Finite State Verification

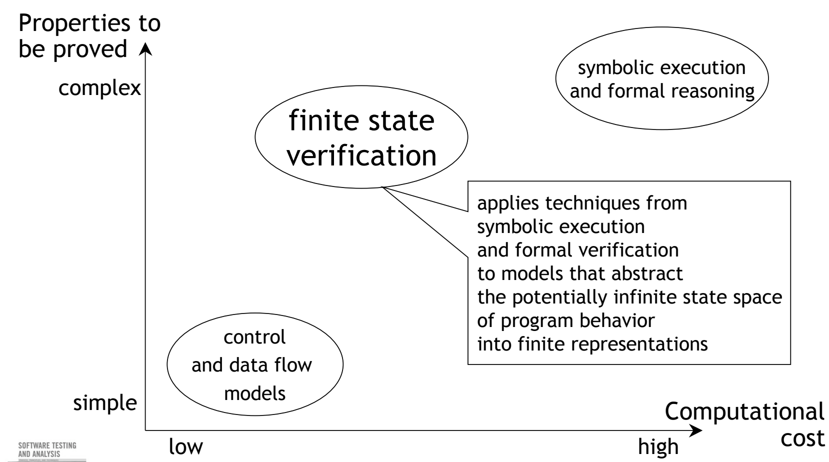
Learning objectives

- Understand the purpose and appropriate uses of finite-state verification (fsv)
  - Understand how fsv mitigates weaknesses of testing
  - Understand how testing complements fsv
- Understand modeling for fsv as a balance between cost and precision
- Distinguish explicit state enumeration from analysis of implicit models
  - And understand why implicit models are sometimes (but not always) more effective

Limits and trade-offs

- Most important properties of program execution are undecidable in general
- Finite state verification can automatically prove *some* significant properties of a *finite model* of the infinite execution space
  - balance trade-offs among
    - generality of properties to be checked
    - class of programs or models that can be checked
    - computational effort in checking
    - human effort in producing models and specifying properties

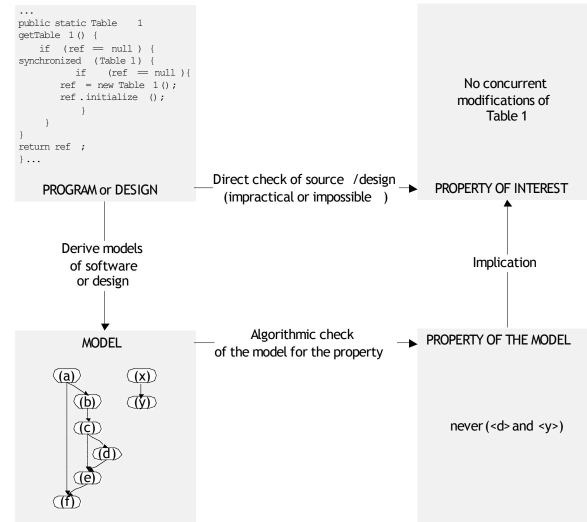
Resources and results



Cost trade-offs

- Human effort and skill are required
  - to prepare a finite state model
  - to prepare a suitable specification for automated analysis
- Iterative process:
  - prepare a model and specify properties
  - attempt verification
  - receive reports of impossible or unimportant faults
  - refine the specification or the model
- Automated step
  - computationally costly
    - computational cost impacts the cost of preparing model and specification, which must be tuned to make verification feasible
  - manually refining model and specification less expensive with near-interactive analysis tools

Analysis of models



Applications for Finite State Verification

- Concurrent (multi-threaded, distributed, ...)
  - Difficult to test thoroughly (apparent non-determinism based on scheduler); sensitive to differences between development environment and field environment
  - First and most well-developed application of FSV
- Data models
  - Difficult to identify "corner cases" and interactions among constraints, or to thoroughly test them
- Security
  - Some threats depend on unusual (and untested) use

Defining the global state space – Concurrent system example

- Deriving a good finite state model is hard
- Example: finite state machine model of a program with multiple threads of control
  - Simplifying assumptions
    - we can determine in advance the number of threads
    - we can obtain a finite state machine model of each thread
    - we can identify the points at which processes can interact
  - State of the whole system model = tuple of states of individual process models
  - Transition = transition of one or more of the individual processes, acting individually or in concert

State space exploration – Concurrent system example

- Specification: an on-line purchasing system
  - In-memory data structure initialized by reading configuration tables at system start-up
  - Initialization of the data structure must appear atomic
  - The system must be reinitialized on occasion
  - The structure is kept in memory
- Implementation (with bugs):
  - No monitor (Java *synchronized*): too expensive*
  - Double-checked locking idiom* for a fast system

*Bad decision, broken idiom ... but extremely hard to find the bug through testing.

