Validating open and component based systems

Lars-Åke Fredlund

Formal methods for specifying components

- We have seen quite a few languages for specifying components and systems:
- Model oriented
  - Mathematical: Z, ...
  - Logics: temporal logic
  - Object oriented: Object-Z
- Algebraic process algebras for concurrency and distribution
  - For basic protocols: CCS
- But there are also a lot of programming language dependent methodologies and tools...

Formal methods for specifying components

- We have seen quite a few languages for specifying components and systems:
- Model oriented
  - Mathematical: Z, ...
  - Logics: temporal logic
  - Object oriented: Object-Z
- Algebraic process algebras for concurrency and distribution
  - For basic protocols: CCS
- But there are also a lot of programming language dependent methodologies and tools...
- **Today:** methods and tools for validating components

Design by Contract

- Associate *contract* to components, specifying what
  - is expected of users of the component
  - is expected of the component when it is used correctly

Design by Contract

- Associate *contract* to components, specifying what
  - is expected of users of the component
  - is expected of the component when it is used correctly
- For general software components we can think of a number of desirable component properties:
  - safety properties: what a component **does not do** – e.g., does not modify its parameters
  - liveness properties: what the component **must do** – e.g., return a value upon correct invocation
  - *security*: does not leak information about its parameters
  - *resource usage*: which resources will the component use – e..g, no more than X KB of stack
  - *timing characteristics*: response time

Design by Contract, part II

- Modern design by contract solutions popularized by the Eiffel language (Bertrand Meyer)
- But now available for many object-oriented and non-object oriented programming languages (Java, C++, C#, Erlang, . . .)
- Existing contract frameworks mostly address only functional behaviour, i.e., they cannot express claims about performance, or about resource usage
- Normally only input-output relations for methods are specified, i.e., not reactive behaviours (sequences of calls)

Contracts in Object-Oriented Languages

- For an object oriented language, a contract specifies for an operation $m(P_1, \ldots, P_n)$ with return value r:

Contracts in Object-Oriented Languages

- For an object oriented language, a contract specifies for an operation $m(P_1, \ldots, P_n)$ with return value r:
  - a pre-condition $pre(P_1, \ldots, P_n)$ which specifies allowed combinations of input parameters

    (an obligation on the caller)

Contracts in Object-Oriented Languages

- For an object oriented language, a contract specifies for an operation $m(P_1, \ldots, P_n)$ with return value r:
  - a pre-condition $pre(P_1, \ldots, P_n)$ which specifies allowed combinations of input parameters

    (an obligation on the caller)
  - a post-condition $\mathrm{post}(P_1, \ldots, P_n, r)$ that specifies the ***effects*** of the method on its parameters (e.g., does it assign values to its parameters), and the returned value r

    (a guarantee from the method to the caller which depends upon the pre-condition being satisfied)

Contracts in Object-Oriented Languages

- For an object oriented language, a contract specifies for an operation $m(P_1, \ldots, P_n)$ with return value r:
  - a pre-condition $pre(P_1, \ldots, P_n)$ which specifies allowed combinations of input parameters

    (an obligation on the caller)
  - a post-condition $\mathrm{post}(P_1, \ldots, P_n, r)$ that specifies the ***effects*** of the method on its parameters (e.g., does it assign values to its parameters), and the returned value r

    (a guarantee from the method to the caller which depends upon the pre-condition being satisfied)

- **Note that the post-condition does not need to completely describe the actions of the method**

Contracts in Object-Oriented Languages

- For an object oriented language, a contract specifies for an operation $m(P_1, \ldots, P_n)$ with return value r:
  - a pre-condition $pre(P_1, \ldots, P_n)$ which specifies allowed combinations of input parameters

    (an obligation on the caller)
  - a post-condition $\mathrm{post}(P_1, \ldots, P_n, r)$ that specifies the ***effects*** of the method on its parameters (e.g., does it assign values to its parameters), and the returned value r

    (a guarantee from the method to the caller which depends upon the pre-condition being satisfied)

- **Note that the post-condition does not need to completely describe the actions of the method**
- The language for specifying pre– and post–conditions is up to the method (normally English or first-order logic)

Contracts in OO languages, part II

- Possibly there is also a specification of an invariant *inv* of the object state, preserved by the pre-conditions such that:

  if the invariant holds and the pre-condition is satisfied (pre **and** *inv) then in the resulting state both the post-condition and the state invariant holds (post* **and** *inv)*

Contracts in OO languages, part II

- Possibly there is also a specification of an invariant *inv* of the object state, preserved by the pre-conditions such that:

  if the invariant holds and the pre-condition is satisfied (pre **and** *inv) then in the resulting state both the post-condition and the state invariant holds (post* **and** *inv)*

- Such invariants help specify the ***internal consistency*** of a class/object:

```
class IntSet {
  int[] a;

  //@ invariant 0 <= size && size <= a.length;
  int size;
  ...
}
```

Checking Specifications and code

- Given a specification of your (component based) system
- What should be checked?
  - Are all contract specifications satisfied?
  - Are the pre– and post–conditions strong enough to guarantee that state invariants are kept?
- What methods are available for checking compliance?
- What tools are available?

Validation Techniques For Contracts

- Runtime Monitoring
- Static Analysis
- Systematic Testing
- Model checking
- Theorem proving
- And many combinations of these techniques. . .

Validation Quality

Too coarse abstractions (in model checking, static analysis etc) can result in imprecise analysis results:

Validation Quality

Too coarse abstractions (in model checking, static analysis etc) can result in imprecise analysis results:

- *False negatives*: bugs which are never detected

Validation Quality

Too coarse abstractions (in model checking, static analysis etc) can result in imprecise analysis results:

- *False negatives*: bugs which are never detected
- *False positives*: reports about possible bugs which turn out to be harmless

Validation Quality

Too coarse abstractions (in model checking, static analysis etc) can result in imprecise analysis results:

- *False negatives*: bugs which are never detected
- *False positives*: reports about possible bugs which turn out to be harmless

A good analysis tool is one that generates few false positives and has good coverage with respect to the actual bugs in the code (has few false negatives)

Runtime Monitoring

- **Idea**: *We modify the program to check whether contracts and state invariants are satisfied when methods are called*

Runtime Monitoring

- **Idea**: *We modify the program to check whether contracts and state invariants are satisfied when methods are called*
- Conceptually we run the program in parallel with a compliance monitor guarding its operation
- If the monitor detects that the program is about to violate a calling contract or invariant, it halts the program

Runtime Monitoring

- **Idea**: *We modify the program to check whether contracts and state invariants are satisfied when methods are called*
- Conceptually we run the program in parallel with a compliance monitor guarding its operation
- If the monitor detects that the program is about to violate a calling contract or invariant, it halts the program
- Very useful for enforcing flexible application security policies (more flexible than the Java sandbox)

Runtime Monitoring

- **Idea**: *We modify the program to check whether contracts and state invariants are satisfied when methods are called*
- Conceptually we run the program in parallel with a compliance monitor guarding its operation
- If the monitor detects that the program is about to violate a calling contract or invariant, it halts the program
- Very useful for enforcing flexible application security policies (more flexible than the Java sandbox)
- **Example**: check all I/O operations by an applet and allow only writes and reads to and from certain locations

Runtime Monitoring: Difficulties

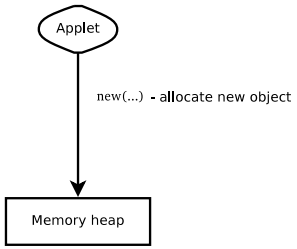
- For some programs halting its execution upon error is not very useful (e.g. Microsoft Word)

Runtime Monitoring: Difficulties

- For some programs halting its execution upon error is not very useful (e.g. Microsoft Word)
- Difficult to implement some contracts (e.g., efficiency concerns, conditions involving quantification, undefinedness)

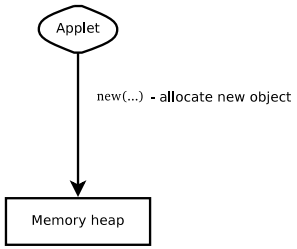
Runtime Monitoring: Java example

- Ensure that a Java applet doesn't allocate too much memory

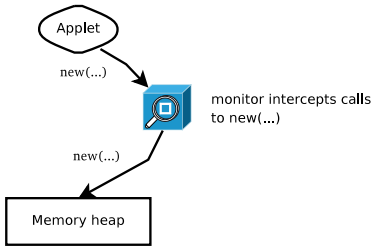


Runtime Monitoring: Java example

- Ensure that a Java applet doesn't allocate too much memory

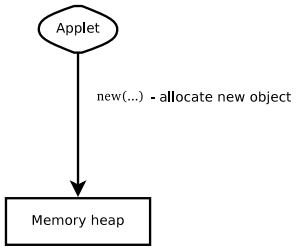


- Design a monitor which is invoked upon every call to `new`, and forbids a new allocation if too much memory has already been allocated

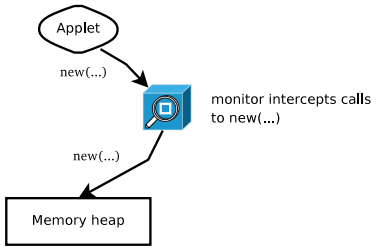


Runtime Monitoring: Java example

- Ensure that a Java applet doesn't allocate too much memory



- Design a monitor which is invoked upon every call to `new`, and forbids a new allocation if too much memory has already been allocated



- **Implementation**: insert code to call the monitor before every instance of a `new` or `newarray` Java bytecode instruction in the Java applet object code

Static Analysis

- Automatically infer properties of the analysed program from its source (or object) code without executing it (a **compile-time** technology)
- Considers a particular phenomenon and tailors an analysis

Static Analysis

- Automatically infer properties of the analysed program from its source (or object) code without executing it (a **compile-time** technology)
- Considers a particular phenomenon and tailors an analysis
- Example: the **Java byte code verifier** ensures that no user program can crash the Java runtime system
- Programs can of course still generate exceptions (null pointer dereferenced) but never crash the host runtime system (caused by e.g. treating an integer as a pointer)
- **Implementation**: the byte code verifier *executes symbolically* the program to be checked but *replaces concrete values with domains* – their types (**abstract interpretation**)

Java Byte code verifier example: the `iadd` instruction

- The `iadd` instruction replaces the two integers on top of the stack with its sum

Java Byte code verifier example: the `iadd` instruction

- The `iadd` instruction replaces the two integers on top of the stack with its sum
- *Abstract exection* of `iadd`: if the top values of the stack are integer types, then replace them with an integer type else byte code verification **fails**

Java Byte code verifier example: the `iadd` instruction

- The `iadd` instruction replaces the two integers on top of the stack with its sum
- *Abstract exection* of `iadd`: if the top values of the stack are integer types, then replace them with an integer type else byte code verification **fails**
- The analysis has to go through every path (method calls, branches, `...`) leading to the particular `inst` instruction to analyse all possible stack shapes when that instruction is executed

