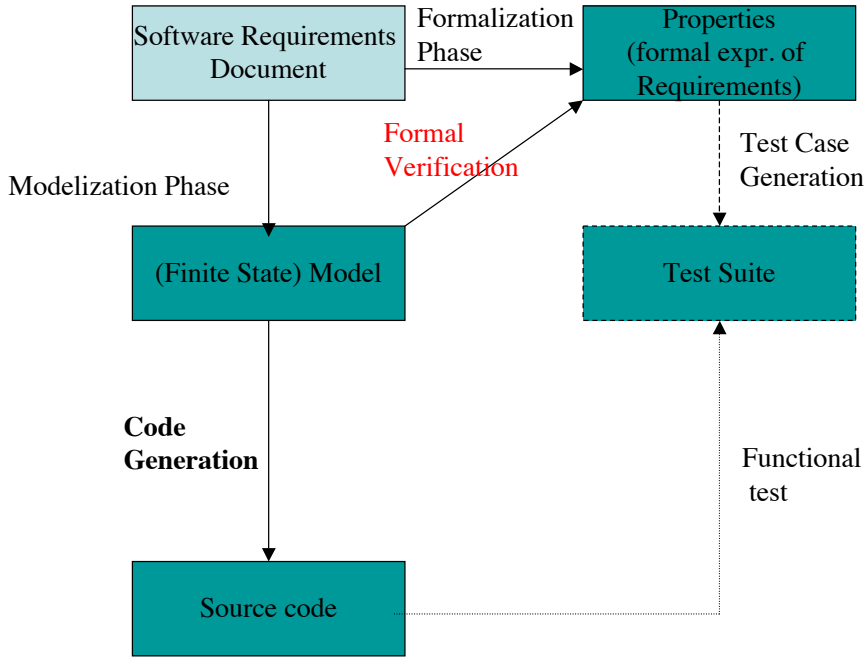


Natural Language

Formalized

Model - Based Development

Automatic coding

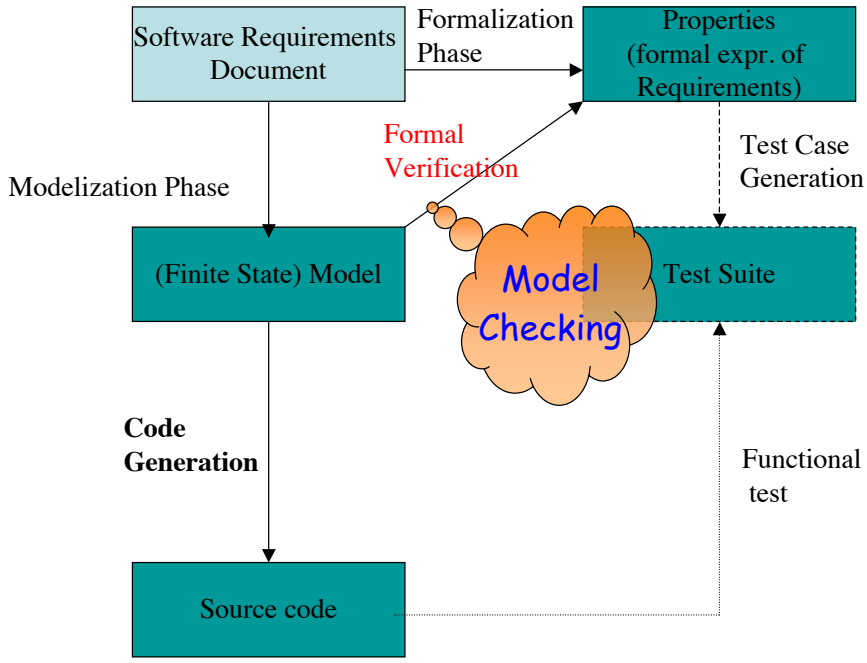


Natural Language

Formalized

Model - Based Development

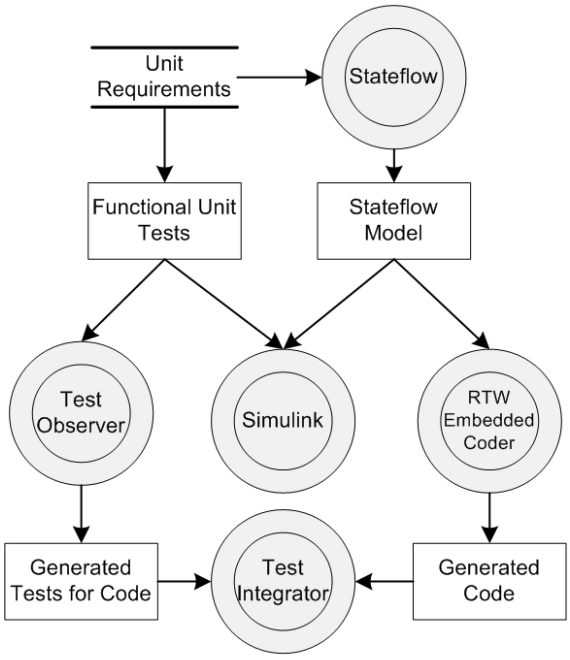
Automatic coding



Natural Language

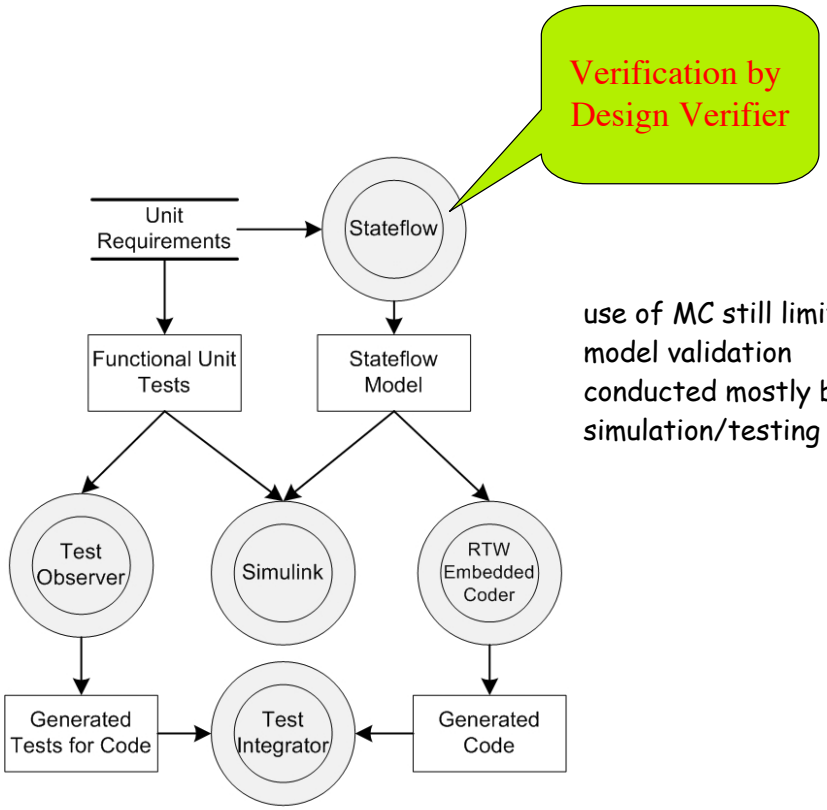
Formalized

GETS model based development cycle



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

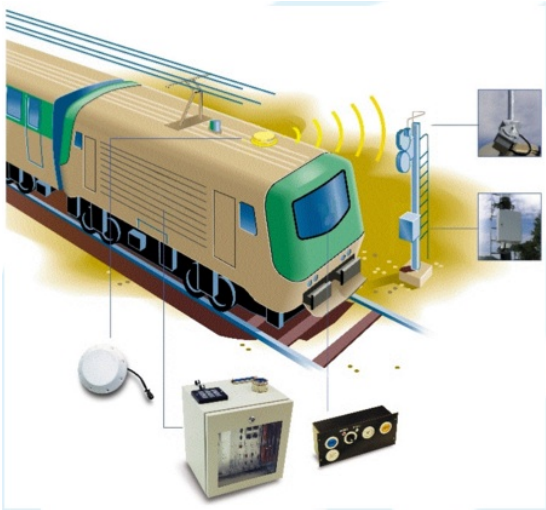
GETS model based development cycle



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

GE Transportation

Automatic Train Protection (ATP) Systems