Concurrent system example – implementation

```
class Table1 {
   private static Table1 ref = null;
   private boolean needsInit = true;
   private ElementClass [ ]
   theValues;
   private Table1() { }

public static Table1 getTable1() {
   if (ref == null)
      { synchedInitialize(); }
   return ref;
}

private static synchronized void
   synchedInitialize() {
    if (ref == null) {
       ref = new Table1();
       ref.initialize();
    }
}
```

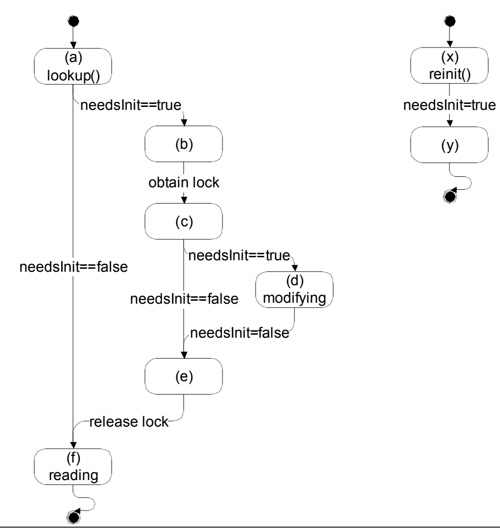
```
public void reinit()
       { needsInit = true; }

private synchronized void
initialize() {
. . .
       needsInit = false;
}

public int lookup(int i) {
   if (needsInit) {
      synchronized(this) {
         if (needsInit) {
            this.initialize();
         }
      }
   }
   return theValues[i].getX()
    + theValues[i].getY();
}
. . .
}
```


Analysis

- Start from models of individual threads
- Systematically trace all the possible interleavings of threads
  - Like hand-executing all possible sequences of execution, but automated

... begin by constructing a finite state machine model of each individual thread ...

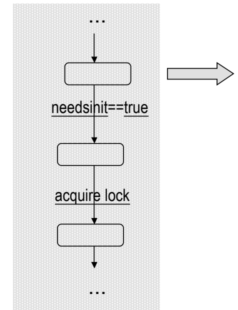
A finite state machine model for each thread



Analysis

- Java threading rules:
  - when one thread has obtained a monitor lock
  - the other thread cannot obtain the same lock
- Locking
  - prevents threads from concurrently calling *initialize*
  - Does not prevent possible race condition between threads executing the *lookup* method
- Tracing possible executions by hand is completely impractical

Express the model in Promela




```
proctype Lookup(int id ) {
  if :: (needsInit) ->
    atomic { ! locked  -> locked = true; };
    if  :: (needsInit) ->
      assert (! modifying);
      modifying = true;
      /*  Initialization happens here */
      modifying = false ;
      needsInit = false;
     :: (! needsInit) ->
      skip;
     fi;
     locked = false ;
  fi;
  assert  (! modifying);}
```

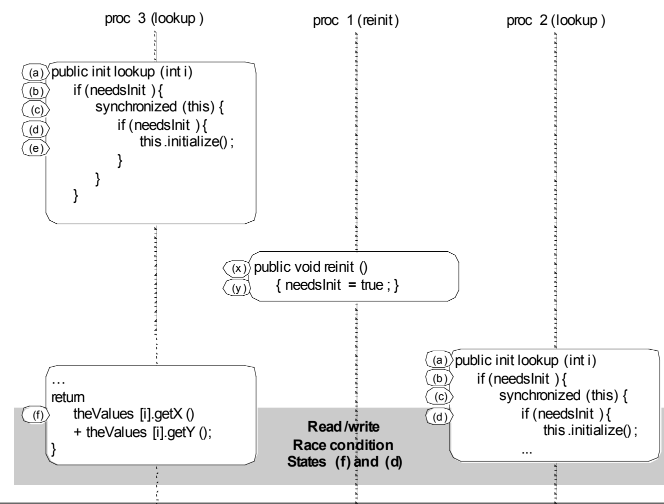

Run Spin; Inspect Output

Spin

- Depth-first search of possible executions of the model
- Explores 10 states and 51 state transitions in 0.16 seconds
- Finds a sequence of 17 transitions from the initial state of the model to a state in which one of the assertions in the model evaluates to false

```
Depth=10 States=51 Transitions=92 Memory=2.302
pan: assertion violated  !(modifying) (at depth 17)
pan: wrote pan_in.trail
(Spin Version 4.2.5 -- 2 April 2005)
…
0.16 real        0.00 user        0.03 sys
```