Java Byte code verifier example: the `iadd` instruction

- The `iadd` instruction replaces the two integers on top of the stack with its sum
- *Abstract exection* of `iadd`: if the top values of the stack are integer types, then replace them with an integer type else byte code verification **fails**
- The analysis has to go through every path (method calls, branches, `...`) leading to the particular `inst` instruction to analyse all possible stack shapes when that instruction is executed
- Recursive behaviour is handled by noticing and stopping at recurring (abstract) states

Typical static analyses

Beyond the byte code verifier for Java:

- Can exceptions be raised at runtime?
- Can null pointers be dereferenced?
- Can any array operations reference an out-of-bound index?
- Calculation of the method call graph for a Java program
- Estimate memory usage of a Java applet
- Estimate time usage for a method call
- Checking pre and post conditions

C programs: static analysis

When you have a poor language – and a poor type system – use static analysis to improve error detection:

- Example in point: a severe X-Windows security bug was detected by the Coverity static analysis tool
- Code fragment to check permissions:

```
if (getuid() == 0 || geteuid() != 0)
```

C programs: static analysis

When you have a poor language – and a poor type system – use static analysis to improve error detection:

- Example in point: a severe X-Windows security bug was detected by the Coverity static analysis tool
- Code fragment to check permissions:

```
if (getuid() == 0 || geteuid() != 0)
```

- But was incorrectly written:

```
if (getuid() == 0 || geteuid != 0)
```

- The bug (expression always true!) was not detected by the GCC compiler, and thus stayed as a security problem . . .
- Many static analysis tools attempts to detect security problems in C programs from buffer overruns etc

C program: static analysis

Other successful applications of static analysis methods:

- Absence of run-time errors for the ADA code executing in Ariane 5 rockets (an overflow error when converting from a floating point to a signed integer caused a rocket launch failure)

C program: static analysis

Other successful applications of static analysis methods:

- Absence of run-time errors for the ADA code executing in Ariane 5 rockets (an overflow error when converting from a floating point to a signed integer caused a rocket launch failure)
- Verifying absence of run-time errors in the flight control software of Airbus aircrafts

C program: static analysis

Other successful applications of static analysis methods:

- Absence of run-time errors for the ADA code executing in Ariane 5 rockets (an overflow error when converting from a floating point to a signed integer caused a rocket launch failure)
- Verifying absence of run-time errors in the flight control software of Airbus aircrafts
- Checking domain specific coding rules in the Linux community: for instance locking schemes for ensuring mutual access in the Linux kernel

C program: static analysis

Other successful applications of static analysis methods:

- Absence of run-time errors for the ADA code executing in Ariane 5 rockets (an overflow error when converting from a floating point to a signed integer caused a rocket launch failure)
- Verifying absence of run-time errors in the flight control software of Airbus aircrafts
- Checking domain specific coding rules in the Linux community: for instance locking schemes for ensuring mutual access in the Linux kernel
- Microsoft has developed a range of tools for static analysis: Microsoft PREfix, PREfast and SDV (Static Driver Verifier)

  These tools check device driver specific rules: does a 3rd party device driver interact correctly with the Windows kernel?

Static analysis: conclusions

- In general static analysis tools are often applied to check (and enforce) domain specific interface usage rules, or coding rules

Static analysis: conclusions

- In general static analysis tools are often applied to check (and enforce) domain specific interface usage rules, or coding rules
- **Advantages**: automatic method, scales to reasonably sized programs

Static analysis: conclusions

- In general static analysis tools are often applied to check (and enforce) domain specific interface usage rules, or coding rules
- **Advantages**: automatic method, scales to reasonably sized programs
- **Disadvantages**: often a rather coarse analysis (need to over-approximate, to over-abstract, results in many false positives)

Systematic Testing

- Testing means subjecting your component (or system of components) to a stimuli, observing the outcome, and deciding whether the outcome is successful or not
- A classical validation technique, what really works in practise
- Key questions:

Systematic Testing

- Testing means subjecting your component (or system of components) to a stimuli, observing the outcome, and deciding whether the outcome is successful or not
- A classical validation technique, what really works in practise
- Key questions:
  - how to get a **good test coverage**, i.e., that tests explore a large part of the whole program behaviour

Systematic Testing

- Testing means subjecting your component (or system of components) to a stimuli, observing the outcome, and deciding whether the outcome is successful or not
- A classical validation technique, what really works in practise
- Key questions:
  - how to get a **good test coverage**, i.e., that tests explore a large part of the whole program behaviour
  - Or. . . how to write as few tests as possible

Systematic Testing

- Testing means subjecting your component (or system of components) to a stimuli, observing the outcome, and deciding whether the outcome is successful or not
- A classical validation technique, what really works in practise
- Key questions:
  - how to get a **good test coverage**, i.e., that tests explore a large part of the whole program behaviour
  - Or. . . how to write as few tests as possible
  - How to **automate** tests? For instance, in a GUI, how to measure that the visual presentation is correct?

Testing Experiences

- Many products are difficult to test because they are deployed in challenging environments that cannot be simulated before deployment (AXD-301 telephone switch) or simply have huge state spaces
- Example: around 10.000 tests are used for the AXD 301 telephony switch

Testing

What to test:

- **Unit testing** (a method, a function, . . .)
- **Integration testing** (testing units together, to detect interface problems)
- **System testing** (testing a whole system)

Testing

What to test:

- **Unit testing** (a method, a function, . . .)
- **Integration testing** (testing units together, to detect interface problems)
- **System testing** (testing a whole system)

Testing trends: **test-driven development (TDD)**

- Write tests early (before software is written)
- Write code to satisfy tests
- Refactor code

Testing with contracts

- Modern testing methods tries to use contract information to avoid having to construct huge numbers of tests
- **QuickCheck/Erlang** is a tool that combines design-by-contract specifications with testing:

Testing with contracts

- Modern testing methods tries to use contract information to avoid having to construct huge numbers of tests
- **QuickCheck/Erlang** is a tool that combines design-by-contract specifications with testing:
  - **specifying** in pseudo-logic the behaviour of functions

Testing with contracts

- Modern testing methods tries to use contract information to avoid having to construct huge numbers of tests
- **QuickCheck/Erlang** is a tool that combines design-by-contract specifications with testing:
  - **specifying** in pseudo-logic the behaviour of functions
  - providing **generators for test data** with which to test the functions (data generators with probability distributions)

Testing with contracts

- Modern testing methods tries to use contract information to avoid having to construct huge numbers of tests
- **QuickCheck/Erlang** is a tool that combines design-by-contract specifications with testing:
  - **specifying** in pseudo-logic the behaviour of functions
  - providing **generators for test data** with which to test the functions (data generators with probability distributions)
  - From function contracts and data generators a large number of plain **tests are generated automatically**

Testing with contracts

- Modern testing methods tries to use contract information to avoid having to construct huge numbers of tests
- **QuickCheck/Erlang** is a tool that combines design-by-contract specifications with testing:
  - **specifying** in pseudo-logic the behaviour of functions
  - providing **generators for test data** with which to test the functions (data generators with probability distributions)
  - From function contracts and data generators a large number of plain **tests are generated automatically**
  - Counterexamples are **reduced** to provide better feedback

Testing with contracts

- Modern testing methods tries to use contract information to avoid having to construct huge numbers of tests
- **QuickCheck/Erlang** is a tool that combines design-by-contract specifications with testing:
  - **specifying** in pseudo-logic the behaviour of functions
  - providing **generators for test data** with which to test the functions (data generators with probability distributions)
  - From function contracts and data generators a large number of plain **tests are generated automatically**
  - Counterexamples are **reduced** to provide better feedback
  - For concurrent systems, test are expressed as properties over execution traces

Implications for Testing

- Normally testing: have to develop many (10000 tests)

Implications for Testing

- Normally testing: have to develop many (10000 tests)
- QuickCheck: much less tests needed (100-1000), and on a higher-level abstraction level

Implications for Testing

- Normally testing: have to develop many (10000 tests)
- QuickCheck: much less tests needed (100-1000), and on a higher-level abstraction level
- Counterexamples are simplified

QuickCheck variants

QuickCheck is rather successful; note the number of tools developed for different languages:

- QuickCheck/Haskell (the original)
- QuickCheck/Erlang (the commercial variant)
- QuickCheck/Java
- QuickCheck/Scala
- QuickCheck/Perl, QuickCheck/Python, . . .

Hint: a good exercise is to use QuickCheck/XXX for working with an exercise. . .

QuickCheck example

- We want to test the Erlang function `sets:union(S1,S2)` $\rightarrow$ `S1` $\cup$ `S2` for computing the union of two sets

QuickCheck example

- We want to test the Erlang function `sets:union(S1,S2)` $\rightarrow$ `S1` $\cup$ `S2` for computing the union of two sets
- How to express its contract? One desidered property is that union is commutative: $X \cup Y = Y \cup X$

QuickCheck example

- We want to test the Erlang function `sets:union(S1,S2)` $\rightarrow$ `S1` $\cup$ `S2` for computing the union of two sets
- How to express its contract? One desidered property is that union is commutative: $X \cup Y = Y \cup X$
- In Quickcheck/Erlang:

```
commutes() ->
    ?FORALL(X,set(),?FORALL(Y,set(),
        sets:union(X,Y) == sets:union(Y,X))).
```

  - `set()` is a test data generator (with a probability distribution)
  - `?FORALL(X,set(),...)` expresses that the variable `X` should be bound to set values,
  - `sets:union(X,Y) == sets:union(Y,X)` uses normal Erlang term equality `==` for set equality

Test Results

- `quickcheck(commutes)` generates a set of test data (sets X, Y), and finds a counterexample: `X={-6,7,11,10,2}` and `Y={7,1,-4,11,-7}`