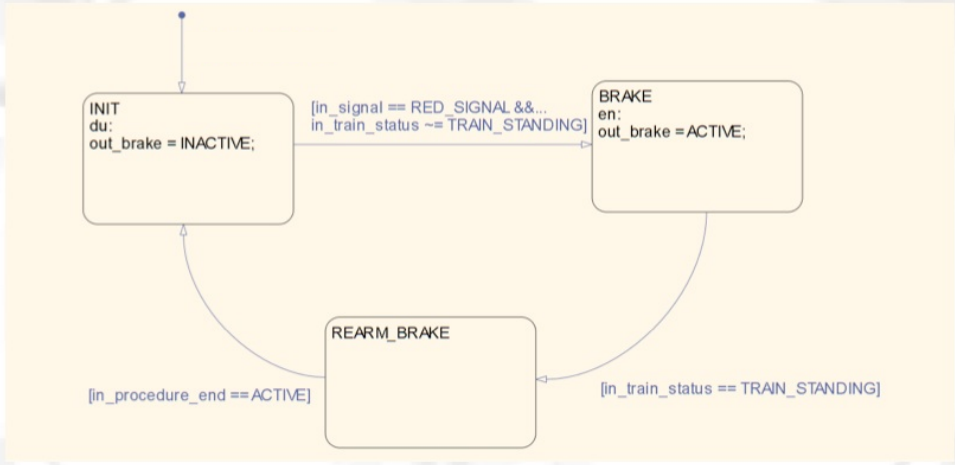


- Stop the train in case of SPAD
- Focus on on-board equipment
- Control modes logic
- Message analysis algorithms

Verification by
Design Verifier

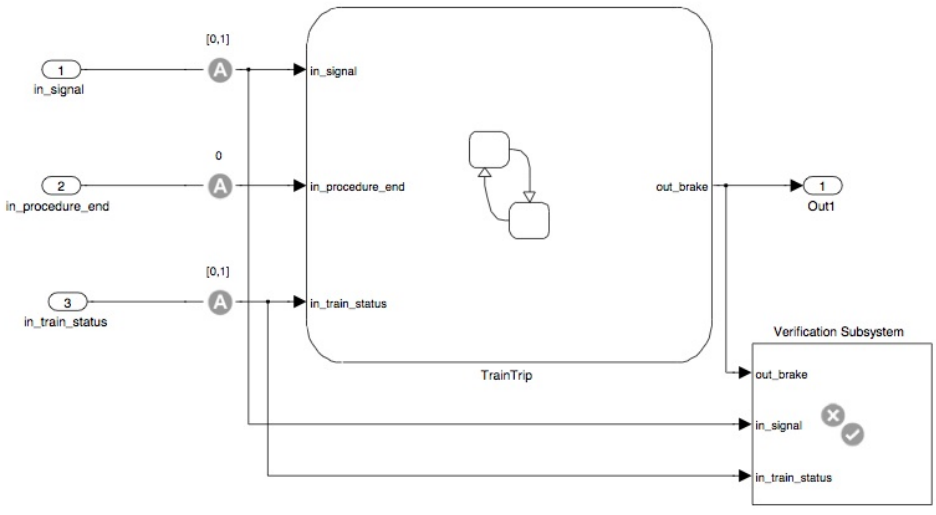
Additional Requirement

During system startup the brake shall be active



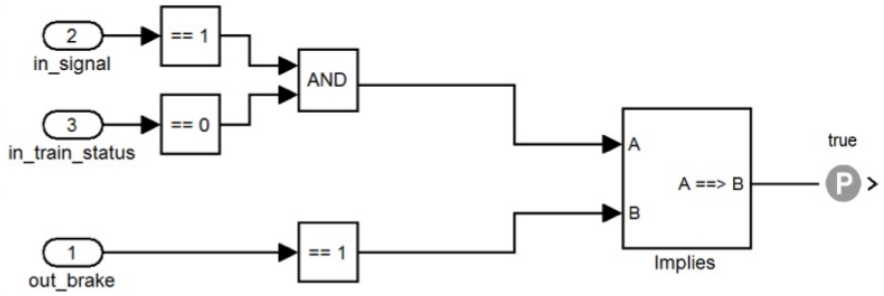
Verification by
Design Verifier

...verification can be performed only on intput/output variables



Verification by
Design Verifier

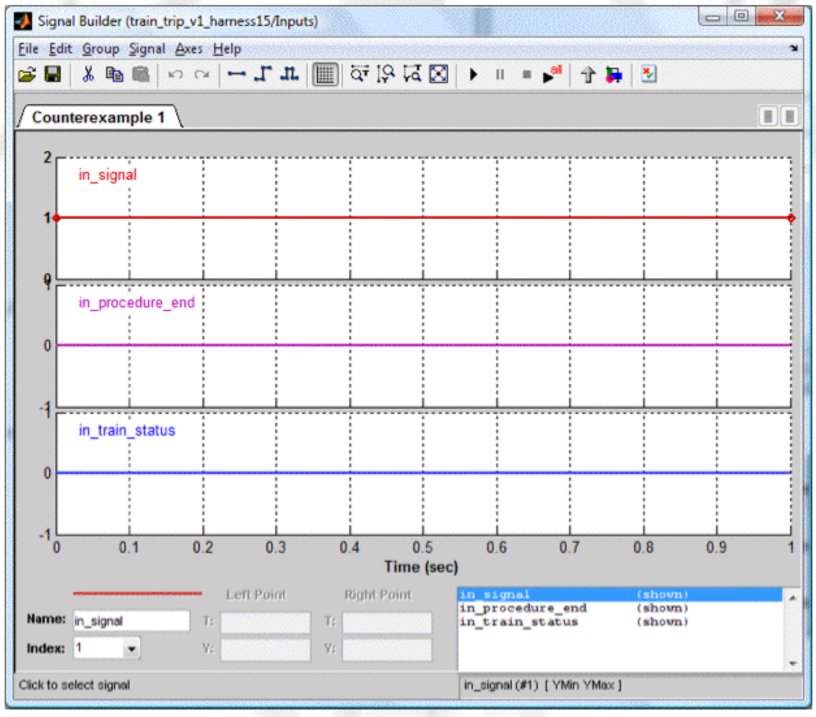
The system shall issue a brake command when a red signal is passed and if the train is not standing



...is it correct?