Interpret the trace



The Promela (Spin) modeling language

- A set of processes described by *process types*
  - Can model threads (Java), processes (Unix), devices, resources, etc.
- C-like syntax, with *guarded commands*
  - *expression* -> *statements*
    - guarded*; not* the same as *if (expression) { statements };*
  - atomic { statements }
    - treat as a single, atomic step (without interleaving)
  - do ... od, if ... fi
    - with multiple :: alternatives, chosen *non-deterministically*

Safety and liveness properties

- Safety: bad things should not happen
  - e.g., two processes should not modify a variable at the same time.
  - Easy to specify in Promela with *assert(...)*
- Liveness: good things should eventually happen
  - e.g., if I push the button, eventually the elevator should arrive
  - Can be specified in temporal logic; more expensive to check
  - Fairness (I should get lucky now and then) is an important and common class of liveness properties

The state explosion problem

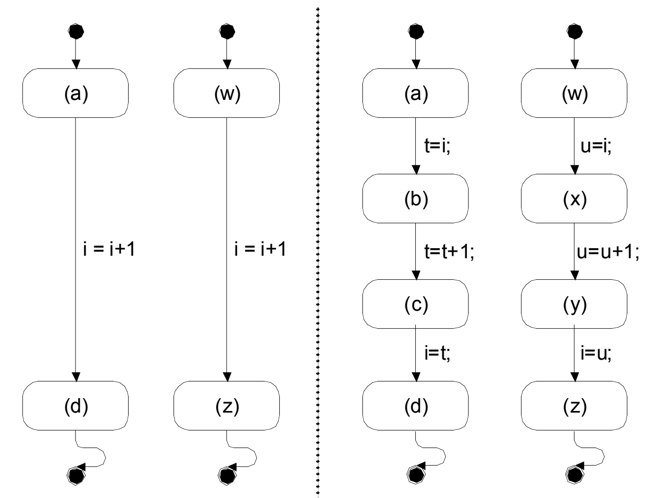
Dining philosophers - looking for deadlock with SPIN

5 phils+forks	145 states deadlock found
10 phils+forks	18,313 states error trace too long to be useful
15 phils+forks	148,897 states error trace too long to be useful

The model correspondence problem

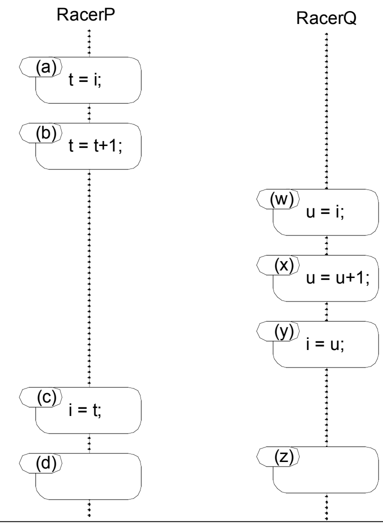
- verify correspondence between model and program:
  - extract the model from the source code with verified procedures
    - blindly mirroring all details ⇒ state space explosion
    - omitting crucial detail ⇒ "false alarm" reports
  - produce the source code automatically from the model
    - most applicable within well-understood domains
  - conformance testing
    - good tradeoff

Granularity of modeling



Analysis of different models

we can find the race only with fine-grain models



Looking for the appropriate granularity

- Compilers may rearrange the order of instruction
  - a simple store of a value into a memory cell may be compiled into a store into a local register, with the actual store to memory appearing later (or not at all)
  - Two loads or stores to different memory locations may be reordered for reasons of efficiency
  - Parallel computers may place values initially in the cache memory of a local processor, and only later write into a memory area
- Even representing each memory access as an individual action is not always sufficient!