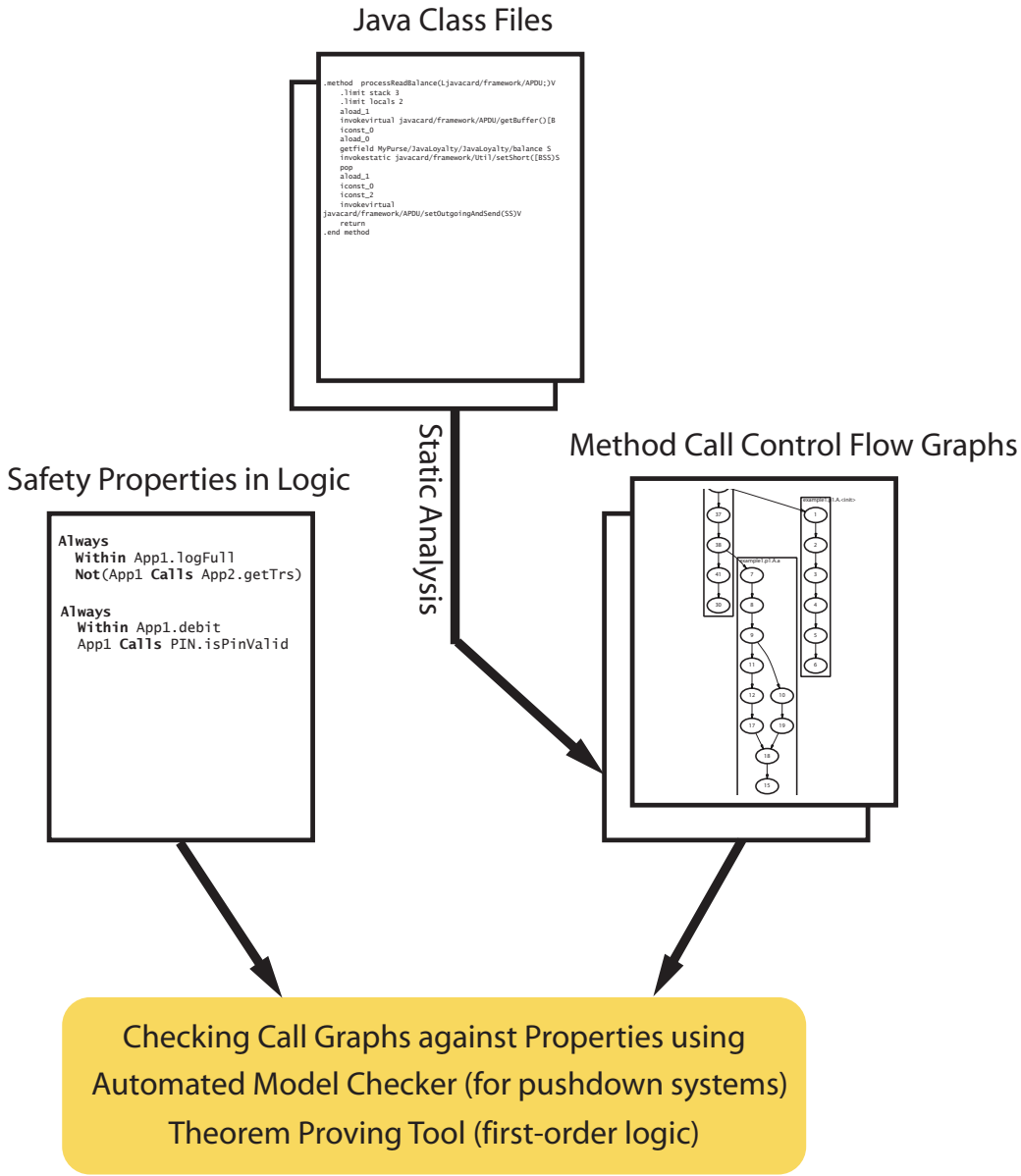
Test Results

- `quickcheck(commutes)` generates a set of test data (sets X, Y), and finds a counterexample: `X={-6,7,11,10,2}` and `Y={7,1,-4,11,-7}`
- Why? `sets:union` can represent sets in different ways: `[1,2]` and `[2,1]` may both be the internal representation of the Erlang set $\{1, 2\}$ but `[1,2] =/= [2,1]`
- Not a bug in the `sets` class but in our test. Happens often.
- How to fix? Use a different equality test.

Model Checking

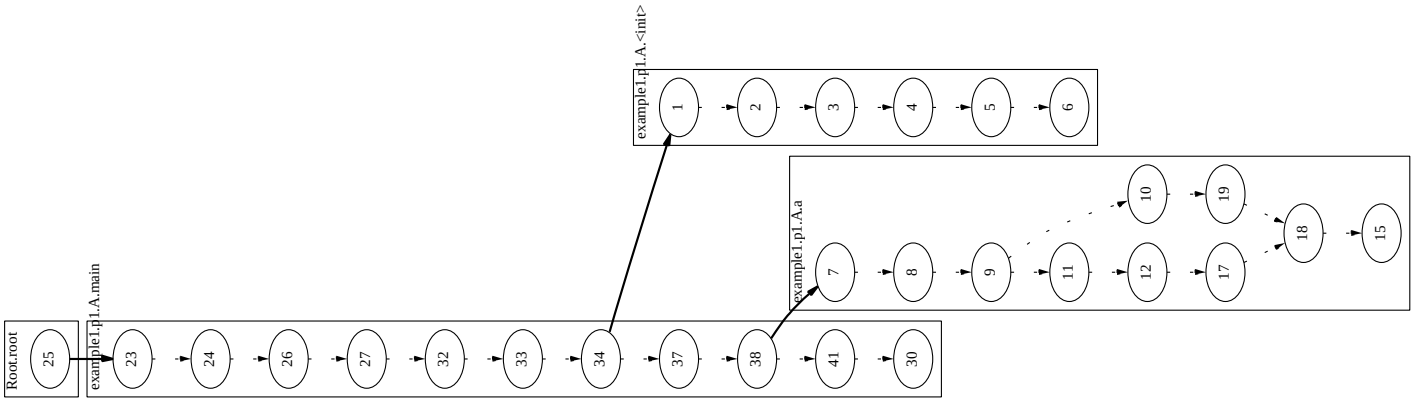
- Idea: **Construct** an abstract **model** of the behaviour of the program, usually a finite state transition graph
- **Check** the abstract model, against some description of desirable/undesirable model properties usually specified in a **temporal logic**
- Usually applied to **reactive systems** (systems that continously react to stimuli)
- Advantage: automatic push button technology
- Disadvantages:
  - Doesn't scale well to larger programs (the model of the behaviour of the program becomes too big)
  - How to construct a faithful model?

Model Checking: an example



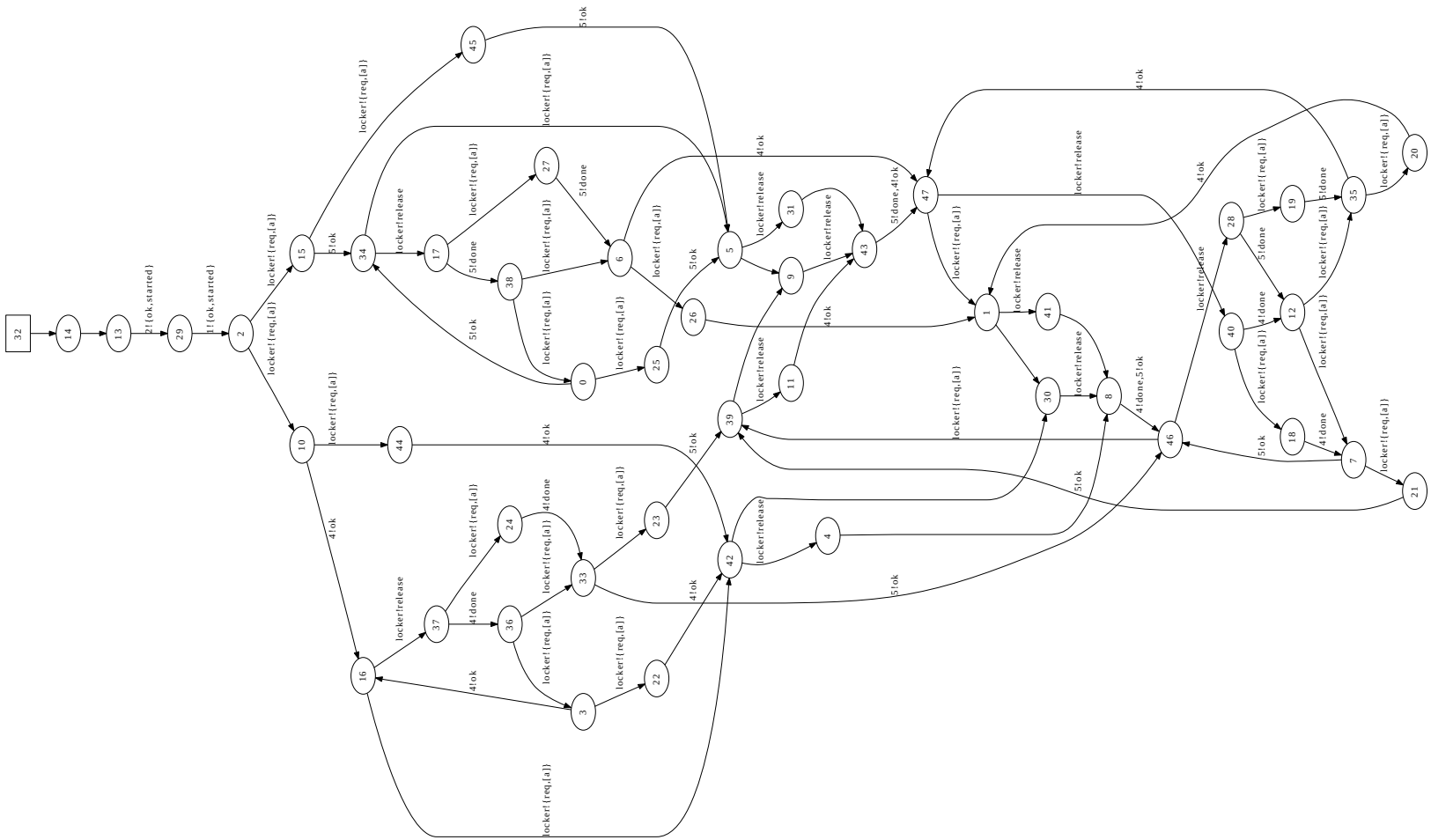
Abstract Model

- Model (method call-graph) extracted using static analysis
- Graph edges are method calls and in–method control flow
- Data is abstracted away – the result is a non-deterministic graph