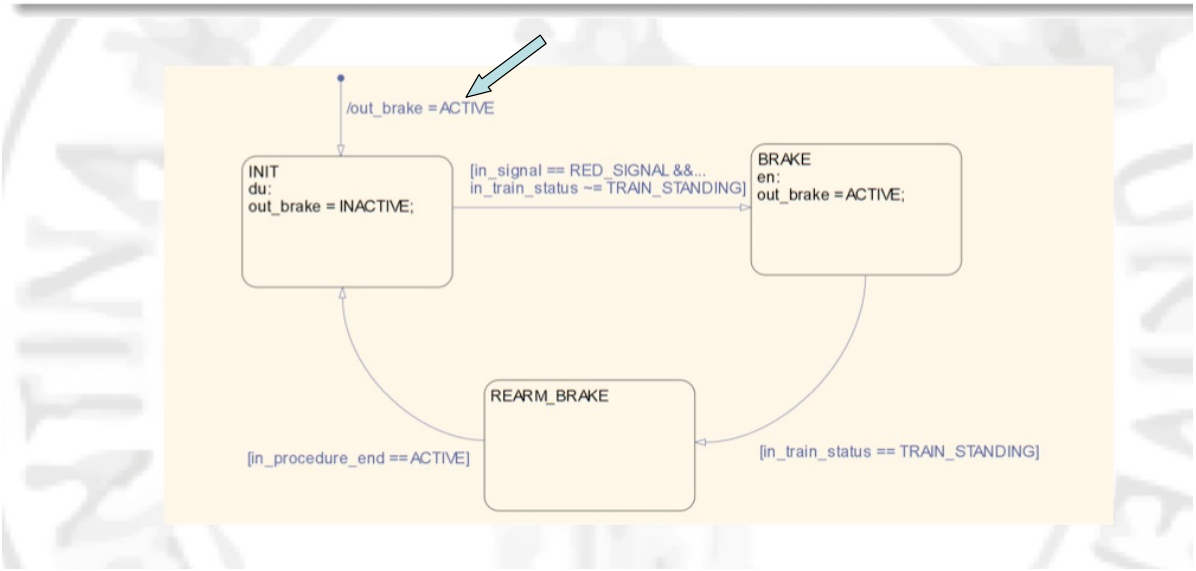
Verification by
Design Verifier

...no? why?



Additional Requirement

During system startup the brake shall be active



An example of family of systems that pose interesting challenges to model checking - railway interlocking

- In the railway signaling domain, an interlocking (IXL) is the safety-critical system that controls the movement of trains in a station and between adjacent stations.
- The IXL monitors the status of the objects in the railway yard (e.g., points, switches, track circuits) and allows or denies the routing of trains in accordance with the railway safety and operational regulations.
- The instantiation of these rules on a station topology is stored in the part of the system named *control table*, that is specific for the station where the system resides.
- Control tables of computerized IXLs are implemented by means of iteratively executed software controls over the status of the yard objects.

Interlocking - representation and implementation of the logic

- For control tables, usually adopted graphical representations such as ladder logic diagrams and relay diagrams (*principle schemata*)
- the graphical representations and the related control tables can be reduced to a set of boolean equations of the form

  $x_i := x_j \wedge \ldots \wedge x_{j+k}$,

  where $x_j \ldots x_{j+k}$ are boolean variables in the form x or ~x. The variables represent the possible states of the signalling elements monitored by the control table (input, output or temporary variables)..
- The model of execution is a state machine where equations are executed one after the other in a cyclic manner and all the variables are set at the beginning of each cycle and do not change their actual value until the next cycle.
- PLC-based semantics, implemented either by interpretation by an off-the shelf PLC, or by a dedicate resolution engine on a dedicated processor

Model Checking the logic of an interlocking

- The most critical part of an IXL is the logic (the engine is reused among several projects…)
- The logic of an interlocking requires a new validation effort for each station: automating this validation would allow significant effort sparing. Several companies are looking at Model Checking (e.g. Siemens)for this purpose
- It is known that IXL logic pose a big challenge to Model Checking for its rapidly incresing dimensions: only small scale IXLs tractable
- General Electric Transportation Sytems wanted to investigate the limits of the technology, by performing model checking runs over IXLs of increasing sizes.
- Joint study with University of Florence

Conducted experiments

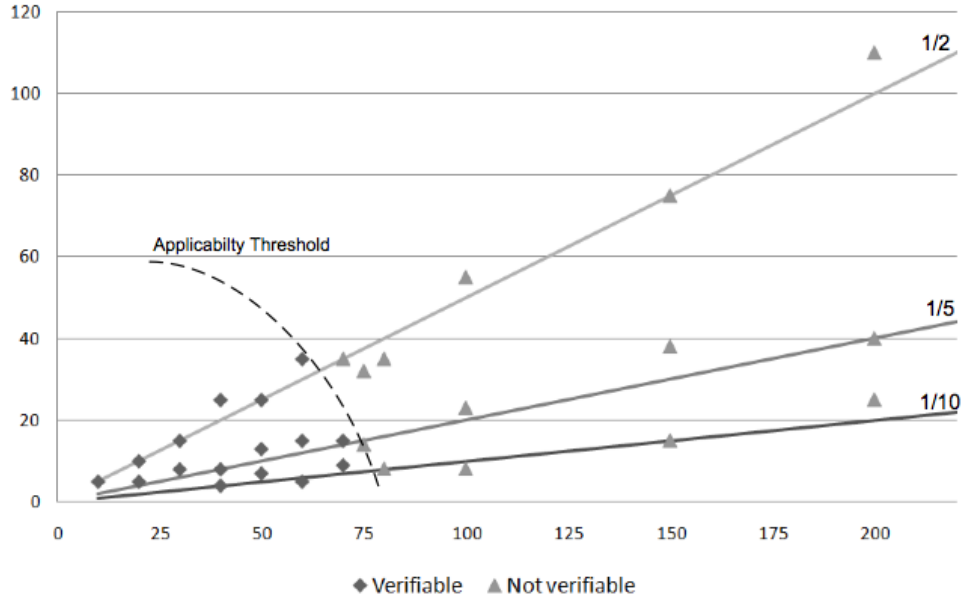
- size of a control table as the couple (m, n), where m is maximum number of inter-dependent equations involved, that means equations that, taken in pairs, have at least one variable in common, and n is the number of inputs of the control table.
- (sets of equations that are independent can be verified separately: slicing can be adopted on the model to reduce the problem size).
- Random generation of set of equations of different size
- Expression of the equations as models for NuSMV and SPIN
- (choice of *mature* model checking tool w.r.t. experimental tools, like PLC model checkers - see later…)