Example

- Suppose we use the double-check idiom only for lazy initialization
- It would still be wrong, but…
- it is unlikely we would discover the flaw through finite state verification:
  - Spin assumes that memory accesses occur in the order given in the Promela program, and ...
  - we code them in the same order as the Java program, but …
  - Java does not guarantee that they will be executed in that order

Intensional models

- Enumerating all reachable states is a limiting factor of finite state verification
- We can reduce the space by using intensional (symbolic) representations:
  - describe sets of reachable states without enumerating each one individually
- Example (set of Integers)
  - Enumeration {2, 4, 6, 8, 10, 12, 14, 16, 18}
  - Intensional rep. {x∈N|x mod 2 =0 and 0<x<20}

characteristic function

Intensional models do not necessarily grow with the size of the set they represent

A useful intensional model: OBDD

- Ordered Binary Decision Diagrams
  - A compact representation of Boolean functions
- Characteristic function for transition relations
  - Transitions = pairs of states
  - Function from pairs of states to Booleans:
    - True if the there is a transition between the pair
  - Built iteratively by breadth-first expansion of the state space:
    - creating a representation of the whole set of states reachable in *k+1* steps from the set of states reachable in *k* steps
    - the OBDD stabilizes when all the transitions that can occur in the next step are already represented in the OBDD

From OBDDs to Symbolic Checking

- An intensional representation is not enough
- We must have an algorithm for determining whether that set satisfies the property we are checking

example:

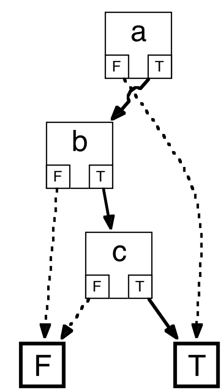
- OBDD to represent
  - the transition relation of a set of communicating state machines
  - a class of temporal logic specification formulas
- Combine OBDD representations of model and specification to produce a representation of just the set of transitions leading to a violation of the specification
  - If the set is empty, the property has been verified

Represent transition relations as Boolean functions

a ⇒ b and c

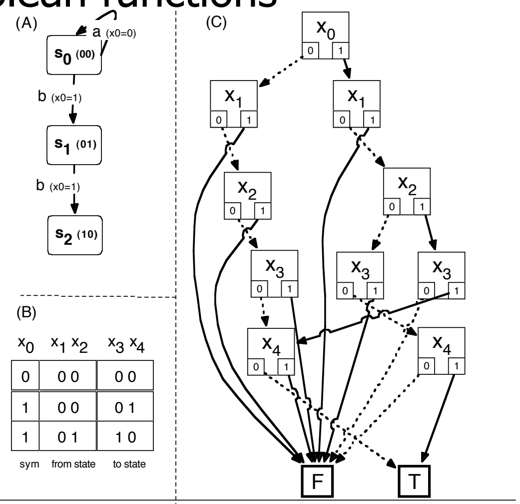
not(a) or (b and c)

the BDD is a decision tree that has been transformed into an acyclic graph by merging nodes leading to identical subtrees



Representing transition relations as Boolean functions

A. Assign a label to each state
B. Encode transitions
C. The transition tuples correspond to paths leading to *true*; all other paths lead to *false*



x_0	x_1 x_2	x_3 x_4
0	0 0	0 0
1	0 0	0 1
1	0 1	1 0
sym	from state	to state

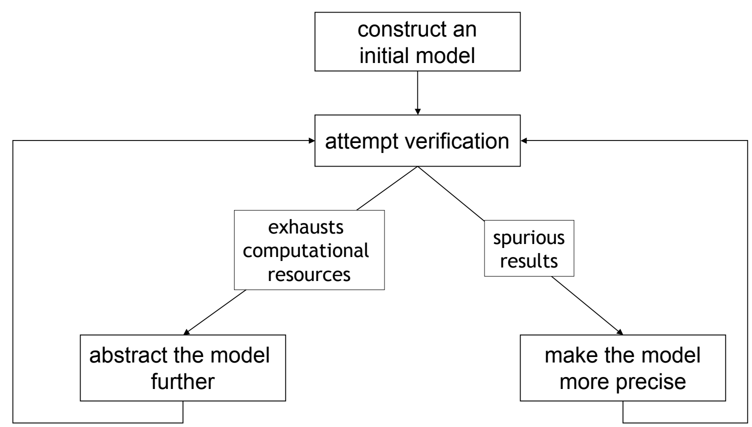
Intensional vs explicit representations

- Worst case:
  - given a large set S of states
    - a representation capable of distinguishing each subset of S
    - cannot be more compact on average
    - than the representation that simply lists elements of the chosen subset.
- Intensional representations work well when they exploit structure and regularity of the state space

Model refinement

- Construction of finite state models
  - balancing precision and efficiency
- Often the first model is unsatisfactory
  - report potential failures that are obviously impossible
  - exhaust resources before producing any result
- Minor differences in the model can have large effects on tractability of the verification procedure
- finite state verification as iterative process

Iterative process



Refinement: Adding details to the model

$M_1 \models P$ initial (coarse grain) model

(the counter example that violates P is possible in M_1, but does not correspond to an execution of the real program)

$M_2 \models P$ refined (more detailed) model

(the counter example is not possible in M_2 but a new counter examples violates M_2 but does not correspond to an execution of the real program)

....