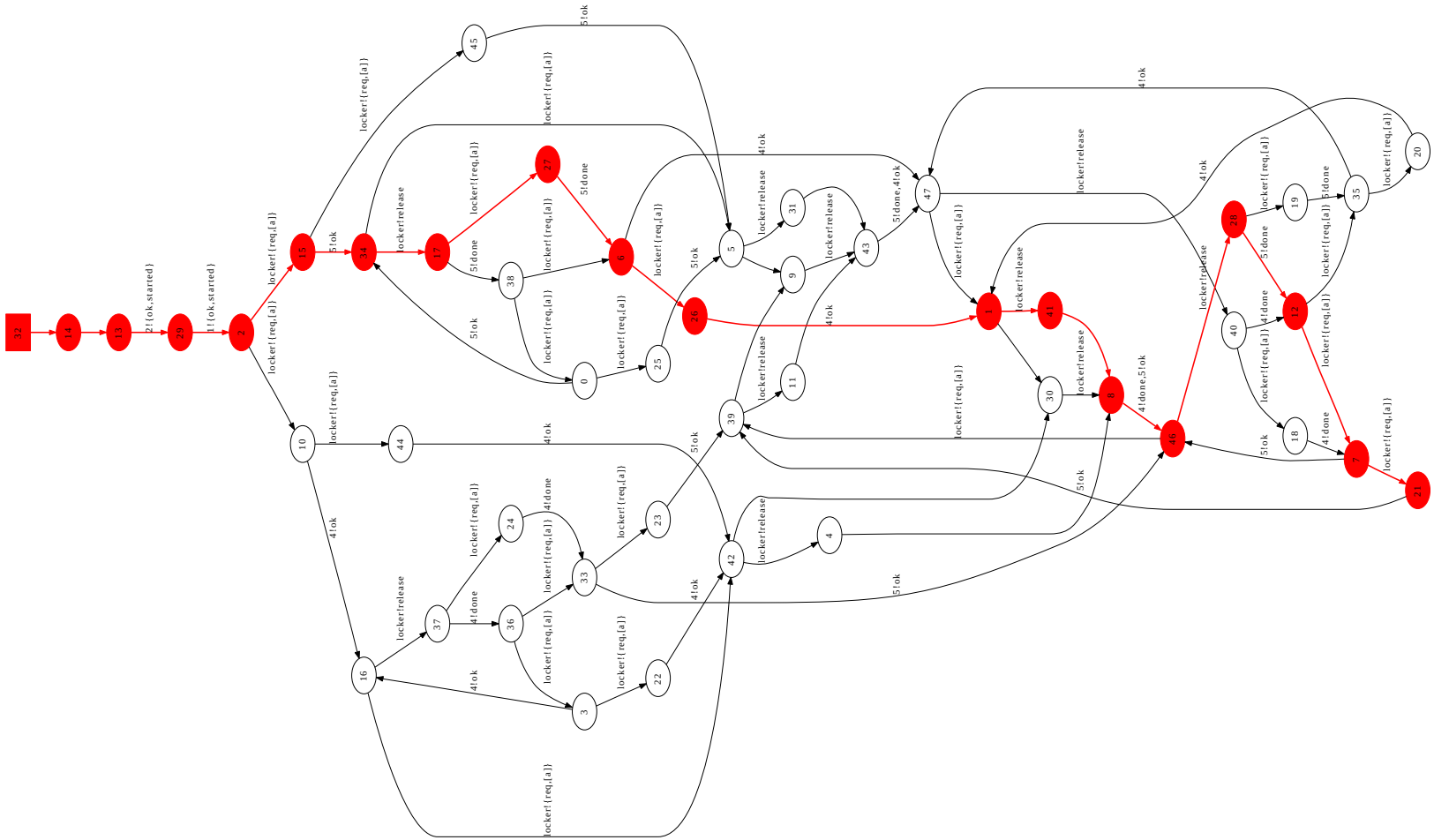
Comparison with Random Testing

The State Space of a small program:



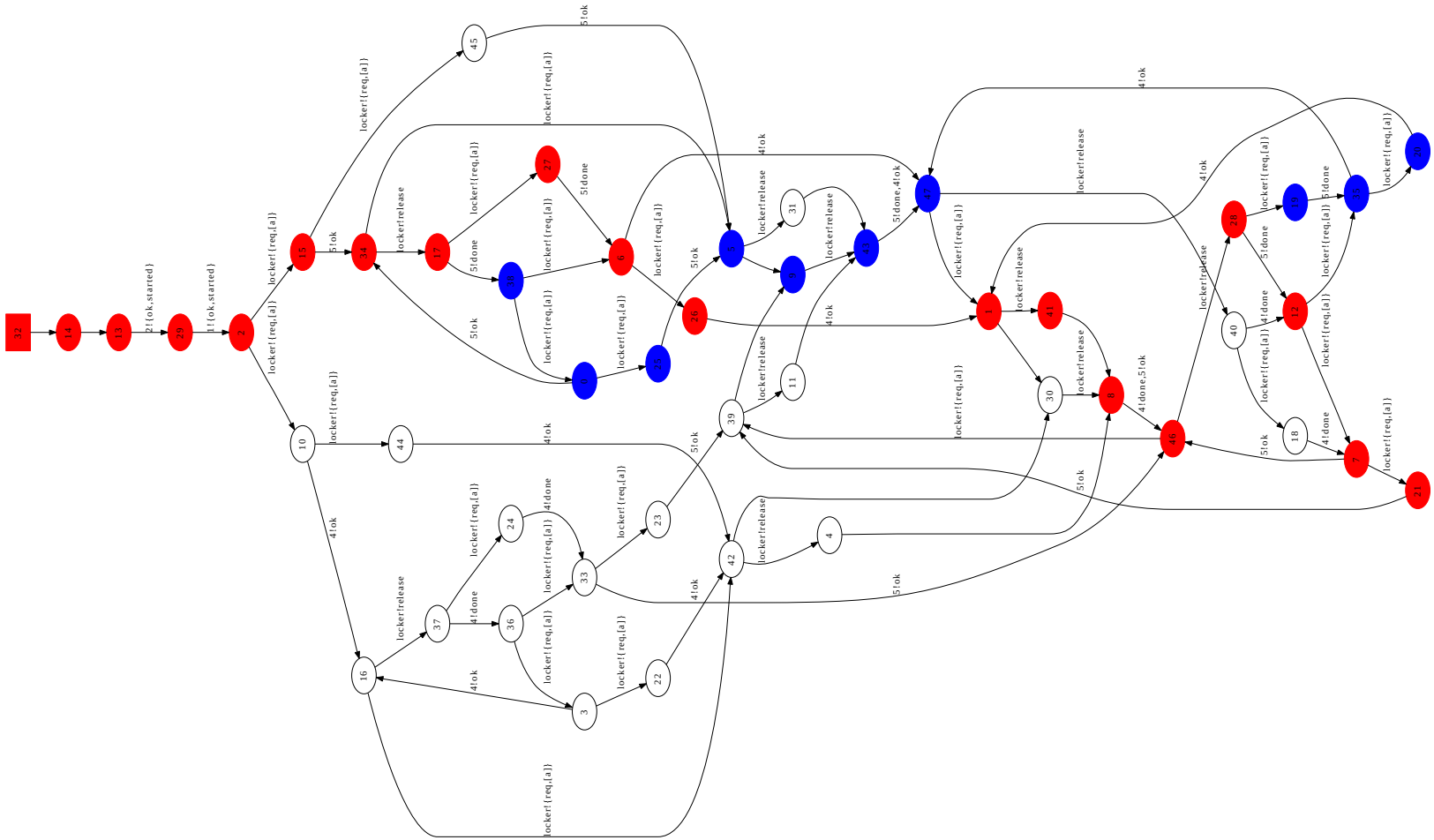
Testing, run 1:

Random testing explores one path through the program:



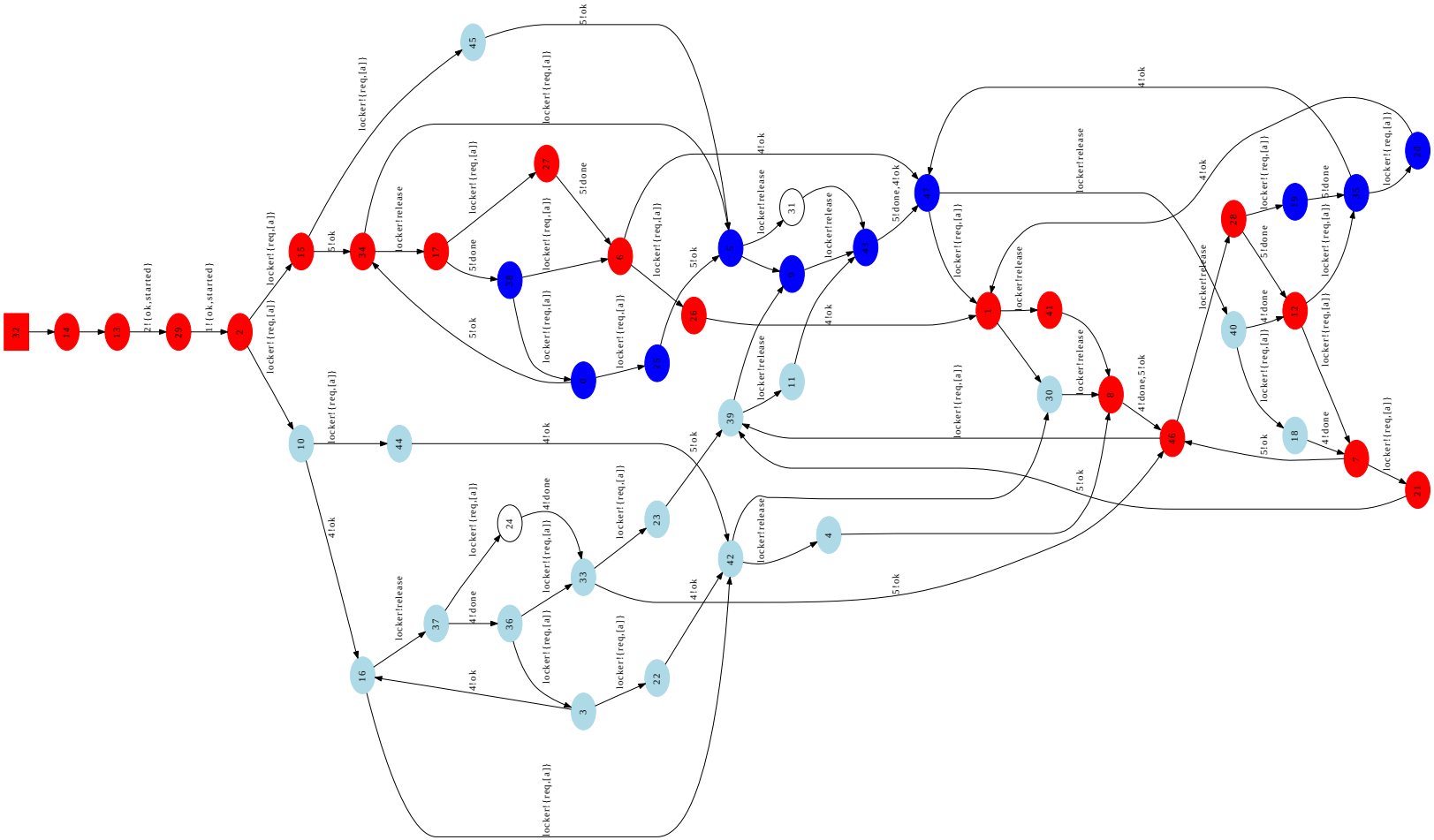
Testing, run 2:

With repeated tests the coverage improves:



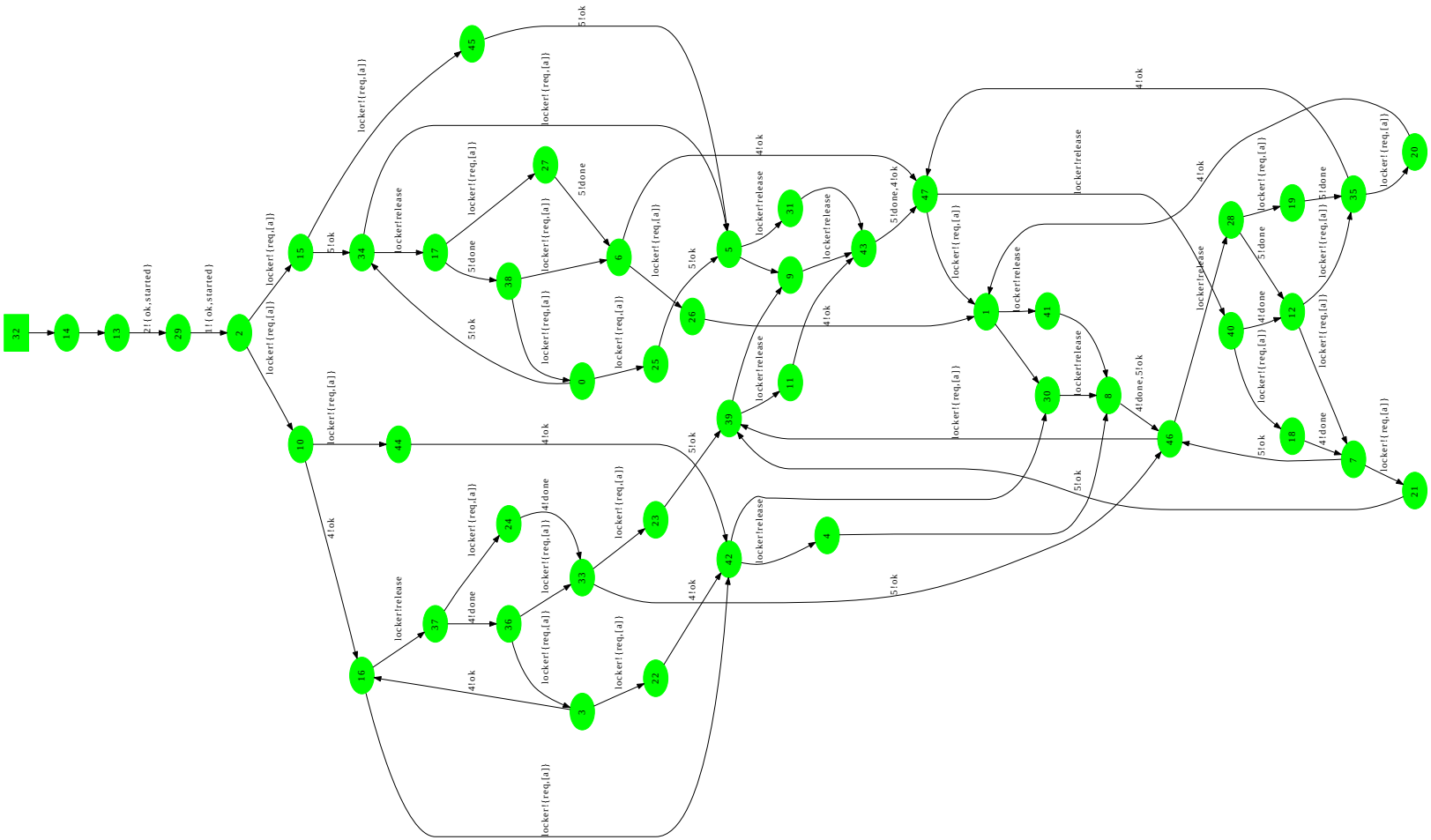
Testing, run n:

But even after a lot of testing some program states may not have been visited:



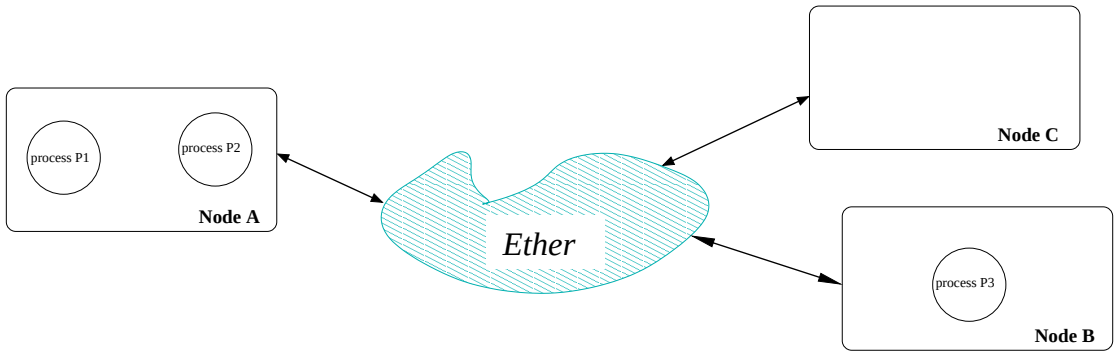
Model checking: 100% coverage

Model checking can guarantee that all states are visited, without revisiting states



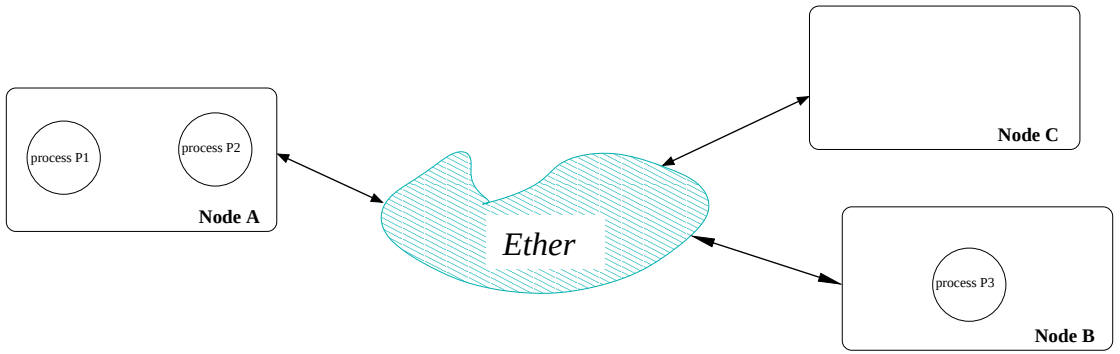
What is the trick? How can we achieve 100% coverage

- Needed: the capability to take a **snapshot** of the model checked program (its program state)
  - A **program state** of an Erlang program is: the contents of all process mailboxes, the state of all running processes, messages in transit, . . .



What is the trick? How can we achieve 100% coverage

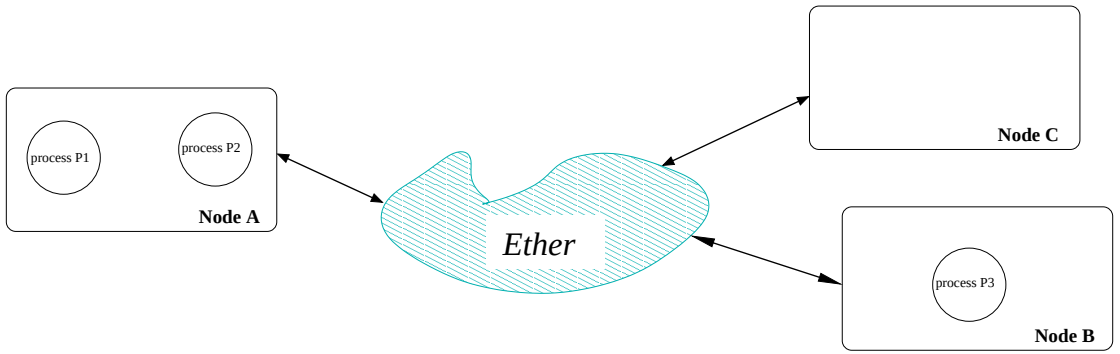
- Needed: the capability to take a **snapshot** of the model checked program (its program state)
  - A **program state** of an Erlang program is: the contents of all process mailboxes, the state of all running processes, messages in transit, . . .



- Save the snapshot to memory and forget about it for a while
- Later continue the execution from the snapshot

What is the trick? How can we achieve 100% coverage

- Needed: the capability to take a **snapshot** of the model checked program (its program state)
  - A **program state** of an Erlang program is: the contents of all process mailboxes, the state of all running processes, messages in transit, . . .



- Save the snapshot to memory and forget about it for a while
- Later continue the execution from the snapshot
- **Difficulties**: too many states (not enough memory/time to save snapshots)

Examples of Model Checkers

- For Java: Java PathFinder
- For C: (SPIN), BLAST, CBMC, …
- For “abstract specifications”: NuSMV, …
- For multi-threaded Win32 programs: Chess

Theorem Proving

- In its pure form: develop a proof manually, have a computer **check the proof** steps
- For example checking our Hoare-style proofs:

$$\frac{A \supset I \qquad \{I \wedge E\}\ C\ \{I\} \qquad (I \wedge \neg E) \supset B}{\{A\}\ \texttt{while}\ E\ \texttt{do}\ C\ \{B\}}$$

- Contemporary theorem provers help in **developing** and **finding proofs** as well
- Many first-order logic theorem provers are **completely automated today!**
- Integrates well with certain formal methods: Z, B–method where proof obligations are generated automatically and theorem provers are used to prove each proof obligation

Theorem proving

Example: checking whether the Z operation schema

$$
\begin{array}{|l}
\hline
\mathit{BadOp} \\
\Delta \mathit{Queue} \\
\mathit{insert}? : \mathbb{N} \\
\mathit{result}! : \mathit{Result} \\
\hline
\mathit{queue}' = \mathit{queue} \frown \langle \mathit{insert}? \rangle \\
\mathit{result}! = \mathit{ok} \\
\hline
\end{array}
$$

violates the state invariant

$$
\begin{array}{|l}
\hline
\mathit{Queue} \\
\mathit{queue} : \mathit{seq}\ \mathbb{N} \\
\hline
\# \mathit{queue} < 10 \\
\hline
\end{array}
$$

is a typical theorem proving task

Theorem Proving

Conclusions:

- **Positive**: very powerful technique, everything we know how to prove can be checked carefully
- **Negative**: generally not very cost effective in time and qualified manpower needed

Java and the Java Modelling Language

- To illustrate some of these validation techniques (design-by-contract, static analysis) we will be using Java and the Java Modelling Language (JML)
- But there are many other tools too. . .
- **Microsoft Research** are very active; tools
  - **Spec#** for C#: uses automatic theorem proving similar to ESC/Java2 below
  - **Code Contracts** for .NET based applications:
    Enables the user to write pre and post conditions
    Implemented using abstract interpretation
    For (C#, Visual Basic, . . .)
  - **Chess** for checking concurrent programs

Hint: a good exercise is to use Spec# or Code Contracts for working with an exercise. . .