Safety properties

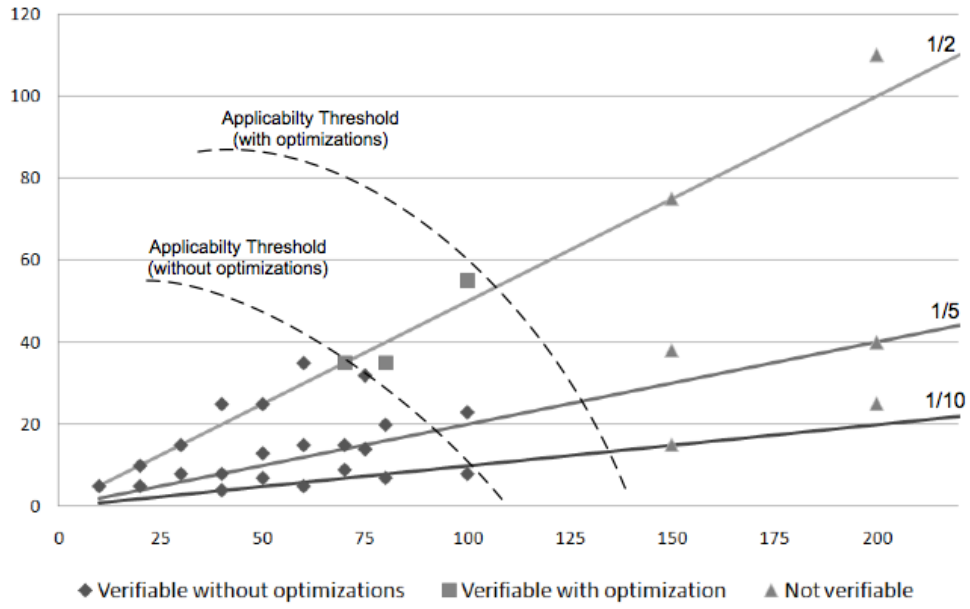
- One of the typical safety properties that is normally required to be verified is the no-derailing property:

  "*while a train is crossing a point the point shall not change its position*".
- This typical system level requirement can be represented in the AGAX form:
- $AG(occupied(tc_i) \wedge setting(pi) = val \Rightarrow AX(setting(p_i) = val))$
  - *whenever the track circuit tc_i associated to a point p_i is occupied, and the point has the proper setting val, this setting shall remain the same on the next state.*
- CTL: AGAX form ---> LTL: GX form
- The experiments have checked over the random generated models properties in the AGAX/GX form, that were true on the model by construction
  - true safety properties tend to be the hardest ones to prove, since require the full state space exploration

NuSMV results



Spin results



Did we win?

- The results have confirmed that the bound on the size of the controlled yard that can be safely addressed by the two tools is still rather small, making general purpose model checking tools not usable for medium and large scale IXLs.
- Medium-size IXLs normally have some hundreds of equations
- Slicing can help since medium size IXL can be decomposed in smaller slices
- Can SAT-based bounded model checking help?
  - Indeed, AGp properties pose a problem to a bounded model checking engine, since this explores only finite length paths.

Bounded Model Checking and SAT

- Suppose we want to check a LTL property of the form Gp on a path of length k.
- We write a formula expressing that at least one state of the path does not satisfy p.

$$Init(x_0) \wedge \bigwedge_{i=0}^{k-1} T(x_i, x_{i+1}) \wedge \bigvee_{i=0}^{k} \sim p(x_i)$$

- where x_i are state bit vectors, $Init$ is a predicate that holds for the initial state, p is the predicate saying that p holds in that state, T is the transition relation.
- Satisfying assignments are counterexamples for the Gp property

Checking safety properties with SAT

- No satisfying assignment means the formula is satisfied on the *k*-length paths: don't know about longer paths.
- Unless you prove that there are no such long paths: your *k*-lenght paths go back to the initial state:

$$Gp \wedge F\ init$$

- Verified on a model with depth k guarantees that the model goes back to the initial state in no more than *k* tranistions
- Boolean encoding:

$$Init(x_0) \wedge \bigwedge_{i=0}^{k-1} T(x_i, x_{i+1}) \wedge (\bigvee_{i=0}^{k} \sim p(x_i) \vee \bigwedge_{i=0}^{k} \sim T(x_i, x_0))$$

- A counterexample means that either the formula is not satisified, or the length *k* is not enough to go back to the initial state.

Embedded systems

- In embedded systems, behaviour is normally cyclic, so if you succeed to prove this formula within a given length, you are done.
- You can give an upper bound to the execution time of a cycle (WCET): very useful in real-time systems.
- Problems: increasing length increases complexity

General form

- In general, prove that every path is of the form:

- that is, that eventually there is a transition to an already traversed state
- need of μ-calculus to express this property as a temporal logic formula:

$$Gp \wedge F\mu x.Fx$$

- need of a quadratic formula to encode this property on state but vectors (but the number of variables is the same)

$$Init(x_0) \wedge \bigwedge_{i=0}^{k-1} T(x_i, x_{i+1}) \wedge (\bigvee_{i=0}^{k} \sim p(x_i) \vee \bigwedge_{i=0}^{k} \bigwedge_{j=0}^{i} \sim T(x_i, x_j))$$

Ad hoc form

- If you know the set S of states where a loop starts:

$$Gp \wedge F \bigvee_{q \in S} q$$

$$Init(x_0) \wedge \bigwedge_{i=0}^{k-1} T(x_i, x_{i+1}) \wedge (\bigvee_{i=0}^{k} \sim p(x_i) \vee \bigwedge_{i=0}^{k} \bigwedge_{q \in S} \sim (T(x_i, x_j) \wedge q(x_j)))$$

- This is easy in a well-structured program, where loops starts at while or for locations.
- Technique used by the CBMC software model checker
- Not easy to do for a general state machine, such as the ones defined by the set of equations of a control table in an IXL