$M_k \models P$

(the counter example that violates P in M_k corresponds to an execution in the real program)

Example: Boolean programs

- Initial Boolean program
  - omits all variables
  - branches *if*, *while*,.. refer to a dummy Boolean variable whose value is unknown
- Refined Boolean program
  - add ONLY Boolean variables, with assignments and tests
- Example: pump controller
  - a counter-example shows that the *waterLevel* variable cannot be ignored
  - a refined Boolean program adds a Boolean variable corresponding to a predicate in which *waterLevel* is tested (*waterLevel* < *highLimit*) rather than adding the variable *waterLevel* itself

Another refinement approach: add premises to the property

initial (coarse grain) model

$M \models P$

add a constraint C_1 that eliminates the bogus behavior

$M \models C_1 \Rightarrow P$

$M \models (C_1 \text{ and } C_2) \Rightarrow P$

.... until the verification succeeds or produces a valid counter example

Other Domains for Finite-State Verification

- Concurrent systems are the most common application domain
- But the same general principle (systematic analysis of models, where thorough testing is impractical) has other applications
- Example: Complex data models
  - Difficult to consider all the possible combinations of choices in a complex data model

Data model verification and relational algebra

- Many information systems are characterized by
  - simple logic and algorithms
  - complex data structures
- Key element of these systems is the data model
  (UML class and object diagrams + OCL assertions)
  = sets of data and relations among them
- The challenge is to prove that
  - individual constraint are consistent and
  - together they ensure the desired properties of the system as a whole

Example: a simple web site

Signature = Sets + Relations

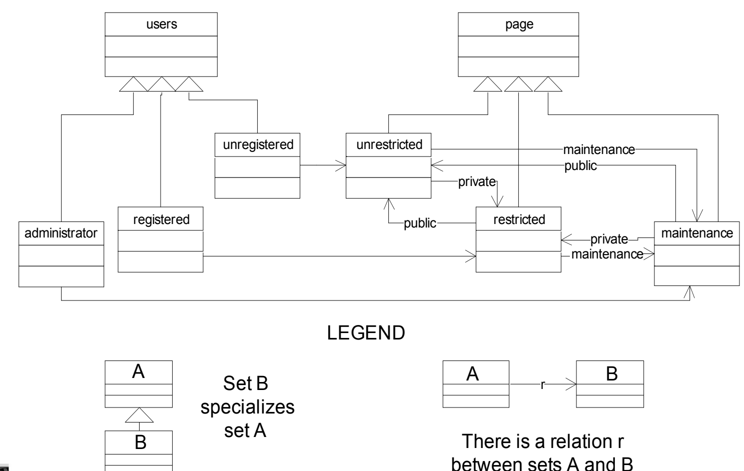
- A **set** of *pages* divided among *restricted*, *unrestricted*, *maintenance* pages
  - unrestricted pages: freely accessible
  - restricted pages: accessible only to registered users
  - maintenance pages: inaccessible to both sets of users
- A **set** of *users*: *administrator*, *registered*, and *unregistered*
- A set of *links* **relations** among pages
  - *private* links lead to *restricted* pages
  - *public* links lead to *unrestricted* pages
  - *Maintenance* links lead to *maintenance* pages
- A set of *access rights* **relations** between users and pages
  - *unregistered users* can access only unrestricted pages
  - *registered* users can access both restricted and unrestricted pages
  - *administrator* can access all pages including maintenance pages

Complete a specification with constraints

Example constraints for the web site:

- Exclude self loops from *links* relations
- Allow at most one type of link between two pages
  - NOTE: relations need not be symmetric: <A, B> ≠ <B, A>
- Web site must be connected
- ...

The data model for the simple web site



Relational algebra to reason about sets and relations

- set union and set intersection obey many of the same algebraic laws as addition and subtraction of integers:
  - commutative law

    $A \cup B = B \cup A$

    $A \cap B = B \cap A$
  - associative law

    $(A \cup B) \cup C = A \cup (B \cup C)$

    $(A \cap B) \cap C = A \cap (B \cap C)$
  - distributive law

    $A \cap (B \cup C) = (A \cap B) \cap (A \cap C)$
  - ...