The Java Modelling Language – JML

- Annotates Java methods and class descriptions with contract information as comments
- Example:

```
/*@ requires x >= 0.0;
  @ ensures JMLDouble.approxEqualTo
  @           (x,\result * \result);
  @*/
public static double sqrt(double x) {
  ...
}
```

- `requires` specifies the callers obligation
- `ensures` specifies the method guarantee
- `ensures` speaks about normal termination of the method; exceptional termination can be specified as well

The Java Modelling Language – JML, part II

- Universal and existential quantifiers are supported:

```
/*@ requires a != null
  @ && (\forall int i;
        0 < i && i < a.length;
        a[i-1] <= a[i]);
  @*/
  int binarySearch(int[] a, int x) { ... }
```

- *Pure* methods can be used in contracts; suppose the *pure* method *sorted* checks whether an array is sorted:

```
//@ requires a != null && sorted(a);
int binarySearch(int[] a, int x) { ... }
```

- Object invariants characterise object state properties (both caller visible and object internal state)

```
//@ public invariant weight>0;
public class person { public int weight; ... }
```

The Java Modelling Language – JML, part III

- `\result` describes the return value of the method
- `assignable` lists the state variables that may be modified by the method
- `old(variable)` references the old (before the method call) value of `variable`

A Class Example

An example class:

```
public class Purse {
 int balance;

int debit(int amount) throws PurseException {
  if (amount <= balance) {
    int saved_balance = balance;
    balance -= amount;
    return saved_balance;
  }
  else throw new PurseException();
} }
```

A Specified Class Example

The same class extended with contract specification information:

```
public class Purse {
 //@ invariant 0 <= balance;
 int balance;

/*@ requires amount >= 0;
  @ assignable balance;
  @ ensures balance == \old(balance)-amount
  @         && \result == \old(balance); @*/

int debit(int amount) throws PurseException {
  if (amount <= balance) {
    int saved_balance = balance;
    balance -= amount;
    return saved_balance;
  }
  else throw new PurseException();
} }
```

Tool Support for JML: Runtime Checking

After lots of API specification. . .

- The JML compiler compiles Java programs with embedded preconditions, postconditions and invariants
- These conditions and invariants are embedded with the code and checked at runtime

JML Testing

- The **jmlunit** tool is used for testing JML annoted Java programs
- Testing errors are reported only for methods called with correct pre–conditions that fail their post–conditions
- Test data (method parameters) is *not* automatically generated; a user must supply such data
- In comparison the testing tool QuickCheck for Haskell and Erlang seem more convenient to use

Tool Support for JML

- The ESC/Java2 tool permits to do limited static checking of Java programs annotated with JML specifications, *at compile-time*, before *runtime*
- Can check simple conditions such as dereferencing the `null` pointer (`person.name` when `person==null`)
- and whether array accesses can be out-of-bounds
- and precondition violation checking (but not fully)
- Checking is done internally using a first-order automatic theorem prover
- Works best for checking against a simple or restricted API:s, e.g., JavaCard (Java on a SmartCard)
- The prover needs domain specific help

A simple ESC/Java2 example

- Lets design a simple Bag class that stores elements in an array `a`, and has a constructor which initialises the array from a parameter:

```
class Bag {
  /*@ non_null */ int[] a;
  int n;
  //@ invariant 0 <= n && n <= a.length;

  public Bag(int[] input) {
    n = input.length;
    a = new int[n];
    System.arraycopy(input, 0, a, 0, n);
  }
  ...
}
```

- Note that the class variable `a` must be non-null always, and that `n` is bounded by the length of the array `a`

A simple ESC/Java2 example, part II

- What happens when we run the ESC/Java2 static checker?

A simple ESC/Java2 example, part II

- What happens when we run the ESC/Java2 static checker?

```
> ../escjava2 Bag.java

Bag: Bag(int[]) ...
-----------------------------------------------
BadBag.java:7: Warning: Possible null dereference (Nu
    n = input.length;
```

- Aha. Maybe there is a bug. . . ?
  Yes! What if the input parameter is null!

Fixing the example

- Lets stipulate a contract that the caller has to provide a non-null parameter:

```
class Bag {
  /*@ non_null */ int[] a;
  int n;
  //@ invariant 0 <= n && n <= a.length;

  //@ requires input != null;
  public Bag(int[] input) {
  ...
  }
}
```

- Lets try:

```
> ../escjava2 Bag.java
```

- No warnings generated!

A simple ESC/Java2 example, part III

- Lets add a user of the Bag class:

```
class User {
  Bag bag;
  User() { bag = new Bag(null); }
}
```

- and run the analysis again. What happens?

A simple ESC/Java2 example, part III

- Lets add a user of the Bag class:

```
class User {
  Bag bag;
  User() { bag = new Bag(null); }
}
```

- and run the analysis again. What happens?

```
> ../escjava2 Bag.java User.java

User: User() ...
-----------------------------------------------------
BadBag.java:16: Warning: Precondition possibly not es
    User() { bag = new Bag(null); }
                       ^
Associated declaration is "BadBag.java", line 6, col
  //@ requires input != null;
      ^
```

A simple ESC/Java2 example, part IV

- Clearly User didn’t fulfil the precondition for creating a Bag

A simple ESC/Java2 example, part IV

- Clearly User didn’t fulfil the precondition for creating a Bag
- Fixing User removes the error message:

```
class User {
  Bag bag;
  User() { bag = new Bag(new int[0]); }
}
```

A simple ESC/Java2 example, part IV

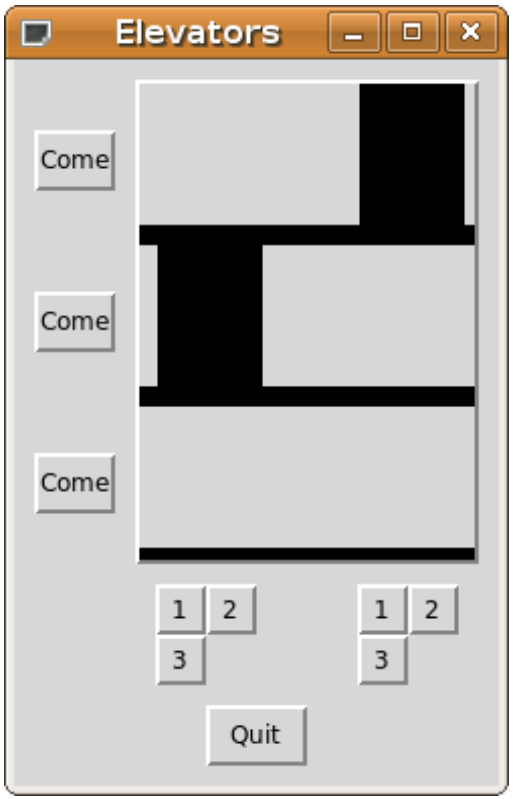
- Clearly User didn't fulfil the precondition for creating a Bag
- Fixing User removes the error message:

```
class User {
  Bag bag;
  User() { bag = new Bag(new int[0]); }
}
```

- **Spec#** for C#: uses automatic theorem proving similar to ESC/Java2 below
- **Code Contracts** for .NET based applications:
  Enables the user to write pre and post conditions
  Implemented using abstract interpretation
  For (C#, Visual Basic, . . .)

Model Checking – and building models

To illustrate model checking of components, and building models, lets us consider a reactive system: **the control software for a set of elevators**



Elevator Control Software

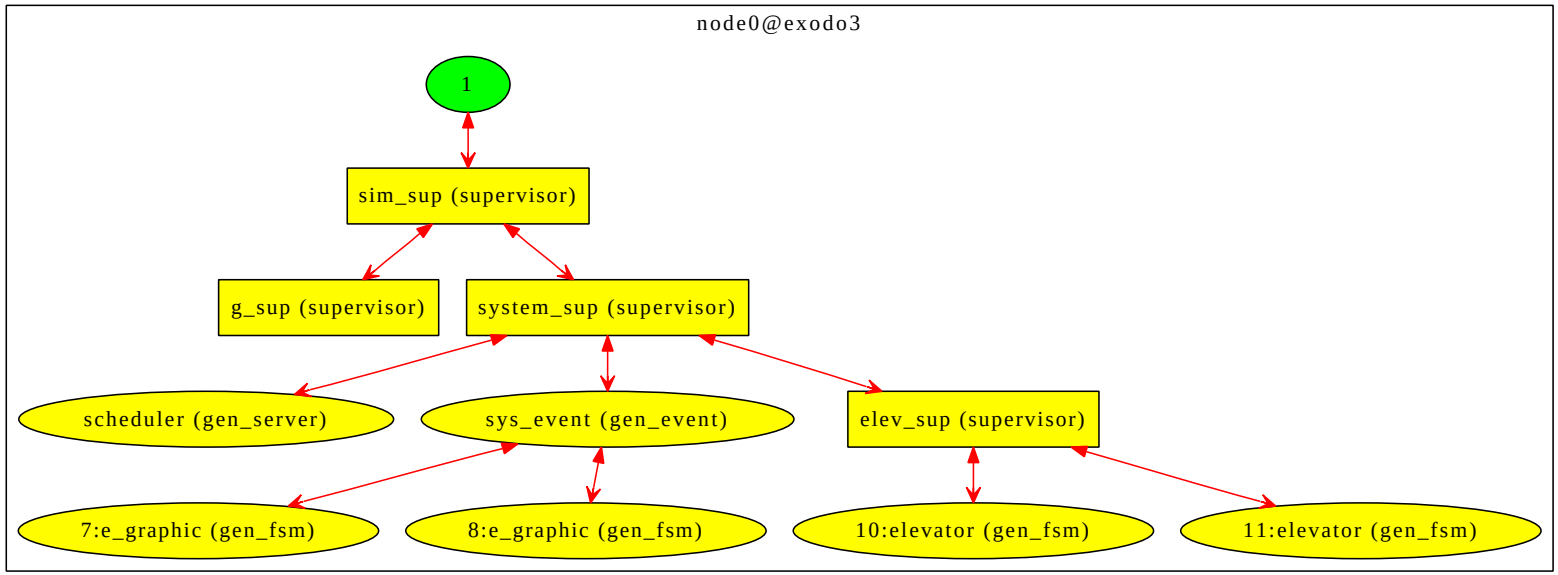
- Control software written in Erlang

Elevator Control Software

- Control software written in Erlang
- Static code complexity: around 1670 lines of code

Elevator Control Software

- Control software written in Erlang
- Static code complexity: around 1670 lines of code
- Dynamic complexity: around 10 processes (for two elevators)



Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*

Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*
- *An elevator only stops at a floor after receiving an order to go to that floor*

Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*
- *An elevator only stops at a floor after receiving an order to go to that floor*
- ...