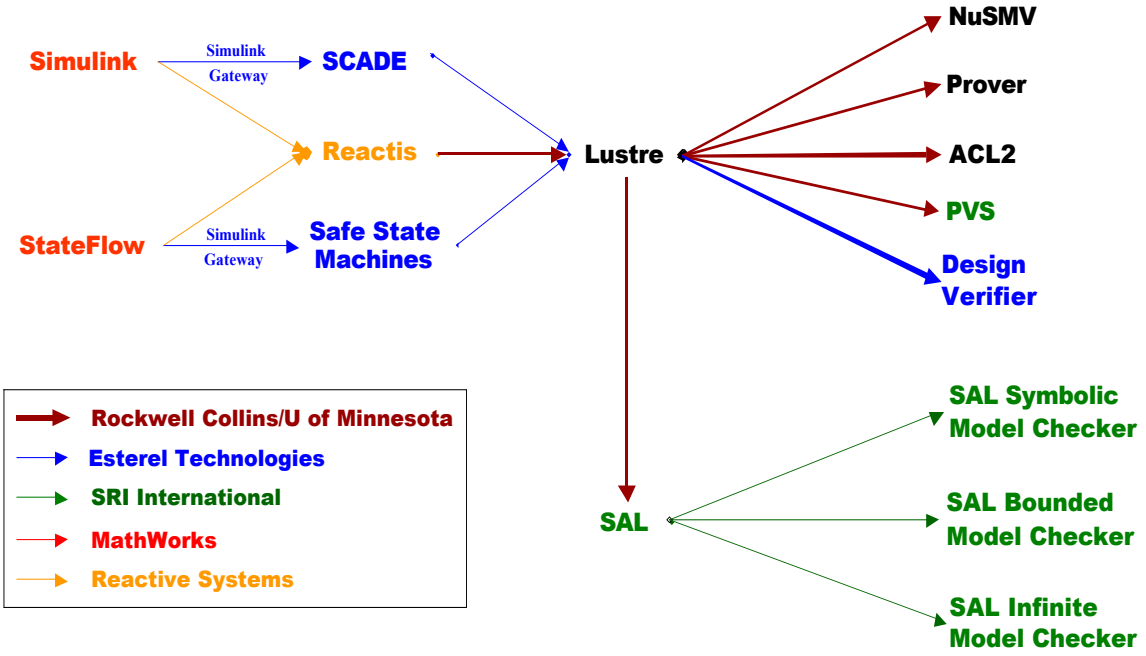
More Examples of ModelBased Design from the avionics sector

- *Rockwell Collins*
- *Airbus*

- These example show that the trend is not dissimilar to the one shown by the cases form the railway sector.
- The different domain pose different challenges, anyway.

Rockwell Collins

Verification - Rockwell Collins Translation Framework



ADGS-2100 Adaptive Display & Guidance System

© Copyright 2008 Rockwell Collins, Inc.

ADGS-2100 Adaptive Display & Guidance System

Modeled in Simulink

Translated to NuSMV

4,295 Subsystems

16,117 Simulink Blocks

Over 10^{37} Reachable States

ADGS-2100 Adaptive Display & Guidance System

Modeled in Simulink

Translated to NuSMV

4,295 Subsystems

16,117 Simulink Blocks

Over 10^{37} Reachable States

Example Requirement:

Drive the Maximum Number of Display Units Given the Available Graphics Processors

© Copyright 2008 Rockwell Collins, Inc.

ADGS-2100 Adaptive Display & Guidance System

Modeled in Simulink

Translated to NuSMV

4,295 Subsystems

16,117 Simulink Blocks

Over 10^{37} Reachable States

Example Requirement:

Drive the Maximum Number of Display Units Given the Available Graphics Processors

Counterexample Found in 5 Seconds

ADGS-2100 Adaptive Display & Guidance System

Modeled in Simulink

Translated to NuSMV

4,295 Subsystems

16,117 Simulink Blocks

Over 10^{37} Reachable States

Example Requirement:

Drive the Maximum Number of Display Units Given the Available Graphics Processors

Counterexample Found in 5 Seconds

Checked 573 Properties - Found and Corrected 98 Errors in Early Design Models

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter		
Failure Processing		
Reset Manager		
Total		

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	
Failure Processing		
Reset Manager		
Total		

© Copyright 2008 Rockwell Collins, Inc.

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing		
Reset Manager		
Total		

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing	**3**	
Reset Manager		
Total		

© Copyright 2008 Rockwell Collins, Inc.

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing	**3**	**0**
Reset Manager		
Total		

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing	**3**	**0**
Reset Manager	**4**	
Total		

© Copyright 2008 Rockwell Collins, Inc.

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing	**3**	**0**
Reset Manager	**4**	**0**
Total		

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing	**3**	**0**
Reset Manager	**4**	**0**
Total	**12**	**0**

- Model-Checking Found 12 Errors that Testing Missed

© Copyright 2008 Rockwell Collins, Inc.

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing	**3**	**0**
Reset Manager	**4**	**0**
Total	**12**	**0**

- Model-Checking Found 12 Errors that Testing Missed
- Spent More Time on Testing than Model-Checking
  - 60% of total on testing vs. 40% on model-checking

CerTA FCS Phase I Errors Found in Redundancy Manager

	Model Checking	Testing
Triplex Voter	**5**	**0**
Failure Processing	**3**	**0**
Reset Manager	**4**	**0**
Total	**12**	**0**

- Model-Checking Found 12 Errors that Testing Missed
- Spent More Time on Testing than Model-Checking
  - 60% of total on testing vs. 40% on model-checking

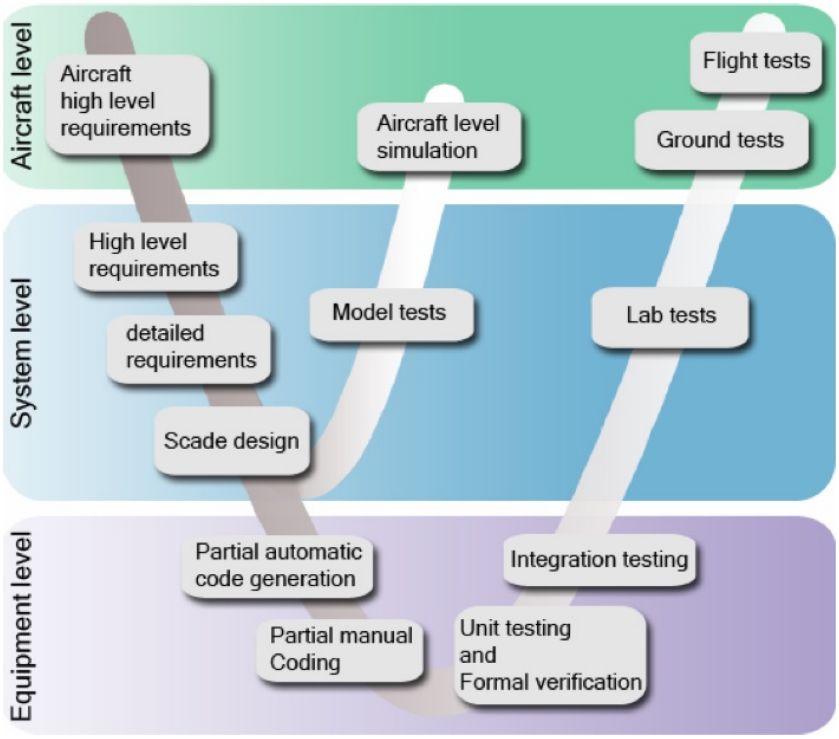
Model-checking was more cost effective than testing at finding design errors.