A relational algebra specification (Alloy): Page

```
module WebSite

// Pages include three disjoint sets of links
sig Page {disj linksPriv, linksPub, linksMain: set Page }

// Each type of link points to a particular class of page
fact connPub {all p:Page, s: Site | p.linksPub in s.unres }
fact connPriv {all p:Page, s: Site | p.linksPriv in s.res }
fact connMain {all p:Page, s: Site | p.linksMain in s.main }

// Self loops are not allowed
fact noSelfLoop {no p:Page| p in p.linksPriv+p.linksPub+p.linksMain }
```

signature:
Set Page

constraints
Introduce
relations

A relational algebra specification: User

```
// Users are characterized by the set of pages that they can access
sig User {pages: set Page }
// Users are partitioned into three sets
part sig Administrator, Registered, Unregistered extends User {}
// Unregistered users can access only the home page, and unrestricted pages
fact accUnregistered {
all u: Unregistered, s: Site|u.pages = (s.home+s.unres)
}
// Registered users can access the home page,restricted and unrestricted pages
fact accRegistered {
all u: Registered, s: Site|u.pages = (s.home+s.res+s.unres)
}
// Administrators can access all pages
fact accAdministrator {
all u: Administrator, s: Site|
u.pages = (s.home+s.res+s.unres+s.main)
}
```

Constraints map
users to pages

Analyze relational algebra specifications

- **Overconstrained** specifications are not satisfiable by any implementation,
- **Underconstrained** specifications allow undesirable implementations
- Specifications identify infinite sets of solutions

  ... so ...

  Properties of a relational specification are undecidable
- A (counter) example that invalidates a property can be found within a finite set of small models

  ... so ...

  We can verify a specification over a finite set of solutions by limiting the cardinality of the sets

Checking a finite set of solutions

- If an example is found:
  - There are no logical contradictions in the model
  - The solution is not overconstrained
- If no counterexample of a property is found:
  - no *reasonably small* solution (property violation) exists
  - BUT NOT that *NO solution* exists
    - We depend on a "small scope hypothesis": Most bugs that can cause failure with large collections of objects can also cause failure with very small collections (so it's worth looking for bugs in small collections even if we can't afford to look in big ones)

Analysis of the web site specification

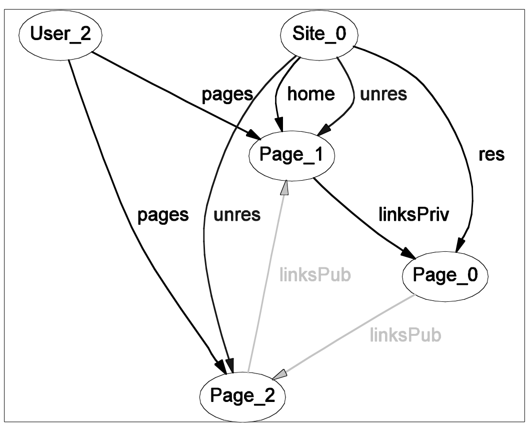
```
run init for 5

// can unregistered users
//     visit all unrestricted pages?
assert browsePub {
all p: Page, s: Site|
       p in s.unres implies s.home in p.* linksPub
}
check browsePub for 3
```

Cardinality limit:
Consider up to
5 objects of each type

Property to be checked

*
Transitive closure
(including home)

Analysis result



Counterexample:

- Unregistered *User_2* cannot visit the unrestricted page *page_2*
- The only path from the home page to *page_2* goes through the restricted page *page_0*
- The property is violated because unrestricted browsing paths can be interrupted by restricted pages or pages under maintenance

Correcting the specification

- We can eliminate the problem by eliminating public links from maintenance or reserved pages:

```
fact descendant {
   all p:Pages, s:Site|p in s.main+s.res
      implies no p. links.linkPub
}
```

- Analysis would find no counterexample of cardinality 3
- We cannot conclude that no larger counter-example exists, but we may be satisfied that there is no reason to expect this property to be violated only in larger models

Summary

- Finite state verification is complementary to testing
  - Can find bugs that are extremely hard to test for
    - example: race conditions that happen very rarely, under conditions that are hard to control
  - But is limited in scope
    - cannot be used to find all kinds of errors
- Checking models can be (and is) automated
- But designing good models is challenging
    - Requires careful consideration of abstraction, granularity, and the properties to be checked. Often requires a cycle of model / check / refine until a useful result is obtained.