Formulating Correctness Properties

- How to formulate a property like: “an elevator only stops at a floor after receiving an order to go to that floor”?

Formulating Correctness Properties

- How to formulate a property like: “an elevator only stops at a floor after receiving an order to go to that floor”?
- We can borrow an idea from runtime monitoring: we write a monitor that *detects* when the above property is violated

Formulating Correctness Properties

- How to formulate a property like: “an elevator only stops at a floor after receiving an order to go to that floor”?
- We can borrow an idea from runtime monitoring: we write a monitor that *detects* when the above property is violated
- Seen from another viewpoint we have created a *model* for the elevator system
- The model only describes a *small subset* of the behaviour of the elevator – but that is fine, it is what models are supposed to do

  So we have to write more monitors and properties. . .

What does a monitor do?

- Is run in parallel with the program

What does a monitor do?

- Is run in parallel with the program
- Has an internal state, which can be updated when the program does a *significant* action (or something happens – *a button press*)

What does a monitor do?

- Is run in parallel with the program
- Has an internal state, which can be updated when the program does a *significant* action (or something happens – *a button press*)
- The monitor should signal an error if an action happens in an incorrect state

A Correctness Monitor

Which elevator actions do the monitor need to react to?

A Correctness Monitor

Which elevator actions do the monitor need to react to?

- Button presses in the elevator

A Correctness Monitor

Which elevator actions do the monitor need to react to?

- Button presses in the elevator
- Button presses at each floor

A Correctness Monitor

Which elevator actions do the monitor need to react to?

- Button presses in the elevator
- Button presses at each floor
- The arrival of the elevator at a floor

State and Correctness Check

- What is the state of the monitor?

State and Correctness Check

- What is the state of the monitor?

  A data structure that remembers orders to go to a certain floor

State and Correctness Check

- What is the state of the monitor?

  A data structure that remembers orders to go to a certain floor

- What is the correctness check?

State and Correctness Check

- What is the state of the monitor?

  A data structure that remembers orders to go to a certain floor

- What is the correctness check?

  When the elevator arrives at a floor, the order to do so is in the monitor state

Checking Monitors

How to **Check** a Monitor?

- Well we can use runtime monitoring = testing

Checking Monitors

How to **Check** a Monitor?

- Well we can use runtime monitoring = testing
- **Note:** since the elevator control system is a nondeterministic reactive system we have to rerun the same test many times! (not so for sequential software)

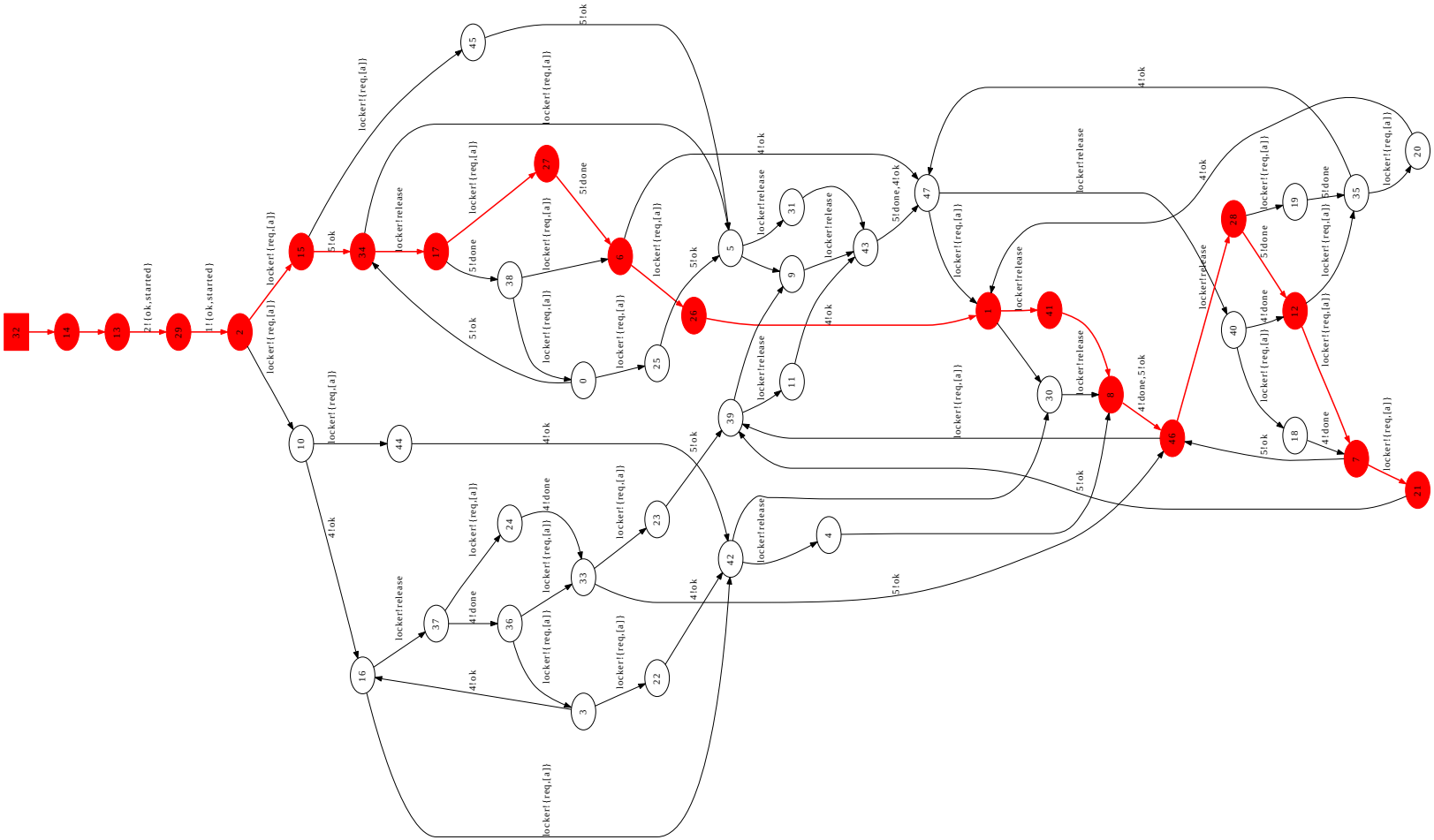
Checking Monitors

How to **Check** a Monitor?

- Well we can use runtime monitoring = testing
- **Note:** since the elevator control system is a nondeterministic reactive system we have to rerun the same test many times! (not so for sequential software)
- Using model checking is a viable alternative

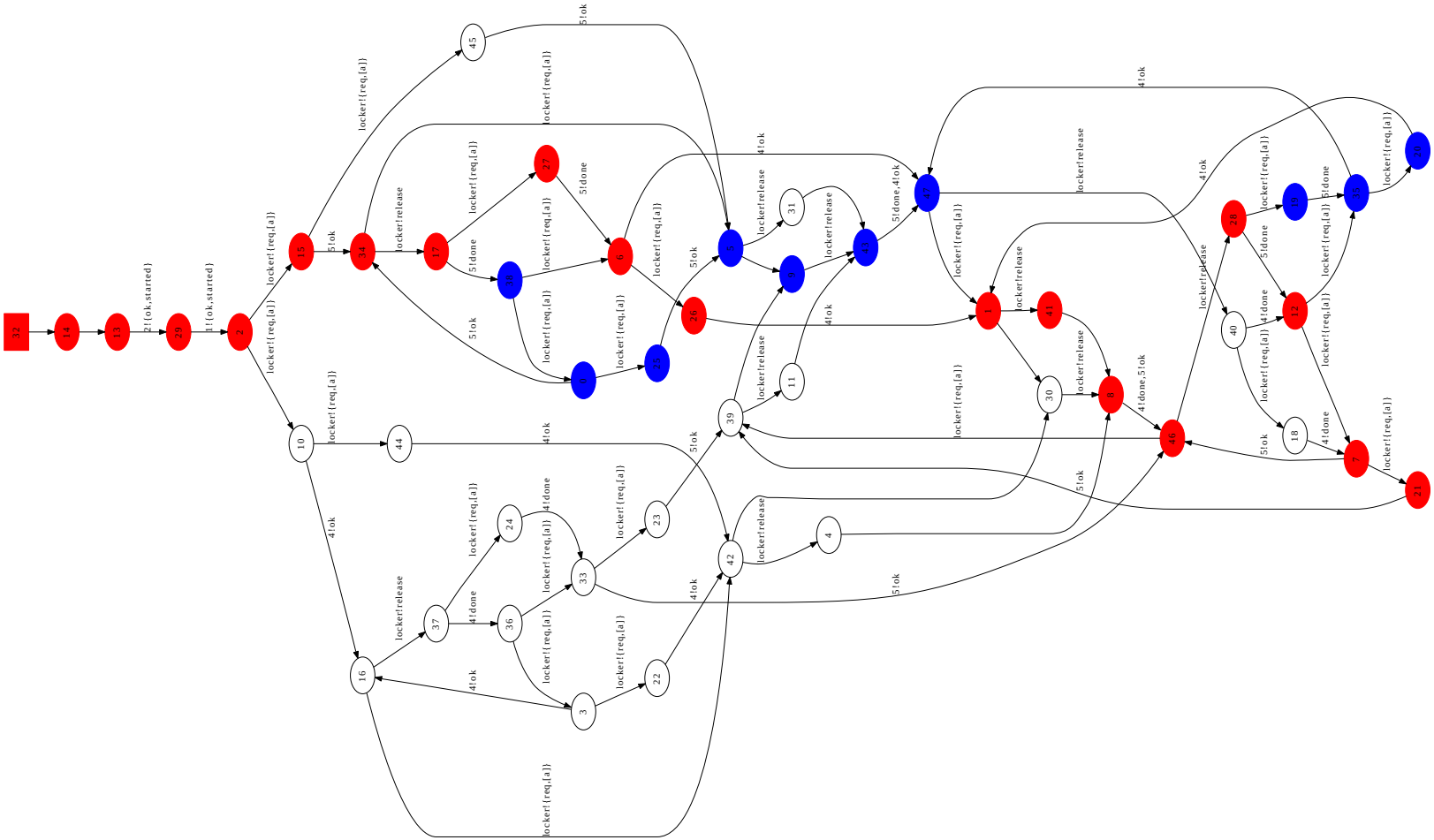
Model Checking vs Random Testing

Random testing explores one path through the program:



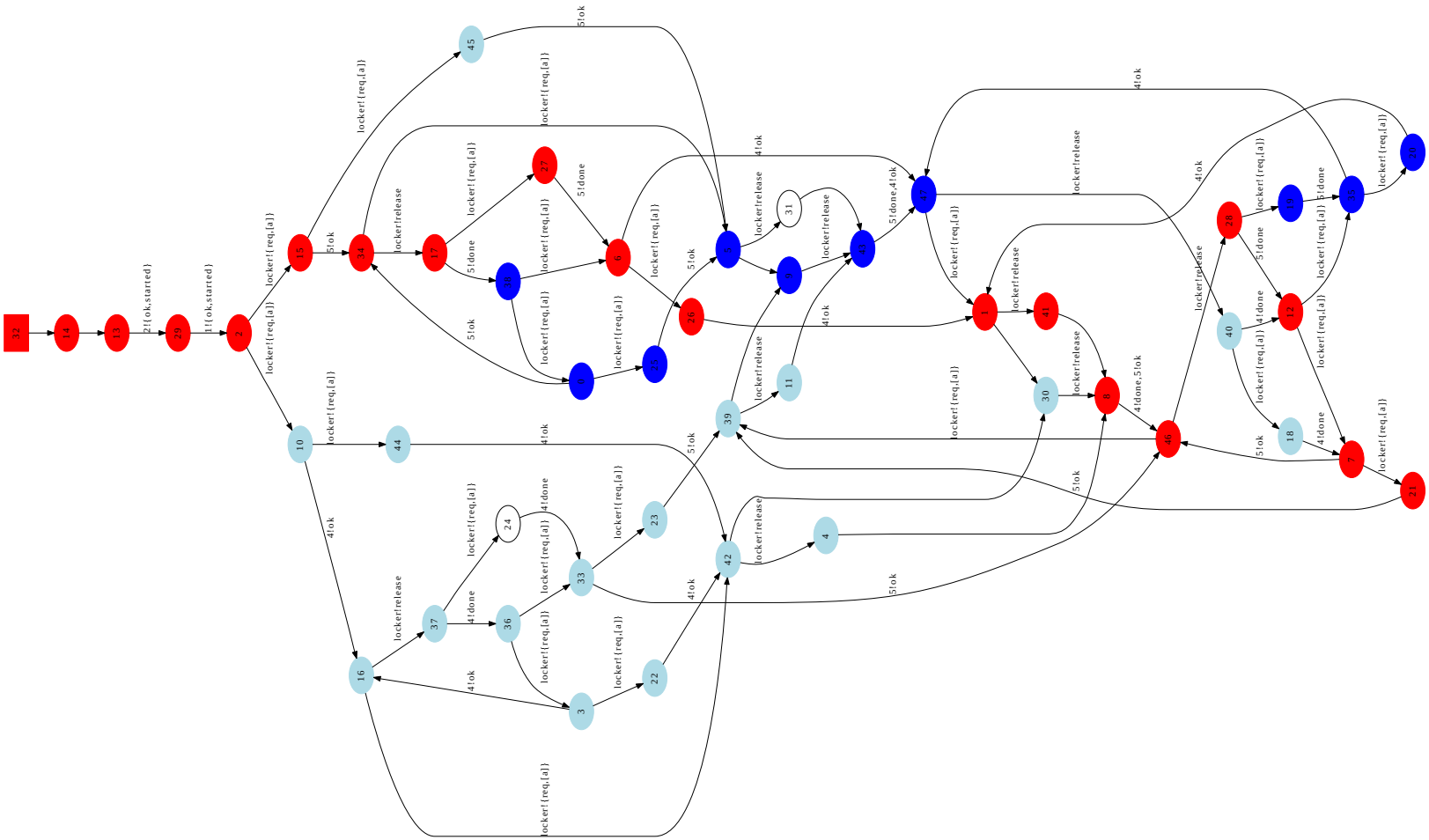
Model Checking vs Random Testing

With repeated tests the coverage improves:



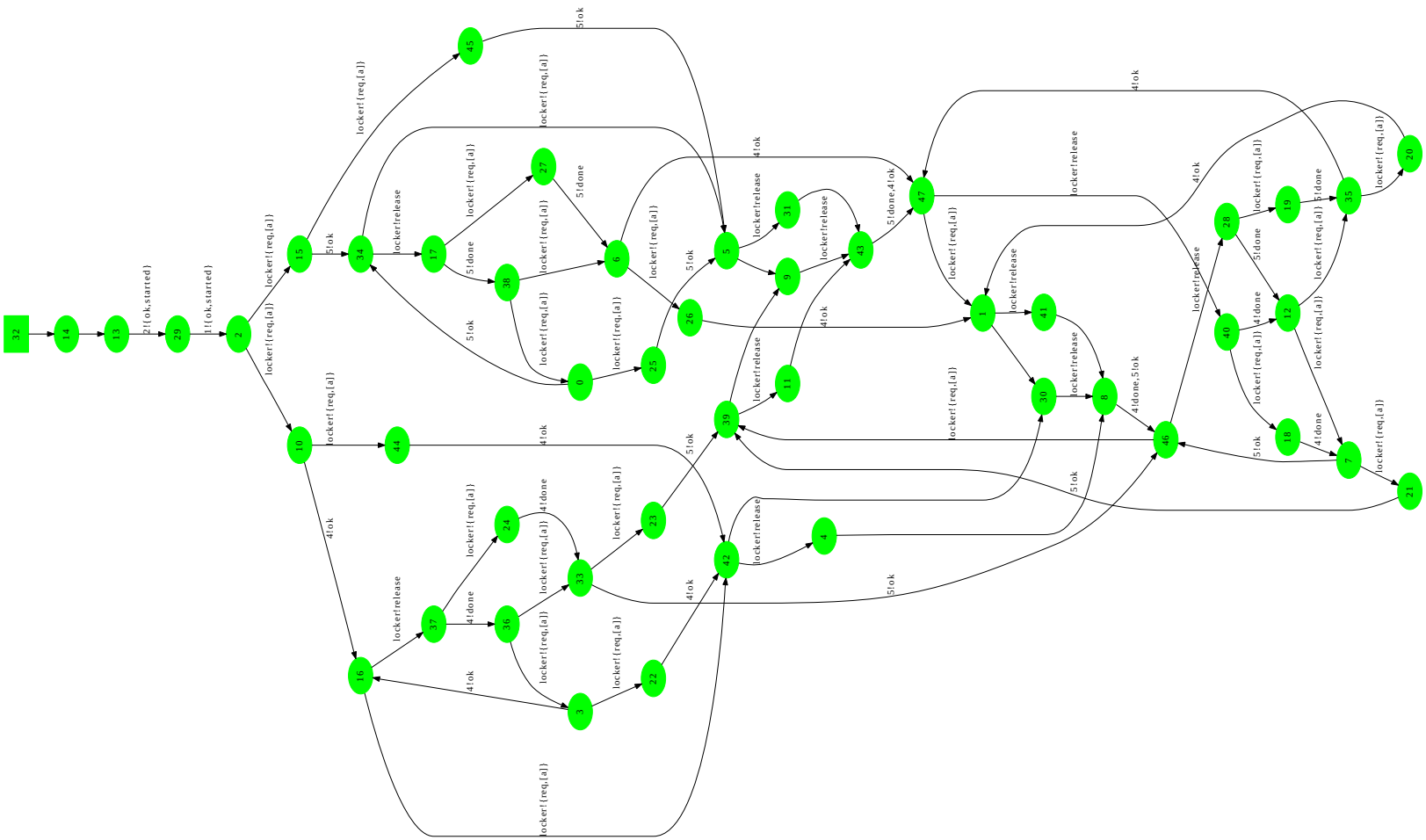
Model Checking vs Random Testing

But even after a lot of testing some program states may not have been visited:



Model Checking vs Random Testing

Model checking can guarantee that all states are visited, without revisiting states



Model Checking the Lift Example

- We use the **McErlang** model checker

Model Checking the Lift Example

- We use the **McErlang** model checker
- Correctness property spec:

```
stateChange(_,FloorReqs,Action) ->
  case Action of
    {f_button,Floor} ->
      ordsets:add_element(Floor,FloorReqs);
    {e_button,Elevator,Floor} ->
      ordsets:add_element(Floor,FloorReqs);
    {stopped_at,Elevator,Floor} ->
      case ordsets:is_element(Floor,FloorReqs) of
        true -> FloorReqs;
        false -> throw({bad_stop,Elevator,Floor})
      end;
    _ -> FloorReqs
  end.
```

- Uses ordered sets (`ordsets`) to store the set of floor orders (the state of the monitor)

Scenarios to Check

- How to choose which scenarios to check?
- Scenario $\equiv$ combinations of button presses

Scenarios to Check

- How to choose which scenarios to check?
- Scenario $\equiv$ combinations of button presses
- We use a “testing style” where a lot of small scenarios are checked:
  - Floor button 1 pressed
  - Floor button 2 pressed, Elevator button 1 pressed
  - Elevator button 2 pressed, Floor button 2 pressed, Floor button 2 pressed
  - …

Scenarios to Check

- How to choose which scenarios to check?
- Scenario $\equiv$ combinations of button presses
- We use a “testing style” where a lot of small scenarios are checked:
  - Floor button 1 pressed
  - Floor button 2 pressed, Elevator button 1 pressed
  - Elevator button 2 pressed, Floor button 2 pressed, Floor button 2 pressed
  - …
- But because we are using model checking we explore every scenario fully

More Correctness Properties

- Refining the floor correctness property:

  An elevator only stops at a floor after receiving an order to go to that floor, if no other elevator has met the request

  (implemented as a monitor that keeps a set of floor requests; visited floors are removed from the set)

Other Correctness Properties

- The floor correctness property is a safety property (*nothing bad ever happens*)

Other Correctness Properties

- The floor correctness property is a safety property (*nothing bad ever happens*)
- A Liveness property:
  If there is a request to go to some floor, eventually some elevator will stop there

Other Correctness Properties

- The floor correctness property is a safety property (*nothing bad ever happens*)
- A Liveness property:
  If there is a request to go to some floor, eventually some elevator will stop there
- In temporal logic:

```
always
   (fun go_to_floor/3) =>
      next(eventually (fun stopped_at_floor/3))
```

- The state predicate `fun go_to_floor/3` is satisfied when an elevator has received an order to go to a floor
- The state predicate `fun stopped_at_floor/3` is satisfied when an elevator stops at a floor

Model Checking

Conclusions:

- **Positive**: automatic technique, can prove difficult (concurrent) programs correct
- **Negative**:
  - model construction is far from automatic, great care needed in constructing a verifiable model
  - how to make sure the right properties are checked?