CerTA FCS Phase II – Verification of Floating Point Numbers

CerTA FCS Phase II – Verification of Floating Point Numbers

- Floating Point Numbers
  - No decision procedures for floating point numbers available
- Solution - Translate Floating Point Numbers into Fixed Point
  - Extended translation framework to automate this translation
  - Convert floating point to fixed point (scaling provided by user)
  - Convert fixed point into integers (use bit shifting to preserve magnitude)
  - Shift from NuSMV (BDD-based) to Prover (SMT-solver) model checker
- Advantages & Issues
  - Use bit-level integer decision procedures for model checking
  - Results unsound due to loss of precision
  - Highly likely to find errors – very valuable tool for debugging

The Airbus software V-shaped development cycle



- Development cycle highly depending on testing
  (according to DO178B)
- However, several studies have been conducted on application of model checking techniques to validate SCADE models.
- These experiments have themselves prompted the development of SCADE's version of Design Verifier

A case study from Airbus

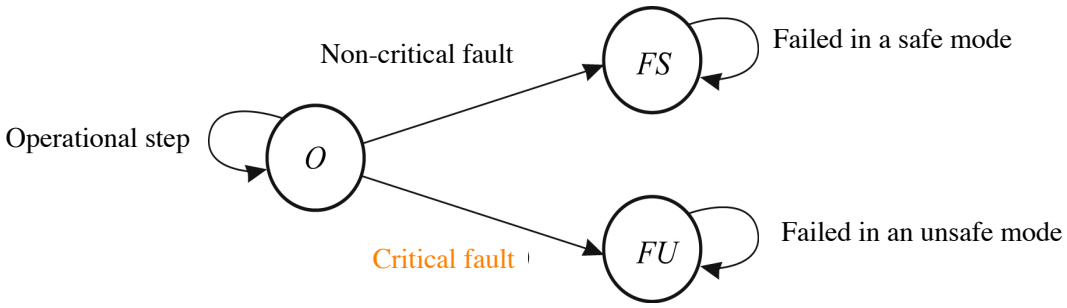
- A380 Ground Spoiler function
- little numerical computation, but sufficiently complex to challenge the verification tool because of the presence of temporal counters.
- 48 hours to exhaustively analyze the correct version, (run on a 1.7-GHz Pentium 4 processor with 256 MB of RAM)
- Production of counterexamples lasted from minutes to hours, depending on the length of the counterexample and the chosen exploration strategy (SCADE offers two strategies).
- Returned counter-examples between 50 and 160 cycles length.

Notice that:

- As in the case of Mathworks' Stateflow Design Verifier, also Esterel Technologies' SCADE Design Verifier is built on top of a proprietary very efficient SAT solver by PROVER Technologies (now part of Mathworks)
- Similar approach - observer based - to property expression
- (This approach aims to easy the work of the property specifier, avoiding awkward logic notations)
- Far from a single push- button experiment. It is rather an iterative process
- Insufficent support from the tools of this process
- Interpratation of counterexamples requires most effort
- Inability of the tools to supply several counterexamples

System Safety

- Safety admits that a system fails with a non-critical failure
- Adoption of "safety nets" mechanism that avoid critical failure
- (often: hardware fault, software safety net)
- Modelling of the possible faulty behaviour of a system as:



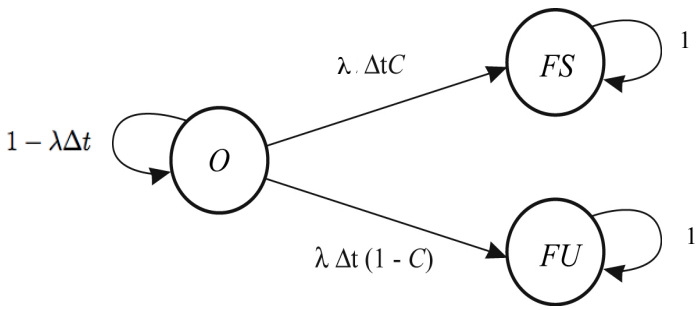
- Safety = prove AG ~ @FU
- (*AGAX form;* AG fault condition => AX ~@FU)
- Counterexample: path leading to the fail unsafe state

Quantitative evaluation of Safety

- Probability that a system, working at time t_0, is still safe (that is still working, or ended up in a stable fail-safe state) at time t.
- To increase safety in case of a faut, it is normally needed to adopt a fault detection mechainsm which can launch a procedure to bring the system in a fail-safe state.
- There is however a non null probability that such detection mechanism is not able to detect some faults, or that the procedure cannot properly complete in a faluty condition.
- This is usually taken care of by recurring to a coverage measure, which expresses the actual capability of the adopted fault detection mechanisms and associated procedures to detect the fault and act accordingly.

Markov chain modeling Safety

- (add probabilities of fault to transitions of a state machine)
- λ = failure rate
- C = coverage



- Probabilistic model checking can be used to evaluate quantitative safety properties

PRISM

- Probabilistic model checker
- probabilistic models supported: discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs)
- property specification language incorporates the temporal logics PCTL, CSL, LTL and PCTL*
- PRISM incorporates state-of-the art symbolic data structures and algorithms, based on BDDs (Binary Decision Diagrams) and MTBDDs (Multi-Terminal Binary Decision Diagrams)
- P<0.002 [F<=10000 "Fail_unsafe"]
  - The probability to have a critical failure before time = 1000 is less than 0.002
- Pmax=? [F<=10000 "Fail_unsafe"]
  - Evaluates the maximum probability that the system is in a fail_unsafe state before time = 10000

Model checking and the Safety Guidelines

- Safety guidelines have been issued in several safety critical systems domains.
- Time of issue dating at the nineties (when model checking was hardly leaving the research labs to the software industry)
- Only mature technologies considered in guidelines for safety-critical systems
- No surprise that model checking is never mentioned
- But formal methods are mentioned and even recommended

Main Software Safety Guidelines

- *Embedded systems:* IEC 61508 - Functional Safety of Electrical / Electronic / Programmable Electronic Safety-related Systems
- *Railway signalling:* CENELEC EN 50128 - Railway Applications - Software For Railway Control And Protection Systems
- *Avionics:* RTCA/DO-178 - Software Considerations in Airborne Systems and Equipment Certification
- *Military:* MoD/DEF-STAN 00-55 - Requirements For Safety Related Software in Defence Equipment

SIL

- ***Safety Integrity Level***
- Number ranging (e.g.) from 0 to 4: 4 indicates the higher criticality, 0 gives no safety concern
- (DO178B: software development assurance level, ranging from E (no safety effect) to A (catastrophic effect)
- SIL is a property of the system, related to the damage a failure of the system can produce
- Apportioned to subsystems and functions at system level in the preliminary safety assessment, it is assigned to software functions (Software Safety Integrity Level)

A definition of SIL, according to EN 50129

Safety Integrity Level		Alternative Descriptive Words			
4	Very High	Vital	Critical	Safety-critical	Fail-safe
3	High	Vital	Critical	Safety-critical	High integrity
2	Medium	Semi-vital	Essential	Safety-involved	Medium integrity
1	Low	Semi-vital	Essential	Safety-involved	Low integrity
0	Not specified	Non-vital	Non-essential	Non-safety-related	Non-safety

Correspondance between safety quantitative requirements and SIL, according to DEF-STAN 00-55

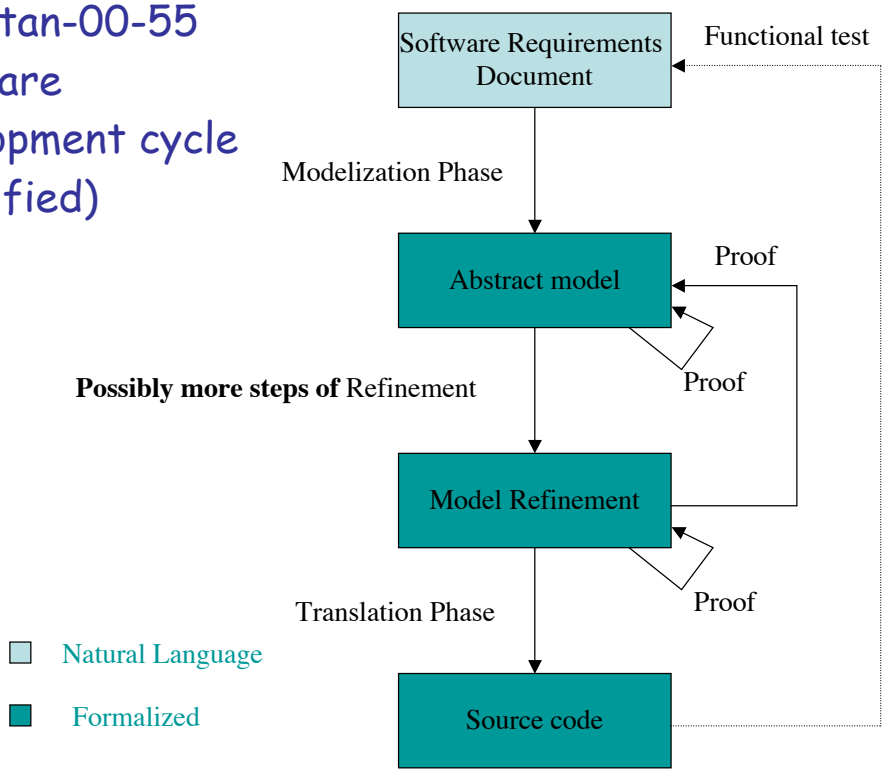
Table 2 Failure Probability for Different Safety Integrity Levels

Safety integrity level	**Safety integrity** (probability of dangerous failure per year, per system)
S4	$\geq 10^{-5}$ to $< 10^{-4}$
S3	$\geq 10^{-4}$ to $< 10^{-3}$
S2	$\geq 10^{-3}$ to $< 10^{-2}$
S1	$\geq 10^{-2}$ to $< 10^{-1}$

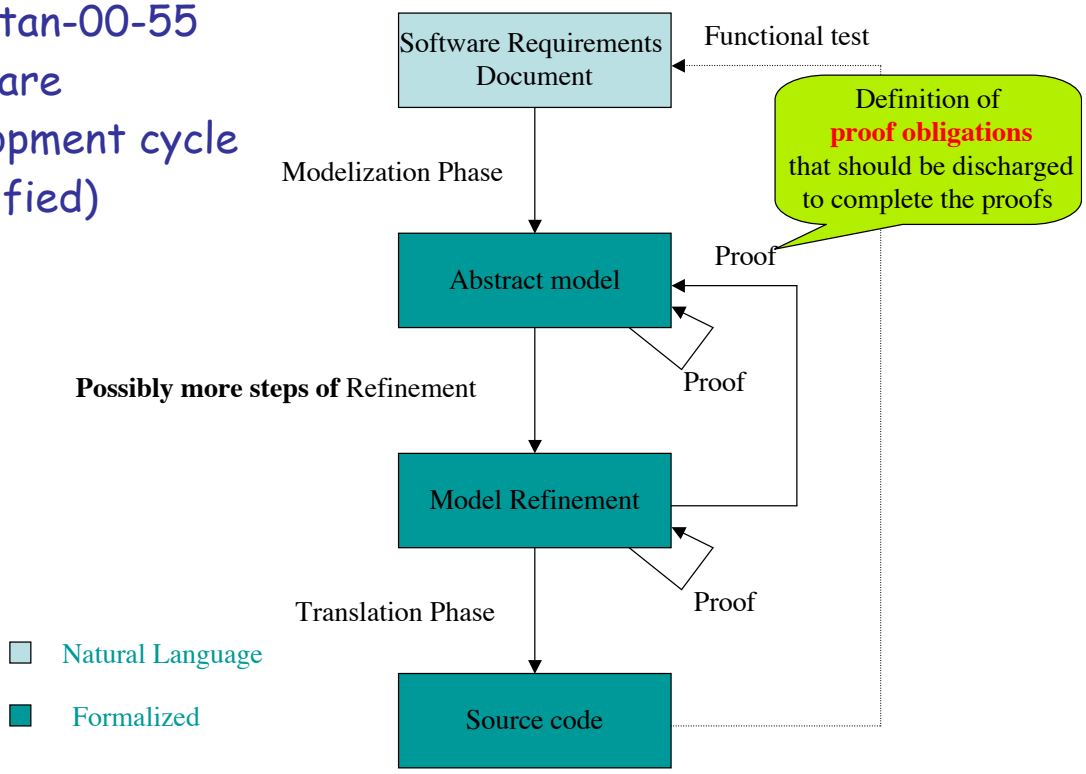
Concentrate on more safety critical software

- SIL apportionment allows software developmers to concentrate the effort on those functions with higher SIL
- Put more effort in verification on higher SIL components
- EN50128/IEC61508 enumerates the development/verification techniques that are mandatory/recommended/forbidden at each SIL level
- DO178B requires different levels of structural coverage for unit testing to different SILs
- **Model checking** could be used to address correctness of higher SIL components, hence addressing the complexity in a divide and conquer fashion
- However, SIL apportionment to software components is made difficult since it requires independence of components (the failure of one should not affect the correct functioning of th eother ones), whcih is hard to prove.

Def-Stan-00-55 Software Development cycle (simplified)



Def-Stan-00-55 Software Development cycle (simplified)



From DEF STAN 00-55

- The **proof obligations** for a particular formal method are the properties that the designer is obliged to discharge in order to have assurance that the specification is self consistent, or that a design correctly implements a specification (refinement).
- Refinement proofs are required to verify the first stage of the design against the specification and to verify each subsequent design stage against the previous one.
- Manual generation of proof obligations is an extremely arduous and error prone task and a more assured method is to use a proof obligation generator.
- Proof obligations are discharged using formal arguments. Formal arguments can be constructed in two ways: by formal proof or by rigorous argument.
- A formal proof is strictly a well formed sequence of logical formulae such that each formula can be deduced from formulae appearing earlier in the sequence or is one of the fundamental building blocks (axioms) of the proof theory.
- **Tools should be used to assist in the generation of formal proofs and checking of formal proofs.**

From DEF STAN 00-55
Formal verification

- **Proof obligations** shall be:

  a) constructed to verify that the code is a correct refinement of the Software Design and does nothing that is not specified;

  b) discharged by means of formal argument.

There is space for Model checking!!

EN50128

Tab. 15.2 Tabella A2 - Software Requirements Specification

TECHNIQUE / MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Formal Methods	B.30	-	R	R	HR	HR
2. Semi-Formal Methods	D.7	R	R	R	HR	HR
3. Structured Methodology	B.60	R	HR	HR	HR	HR

Tab. 15.3 Tabella A5 – Verification and Testing

TECHNIQUE / MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Formal Proof	B.31	-	R	R	HR	HR
2. Probabilistic Testing	B.47	-	R	R	HR	HR
3. Static Analysis	D.8	-	HR	HR	HR	HR
4. Dynamic Analysis and Testing	D.2	-	HR	HR	HR	HR
5. Metrics	B.42	-	R	R	R	R
6. Traceability Matrix	B.69	-	R	R	HR	HR
7. Software Error Effects Analysis	B.26	-	R	R	HR	HR

Revision of EN50128
where Model Checking first appears!
(prEN 50128:2009 E - Draft for Enquiry)

- Model Checking appears as one of the paragraphs within Formal Methods
- Formal Methods appear as one of the techniques recommended for the activities of: Software Requirements Specification, Software Development, Modelling *(inherited by 50128)*
- Still, Model Checking is not included among the Formal Proof techniques

DO178B review: DO178C

- DO-178C nears finish line (Sep 1, 2010)
- Avionics for new aircraft such as the Boeing 787 Dreamliner will be certified under DO-178C.
- After five years, RTCA and EUROCAE, the U.S. and European avionics standards organizations, are nearing the finish line in updating DO-178B, the bible for developers of safety-critical software.
- A cast of 1,000-plus people have observed or participated in the process and about 100 people show up at every meeting, according to one member of RTCA Special Committee 205 (SC-205).
- The industry expects the final package -- DO-178C -- to be released in the first quarter of 2011 and be mandated six to nine months after ratification.

DO 178C and Formal Methods

- DO-178B only mentioned Formal Methods among the "Addiotional Considerations"
- DO-178C will, for the first time, officially recognize the validity of using Formal Methods within the avionics software development process.
- Subject to DO-178C guidelines, formal methods can be used to augment or replace verification steps which must normally be performed via DO-178B.
- Formal methods will be allowed to verify requirements correctness, consistency, and augment reviews.
- DO-178C source code reviews can utilize formal methods, particularly for auto-generated code (typically developed via Model Based Development).
- Also, DO-178C will allow formal methods to verify or replace test cases used to verify low level requirements and replace some forms of testing via formal method based reviews.
- ***(will Model Checking appear in the final text???)***

Tool qualification

- One of the issues that is raised in regulated safety critical domains is:
- *Is the model checker itself bug-free?*
- *Can I trust the model checker tool when it says that a system is safe?*
- The model checker itself should be programmed following the same guidelines
- Which SIL should be assigned to a model checker?

Tool Qualification, according to DO178B

Software tools are classified as one of two types:

- *Software development tools:* Tools whose output is part of airborne software and thus can introduce errors.
- *Software verification tools:* Tools that cannot introduce errors, but may fail to detect them.

Qualification Criteria for Software Development Tools

- a. the software development processes for the tool should satisfy the same objectives as the software development processes of airborne software.
- b. The software level assigned to the tool should be the same as that for the airborne software it produces, unless the applicant can justify a reduction in software level of the tool to the certification authority.
- c. The applicant should demonstrate that the tool complies with its Tool Operational Requirements
- d. Software development tools should be verified to check the correctness, consistency, and completeness of the Tool Operational Requirements and to verify the tool against those requirements

Qualification Criteria for Software Verification Tools

- Demonstration that the tool complies with its Tool Operational Requirements under normal operational conditions.
- Tool Operational Requirements;
  - "A description of the tool's functions and technical features"
  - "User information, such as installation guides and user manuals"
- Documented configuration management / development of tool
- Independent quality assurance in tool development
- Documented tests satisfying tool requirements
- Tests that can be run on the tool deployed on the project environment
- Demonstration that use of the tool is controlled correctly on the project

Alternatives to qualification

- Only DO-178 qualified testing support tools exist, but no model checker up to now (and to my knowledge) has been qualified
- *Proven in use* concept:
  - a tool that has a long record of usage within similar projects with no known failure.
  - Again, there is not a recorded long story of usage of a model checker: up to now,this can be said only for applications of Model Checking to hardware.
  - Anyway, preference for mature tools
- *Duplication and comparison*: equal results from two mature model checkers

Conclusions (has the promise been fulfilled?)

- Model Checking advantages more and more recognized in several safety-critical systems domains
- However, still not routinely used
- Still problems of complexity, scalability, tool support and so on make MC at best appear as a side validation possibility to achieve more confidence on what is developed
- Use of MC to find bugs more easily vs. use of MC to demonstrate safety (in fornt of ian assessor)
- Need of industrial strength tools (although something is moving: see Design Verifier for SCADE and for Stateflow)
- What about UML? Increasing industrial interest, and MBD tools (Rhapsody): still no commercial Model Checker
- Many areas still to be developed: interest in particular areas is domain-dependent
- No push-botton technology
- On the other hand, trend towards hidden MC engines in development tools.
- Model checking slowly slipping in safety critical systems development guidelines (anyway, mentioning formal methods has apparently favoured more penetration of MC w.r.t. unregualted domains, such as automotive)
- Next decade will probably see a fast growth in Model Checking application to safety critical systems

Some References and Credits

- Willem Visser. ASE 2002 Tutorial on Software Model Checking www.visserhome.com/willem/.../ASE2002TutSoftwareMC-fonts.ppt
- Nishant Sinha. Lectures on Abstraction in Model Checking (ppt), 15817, Mar 2005.
- Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, Virginie Wiels: Model checking flight control systems: The Airbus experience. ICSE Companion 2009:18-27
- Steven P. Miller, Michael W. Whalen, Darren D. Cofer: Software model checking takes off. Commun. ACM 53(2): 58-64 (2010)
- Cary R. Spitzer. Digital Avionics Handbook - Avionics: Elements, Software and Functions. CRC Press LLC, 2001
- A. Ferrari, D. Grasso, G. Magnani, A. Fantechi, The Metro Rio ATP case study, 15th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2010) September 20-21, 2010, Antwerp, Belgium, Lecture Notes in Computer Science 6371.
- A. Ferrari, D. Grasso, G. Magnani, A. Fantechi, Model Checking Interlocking Control Tables, 8th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2010) December 2-3, 2010, Braunschweig, Germany.